

Functional abstraction for programming multi-level architectures : formalisation and implementation

Victor Allombert

▶ **To cite this version:**

Victor Allombert. Functional abstraction for programming multi-level architectures : formalisation and implementation. Programming Languages [cs.PL]. Université Paris-Est, 2017. English. NNT : 2017PESC1016 . tel-01693568

HAL Id: tel-01693568
https://theses.hal.science/tel-01693568

Submitted on 26 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Paris-Est

École doctorale MSTIC

Thèse de Doctorat

pour obtenir le titre de
Docteur de l'Université Paris-Est
Spécialité : Informatique

défendue par
Victor Allombert

Functional abstraction for programming Multi-level architectures: Formalisation and implementation

Directeur de thèse : Frédéric Gava

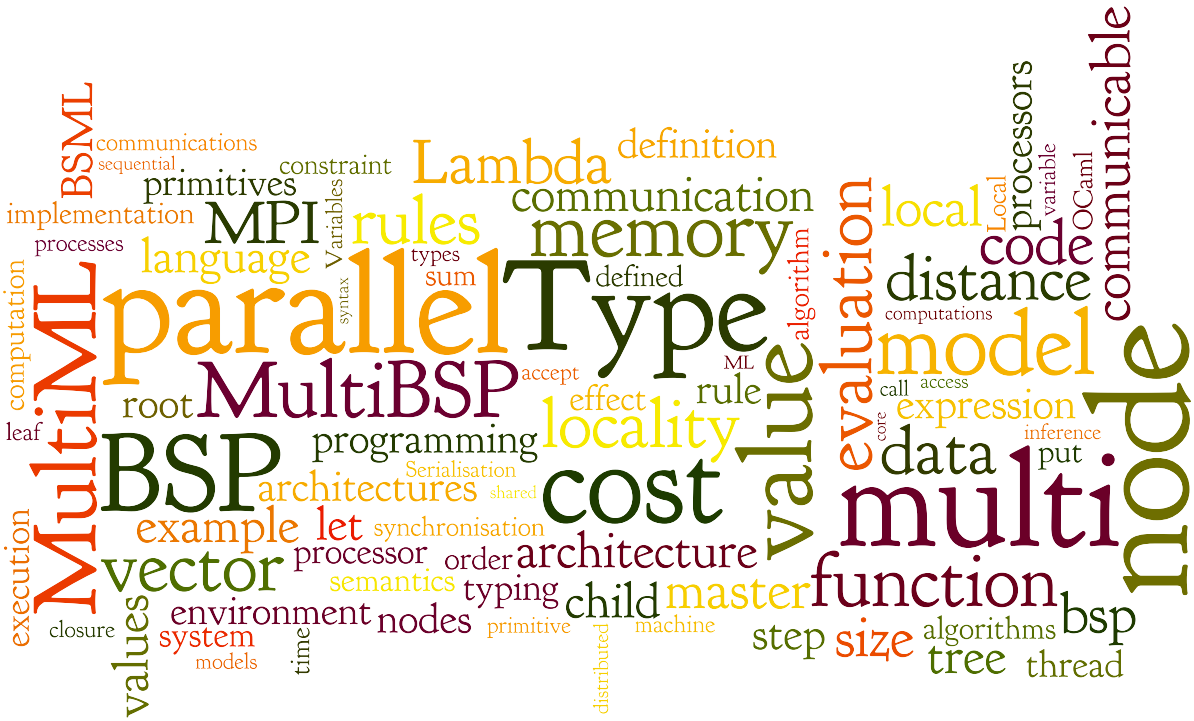
soutenue le 7 Juillet 2017 devant le jury composé de :

Rapporteurs/Referees :
Kevin hammond ..... University of Saint Andrews
Christoph kessler ..... Linköping University

Examinateurs/Examiners :
Catherine dubois ..... ensiie
Julia lawall ..... inria
Daniele varacca ..... Université Paris-Est Créteil

Encadrants/Supervisors :
Frédéric gava ..... Université Paris-Est Créteil
Julien tesson ..... Université Paris-Est Créteil

UNIVERSITÉ PARIS-EST



Thèse préparée au LACL
(Laboratoire d'Algorithmique, Complexité et Logique)
https://lacl.fr

LACL
Faculté des Sciences et Technologie
61 avenue du Général De Gaulle
94 010 Créteil

Abstract

Functional abstraction for programming multi-level architectures: Formalisation and implementation

From personal computers using an increasing number of cores, to supercomputers having millions of computing units, parallel architectures are the current standard. The high performance architectures are usually referenced to as *hierarchical*, as they are composed from clusters of multi-processors of multi-cores. Programming such architectures is known to be notoriously difficult. Writing parallel programs is, most of the time, difficult for both the algorithmic and the implementation phase. To answer those concerns, many structured models and languages were proposed in order to increase both expressiveness and efficiency. Among other models, MULTI-BSP is a bridging model dedicated to hierarchical architecture that ensures efficiency, execution safety, scalability and cost prediction. It is an extension of the well known BSP model that handles *flat* architectures.

In this thesis we introduce the MULTI-ML language, which allows programming MULTI-BSP algorithms "à la ML" and thus, guarantees the properties of the MULTI-BSP model and the execution safety, thanks to a ML type system. To deal with the multi-level execution model of MULTI-ML, we defined formal semantics which describe the valid evaluation of an expression. To ensure the execution safety of MULTI-ML programs, we also propose a typing system that preserves replicated coherence. An abstract machine is defined to formally describe the evaluation of a MULTI-ML program on a MULTI-BSP architecture. An implementation of the language is available as a compilation toolchain. It is thus possible to generate an efficient parallel code from a program written in MULTI-ML and execute it on any hierarchical machine.

Keywords: Parallel programming • MULTI-BSP • ML • Programming language • Parallel execution safety

Résumé

Abstraction fonctionnelle pour la programmation d'architecture multi-niveaux : Formalisation et implantation

Les architectures parallèles sont de plus en plus présentes dans notre environnement, que ce soit dans les ordinateurs personnels disposant des dizaines d'unités de calculs jusqu'aux super-calculateurs comptant des millions d'unités. Les architectures haute performance modernes sont généralement constituées de grappes de multiprocesseurs, elles-mêmes constituées de multi-cœurs, et sont qualifiées d'architectures *hiérarchiques*. La conception de langages pour de telles architectures est un sujet de recherche actif car il s'agit de simplifier la programmation tout en garantissant l'efficacité des programmes. En effet, écrire des programmes parallèles est, en général, plus complexe tant au point de vue algorithmique qu'au niveau de l'implémentation. Afin de répondre à cette problématique, plusieurs modèles structurés ont été proposés. Le modèle logico-matériel BSP définit une vision structurée pour les architectures parallèles dites *plates*. Afin d'exploiter les architectures actuelles, une extension adaptée aux architectures hiérarchiques a été proposée : MULTI-BSP. Tout en préservant la philosophie BSP, ce modèle garantit efficacité, sécurité d'exécution, passage à l'échelle et prédiction de coût.

Cette thèse s'articule donc autour de cette idée et propose de définir MULTI-ML, un langage basé sur le modèle logico-matériel MULTI-BSP, garantissant les propriétés énoncées ci-dessus. Afin de pouvoir garantir la sécurité d'exécution des programmes MULTI-ML, nous proposons une sémantique formelle ainsi qu'un système de type afin d'accepter uniquement des programmes bien formés. De plus, nous proposons une machine abstraite permettant de décrire formellement l'évaluation d'un programme MULTI-ML sur une machine MULTI-BSP. Une implantation du langage, développé dans le cadre de cette thèse, permet de générer un code exécutable. Il est donc possible d'exécuter, efficacement, des algorithmes MULTI-BSP écrits à l'aide de MULTI-ML sur diverses machines hiérarchiques.

Mots-clés: Programmation parallèle • MULTI-BSP • ML • Langage de programmation • Sûreté des langages

Résumé long

Abstraction fonctionnelle pour la programmation d'architecture multi-niveaux : Formalisation et implantation

Les architectures parallèles sont de plus en plus présentes dans notre environnement, que ce soit dans les smartphones disposant des dizaines d'unités de calculs jusqu'aux super-calculateurs comptant des millions d'unités. À la vitesse où la technologie évolue, les machines haute performance d'aujourd'hui seront semblables à l'embarqué de demain. Comme une grande diversité de machines aux caractéristiques différentes existe, il est nécessaire de simplifier leur programmation tout en apportant des garanties en terme de sûreté d'exécution. Nous pouvons distinguer deux types d'architectures dominantes : les machines à mémoire partagée et les machines à mémoire distribuée. Les machines à mémoire partagées sont souvent assimilées aux multi-cœurs, où un ensemble de processus accèdent à une mémoire commune afin de communiquer. Au contraire, les machines à mémoire distribuées, comme les clusters, nécessitent l'utilisation de primitives de communication explicite afin d'échanger des données. Dans le monde du calcul haute performance (High Performance Computiong : HPC), il est courant d'être confronté à des machines combinant plusieurs niveaux de mémoires partagées et distribuées. Ces dernières, qualifiées d'architectures hiérarchiques, disposent de caractéristiques diverses liées à leur nature multi-niveaux. Les dernières générations de puces dites *many core* sont également hiérarchiques, au vu des différents niveaux de mémoire et aux réseaux dédiés, directement intégrés à la carte. Des langages pour programmer de telles architectures existent, mais leur utilisation n'est pas triviale. En effet, écrire des programmes parallèles est, en général, plus complexe tant au point de vue algorithmique qu'au niveau de l'implémentation. La gestion fine des communications et des niveaux de mémoires, les inter-blocages et le non-déterminime sont autant de problèmes difficiles à appréhender en programmation parallèle.

Afin d'éviter ces problèmes d'implémentation, plusieurs modèles ont été proposés. En plus de structurer la programmation, ces modèles apportent des propriétés importantes et définissent des modèles de coût ainsi que des modèles d'exécutions pertinents. De nombreuses familles ont été suggérées au fil du temps. On peut notamment distinguer des modèles adaptés aux architectures dites "*plates*" (composées d'un seul niveau de mémoire), proposant (ou non) des capacités de sous-synchronisation des unités de calculs, jusqu'à des modèles prenant en charge les architectures hiérarchiques. Bien que certaines propositions soient dédiés à des architectures précises, d'autres sont beaucoup plus génériques. Par exemple, une approche par squelette algorithmique permet d'utiliser des *motifs* de programmation adaptés à différentes problématiques tout en étant approprié à de nombreuses architectures. Dans le cadre de cette thèse, nous allons prin-

cipalement nous intéresser à des modèles *logico-matériels* permettant de structurer l'approche algorithmique tout en reflétant l'architecture physique des machines. Une telle méthodes permet de s'affranchir des détails d'implémentation liés aux communications et aux optimisations spécifiques à chaque architecture.

Bien que l'usage de primitives de communication bas niveau est souvent considéré comme un gage de performance, l'utilisation de modèles de programmation structurée ne se fait pas toujours au détriment de l'efficacité. En plus de permettre au programmeur de se concentrer sur l'algorithme en tant que tel, ces modèles assurent des propriétés intéressantes, comme l'efficacité, la sécurité d'exécution et la scalabilité. De plus, un modèle suffisamment générique permet d'écrire des algorithmes qui seront compatibles avec des nombreuses architectures, sans aucune modification.

Le modèle logico-matériel quasi synchrone BSP (Bulk Synchronous Parallelism), propose une approche structurée, basée sur un modèle d'exécution en super-étapes tout en garantissant les propriétés énoncées précédemment. Une super-étape est composée d'une phase de calcul suivie d'une phase de communication, où tous les processeurs de la machines sont concernés. Une barrière de synchronisation globale met fin à une super-étape, afin de synchroniser toute la machine, pour pouvoir exécuter une nouvelle super-étape. Cette approche quasi synchrone permet de structurer un algorithme et ainsi de garantir l'absence d'inter-blocages lors de l'exécution de celui-ci. De plus, BSP propose un modèle de coût qui permet de raisonner sur la complexité d'un algorithme. Il prend également en compte les caractéristiques de la machine exécutant un algorithme, en se basant sur : (1) le nombre de processeurs et leur vitesse de calcul ; (2) la vitesse de communications inter-processeurs ; (3) le temps de synchronisation de la machine. Ainsi, il est possible d'offrir une prédiction de performances détaillée pour un algorithme BSP, en se basant sur sa complexité et sur les caractéristiques d'une machine.

Bien que ce modèle soit largement adopté par la communauté du calcul parallèle, il n'est pas complètement adapté aux architectures hiérarchiques actuelles. En effet, BSP a une approche qui consiste à manipuler une machine comme un ensemble de processeurs communiquants par un seul et même réseau. Comme les architectures hiérarchiques sont munies de nombreux niveaux ayant des capacités bien différentes, BSP ne permet pas de tirer partie de tout le potentiel de telles machines. Afin de ternir compte du caractères multi-niveau des machines hiérarchiques, une extension du modèle BSP a été proposée : MULTI-BSP. Ce nouveau modèle logico-matériel propose une approche dédiée aux architectures hiérarchiques tout en préservant la marque de fabrique propre à BSP. Une machine est vue comme un ensemble de composants imbriqués sous forme d'arbre, où les nœuds sont des mémoires et les feuilles sont des unités de calcul. Une machine MULTI-BSP est donc représentée comme un arbre de composants imbriqués caractérisant les différentes spécificités d'une machine hiérarchique : chaque nœud dispose d'une quantité de mémoire ainsi qu'une capacité à communiquer avec ses sous-nœuds. Le modèle d'exécution de MULTI-BSP permet de mettre en place un système de sous-synchronisation des composants. Ainsi, la synchronisation d'un composant nécessite que tous ses sous-composants soient également synchronisés. À l'image de BSP, MULTI-BSP garantit la sécurité de l'exécution d'un programme en excluant les problèmes d'inter-blocages. La généricité de ce modèle permet d'exprimer des algorithmes sur une grande variété d'architectures, et notamment sur les architectures hiérarchiques.

Bulk **S**ynchronous Parallel **ML** (BSML) étend le langage fonctionnel d'ordre supérieur OCAML pour la programmation parallèle structurée quasi synchrone BSP. Déterministe et sans interblocage, ce paradigme est adapté à la conception et à la vérification de programmes parallèles sûrs. Le principe de BSML repose sur un ensemble restreint de primitives utilisées pour manipuler une structure de donnée parallèle particulière, appelée *vecteur parallèle*. Un vecteur parallèle représente une donnée qui est distribuée sur l'ensemble des processeurs d'une machine BSP. Des primitives de communication synchrones permettent d'effectuer des communications, via des vecteurs parallèles, entre les processeurs de la machine. BSML est un langage conçu pour

la programmation parallèle et est également adapté à la vérification de programmes. Il est possible de générer un programme parallèle certifié à partir d'une preuve de programme écrite via l'assistant de preuve COQ. Cependant, BSML est défini pour programmer des architectures plates de type BSP. Il semble naturel de vouloir proposer une approche similaire à BSML permettant la programmation efficace d'architectures hiérarchiques, basée sur le modèle MULTI-BSP.

Cette thèse s'articule naturellement autour de cette idée et propose de définir MULTI-ML, un langage basé sur le modèle logico-matériel MULTI-BSP, garantissant efficacité, prédiction de performances, sécurité d'exécution et passage à l'échelle. Ce nouveau langage de programmation peut être vu comme une extension de BSML. En effet, MULTI-ML utilise des primitives empruntées à BSML ainsi que la notion de vecteur parallèle, tout en ajoutant une syntaxe particulière permettant d'exprimer une récursivité sur les différents niveaux des machines hiérarchiques. Chaque nœud de la machine est donc une unité qui peut effectuer des calculs locaux mais également distribuer données et calculs sur ses fils. L'approche MULTI-ML consiste à interpréter une machine MULTI-BSP comme un arbre où les calculs vont être propagés de la racine (composant le plus haut) vers les feuilles (composants les plus bas). Cette approche *top-down* permet une programmation récursive et élégante exprimant un parallélisme multi-niveau. Cette notion de récursion permettant de parcourir une machine hiérarchique est au cœur de MULTI-ML, et est implémentée sous forme d'une fonction récursive particulière : la *multi-fonction*. Une multi-fonction est composée de deux parties, l'une visant à manipuler un nœud et l'autre une feuille. Le code du nœud permet de manier et distribuer des données ainsi que de propager des calculs sur ses fils. Au contraire, le code des feuilles est exclusivement dédié au calcul intensif. Lors de l'évaluation d'une multi-fonction, la récursion est initiée sur la racine de l'arbre MULTI-BSP, en exécutant le code du nœud. La récursion est effectuée via l'appel récursif de la multi-fonction à l'intérieur d'un vecteur parallèle. En effet, un vecteur parallèle défini dans un nœud manipule directement la mémoire de ses fils. Ainsi, la récursion se propage sur toute l'architecture : de la racine vers les feuilles. Un nœud étant un niveau de l'architecture représentant une sous-machine BSP, il peut utiliser certaines primitives empruntées à BSML. La construction des vecteurs parallèles peut se faire par l'intermédiaire de la syntaxe `<< >>` (section parallèle) et il est possible d'utiliser le sucre syntaxique `$v$` afin d'accéder à la valeur du vecteur `v` à l'intérieur de la section parallèle. Afin de pouvoir faire référence aux valeurs définies dans la mémoire d'un nœud à partir d'une section parallèle, nous proposons un nouveau sucre syntaxique. En effet, une valeur d'un nœud doit être explicitement copiée dans la mémoire de ses fils afin d'y faire référence dans une section parallèle. Pour ce faire, annoter une variable `x` par `#x#`, permet d'exprimer une communication explicite de la variable `x` (contenue dans la mémoire d'un nœud) vers la mémoire des fils. En MULTI-ML, il est également possible de créer des vecteurs parallèles en utilisant la primitive `mkpar`. Cependant, son comportement est différent de sa version BSML. Ici, la fonction donnée en argument de `mkpar` ne va pas être exécutée sur chacun des fils d'un nœud, mais va l'être localement. La fonction est donc exécutée autant de fois que le nœud a de fils, et l'ensemble des résultats générés sont ensuite distribués, un à un aux fils, afin de créer un vecteur parallèle. L'intérêt de cette modification est de minimiser les communications lors de la création d'un vecteur parallèle. Par exemple, le découpage d'une liste via `mkpar` se fait localement et chaque sous-partie est communiquée. Au contraire, avec l'utilisation de la section parallèle, il aurait été nécessaire de répliquer la liste sur tous les fils pour finalement la découper localement. Des primitives de communication synchrones (impliquant une barrière de synchronisation) sont également disponibles. La primitive `proj`, qui est le dual de la création d'un vecteur, est utilisée pour rendre les éléments d'un vecteur parallèle disponible, sous forme fonctionnelle, au niveau du nœud appelant. `put` permet d'effectuer des communications entre tous les processeurs disponibles à un nœud de l'architecture. L'utilisation des primitives de communication synchrone

`proj` et `put` est similaire à leur version BSML.

Afin de pouvoir exploiter les différents niveaux de mémoire de l'architecture, nous proposons une nouvelle structure de données parallèle, nommée tree (*arbre*), représentant la distribution des données à travers tous les niveaux. Un ensemble restreint de primitives permet de créer et de manipuler de telles structures. Par exemple, l'opérateur `at` permet, à un composant de l'architecture, d'accéder à l'élément d'un arbre le concernant.

Contrairement au modèle MULTI-BSP, il est possible d'exécuter des calculs sur les nœuds d'une architecture. Cette petite dissemblance s'explique dans le fait qu'il paraît plus naturel d'avoir la possibilité de distribuer des données directement dans le code des nœuds concernés. Cependant, une implémentation de MULTI-ML qui s'apparente effectivement au modèle MULTI-BSP est tout à fait possible. Il suffit de simuler l'exécution des calculs des nœuds sur les feuilles de l'architecture.

Le mécanisme d'exécution d'un programme parallèle, et à fortiori d'un programme MULTI-ML, est particulier. En effet, l'exécution en parallèle de nombreux processus est généralement difficile à appréhender. Afin de clarifier les comportements acceptables d'un programme MULTI-ML, nous formalisons une sémantique opérationnelle (grand pas). Celle-ci permet de décrire en détail les comportements d'une expression (ou programme) exprimée dans la syntaxe MULTI-ML. Par exemple, nous décrivons le mécanisme de distribution des calculs et des données d'un nœud vers ses fils. Par l'intermédiaire d'une sémantique grand pas co-inductive, il nous est également possible de décrire les évaluations qui divergent, en plus des évaluations qui terminent.

De plus, comme énoncé précédemment, le modèle MULTI-BSP propose un modèle de coût. Nous définissons donc une sémantique annotée décrivant les différents coût des différentes constructions du langage. Ainsi, nous décrivons le comportement des composants engagés dans l'évaluation d'une expression, tout en quantifiant les calculs et les communications nécessaires. Il est ainsi possible de raisonner sur le coût algorithmique d'un programme écrit en MULTI-ML via une sémantique formelle.

Comme en ML, nous proposons un système de type permettant de nous assurer que les programmes soient bien formés. Bien que proche du système de type d'OCAML, celui de MULTI-ML doit prendre en charge un degré de complexité supplémentaire lié au parallélisme. En effet, la possibilité d'évaluer des expressions sur différents niveaux implique une gestion fine de ces niveaux à l'intérieur du système de type. Pour ce faire, nous avons choisi de développer un système de type avec effet, annotation de types et contraintes. Ces annotations reflètent les capacités d'un niveau de l'architecture. Nous avons : `m` pour une évaluation concernant tous les composants de la machine ; `b` pour un nœud, c'est-à-dire une machine pouvant exécuter du code BSML ; `c` pour les expressions communicables dans un vecteur parallèle et `l` pour ces mêmes expressions lorsqu'elles sont non communicables ; finalement, `s` concerne les exécutions séquentielles des feuilles. Une expression dite *communicable* est une expression qui a un sens dans la mémoire d'une autre unité. Typiquement, les *pointeurs* (ou références) sur les données d'un nœud ne sont pas exploitables à un autre niveau de l'architecture, ils sont donc annotés `l` (pour *local*). Un type est donc annoté par une localité représentant l'endroit où l'expression a été évaluée et donc, où une donnée réside. Par exemple, le type `int_m` dénote un entier qui est défini au niveau `m`. De plus, la notion d'effet est intégré à notre système de type via, notamment, l'annotation des types flèches. La notation `-(b)->` représente un effet latent, c'est-à-dire, un effet qui sera émis lors de l'application d'une fonction. Dans ce cas, l'exemple représente une fonction effectuant des calculs parallèles devant être effectués au niveau `b`. Ainsi, l'application d'une telle valeur à un autre niveau doit être prohibée.

Le système de type conçu dans cette thèse utilise ces localités afin de déterminer si une expression est valide à un niveau en particulier de l'architecture mais également afin de vérifier si

une expression peut être communiquée vers un autre niveau. Il est également possible de prévenir l'imbrication inappropriée des structures parallèles comme l'imbrication de vecteurs parallèles (comme en BSML) ou l'imbrication d'arbres. De plus, le système de type garantit la cohérence répliquée afin d'éviter les inter-blocages ainsi que les comportements non déterministes. Par exemple, l'exécution d'un code menant à l'exécution d'une barrière de synchronisation sur une sous-partie des processeurs d'un nœud n'est pas acceptée. Le système de type de MULTI-ML garantit donc la sécurité d'exécution de tels programmes sur des architectures hiérarchiques.

Afin de décrire formellement le modèle d'exécution d'un programme MULTI-BSP, nous proposons une machine abstraite. Celle-ci décrit en détail les suites d'instructions nécessaires à l'exécution d'un programme MULTI-ML et permet de lier le comportement d'un langage de haut niveau avec celui de la machine physique, dit *bas niveau*. Ce langage intermédiaire est utilisé pour dérouler, étape par étape, l'exécution d'un programme sans se soucier des détails d'implémentation liés aux composants physiques (hardware). L'utilisation d'un schéma de compilation générique pour une machine abstraite parallèle permet également de garantir une certaine confiance par rapport à l'implémentation d'un compilateur. Dans un premier temps, nous proposons une machine abstraite permettant une implantation rapide et relativement efficace. Afin de simuler une exécution hiérarchique, un processus *démon* est disponible pour chaque nœud et feuille de l'architecture. Ces processus vont attendre des ordres consistant à lancer l'exécution de différents codes. Les communications entre processus, bien que hiérarchiques, sont effectuées en utilisant des routines de communication asynchrones *point à point* (de type send/receive). Les ordres d'exécutions consistent, principalement, en une fermeture (contenant une fonction à exécuter) et en un ensemble de valeurs (nécessaires pour l'évaluation de la fermeture). Un ensemble de *briques élémentaires* est ainsi utilisé pour décrire l'ensemble des comportements et actions nécessaires pour l'exécution d'un programme MULTI-ML.

Grâce au modèle de coût proposé par le modèle MULTI-BSP, il est possible de définir formellement un coût d'exécution pour un programme MULTI-ML. Pour ce faire, nous proposons une sémantique naturelle (ou grand pas) qui intègre un coût pour l'évaluation de chaque expression. Ainsi, il est possible de comptabiliser l'ensemble des coûts engendrés par l'utilisation des primitives asynchrones lors des phases de synchronisation. L'utilisation d'une algèbre de coût, utilisant les paramètres MULTI-BSP liés à l'architecture permet de proposer une prédiction de coût réaliste, pour un algorithme MULTI-ML donné.

Actuellement, les sources du langage MULTI-ML (disponible sous licence *open source*) représentent approximativement 8.000 lignes de codes, réparties dans environ 100 fichiers (la vaste majorité étant écrite en OCAML). La distribution est structurée en différentes parties afin de proposer un langage facilement extensible. Nous pouvons notamment distinguer les différents étages de compilations, le typage, les modules de communications ainsi que le support d'exécution séquentiel (et *toplevel*) et parallèle. Afin de pouvoir expérimenter la programmation d'algorithmes en MULTI-ML, nous proposons une implémentation du langage qui peut être vue comme une extension du langage de programmation fonctionnelle OCAML. Une chaîne de compilation est disponible afin de pouvoir transformer un code MULTI-ML en exécutable. La chaîne de compilation est structurée en trois phases principales : (1) Le *frontend* (ou "partie dépendante du langage") permet de générer un arbre de syntaxe abstrait (AST) à partir d'un programme écrit en MULTI-ML. Pour ce faire, cette première phase doit effectuer une analyse lexicale et syntaxique, afin de vérifier que la norme du langage est respectée. De plus, le *front end* comprend la phase de typage, utilisée pour vérifier que le programme est bien formé et ainsi, garantir sa sécurité d'exécution. Cette phase permet également de récupérer de nombreuses informations comme les types, les erreurs ou les avertissements. (2) Le *middle-end* (ou "partie commune") permet de manipuler l'AST produit par le *frontend* afin d'y effectuer des transformations et optimisations.

Il produit un code intermédiaire générique. (3) Le *backend* utilise le résultat du *middle-end* afin de terminer la phase de compilation et de générer un code exécutable. Le produit du BACKEND est donc un exécutable qu'il est directement possible d'utiliser sur une architecture concrète. Pour la mise au point de MULTI-ML, nous avons choisi de reprendre les sources d'OCAML et de modifier directement l'analyse syntaxique (*lexer*) et l'analyse lexicale (*parser*). Ainsi, nous avons pu retirer les constructions qui ne sont pas encore disponibles en MULTI-ML (structures, modules, *etc.*), et qui seront intégrées au fur et à mesure de l'évolution du langage. Ensuite, nous avons ajouté les constructions et sucres syntaxiques propres à MULTI-ML (primitives de communication, multi-fonctions, *etc.*). Ces modifications nous permettent de reconnaître un ensemble d'expression limité, afin de générer un arbre de syntaxe abstrait (AST) valide. La validité de cet arbre est vérifiée par le système de type. Une fois que l'AST validé est généré, le MIDDLE-END se charge d'effectuer des transformations permettant d'obtenir un code intermédiaire adapté à l'architecture ciblée. Actuellement, nous sommes en mesure de générer un AST adapté à une machine séquentielle ainsi qu'une version dédiée aux architecture hiérarchiques (en utilisant des primitives MPI). C'est à partir de ce code intermédiaire que le BACKEND se charge de générer une exécutable, en utilisant le compilateur OCAML. En effet, l'AST produit par le *middle-end* est structuré à la manière d'un programme OCAML. La phase de compilation d'un programme MULTI-ML consiste en la transformation d'un code MULTI-ML en un code OCAML contenant une ensemble de primitives de communication. Le code généré est adapté et optimisé à l'architecture ciblée. L'exécutable final est obtenu via l'utilisation du compilateur OCAML optimisé.

Bien que nous avions la possibilité de reprendre le système de type d'OCAML afin de l'étendre, nous avons choisi une autre solution. En effet, à cause de la complexité des sources du système de typage d'OCAML, nous avons préféré construire le système de type ainsi que son inférence, à partir de zéro. Pour ce faire, nous avons choisi de mettre en place un système d'inférence basé sur un système de contraintes : PCB(X). Ce système d'inférence de type est composé de deux phases principales : (1) la génération de contraintes et (2) la résolution de contraintes. La génération de contraintes est effectuée à partir de l'AST produit pendant la phase de compilation. Les contraintes générées sont directement tirées des règles d'inférence de types proposées pour MULTI-ML. Les contraintes sont ensuite données à un algorithme de résolution de contraintes qui est composé de trois principales phases : (1) Unification de types, (2) unification de localités et (3) résolution des contraintes. C'est lors de la résolution de contraintes qu'il est possible de détecter les erreurs de programmation qui pourraient remettre en cause la sûreté d'exécution d'un programme MULTI-ML. Ainsi, nous sommes capable de détecter des erreurs spécifiques à MULTI-ML, comme l'imbrication de vecteurs parallèles, la manipulation de données avec des localités de définition incompatibles ou la tentative de communication de données *non communicables*.

En ce qui concerne l'implémentation MPI, nous avons choisi de proposer une approche modulaire. Celle-ci repose sur un module de communication qu'il est possible de choisir, en fonction du contexte d'exécution. Ce module décrit un ensemble de primitives élémentaires de bas niveau, qui permettent d'exprimer l'ensemble des constructions du langage. Actuellement, nous proposons un module de communication basé sur la librairie MPI, du fait de ses bonnes performances et de sa compatibilité avec de nombreuses architectures. Cependant, nous pouvons facilement imaginer l'utilisation d'une autre librairie, adaptée à un type d'architecture en particulier. De plus, cette approche modulaire nous permet de proposer une implémentation séquentielle. Celle-ci peut être utilisée à des fins de débugage ou pour simuler l'exécution de programmes sur des architectures imaginaires. Une boucle interactive (*toplevel*), à la manière d'OCAML et BSML, est également disponible.

Comme ce travail s'inscrit dans la volonté de proposer un langage efficace adapté au calcul parallèle, nous décrivons un algorithme implémenté à l'aide de MULTI-ML. Pour ce faire,

nous proposons un algorithme de calcul de nombres premiers basé sur le crible d'Eratosthène. Cet algorithme relativement naïf est implémenté par l'intermédiaire d'un algorithme de somme de préfixes où l'opérateur est défini conformément aux règles du crible d'Eratosthène. Afin de pouvoir évaluer les performances de cet algorithme, nous implémentons également une version BSML, qui fonctionne pareillement. Ainsi, nous pouvons comparer les performances de l'algorithme écrit en MULTI-ML par rapport à la version BSML, et ce, sur différentes quantités de données. Les tests pratiques sur différentes machines parallèles montrent que les performances de la version MULTI-ML sur de petites données est similaire à la version BSML. Cependant, avec une quantité importante de données, l'algorithme écrit MULTI-ML surpasse celui en BSML. Cette différence de performances s'explique par le fait que, grâce au modèle MULTI-BSP, la version MULTI-ML exploite au maximum les différents niveaux de mémoires. Les communications sont d'abord effectuées via les mémoires *rapides*, les mémoires *lentes* étant utilisées à la fin du calcul seulement. Au contraire, l'algorithme BSML effectue toutes ses communications au niveau de la mémoire *la plus haute*, c'est-à-dire via le réseau. Les nombreuses communications vont avoir tendance à engorger le réseau, ce qui résulte en une augmentation drastique du temps de communication, à cause du goulot d'étranglement généré par un trafic intense.

Mots-clés: Programmation parallèle • MULTI-BSP • ML • Langage de programmation • Sûreté des langages

Remerciements

Je tiens tout d'abord à remercier mon directeur de thèse, Frédéric GAVA, qui m'a guidé tout au long de mes travaux. Merci pour ce flot d'idées partagées durant de nombreuses sessions de travail et pour sa disponibilité. Je souhaite également remercier Julien TESSON pour ses conseils bienveillants, sa sympathie, et son accueil, chez lui, dans ses grands fauteuils propices à la réflexion. Je n'oublie pas non plus Frédéric LOULERGUE, sans qui cette thèse n'aurait certainement jamais débuté.

Un grand merci à Catherine DUBOIS pour avoir présidé mon jury. Merci à Kevin HAMMOND et Christoph KESSLER de m'avoir fait l'honneur de rapporter ma thèse. Je remercie également Julia LAWALL et Daniele VARACCA d'avoir participé à mon jury. Merci pour vos remarques et vos questions qui m'ont permis d'améliorer mon manuscrit et de prendre du recul sur mon travail.

Merci également à Quentin, Martin, Sergiu, Thomas, Nghi, Yoann et aux autres. Cette génération de doctorants extraordinaires avec qui j'ai eu la chance de partager de bons moments. Je n'oublierai pas ces discussions tordues et encore moins ces tableaux aux dessins toujours plus improbables. C'est tout naturellement que je remercie les membres du LACL mais aussi ceux du LIFO pour leur accueil et leur soutien durant ces années.

À mes amis qui m'ont toujours permis de déconnecter après de longues journées de concentration, merci. J'en suis certain, ils se reconnaîtront.

Marie, je te remercie pour ce soutien indispensable jusqu'au sprint final.

Merci à mes parents, ma sœur et plus généralement à ma famille, pour m'avoir soutenu et encouragé tout au long de ces années.

Finalement, cher lecteur, merci de consacrer un peu de temps à la lecture de ce manuscrit. J'espère que vous y trouverez bon nombre d'informations pertinentes.

Contents

Abstract (English) iii

Résumé (French) v

Résumé long (French) vii

Remerciements xv

Contents xvii

List of Figures xxi

List of Tables xxv

1 Introduction 1
- 1.1 Context of work: generalities and background . . . 1
  - 1.1.1 Background . . . 2
  - 1.1.2 Generalities concerning solutions proposed in this thesis . . . 3
- 1.2 Parallel and distributed architectures . . . 3
  - 1.2.1 The different architectures . . . 4
    - 1.2.1.1 Flynn's taxonomy . . . 4
    - 1.2.1.2 Shared memory model . . . 4
    - 1.2.1.3 Distributed memory model and the Message Passing Interface (MPI) . . . 5
    - 1.2.1.4 Hybrid architectures . . . 6
  - 1.2.2 Critical point of parallel programming . . . 7
    - 1.2.2.1 Common bugs . . . 8
    - 1.2.2.2 Coveted properties . . . 10
  - 1.2.3 A solution: a bridging model for abstracting HPC architectures . . . 11
- 1.3 Programming BSP algorithms in ML . . . 13
  - 1.3.1 The BSP model of computation . . . 13
    - 1.3.1.1 The BSP architecture . . . 13
    - 1.3.1.2 The execution model . . . 14
    - 1.3.1.3 The cost model . . . 14
  - 1.3.2 The BSML language . . . 15
    - 1.3.2.1 The essence of BSML . . . 15
    - 1.3.2.2 Model of execution . . . 16
    - 1.3.2.3 Parallel primitives and their informal semantics . . . 17
  - 1.3.3 Some simple examples . . . 18
  - 1.3.4 Low-level primitives and the Coq proof assistant . . . 20

1.4 From BSP to multi-BSP . . . 21
1.4.1 Pro and Cons of BSP and BSML . . . 21
1.4.1.1 Benefits . . . 21
1.4.1.2 Disadvantages . . . 22
1.4.1.3 The problem of congestion using a cluster of many-cores . . . 23
1.4.2 The Multi-BSP model of computation . . . 24
1.4.2.1 The Multi-BSP architecture . . . 24
1.4.2.2 The execution model . . . 26
1.4.2.3 The cost model . . . 27
1.5 Outline . . . 30

2 State of the Art **33**
2.1 Parallel programming models . . . 33
2.1.1 The PRAM family . . . 33
2.1.2 The LogP family . . . 34
2.1.3 The sub-synchronous family . . . 35
2.1.4 The hierarchical family . . . 37
2.1.5 The BSP like models . . . 38
2.2 High-level parallel programming . . . 40
2.2.1 Algorithmic skeletons . . . 40
2.2.2 BSP programming . . . 49
2.2.3 Other functional parallel languages . . . 62

3 Multi-ML in a Nutshell **65**
3.1 The Multi-ML concepts . . . 65
3.1.1 What Multi-ML is not . . . 65
3.1.2 The execution levels . . . 65
3.1.3 A short introduction to the syntax of Multi-ML types . . . 67
3.2 The Multi-ML features . . . 68
3.2.1 The multi-functions definitions . . . 68
3.2.2 The recursive multi-functions calls . . . 69
3.2.3 The tree structure . . . 70
3.2.4 The multitree-functions . . . 71
3.2.5 Tree manipulations . . . 72
3.2.6 A global identifier . . . 73
3.3 Communications . . . 73
3.3.1 Vertical communications . . . 73
3.3.2 Communication friendly vector construction . . . 74
3.4 A complete example . . . 76

4 Type system **79**
4.1 Safety in parallel and distributed programming . . . 79
4.1.1 Replicated coherency . . . 79
4.1.2 Level compatibility . . . 80
4.1.3 Parallel structure nesting . . . 80
4.2 Operational semantics . . . 81
4.2.1 A core language . . . 81
4.2.2 Semantics rules . . . 84
4.2.3 Terminating semantics rules . . . 85
4.2.4 Terminating semantics is deterministic . . . 93

4.2.5 Diverging semantics rules . . . 93
4.2.6 Diverging semantics properties . . . 93
4.3 A typing system for μMULTI-ML . . . 97
4.3.1 Type definitions . . . 97
4.3.2 Type constraints definition . . . 101
4.3.2.1 Accessibility . . . 102
4.3.2.2 Definability . . . 103
4.3.2.3 Propagation . . . 103
4.3.2.4 Serialisation . . . 104
4.3.2.5 Weakening . . . 105
4.3.3 Typing rules . . . 106
4.3.4 Examples of rule derivation . . . 112
4.3.5 Discussions on closure communication . . . 113
4.3.6 Type soundness . . . 113
4.4 Extensions to imperative features and exceptions . . . 117
4.4.1 A typing system for μ**multi**-ML^{ref} . . . 117
4.4.1.1 Definitions . . . 118
4.4.1.2 Generalisation and references . . . 119
4.4.1.3 References and communications . . . 120
4.4.2 A typing system for μ**multi**-ML^{exn} . . . 121
4.5 A purely constraint-based type system for μMULTI-ML . . . 122
4.5.1 PCB(X) . . . 122
4.5.2 Constraint generation . . . 122
4.5.3 Constraints solving . . . 123
4.5.3.1 Type unification . . . 123
4.5.3.2 Locality unification . . . 126
4.5.3.3 A constraint solver . . . 127

5 Implementation **131**
5.1 Formalisation of the compilation . . . 131
5.1.1 Current implementation . . . 131
5.1.2 A core language . . . 132
5.1.2.1 Compilation . . . 133
5.1.3 Execution and Correctness . . . 139
5.1.4 Distributed Garbage collector . . . 142
5.1.5 Limitations of the implementation . . . 143
5.1.6 Modular Implementation . . . 143
5.2 Big step semantics with costs . . . 144
5.3 The compilation toolchain . . . 150
5.3.1 Lexing and parsing . . . 151
5.3.2 Type system implementation . . . 152
5.3.3 Code transformation . . . 152
5.3.4 Code execution . . . 153
5.3.4.1 Parallel execution . . . 153
5.3.4.2 The interactive loop . . . 154
5.4 Experimentation and benchmarks . . . 155
5.4.1 The execution platforms . . . 155
5.4.2 Eratosthenes's sieve . . . 156

6 Conclusion and perspectives **163**
6.1 Contributions . . . 165
6.1.1 The Multi-ML language . . . 165
6.1.2 Semantics and typing . . . 166
6.1.3 Implementation . . . 166
6.1.4 Benchmarks and results . . . 167
6.2 Perspectives . . . 167
6.2.1 Certified parallel programming . . . 167
6.2.2 GPU integration . . . 168
6.2.3 Examples . . . 169
6.2.4 Costs analysis . . . 169
6.2.5 Multi-ML implementation . . . 170
6.2.6 Multi-ML extensions . . . 170

Bibliography **173**

Table of notations **187**

List of URLs **188**

Appendix **189**
A.1 Semantics . . . 189
A.1.1 Terminating semantics is deterministic . . . 189
A.1.2 Diverging semantics properties . . . 190
A.2 Typing . . . 191
A.2.1 Type soundness . . . 191

List of Figures

1.1 The Sequoia super computer. . . . 7
1.2 The scheme idea of a bridging model. . . . 12
1.3 A BSP computer. . . . 13
1.4 A BSP superstep 14
1.5 **g** and **L** BSP parameters of Mirev2 in flops. . . . 24
1.6 Times of a total exchange on Mirev2. . . . 25
1.7 The MULTI-BSP components. . . . 25
1.8 The MULTI-BSP computer model. . . . 26
1.9 Sub-synchronisation capabilities of MULTI-BSP 27
1.10 The MULTI-BSP parameters 28
1.11 The difference between the MULTI-BSP and BSP models for a multi-core architecture. 28
1.12 A cluster of multi-core modeled with MULTI-BSP 29

2.1 The sub-synchronisation scheme. . . . 35
2.2 Data parallel skeletons. . . . 41
2.3 The mapreduce skeleton. . . . 42
2.4 The divide and conquer skeleton. . . . 42
2.5 The stencil skeleton. . . . 42
2.6 The *Google*'s mapreduce skeleton. . . . 44
2.7 A MapReduce example. . . . 45
2.8 A FASTFLOW example. . . . 46
2.9 A MUESLI example. . . . 47
2.10 A SKEPU example. . . . 48
2.11 A QUAFF example. . . . 49
2.12 DRMA BSP operations: "put" and "get". . . . 53
2.13 A MulticoreBSP example. . . . 57
2.14 A BSP++ code example. . . . 58
2.15 A BSP PYTHON example. . . . 59
2.16 A PREGEL code example. . . . 60
2.17 A NESTSTEP example. . . . 62
2.18 A NESL code example. . . . 63

3.1 The MULTI-ML levels. . . . 67
3.2 The small architecture used as example. . . . 68
3.3 The multi-function recursion. . . . 70
3.4 The multi-function evaluation. . . . 70
3.5 Tree construction. . . . 72
3.6 The **at** operator. . . . 72
3.7 The **gid** tree. . . . 74

3.8 **mkpar** execution steps. . . . 75
3.9 End of the distribution. . . . 77
3.10 Resulting tree. . . . 77

4.1 The grammar of the μMULTI-ML expressions. . . . 82
4.2 The communication predicate. . . . 86
4.3 Generic evaluation rules. . . . 87
4.4 BSP evaluation rules. . . . 88
4.5 Local evaluation rules. . . . 89
4.6 MULTI-BSP evaluation rules. . . . 89
4.7 The lookup predicate. . . . 90
4.8 The select predicate. . . . 90
4.9 The free variable predicate. . . . 91
4.10 Generic coinductive evaluation rules. . . . 94
4.11 BSP coinductive evaluation rules - part 1. . . . 95
4.12 BSP coinductive evaluation rules - part 2. . . . 96
4.13 Local coinductive evaluation rules. . . . 96
4.14 MULTI-BSP coinductive evaluation rules. . . . 97
4.15 Annotated types definition. . . . 98
4.16 Locality definition. . . . 98
4.17 Constraints definition. . . . 102
4.18 The *accessibility* relation. . . . 102
4.19 The *definability* relation. . . . 103
4.20 The *propagation* predicate. . . . 104
4.21 Serialisation rules. . . . 105
4.22 *Weakening* rules. . . . 106
4.23 Types of primitives and basic operators. . . . 107
4.24 Inductive rules - Generic level. . . . 109
4.25 Inductive rules - BSP level. . . . 110
4.26 Inductive rules - MULTI-BSP level. . . . 110
4.27 The MmlPrimitives(e) predicate. . . . 111
4.28 The MULTI TREE FUN inference rule. . . . 112
4.29 BSP function derivation. . . . 114
4.30 Parallel vector derivation. . . . 114
4.31 Derivation of a non-communicable vector projection. . . . 114
4.32 Operators of references. . . . 118
4.33 Non-expansive expressions grammar. . . . 120
4.34 Typing rules for references. . . . 120
4.35 Serialisation with references. . . . 121
4.36 Constraints and type scheme syntax. . . . 122
4.37 Constraints generation rules. . . . 124
4.38 Type unification algorithm. . . . 125
4.39 Constraints on localities. . . . 126
4.40 Locality unification. . . . 127
4.41 The constraint solver. . . . 128
4.42 Stack examination. . . . 129

5.1 Syntax of core-MULTI-ML. . . . 132
5.2 Free variables of core-MULTI-ML. . . . 133
5.3 Well formed terms of core-MULTI-ML. . . . 134

5.4 Flattening the MULTI-ML tree into cores. . . . 135
5.5 Compilation scheme of core-MULTI-ML. . . . 139
5.6 Abstract context syntax. . . . 140
5.7 Cost algebra definitions. . . . 145
5.8 Computation of the size of data. . . . 145
5.9 Generic evaluation rules with costs. . . . 146
5.10 BSP evaluation rules with costs. . . . 147
5.11 Local evaluation rules with costs. . . . 148
5.12 MULTI-BSP evaluation rules with costs. . . . 148
5.13 The MULTI-ML compilation toolchain. . . . 151
5.14 A MULTI-ML toplevel configuration file. . . . 155
5.15 Interactive tree visualisation. . . . 155
5.16 A MULTI-ML scan. . . . 157
5.17 The sieve of Eratosthenes. . . . 159

6.1 Summary of this thesis. . . . 164
6.2 Unbalanced GPU integration. . . . 169
6.3 Balanced GPU integration. . . . 169

List of Tables

1.1 Summary of the BSML primitives. 17

2.1 The MPI primitives. 51
2.2 The PUB/BSPLIB primitives. 54
2.3 Comparison between BSP++ and BSML primitives. 57

4.1 The informal description of μMULTI-ML primitives. 84

5.1 Multi-BSP parameters of Mirev3 (left) and Mirev2 (right). 156
5.2 Execution time (in seconds) of Eratosthenes using MULTI-ML and BSML on Mirev3.160
5.3 Execution time (in seconds) of Eratosthenes using MULTI-ML and BSML on Mirev2.160

1

Introduction

The main topic of this thesis is the design and the implementation of a high-level parallel language designed for programming hierarchical architectures in a functional way.

In this introduction, we provide a general background on high-level parallel programming (Section 1.1), and present different HPC architectures (and thus, forms of parallelism) in Section 1.2. We will then introduce some technologies that are the foundations of our work in Section 1.3: the **B**ulk **S**ynchronous **P**arallelism model (BSP) and the BSML language which is a BSP extension of the high-level, general purpose and mainly functional ML language. We finally show, in Section 1.4, their inherent limitations and introduce our solution, which is the core and the purpose of this thesis.

1.1 Context of work: generalities and background

Nowadays, it is accepted that parallel and distributed programming is the *norm* in many areas [82] such as meteorology, molecular biology, geology, *etc.* Personal computers are using an increasing number of *cores* and even some smartphones now include octo-core processors and beyond. On another scale, *supercomputers* reach new records of performance using a growing number of *hierarchical* architectures [6] such as GPUs, multi processors of multi-cores, clusters of multi-cores, *etc.*

In this context of "Think Parallel or Perish", *easy* and *safe* programming of all parallel architectures should be of paramount importance. Moreover, the additional *costs* of erroneous computation induced by deadlocks or unexpected messages on large-scale simulations can be very important. Today, popular parallel and distributed programming methodologies are sadly dominated by low-level techniques. High-level approaches offer many possible advantages by abstracting implantation details. Such structured approaches have a key role in scalable, portable and ubiquitous parallelism.

Unfortunately, it is a hard task to conceive such structured approaches and there is no universal solution. In fact, the design of a parallel language is a trade-off between two key features. First, the possibility, for the programmer, to finely control the parallel aspects that are necessary for efficiency (but which makes programs harder to write, to prove correct and to make portable). Then, the abstraction of such features which are necessary to make parallel programming easier (but which hampers efficiency and performance prediction).

1.1.1 Background

In [70], Gorlach demonstrates that programming distributed and parallel architectures[1] is a hard task and that only using "send" and "receive" primitives (to transmit data between processes) is not a good way of programming. He compares this practice to the "Goto statement" [42] programming and "spaghetti code" developing. Most parallel programming tasks are still performed using low-level tools and languages, leading to poor quality code, high debugging and upkeep costs [70]. Architectures are also becoming more and more complicated by aggregating thousands of processors and cores with different levels of memory and networks. Thus, a perfect and universal programming language to *rule them all* seems inconceivable. We recall why parallel programming is such a hard task:

- **Additional complexity** It is not easy to move from a sequential language to a parallel one: the way of *thinking* parallel algorithms is completely different. Transform a sequential program into a parallel one incrementally is not always possible. Moreover, tools for automatic parallelisation mainly work on trivial loops and basic data structures. There is no *magic* method to decompose a task and distribute data. If the wrong decomposition is made, one might not see any performance improvement on HPC applications. The only thing that really helps is the experience and/or the cost analysis, which is also an arduous task with complex algorithms.

- **Parallel programming is error-prone** even for "Gurus". Some subtle errors (*e.g. deadlocks*, *data-races*, *etc.*) can be introduced and may arise after a large number execution of a given program. Furthermore, compilers are not tested as well as their sequential counterparts, mainly because of the lack of standards and portability of programs. Due to an endless quest for efficiency, the implementation of MPI [[1]] does not always meet the specifications — even if these specifications are not always very accurate. Indeed, in the MPI standard, there are many "unspecified cases" (at least dependent on the manufacturer); This lack of "understanding" makes the "rigorous" requirement difficult to satisfy. In fact, many libraries and programming interfaces (*e.g.* MPI, OPENMP, *etc.*) allows to test your parallel programs on your personal computer by emulating parallelism using the operating system's processes. However, not all interleaving can be tested that way and some combinations that generate deadlocks may be missed. Regarding HPC, testing and debugging such errors (detailed in Section 1.2.2.1) are usually a challenge.

- **Powerful abstractions are scarce** in the languages used for parallel programming, as described in [103]. They are often too low-level and each communication or synchronisation operations has to be finely managed by the programmer. Programming becomes way easier, when the programmer can rely on powerful libraries that encapsulate complex behaviours. Works based on *high-level patterns* date back to several decades but there is still not much knowledge of innovative parallel programming languages and tools for verifying such codes.

Parallel programming is thus a strenuous task. When every detail of parallelism is left to the programmer, the program complexity becomes excessive and large sections of code deal with purely technical issues.

[1] We will explain in details, in the next section, what parallel and distributed architectures are.

1.1.2 Generalities concerning solutions proposed in this thesis

A solution proposed in this document is the use of a *functional programming* language with structured parallelism. Functional languages [81] have the advantages to be high-order, safe and code composing friendly. This is also the case with structured parallelism which composes patterns of communications and computations.

We are also exploring thoroughly the intermediate position of the paradigm of structured parallelism and algorithmic skeletons [15, 31, 39, 139] in order to obtain universal parallel languages where execution cost can easily be determined from the source code (in this context, cost means the estimated parallel execution time). This last requirement forces the use of explicit processes corresponding to the processors of the parallel machine.

Using structured parallelism also eases to describe, more formally, the behaviours of the programs. For example, by designing operational semantics and abstract machines.

Another important point is *safety* [88]. Parallel libraries and languages are generally well documented for the common cases. Nevertheless, even experienced programmers can misunderstand those APIs, especially when they are informally described and concern complex operations leading to multiple sources of potential errors. By designing a type system and by limiting (not too much) the the way of programming, one can prove that programs can be safely executed and do not crash the whole machine (or stop it in a inconsistent state). Types can also help to understand codes. Because parallel code is the norm in many areas, having parallel programs that can be executed without any error is necessary. Indeed safety seems essential when considering the growing number of parallel architectures (GPUs, multi-cores, *etc.* [6]), the complexity of such architectures and the *cost* of conducting large-scale simulations (the losses due to faulty programs, unreliable results, unexpected crashing simulations, *etc.*). This is especially true when parallel programs are executed on architectures which are expensive and consume many resources, and also on critical systems.

In this thesis, we keep in mind all those considerations to design a parallel extension of a ML (functional) language. This extension is based on a structured model of parallelism that also allows to estimate the execution time of programs. Given the strong heterogeneity of massively parallel architectures and their complexity, a frontal attack of the problem is a daunting task which is unlikely to materialise. Therefore, it seems more appropriate to design languages that limits the way of using such architectures but also preserve the performances. In fact, designing a programming language requires to choose a trade-off between the possibility to control parallel aspects — necessary for predictable efficiency but which makes programs more difficult to write, to prove and to port — and the abstraction of such features — which makes programming easier but which may hamper efficiency and performance prediction. The language designed in this thesis aims at have both efficient and safe program execution for hierarchical architectures. The chosen solution uses a bridging model that abstract efficiently many HPC architectures. Using a functional language (ML) allows us to have the required programming safety.

In the rest of this introduction, we will explain what are HPC architectures, what is a bridging model (by presenting one of them: BSP) and how programming algorithms with such a model (using a extension of ML call BSML). We will also show some limitations and how to overcome them: which is the main goal of this thesis.

1.2 Parallel and distributed architectures

In this section, we first recall some concepts of parallel computing in order to define a common vocabulary that will be used along this thesis. Indeed, in the parallel and HPC community, several terms are not used in a proper way and can lead to misunderstandings. We will also recall the

traditional problems of parallel and distributed programming. These difficulties will allow us to reveal the *philosophy* of the solutions that are proposed in this thesis.

1.2.1 The different architectures

A parallel computer, or a multi-processor system, is a computer composed by more than one processor (or unit of computation). It is common to classify parallel computers by distinguishing them by the way they access to the system memory. Indeed, the memory access scheme influences heavily the programming method of a given system.

1.2.1.1 Flynn's taxonomy

Flynn[52] defines a taxonomy of computer architectures that are based upon the number of concurrent instructions (or controls) and data streams available in the architecture [45]. Flynn's taxonomy is the following:

	Single Instruction	Multiple Instructions
Single data	**SISD**	**MISD**
Multiple data	**SIMD**	**MIMD**

where:

- **SISD** is "**S**ingle **I**nstruction, **S**ingle **D**ata stream" that is a sequential machine;
- **SIMD** is "**S**ingle **I**nstruction, **M**ultiple **D**ata streams" that is mostly array processors;
- **MISD** is "**M**ultiple **I**nstruction, **S**ingle **D**ata stream" that is pipeline of data;
- **MIMD** is "**M**ultiple **I**nstruction, **Mu**ltiple **D**ata streams" that is clusters of CPUs.

Two major classes of parallel computers can be distinguished: distributed memory and shared memory systems. The shared memory model allows multiple processes to read and write data from the same location. Otherwise, in distributed memories, the message passing model, where each process sends messages to the other processes, is used. Those different approaches are respectively adapted to various domains of application. For example, distributed memory systems are needed for computations using a large amount of data which does not fit in the memory of a single machine.

1.2.1.2 Shared memory model

In the shared memory model, a program often starts as a single process (known as a master thread) which is executed on a single processor. We consider a program where some code *regions* are defined as parallel regions. When the master thread reaches a parallel region, a fork operation is executed. It creates a group of threads which execute the parallel region on multiple processors. At the end of this parallel computation phase, a join operation terminates the fork, leaving the master thread as a lonely unit that continues to execute the program.

Most of the time, when programmers talk about parallel architecture or *concurrent* programming languages, they refer to the shared memory models. Some multi-core systems or specific supercomputers (such as Cray ones) are part of this category. In the following, we introduce some well known examples of libraries for the shared memory programming model.

OPENMP [27, 152] is a directive-based programming interface for the shared memory programming model. It consists of a set of directives and runtime routines for FORTRAN, and a corresponding set of pragmas for C and C++ (since 1998). OPENMP directives take the form of special code annotations that are interpreted by the compiler. The advantage of such directives is that the code is still sequential and can be executed on sequential machines, by ignoring the directives (pragmas). The main benefit of this approach is to avoid to maintain both a sequential and a parallel version.

INTEL **T**hreading **B**uilding **B**locks (Intel TBB [[2]]) [102] is a C++ library for task parallelism. It allows to write programs that take full advantage of multi-core performance, that are portable, composable and scalable. It also proposes features such as parallel algorithms, parallel data structures, task scheduling and scalable memory allocation. Different applications are developed using the library and are evaluated in terms of their usability and expressibility, as shown in [144].

Graphic **P**rocessing **U**nits (GPU) are used in HPC for a *short* while. It is mainly due to their tremendous computing power, favourable cost effectiveness, and energy efficiency. The **C**ompute **U**nified **D**evice **A**rchitecture (CUDA) [[3]] [118] and **Open C**omputing **L**anguage (OPENCL) have enabled GPUs to be explicitly programmed as general-purpose shared-memory multi-core processors with a high-level of parallelism. The **G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits (GPGPU) was born. In a short period, GPUs have moved from graphical-centred computing units to programmable and parallel computing devices. In CUDA, a GPU can be exploited by the programmer as a set of general-purpose shared-memory SIMD multi-core processors.

1.2.1.3 Distributed memory model and the Message Passing Interface (MPI)

Distributed memory **N**o **R**emote **M**emory **A**ccess (NORMA) computers (or shared nothing) do not have any special hardware to support a direct access to another machine's local memory. The nodes are only connected through a computer network (which can be specific to a supercomputer and therefore be very efficient). Processors can obtain data from a remote memory by exchanging messages, through the network, between the requesting and the supplying node. Such computers are sometimes also called **N**etwork **O**f **W**orkstations (NOW). When shared memory systems are used, we talk about **R**emote **M**emory **A**ccess (RMA) computers. They grant access to remote memory via specific operations that may be implemented by hardware. However the hardware does not provide a global addressing space.

The major advantage of distributed memory systems is their ability to scale to a very large number of nodes. A shared memory architecture provides a global addressing space defined by the hardware, *i.e.* all memory locations can be accessed via load and store operations. Such a system may appear to be much easier to program. However, it is hard to handle a high number of processes, as the concurrent read and write makes the program difficult to design. Furthermore, there may be unexpected performance loss due to **N**on **U**niform **M**emory **A**ccess (NUMA) shared memory models.

The MIMD streams model, and especially the so-called **S**ingle **P**rogram **M**ultiple **D**ata (SPMD) model, is widely used for programming parallel computers. In the SPMD model, the same program is executed on each processor but the computations are done on different pieces of data that are distributed over the processors.

The message passing model is based on a set of processes with private data structures. Processes communicate by exchanging messages with special send and receive operations. It is widely used for programming distributed memory machines but it can be also used on shared memory computers. The most popular message passing technology is the **M**essage **P**assing **I**nterface (MPI) [142]. It is a C and FORTRAN message passing library. MPI is an industry

standard and is implemented on a wide range of parallel computers. Details of the underlying network protocols and infrastructure are hidden from the programmer. This helps to achieve portability while enabling programmers to focus on writing parallel code rather than networking code. It includes routines for point-to-point communication, collective communication, one-sided communication, parallel IO and dynamic task creation.

Message passing and distributed memory models also subsume the well-known master-slave paradigm. In a sense, mobile agents (and nomadic computations) can also be referenced as this model. Concurrent programming such as the famous π-calculus [117] (or its JAVA version [122]) can also be referenced as message passing models. Some more general models, such as in [21], were defined in order to unify those models. Nevertheless, they appear to be unsuitable to HPC and do not provide a structured enough cost model.

1.2.1.4 Hybrid architectures

In addition to shared-memory and distributed models, modern parallel architectures now provide hybrid models. Many architectures are composed by clusters of clusters of … of multi-processors of multi-cores with GPUs. Such architectures are referenced as *hybrid* due to the diversity of its components. Furthermore, as the computations taking place on these machines are organised as a tree, they are also mentioned as *hierarchical* architectures. For example, we give the top five[4] of the 500 most powerful supercomputers in the world:

1. (China) Sunway TaihuLight(NRCPC): 10.649.600 cores @1.45Ghz with 1.31PiB of memory
2. (China) Tianhe-2(NUDT): 3.120.000 cores @2.2Ghz with 1TiB of memory
3. (USA) Titan(Cray): 299.008 cores @2.2Ghz with 584TiB of memory and 18.688 GPUs with 109TB of memory
4. (USA) Sequoia(IBM): 1.572.864 cores @2.3Ghz with 1.5PiB of memory
5. (USA) Cori(Cray): 64.128 cores @2.3Ghz with 203TiB of memory and 632.672 Xeon Phi @1.4Ghz with 1PiB of memory

To illustrate this, we choose to describe the Sequoia supercomputer of the Lawrence Livermore National Laboratory in Livermore, California, USA. This is a *Bluegene* version Q model of supercomputer which is offered by IBM. The Figure 1.1[2] shows, in detail, the hierarchy of all the components that are used to build such a machine. A 16 core chip is associated with a memory in a *compute card* and put together in a *node card* containing 32 *compute cards*; 16 *node cards* are assembled inside a *midplane*; a *rack* contains 2 *midplanes*. Finally, the whole system is composed by 96 *racks*. Thus, we have $16 \times 32 \times 16 \times 2 \times 96 = 1.572.864$ computing units that are hierarchically organised.

Clusters of multi-processors of multi-cores have become the de-facto standard in parallel processing due to their high performance to price ratio. **S**ymmetric **M**ulti-**P**rocessing (SMP) clusters are also gaining popularity, mainly because of the fast interconnection networks and memory buses. SMP clusters can be seen as a hierarchical two-level parallel architecture, since they combine the features of both shared and distributed memory machines. Aggregating SMP clusters (clusters of multi-cores) can combine the efficiency of shared-memory multiprocessors with the scaling advantages of distributed-memory architectures. The result of such a combination

[2]LLNL. Lawrence Livermore National Laboratory: BlueGene/Q, 2012. https://computing.llnl.gov/tutorials/bgq/.

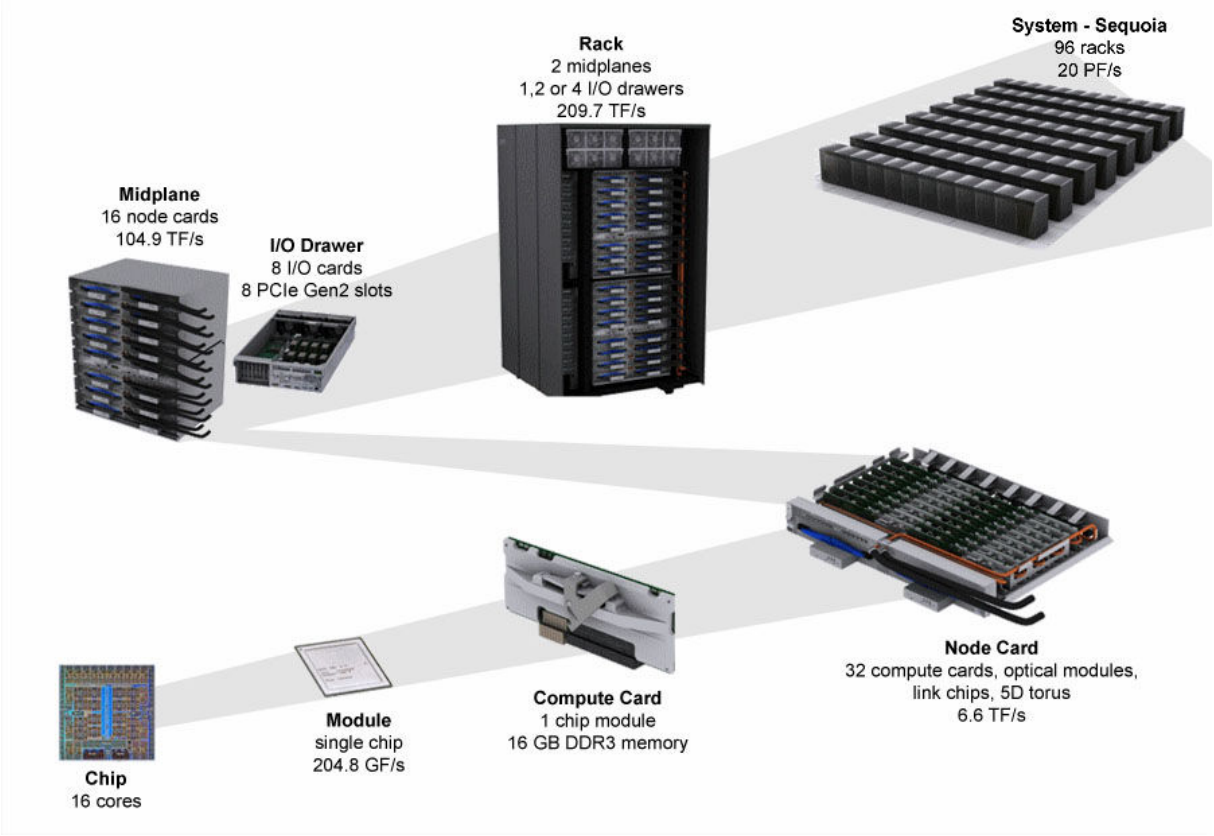


Figure 1.1: The Sequoia super computer.

leads to very large computer architectures with a good cost/efficiency ratio. Nevertheless, programming a machine combining both shared and distributed memories induces a more complex programming model. As a consequence, hybrid architectures are interesting in terms of performance but they require a new way of programming.

Intuitively, a paradigm using memory access for intra-node communication and message passing for inter-node communication seems to fully exploit SMP clusters [44]. Hybrid models have already been used for scientific computations [83], including more symbolic ones such as probabilistic model-checking [75].

These kinds of architectures are those that we are targeting in this thesis. However, as GPUs require particular programming instructions [16, 120], they are excluded from the *core* of the thesis.

1.2.2 Critical point of parallel programming

We recall that writing parallel/distributed/hybrid programs (for HPC architectures) is known to be notoriously difficult. Most of the time, programmers need to reason in terms of message-passing algorithms or shared data structures, which could be hard to deal with.

One can say that models based on shared memory (generally, concurrent languages) are easier to program. As such models provide a single address space that is shared, it seems easier to access to any pieces of data. However, it is mandatory to control simultaneous access to resources. To do so, it requires to write programs carefully and manage expensive system locks. Most of the time, implementing shared-memory abstractions requires to devote numerous resources of a machine to communication and coherence maintenance task. As shown in [103] such an abstraction is not easy to handle and introduces costs.

Concerning the distributed architectures, the control of message-passing algorithms could also be a hard task. For example, managing communication of different processes executing different codes doing different tasks could be hazardous.

Hence, both shared memory and message passing models introduces complexity and bugs. In the following, we recall the most common of them.

1.2.2.1 Common bugs

Here, we give a non exhaustive list of the most common bugs that parallel programmers are facing.

Data race condition

When evaluating a sequential program, the execution order of the instructions is quite obvious (assuming no compilation or execution optimisation). On the contrary, the evaluation of a parallel program where two processors share data is not that simple. For example, one can imagine that one processor might progress faster than the other one. We illustrate with the following example:

Process 0	Process 1
`x=0`	`x=0`
`x=x+1`	`x=x+1`
`y=read(x,1)`	`y=read(x,0)`
`print(y)`	`print(y)`

Here, `read(x,i)` returns (or *read*) "immediately" the value of `x` at processor `i`. In this example, every processor executes the same sequence of instructions. This is consistent using the SPMD model of parallelism. Indeed, if the processors execute always the same instruction simultaneously, they would both store in `x` the result of the computation, get the result from the other processor, and, then, display `y`. However, in parallel programming, it is (almost) impossible to ensure that all processors are executing the same statement at the same time.

In this example, if the processor 0 finishes its computations faster than the processor 1, he might execute the instruction `y=read(x,1)` while the processor 1 is still computing. The value of `x` is still `0`. Then, the output will be `0` for processor 0 and `1` for processor 1.

In this example the problem was fairly obvious. However, in a complex code, this kind of *bug* can be very difficult to spot. The behaviour of such programs is often non-deterministic, which make the bug even harder to track and correct. Concurrent analysers can be used to try to identify *obvious* bugs.

A similar behaviour can occur when using the famous MPI library. As the timing of a message arrival can impact the execution logic of a program, it is very common to introduce non-deterministic behaviour using message-passing libraries. A typical form of non-determinism occurs in a code where multiple concurrent incoming messages are handled in an unspecified arrival order (using `MPI_Waitany` for example). The following code illustrates this non-deterministic behaviour:

Process 0	Process 1	Process 2
`send(2,0)`	`send(2,42)`	`x=recv()`
...	...	`y=recv()`
...	...	`print(x/y)`

Here, both process 0 and 1 are going to send a value (0 and 42) to process 2. The values are received and used in order to perform a division. As the values are received without any explicit reception order (which is possible in MPI) the process 2 has two different behaviours: performing a valid computation or raising a *division by 0* exception. This kind of non-determinism is difficult to spot and requires meticulous code writing to avoid such behaviour.

Deadlock

A deadlock is a state where at least two processes are waiting for each others. This problem mainly appears when adding synchronisation primitives in a language to avoid data races. Using a shared data structure, the classical deadlock example is the following:

Process 0	Process 1
`Lock(x)`	`Lock(y)`
`...`	`...`
`Lock(y)`	`Lock(x)`
`...`	`...`
`UnLock(y)`	`UnLock(x)`
`UnLock(x)`	`UnLock(y)`

Where `Lock(x)` forces all other processes to wait until the variable `x` (a mutex) is unlocked using `UnLock(x)`. In this example, process 0 locks `x` whereas process 1 locks `y`. Later on, process 0 is blocked when accessing to `y` and, reciprocally, process 1 is waiting for process 0 to free `x`.

Concerning message passing, this situation may also appear in a similar way. For example:

Process 0	Process 1
`SendBlock(1,mess0)`	`SendBlock(0,mess1)`
`x=recv(1)`	`x=recv(0)`
`...`	`...`

Where `SendBlock(i,mess)` is a synchronous send of a message `mess` to process `i`, where the process waits until it receives the notification of complete reception. In this case, both process 0 and 1 are waiting for each other to receive the two messages.

Another common situation is when using a communication pattern involving all the processes: a collective operation. For example:

```
if pid<>0 then
  broadcast(0,buffer)
else
  asynchronous_computations
```

Where `broadcast(i,buffer)` makes the process `i` broadcast its values to all other processes, into their respective buffers. In this case, the processors with a non-zero identifier are waiting for a value broadcasted by processor 0. As processor 0 is executing some `asynchronous_computations`, we are facing a deadlock concerning the used partition of the parallel machine.

Many other examples of erroneous parallel computation exist, such as: bad size/ number of messages, sending data to a nonexistent processor and wrong memory accesses relying on pointers. Using a programming language that strongly limits these incorrect uses of parallelism would increase the confidence concerning parallel codes.

1.2.2.2 Coveted properties

Here, we list the most important properties that makes parallel programmer's life easier.

Determinism. The main thing that a programmer is expecting when executing a program is to always get the same output for a given input — excepted for randomised algorithms. Determinism is a property assuring that: what ever the execution order of the processes is, the final value will be the same. This property is close to confluence of sequential programs. This feature is uncommon in the world of parallel languages [11]; nevertheless it is a worthwhile property [12].

For many computational problems, there is no inherent non-determinism in the problem statement. As said in [19] about HPC applications: *it is typically the case that the outcome of the computation will not depend on the arrival order: The execution of the process will consist of multiple sections, where there is non-determinism within each section, but the state at the end of each section is deterministic. Furthermore, non-determinism "does not propagate": the order and content of sent messages is not affected by receive order.* In [19], this property was called: "send-determinism". Otherwise, the notion of "internally deterministic", studied in [11], concerns evaluations that generates a unique *trace* for each input (which depends on the level of abstraction of trace's equivalence).

Determinism is convenient for safety: no deadlock nor data-race can occur; a deterministic program is free of the problem of execution interleavings (and complex memory consistency models). Furthermore, the composition of separately developed deterministic codes is easier as their behaviour is similar, regardless of the execution context. Determinism also allows the programmer to reason on an algorithms (the behaviour is completely defined by the sequential equivalent) and their optimisations without dealing with the previously defined safety problems. The programmer can reason "as" for sequential computation and add, incrementally, parallelisation strategies instead of sequential computations. Generally speaking, this method is simple but does not always offer the best performance.

Therefore, it seems to be a growing consensus on the fact that determinism is worthwhile [12].

Reasoning about execution time Another pleasant property is the ability of prediction concerning the execution time of HPC programs. For sequential algorithms, the common solution uses the RAM/VonNeumann model and describes the complexity of a problem according to the input size n and relate it with a $\mathcal{O}$ function (worst case) or θ function (average case), *etc.* Based on these complexities analysis, some tools are able to accurately predict the execution time of programs by estimating their **W**orst **C**ase **E**xecution **T**ime. For example, MERASA was used to analyse parallel computation on multi-cores [126, 148].

Concerning parallel and distributed algorithms, such a property is obviously valuable. Curiously, in HPC, the cost measurement is not based on the parallel complexity of an algorithm but is rather based on the execution time (measured using empirical benchmarks). Programmers are benchmarking load balancing, communication (size of data), *etc.* Using such a technique, it is very difficult to explain why a code is faster than another one and which one is more suitable for one or another architecture.

Even if the PRAM cost analysis (which is detailed in Chapter 2) proposes a method to estimate the *work* of a parallel algorithm, such a technique is not widespread. One explanation

of such a lack can be found in [9]: it seems that too many programmers still prefer using asynchronous send/received primitives (with potential data-races, deadlocks, *etc.*) instead of a structured model. Thus, measuring the overlap of computations and communications is hard to determine. Let's have a look to the following situation:

Process 0	Process 1	Process 2

`for i:=0 to n do`
`  x=received(AnyProcess)`
`  compute(x)`
`done`
	`for i:=0 to n/2 do`
	`  send(0,...)`
	`  compute(...)`
	`done`
		`for i:=0 to n/2 do`
		`  send(0,...)`
		`  compute(...)`
		`done`

Here, both processes 1 and 2 are going to send values to process 0. Each time process 0 receives a value, it uses it to make some computations. We assume synchronous sending (a process is blocked until the value has been received), constant size of data and constant computations.

One can ask how is it possible to determine the cost of this code. We assume that we just measure the efficiency of the communications and the latency of the network. As there are $\mathcal{O}(n)$ local steps, process 0 will receive n pieces of data and other processes will send $n/2$ pieces of data. If we assume that the overlap is possible, it is possible to both send and receive values while computing. If process 0 computes while receiving both the values of process 1 and 2, one value is directly available when the computation of 0 is done, for example 1. Then, both process 0 and 1 can compute; the value of process 2 is thus available for the next computation of 0 while 1 is sending a new one. The problem here is that we cannot distinguish the network efficiency of two distinct cases: receiving one value and receiving two values. Furthermore, the latency is hard to determine. Indeed, we should distinguish the latency of a value that has just been received and the latency of a value already received.

Therefore, it seems hard to reason about costs of message-passing algorithms. Reasoning, on a combination of high-level patterns to deduce the cost of a parallel algorithm seems wiser. It also allows to give some measures that takes into account worst cases (at least average) execution of programs on a given architecture.

1.2.3 A solution: a bridging model for abstracting HPC architectures

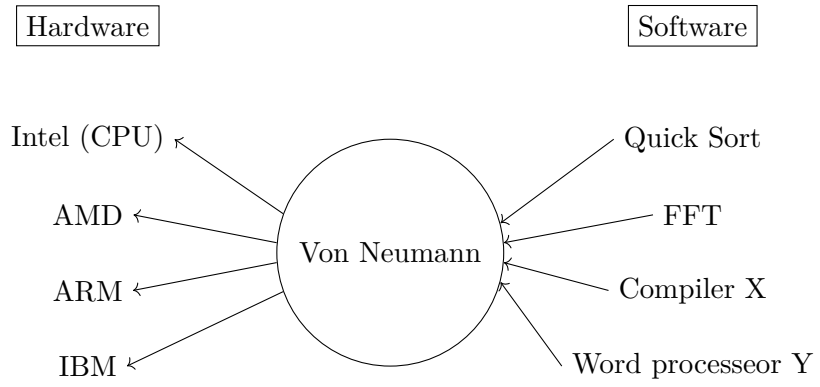
As we have just seen, one inherent difficulty in the development of scientific software is to have codes that are correct, easy to develop and efficient. Today, the number of parallel languages may exceed the number of different architectures available. Most languages are inadequate to such purposes because they are not made for portability, correctness and performance. Furthermore, the performances of such programs are typically difficult to predict because of the interaction of a large number of individual data transfers. On another side, placing too much emphasis on abstraction may result in inefficient code. Nevertheless, striving for optimal efficiency can run the risk of compromising software correctness and can result in the employment of architecture-specific programming models.

A solution is the use of a *bridging model*. This term was introduced in Leslie Valiant's 1990 paper: "A Bridging Model for Parallel Computation" [150]. This paper argued that the strength of the von Neumann model was largely responsible for the success of computing.

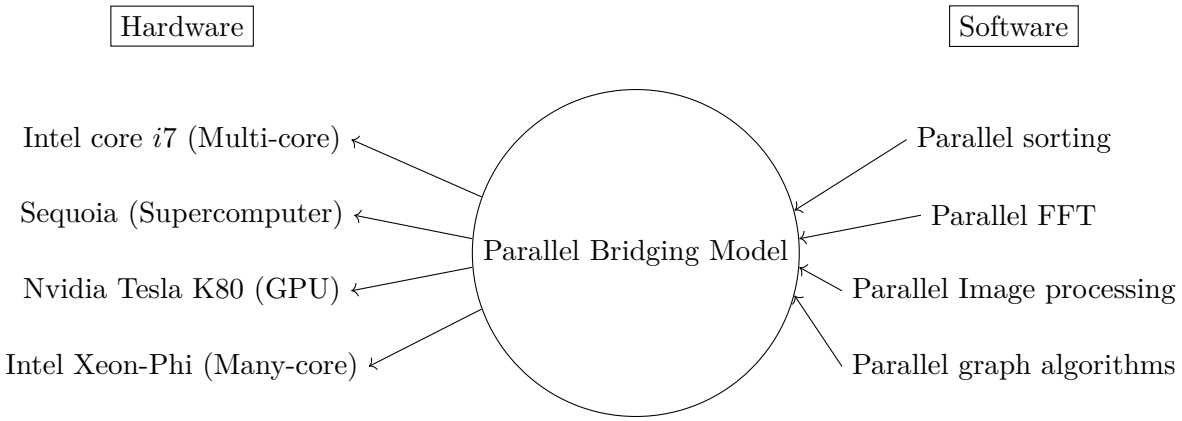
Figure 1.2 schemes the idea of a *bridging model* where the signification of the directed arrow is (from the tail to the head) *"can efficiently be simulated on"*. We can distinguish a sequential version in Figure 1.2a, based on the VON NEUMANN model; and a parallel one, Figure 1.2b. In computer science, a bridging model is an abstract model of a computer which provides a conceptual bridge between the physical implementation of the machine and the abstraction

available to a programmer for the corresponding machine; in other words, it is intended to provide a common level of understanding between hardware and software engineers. A bridging model provides to software developers a way to escape architecture-dependent development.

A flourishing bridging model is able to be efficiently implemented on real architectures and efficiently targeted by programmers. In particular, it should also be possible for a compiler to produce efficient code from a high-level programming language.



(a) A sequential bridging model.



(b) A parallel bridging model.

Figure 1.2: The scheme idea of a bridging model.

To evaluate a bridging model and a programming language based on this model, it is necessary to consider them in terms of all its properties: (1) its benefits as a basis for the design and analysis of algorithms in order to obtain a relevant complexity; (2) its ability to be applied onto the whole range of general-purposes architectures while providing efficiency and scalability; (3) its support of a fully-portable design; and (4) software engineering tools able to check correctness or debug programs based on a given model. For example, an efficient GPU-like program could be interesting in terms of performance and scalability, nevertheless it is not fully portable as it cannot be directly used on a cluster.

The main advantage of a bridging model is that if an algorithm is efficient, then it is also efficient on "all" physical architectures. It will always be possible to have models adapted to particular architectures that are more detailed than a bridging model. Nevertheless, such a detailed model raises much more difficulties concerning cost analysis and portability. The

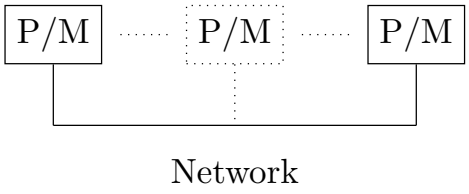


Figure 1.3: A BSP computer.

usage of a bridging model also enables to write an algorithm, once; then, for all inputs and all architectures, an efficient and seasoned version is available.

1.3 Programming BSP algorithms in ML

As a partial solution to the aforementioned problems, some researchers have designed bridging models such as the BSP (**B**ulk **S**ynchronous **P**arallelism) [9, 150] model of computation. Many people are working on the way of programming BSP algorithms; and some of them program them in a functional way (using ML). In this section we are going to introduce the BSP model and its functional implementation: the BSML language. We also present the "philosophy" of BSP and BSML which are part of the foundation of this thesis.

The next section is dedicated to the limitations of BSP and BSML on modern architectures. It justifies the recursive and hierarchical extension of the BSP model called the MULTI-BSP model, and hence, the work of this thesis which is to propose a way to program MULTI-BSP algorithms in ML.

1.3.1 The BSP model of computation

The BSP model (**B**ulk **S**ynchronous **P**arallelism) [9, 150] is a bridging model designed for parallel architectures. We recall that the aim of a bridging model [109] is to ease the way of programming various parallel architectures using a certain level of abstraction. It allows to reduce the gap between an abstract execution (programming an algorithm) and concrete parallel systems (using for example a compiler).

The initial assumption of the BSP model is to have a portable and scalable performance prediction for parallel programs. Without dealing with low-level details of parallel architectures, the programmer can focus on the design of an algorithm, its complexity, correctness, *etc.* A nice introduction concerning the BSP "philosophy" can be found in [140] and a complete book of numerical algorithms is also available [9]. A comparison of the BSP model with other different models can be found in [100].

The BSP bridging model describes (1) a parallel architecture, (2) an execution model (3) and a cost model in order to allows the performance prediction of a BSP algorithm on a given architecture. In the following, we present the three key points of the BSP model.

1.3.1.1 The BSP architecture

A BSP computer is structured by three main components:

1. A set of homogenous pairs of processors-memories;
2. A communication network to exchange messages between pairs;
3. A global synchronisation unit to execute global synchronisation barriers.

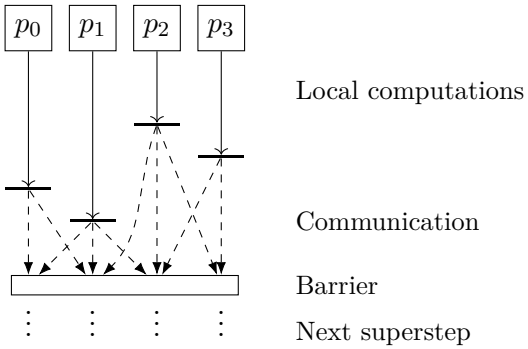


Figure 1.4: A BSP superstep

A wide range of current architectures can be seen as a BSP computers. For example, shared memory machines could be used in a way such that each processor only accesses a sub-part of the shared memory (which is then "private") and communications could be performed using a dedicated part of the shared memory. Moreover the synchronisation unit is very rarely a hardware feature but is rather software [85]. Supercomputers, clusters of computing units [9], multi-cores [156] and GPUs [90], *etc.* can be thus considered as BSP computers. Figure 1.3 shows a representation of a BSP computer which is composed by a set of pairs of processor/memory connected by a network.

1.3.1.2 The execution model

The execution of a BSP program is a sequence of *supersteps* (Figure 1.4) where each one is divided into three successive disjoint phases:

1. Each processor only uses its local data to perform sequential computations and to request data transfers to other nodes (thick arrows);
2. The network delivers the requested data (dotted arrows);
3. A global synchronisation barrier occurs, making the transferred data available for the next superstep (empty rectangle).

This *structured* model enforces a strict *separation* of communication and computation: during a superstep, no communications between the processors are allowed, only requests of transfer; information exchanges only occur at the synchronisation *barrier*. Note that a BSP library[3] can send data during the computation phase of a superstep, but this is hidden from the programmers.

1.3.1.3 The cost model

The *performance* of a BSP computer is characterised by four *parameters*:

1. The local processing speed **r**;
2. The number of processors **p**;
3. The time **L** required for a barrier;
4. The time **g** for collectively delivering a 1-relation.

Where **r**, **g** and **L** can be expressed using FLoating-point Operations Per Second (FLOPS).

[3] BSP libraries are generally implemented using MPI [142] or low-level routines of the given specific architectures.

A 1-relation is a collective exchange where every processor receives/sends at most one word. The network can deliver an h-relation in time $\mathbf{g} \times h$. To accurately *estimate* the execution time of a BSP program, these 4 parameters can be easily benchmarked [9]. The execution time (cost) of a superstep s is the sum of the maximal local processing time, the data delivery and the global synchronisation times. The total cost (execution time) of a BSP program is the sum of its supersteps's costs. It is expressed by the following formula:

$$\text{Cost}(s) = \max_{0 \leq i < \mathbf{p}} w_i^s + \max_{0 \leq i < \mathbf{p}} h_i^s \times \mathbf{g} + \mathbf{L}$$

Where w_i^s is the local processing time on processor i during superstep s and h_i^s is the maximal number of words transmitted or received by processor i during superstep s.

The BSP model considers homogeneous processors only. Indeed, with heterogeneous processors, due to global synchronisation, the speed $\mathbf{r}$ is lowered to the slowest processor. In practice, many HPC architectures have heterogeneous networks and memories but a homogeneous set of processors. Moreover, the design of algorithm made for heterogeneous architectures is known to be much more difficult. For example, managing many parameters to load-balance computations and data automatically introduces complexity in the designing of algorithms.

We now present how to program BSP algorithms in a functional way using the the BSML language. It is an attempt at providing a good balance between the two aforementioned opposite approaches of parallel programming: low-level (and subject to concurrency issues), and high-level (with loss of flexibility and efficiency).

1.3.2 The BSML language

1.3.2.1 The essence of BSML

BSML combines the high degree of *abstraction* of ML (with performances relatively close to C) with the *scalable* and *predictable* performance of the BSP model. On another side, we can find libraries such as MPI [142], generally used with FORTRAN or C, which are unsafe approaches leaving the programmer responsible for deadlock or non-determinism issues. Otherwise, algorithmic skeletons [32] proposes a safe programming model, but it is limited to a restricted set of algorithms. Discussions concerning those approaches are available in Chapter 2.

BSML follows the BSP paradigm to structure the computations and communications between the processors in a data-parallel fashion. All communications in BSML are collective (they involve all processes) and deadlocks are avoided thanks to global synchronisation barriers.

An important feature of BSML is its *confluent* semantics: whatever the order of execution of the processes, the final value will be the same. Confluence is convenient for *debugging* since it allows to get an *interactive loop* (also know as *toplevel*), with the exact same execution as in parallel. It also eases the *incremental* transformation of a ML program to a parallel version written in BSML. Last but not least, it is possible to reason about BSML programs using the COQ ([[5]]) proof assistant [61, 68] and to extract actual BSP programs from proofs. This feature is uncommon in the world of parallel languages [11], even if it is a worthwhile property.

BSML [67] uses a *small set of primitives* and is currently implemented as a library [[6]] for the ML programming language OCAML [[7]]). Using a safe high-level language as OCAML to program BSP algorithms allows performance, scalability and *expressiveness*. The OCAML language was chosen among different variants of ML because of its *close to* C performance. As the evaluation strategy of OCAML is eager, it eases the cost analysis of a program. Moreover, the OCAML community is large and a generous set of libraries and tools are available.

The BSML primitives work over a parallel data structure called a *parallel vector*. To handle this parallel data structure, the language proposes two functions to manipulate it. As well as four constants to access the BSP parameters of the underlying architecture. By comparison, the standard BSPLIB [[8]] [86] library for BSP programming in C offers about twenty primitives, and MPI more than a hundred. The BSML approach is uncommon since the majority of parallel (or distributed) programming languages (or libraries) offers a lot of primitives. Most of the time, the programmer must choose between *basic* and *optimised* (and complex) primitives to program an algorithm. By reducing the primitives, BSML aims to simplify and structure the way of programming parallel algorithms. As BSML is conformant to the BSP cost model, it enables a seamless cost analysis of algorithms.

1.3.2.2 Model of execution

A BSML program is written as a ML one where it is possible to express parallel computation using parallel vectors. The core syntax of BSML is alike the OCAML one, with few restrictions. BSML programs can be read as OCAML ones. In particular, the execution order should not appear unexpected to a programmer used to OCAML, even if the program is parallel. Moreover, most OCAML programs can be considered as BSML programs that do not use parallelism. Such a program would be executed sequentially on each processor of the parallel machine and would return their results, as usual. This behaviour allows the parallelisation to be done incrementally from a sequential program.

The ML type of parallel vectors is `'a par`. A vector expresses that each of the $\mathbf{p}$ processors *embeds* a value of any type `'a`. The processors are *labelled* with *ids* from 0 to $\mathbf{p}-1$ where $\mathbf{p}$ is the number of processors. It is important to note that the number of processors is constant throughout the whole execution of the program. $\mathbf{p}$ can be accessed in BSML using the integer constant `bsp_p`. The other BSP parameters are accessible as float values through the constants `bsp_g`, `bsp_l` and `bsp_r`.

We use the following notation to describe a (parallel) vector: $\langle v_0, v_1, \ldots, v_{\mathbf{p}-1}\rangle$. We distinguish a vector from a usual array because each local value, that will be called *local* values, are not aware of others. It is possible to access the local value v_i of a vector locally: on processor i (using a specific syntax); or after some communications. The nesting of parallel vectors is not allowed in BSML– see Section 4.1.3 of Chapter 4 for discussions and a type system that prevents this forbidden use of vectors. Thus, the vector $\langle x_0, x_1, \ldots, x_{\mathbf{p}-1}\rangle$ holds the value x_i on processor i.

Since a BSML program deals with the whole parallel machine and individual processors at the same time, a *distinction* between the 3 existing *levels of execution* is needed:

1. **Replicated** or r is the default level of execution. Code that does not involve BSML primitives is run by the parallel machine as it would be by a single processor. Replicated code is executed at the same time by every processors, and leads to the same result everywhere.

2. **Local** execution (l) represent an evaluation within the scope of a parallel vector. Each processor uses its local data to do their own computations.

3. **Global** execution (g) concerns the set of all the processors together, as for BSML communication primitives.

The distinction between local and replicated is strict: the replicated code cannot depend on local information. If such a thing happened, it would lead to a *replicated inconsistency*.

Primitive	Level	Type	Informal semantics
`<< e >>`	g	`'a par` (if `e:'a`)	$\langle$ `e` $,\ldots,$ `e` $\rangle$
`pid`	g	`int par`	A predefined vector: i on processor i
`$v$`	l	`'a` (if `v: 'a par`)	`v` $_i$ on processor i, assumes `v` $\equiv \langle$ `v` $_0,\ldots,$ `v` $_{\mathbf{p}-1}\rangle$
`proj`	g	`'a par-> (int-> 'a)`	$\langle x_0,\ldots,x_{\mathbf{p}-1}\rangle \mapsto$ (`fun` $i \to x_i$)
`put`	g	`(int-> 'a)par-> (int-> 'a)par`	$\langle f_0,\ldots,f_{\mathbf{p}-1}\rangle \mapsto \langle$ `fun` $i \to f_i$ $0,\ldots,$ `fun` $i \to f_i$ $(\mathbf{p}-1)\rangle$

Table 1.1: Summary of the BSML primitives.

1.3.2.3 Parallel primitives and their informal semantics

Parallel vectors are handled through the use of different *communication primitives*. Table 1.1 summarises the BSML primitives. Informally, the BSML primitives works as follow: let `<< e >>` be the vector holding `e` everywhere (on each processor), the `<< >>` piece of notation indicates that we enter the scope of a parallel vector and, thus, we switch to the local level. Here, replicated values are directly available inside the vector. To access to the local value of a vector named x inside the scope of a parallel vector, we add the syntax `$x$`. The *ids* can be accessed with the predefined vector `pid`. Note that, in the BSML implementation, *ids* are accessible through `$this$`. For the sake of coherence with the rest of the thesis, we choose to replace the keyword `this` by `pid`. Nevertheless, they do have the same meaning.

To illustrate the previous primitives with a small piece of code, we use the BSML toplevel that *simulates* a BSP machine with 3 processors:

```
# let vec1 = << "Hello" >> in
    << $vec1$^", proc "^(string_of_int $pid$) >>;;
- : string par = <"Hello, proc 0", "Hello, proc 1", "Hello, proc 2">
```

The `#` symbol is the prompt that invites the user to enter an expression to be evaluated. Then, the toplevel gives the evaluated value with its type. Thanks to BSML confluence, it is ensured that both the results of the toplevel and the distributed implementation are identical.

In this code, we create a vector `vec1` containing a string and we built a new vector by concatenating `vec1` with the vector identifier (using `$pid$`).

The `proj` primitive is the only way to *extract* local values from a vector. Given a vector, `proj` returns a function such that, applied to a valid identifier of a processor, it returns the corresponding value of the vector. `proj` performs communications to make local values available globally (a single-broadcast), and ends the current superstep. One can say that `proj` is a parallel vector *destructor*. For example, if we want to convert a vector into a replicated list, we write:

```
# let list_of_par vec = List.map (proj vec) procs;;
- : val list_of_par : 'a par -> 'a list = <fun>
# list_of_par << $pid$ >>;;
- : int list = [0; 1; 2]
```

In this code example we *map* the function produce by `(proj vec)` on `procs`, a list of *ids* $[0;1;\ldots;\mathbf{p}-1]$. The function `list_of_par` extracts, for each process, all the values of `vec`. Thus, the evaluation of `list_of_par` on a parallel vector containing the pids produces the list of pids (on each processes). The BSP cost of such a function is $(\mathbf{p}-1)\times s\times\mathbf{g}+\mathbf{L}$ where s is the size

of the biggest sent value. Indeed, each processor will receive all the values of other processors (to be replicated) but only one processor will send the biggest value $(\mathbf{p}-1)$ times.

It is interesting to note that the result of `proj` is of type `(int -> 'a)`. It could have returned an array or a list of size $\mathbf{p}$ instead, but the interface is more functional. Furthermore, as seen in the examples, the conversion between one style to the other is easy.

The `put` primitive is another communication primitive. It allows any local value to be *transferred* to every other processor. It is also synchronous, and ends the current superstep. The parameter of `put` is a vector of functions (of type `(int -> 'a)`) that describes, for each processor, the values that one processor wants to send to each other. Thus, the function returns the data to be sent to processor j when applied to j. The result of `put` is a new vector of functions: at a processor j the function, when applied to i, yields the value *received from* processor i by processor j. The canonical use of `put` is: `(put << fun sendto -> e($pid$, sendto, $x$) >>)` where expression `e` computes (or usually, selects) the data that should be sent depending on `sender` to `sendto`. For example, a `total_exchange` could be written:

```
# let total_exchange vec =
    let msg = put <<fun dst -> $vec$>> in
      << List.map $msg$ procs >>;;
- : val total_exchange : 'a par -> 'a list par = <fun>
# total_exchange << $pid$ >>;;
- : int list par = <[0;1;2], [0;1;2], [0;1;2]>
```

1.3.3 Some simple examples

In the following, we give three examples that are useful in BSP programming and some are already available in additional BSML libraries. As they are typical of the way of programming BSML algorithm, it is interesting to detail them.

Firstly, it is often needed to scatter a data set among the processors. The following function is a way to select a sub-part of a list on every processor of a BSP computer:

```
#(* select_list:'a list -> 'a list par *)
let select_list l =
  let len = List.length l in
    <<cut_list l ($pid$ * len / bsp_p) (($pid$ + 1) * len / bsp_p))>>
```

where `cut_list l a b` returns the sub-list of the elements of `l` from index `a` (inclusive) to index `b` (exclusive). Now, in order to *map* a function, in parallel, on a distributed list (which is a classical data-parallel skeleton [3, 32]), we can write this simple code:

```
(* parmap : ('a->'b)->'a list par -> 'b list par *)
#let parmap f parlist = << (List.map f) $parlist$ >>
```

Secondly, reducing and scanning are also important operators which are both implemented as MPI collective operations. A simple one-superstep reduce (having $\oplus_{k=0}^{\mathbf{p}-1} v_k$ on each processor from the parallel vector $\langle v_0, v_1, \ldots, v_{\mathbf{p}-1}\rangle$) can be done with:

```
(* simple_reduce: ('a->'a->'a)->'a->'a par -> 'a
let simple_reduce op e v = List.fold_left op e (proj_list v)
```

here e is a neutral element and `op` the associative operator $\oplus$. This reduce function does not use the capabilities of the parallel machines. If the combination operator $\oplus$ has an important cost, we may use a multi-step reduce (in a logarithmic way), by applying the operator locally. The following algorithm combines the values of processors i and $i + 2^n$ at processor i for every superstep n from 0 to $\lceil \log_2(\mathbf{p}) \rceil$:

```
(* scan': int->('a->'a->'a)->'a->'apar->'apar *)
let rec scan' step op e v =
  if step >= bsp_p then v else
    let comm =
      put << fun j ->
        if (j mod (2*step) = 0) && ($pid$ = j + step)
          then $v$
        else e >> in
     let v' =
       << if $pid$ mod (2*step) = 0
         then
           if $pid$ + step < bsp_p
             then op $v$ ($comm$ ($pid$ + step))
           else $v$
         else e >>
     in
     scan' (step*2) op e v'

(* reduce: ('a->'a->'a)->'a->'a par->'a *)
let reduce op e v = (proj (scan' 1 op e v)) (bsp_p-1)
```

The third example concerns the *parallel sorting by regular sampling* algorithm (PSRS) [106, 135] in its BSP version [146]. Given a distributed list, the PSRS algorithm proceeds in four steps:

(1) The p sub-lists of each processor are sorted locally by a sequential algorithm. The problem now consists in merging the $\mathbf{p}$ sorted lists.

(2) To do so, we must choose globally $\mathbf{p} - 1$ pivots in order to split the data into $\mathbf{p}$ sub-sets.

(3) We now need to communicate those sub-sets: processor i will receive, from every other process, the sub-list corresponding to the elements from $(i-1)^{th}$ to i^{th} pivot.

(4) Finally, each processors must merge the received sub-lists.

After these steps, the result consist in a distributed list ordered through the $\mathbf{p}$ processes. The efficiency of the algorithm relies on a suitable selection of the pivots, done in step (2). This selection, called *regular sampling*, can be done in three steps:

(a) Each processor extracts $\mathbf{p} - 1$ pivots from its ordered local list, in a regular way.

(b) Those pivots lists are merged globally, in order to form an ordered list of size $\mathbf{p}(\mathbf{p} - 1)$.

(c) $\mathbf{p} - 1$ pivots are chosen regularly spaced in the list.

A BSML version of the corresponding algorithm can be coded as follows:

```
(* psrs: int par -> 'a list -> 'a list *)
let psrs lvlengths lv =
     (* step 1: local sorting *)
   let locsort = << List.sort compare $lv$ >> in
     (* step 2(a): selection of the (p-1) primary samples  *)
   let regsampl = << extract_n bsp_p $lvlengths$ $locsort$ >> in
     (* step 2(b): total exchange of the primary samples  *)
   let glosampl = List.sort compare (gather_list regsampl) in
    (* step 2(c): selection of the secondary samples *)
   let pivots = extract_n bsp_p (bsp_p*(bsp_p-1)) glosampl in
     (* step 3 : building the communicated lists of values *)
   let comm = << slice_p $locsort$ pivots >> in
    (* step 3: send the values *)
   let recv = put << List.nth $comm$ >> in
    (* step 4: local fusion of the received data*)
    << p_merge bsp_p (List.map $recv$ procs_list) >>
```

where `vlengths` is the vector of list sizes, `extract_n n` extracts $n-1$ evenly distributed elements; `slice_p` transforms a sorted list to a list of list depending on the samplings and `p_merge` merges sorted lists. The cost of steps 1 to 2(b) is thus:

$$\frac{n}{\mathbf{p}} \times \log(\frac{n}{\mathbf{p}}) \times c_c + \frac{n}{\mathbf{p}} + (\mathbf{p} \times (\mathbf{p}+1) \times s_e) \times \mathbf{g} + \mathbf{L}$$

where c_c is the time (supposed constant) to compare two elements and s_e is the size (also supposed constant) of an element. The cost of step 3 is [146]:

$$\frac{n}{\mathbf{p}^2} \times \log(\frac{n}{\mathbf{p}^2}) \times c_c + \frac{n}{\mathbf{p}^2} + \frac{3n}{p} \times s_e \times \mathbf{p} + \mathbf{L} + \text{time}_{\text{fusion}}$$

where $O(n/\mathbf{p})$ is the time needed to merge values.

1.3.4 Low-level primitives and the Coq proof assistant

Before [17], BSML had only the `put` and `proj` communication primitives combined with:

```
mkpar: (int -> 'a) -> 'a par
apply: ('a -> 'b) par -> 'a par -> 'b par
```

where `mkpar` aims to create a parallel vector as an array and `apply` allows the local application of a vector of functions onto a vector of values. Those primitives are the ones that are used in the current implementation [61, 68], and inside the Coq [8] proof assistant. Introduced in [17], `$v$` and `<< e >>` is a syntactic sugar of `mkpar` and `apply` that aims to ease the way of programming algorithm in BSML. It is not suitable for formal purposes. For example:

```
<<f ($pid$+1) >>
```

is transformed by BSML into:

```
let pid=mkpar (fun i ->i);;
apply (mkpar (fun _ -> f)) (apply (mkpar (fun _ pid -> pid+1)) pid)
```

The syntax of vectors (`<< e >>`) and *low-level* primitives are equivalent. However, the use of those primitives can sometimes become awkward. Indeed, every operation inside the scope of a parallel vector has to call a primitive and define an "ad hoc" function. This gets worse when working with multiple vectors and nested calls to `apply`.

In Coq, we also used these *low-level* primitives as the syntactic transformation is not available. Moreover, studying the semantics without the syntactic sugar is easier. For example, we do not have to differentiate variables annotated with a $ and the ones which are not inside the scope of a parallel vector.

1.4 From BSP to multi-BSP

1.4.1 Pro and Cons of BSP and BSML

1.4.1.1 Benefits

The BSP's execution policy has two main advantages. First, it removes non-determinism (no concurrent accesses) and guarantees the absence of deadlocks, as there is no shared memory and communications are always collective. This is a worthwhile point for proving the correctness of algorithms, as it is done in [54]. Secondly, it proposes an accurate model of performance prediction based on the bandwidth and the latency of the interconnection network, and on the speed of processors. This performance prediction model can even be used at runtime to dynamically make decisions, for example, choose whether or not to communicate a data in order the re-balance the workload.

On many distributed architectures, synchronisation barriers are often expensive when the number of processors dramatically increases (more than 1 000). Even if proprietary architectures usually provide much faster synchronisation systems, this cost may be important. However, efficient BSP algorithms tend to reduce the use of synchronisation barriers, making them *almost* negligible (in term of speed-up constraint). In spite of the performances issues, synchronisation barriers have several non negligible benefits: it is harder to introduce the livelock, since barriers do not create circular data dependencies; barriers also permit new forms of fault tolerance, as shown in [140].

The BSP model also considers communication actions *en masse*. This is less flexible than asynchronous messages, but easier to debug since there are many simultaneous communication actions in a parallel program, and their interactions are typically complex. Bulk sending also provides better performance since, from an implementation point of view, grouping communication together in a separate program phase allows global optimisations of the data exchange by the communications library. Moreover it is easy to measure, during the execution of a BSP program, the time spent to communicate and to synchronise (by adding timers before and after the primitive of synchronisation). This capability is mainly uses to compare different algorithms. Since BSP programs are portable and cost predictions can lead to power consumption estimation, they are interesting in cloud-computing [5]. Thus it is not surprising to read that both the famous Google's MapReduce [40] and the Pregel framework [110] (for large graph processing) are based on the BSP model of execution.

In respect to the level of abstraction of the chosen language, the programmer can gain in programming simplicity but lose in terms of flexibility and performance. Indeed, it is known that a skilled programmer can produce an efficient and rather optimised code using a low-level language combined with a message passing interface. Nevertheless, to write such a code is not always possible with very complicated simulation running on complex architectures. Moreover, it is often a long task (that may introduce bugs) which is rarely profitable when using a wide range of different machines [70, 103].

Another advantage of BSP is the way it eases debugging. As the computations during a super-step are independent, they can be debugged independently. This capability is used to formally prove the correctness of algorithms.

The simplicity of programming, debugging and proving, but also the efficiency of the BSP model makes it a good *framework* for teaching: a few hours are needed to teach BSP programming and algorithms.

The BSP model has also been used with success in a wide variety of problems. In the following, we propose a non-exhaustive list of the most recent works from our knowledge: scientific computing [9], string search and genetic algorithms [41, 101], graphs [26], search engine (queries to textual databases) [33], 3-SAT solver [13], algebra [128], scheduling threads [35], multi-agent services [93], image processing [90, 125], *etc.* BSP was adopted because it represents a common model for writing successful parallel programs that exhibit phase-based computational behaviours [82].

In addition to being convenient in practice, BSP allows the design of what could be called *"immortal algorithms"*. Proposed by Leslie Valiant and Bill McColl in the 1990's, an *immortal algorithm* is an algorithm that is written once and can be run optimally on any parallel machine of today or tomorrow, because its BSP algorithm cost is optimal. For example, for sequential computing, the merge sort algorithms is known to be of time complexity of $\mathcal{O}(n \log(n))$; such a complexity will be valid forever. For parallel computing, because the BSP cost model is a realistic cost model for standard parallel architectures, the cost (formulae) of BSP algorithms will also remain the same in the future. Thus, algorithm designers can formulate *immortal algorithms* that are proved optimal regardless of the values of ($\mathbf{p}$, $\mathbf{g}$, $\mathbf{L}$): algorithms that need to be written just once, and remain forever optimal. In the absence of a proof of optimality (but with an algorithm that still reduce and balance both computation and communication, while avoid synchronisation), the cost model allows derivation of the asymptotic behaviour of the algorithm to determine its scalability. Such a property allows the design of parallel algorithms for long-time support.

1.4.1.2 Disadvantages

All the benefits of the BSP model are possible because the runtime system knows precisely which computations are independent. In an asynchronous message-passing system like MPI, the independent sections tend to be smaller and identifying them is harder. Nevertheless, using BSP, programmers and designers have to keep in mind that some parallel patterns are not really BSP *friendly*. For example, BSP does not handle with ease pipeline and master/slave paradigms (also known as farm of processes). Nevertheless, it is possible to get a *not too inefficient* BSP version of such patterns, as we will see later in this document. Therefore, some parallel computations and other optimisations would never be BSP [46]. This is the drawback of BSP that also concern all the restricted models of computations as well.

In parallel programming, it is convenient to stop a computation if an exceptional case arrives (typically an error). It seems natural that one processor could warn other processors that an error was raised. Using the strict BSP model, it is impossible to do so until the end of the super-steps. Such a behaviour may let other processors to continue their computations for nothing.

Another example is the pipe-lining pattern (where each processor sends its value to the next processor and waits for the data of its preceding processor). As in BSP the communication cannot be asynchronous, such a code is clearly not adapted to the BSP model. Nevertheless, with some restrictions on the operations on the data, it is possible to get a not too inefficient BSP algorithm to solve this problem (and more) [35].

The BSP cost model reveals two increasing and unavoidable sources of costs that are ruled by the physical realities of the architectures: a cost $\mathbf{g}$ for bandwidth and a cost $\mathbf{L}$ for latency.

As architectures are growing bigger, those cost are becoming more and more expensive. Such concerns are important but quite complicated to handle with BSP. The BSP model considers homogeneous processors only. Indeed, with heterogeneous processors, due to global synchronisation, the speed **r** is lowered to the slowest processor. In practice, many HPC architectures have heterogeneous networks and memories but a homogeneous set of processors. Moreover, the design of algorithms made for heterogeneous architectures is known to be much more difficult. For example, managing many parameters to load-balance computations and data automatically introduces complexity in the designing of algorithms.

The worst case concerns is the congestion problem that arises when using BSP algorithms on a cluster of multi(many)-cores [149]. In fact, the design of a BSP algorithm makes the hypothesis that communications are nearly identical over all the network (the idea of the h-relations). But this is not accurate when comparing the communication between the cores of the same machine (shared memory) and cores of different machines (over the network). Let as see this problem with a practical example.

1.4.1.3 The problem of congestion using a cluster of many-cores

As said above, one of the main advantages of the BSP model is its cost model: it is quite simple and quite accurate on a cluster or on multi-cores. Moreover, the cost model allows to reason on algorithm efficiency. One might wonder if the BSP model is also accurate with clusters *of* multi-core. Many works [65, 149] think not; therefore we would like to try to answer the question using BSML. To do so, we use a BSML version of a program from [9] that aims to benchmark and determine the BSP parameters of a parallel machine. It measures the time to perform some random collective all-to-all operations of growing size; then a least squares methods is applied to compute the values **g** and **L**.

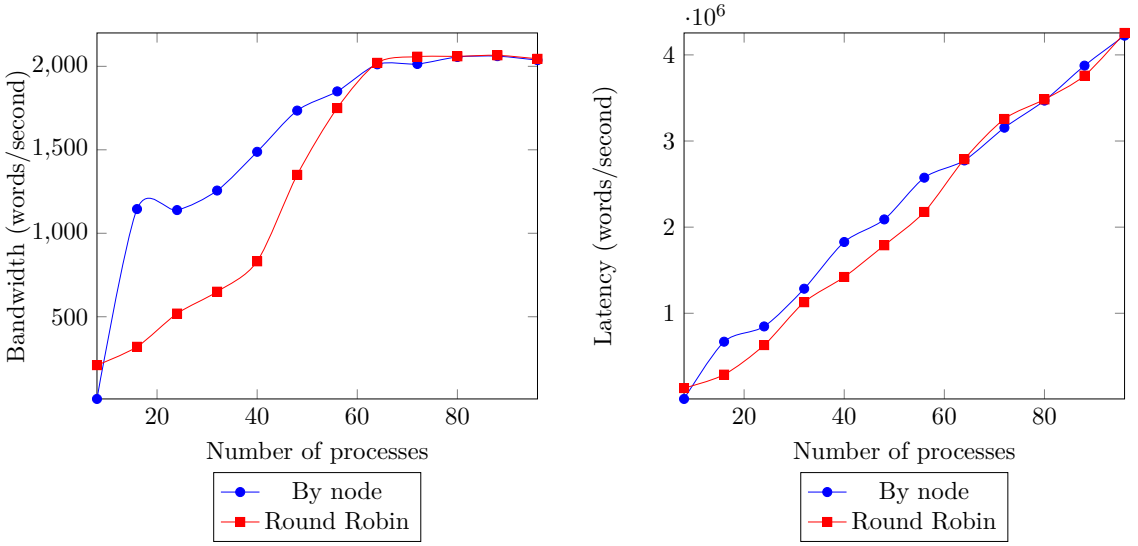
The benchmark were done on two parallel architectures named Mirev2 and Mirev3 of the LIFO: "**L**aboratoire d'**I**nformatique **F**ondamental d'**O**rléans, Orléans, France" [[9]]. Here are the main specifications of these machines:

- Mirev2: 8 nodes with 2 quad-core (AMD 2376) at $2.3Ghz$ with $16Gb$ of memory per node and a $1Gb/s$ network.
- Mirev3: 4 nodes with 2 hyper-threaded octo-core (16 threads)(Intel XEON $E5-2650$) at $2.6Ghz$ with $64Gb$ of memory per node and a $10Gb/s$ network.

On both machines, the version 4.02.1 of OCAML, GCC 4.9 and MPICH 3.1 were used. The **g** and **L** BSP parameters are given in Figure 1.5. The number of cores growth in abscissa and we distribute the computations on cores with two different strategies: (1) cycling on nodes in a round-robin fashion; (2) using as much as possible of cores on each node. One can notice that until 64 cores **g** evolves as expected: (1) the round robin distribution lacks from its intense network access in the first time; (2) on the contrary, the distribution by nodes achieves better performances as it minimises network accesses. However, when an equivalent number of processes are accessing the network the performances of both distributions are similar. With more than 64 processes, which is the number of physical cores of our architecture, the performances reach a plateau. Too many cores are accessing the network, resulting in a bottleneck which severely hindering the performance. Then, the value of **g** is not accurate making impossible to use the BSP cost model correctly.

Regarding the **L** parameter, we can observe an important growth of the parameter as the number of processing grows.

To show this problem of accuracy and hence of bottleneck, we can grow the size of the data exchanged during a total-exchange. The BSP cost prediction will exhibit a linear growing:



(a) **g** BSP parameters of Mirev2.

(b) **L** BSP parameters of Mirev2.

Figure 1.5: **g** and **L** BSP parameters of Mirev2 in flops.

$\mathbf{p} \times n \times \mathbf{g} + \mathbf{L}$. Where n is the constant size of data on each processor. On Figure 1.6 we show the computation time of a total-exchange (with a fixed data size) concerning a growing number of processes. The reader can see that the actual execution time differs from the cost prediction: the BSP cost prediction is not precise enough.

Without the cost prediction, we lost one of the main advantage of a bridging model such as BSP. Different solutions have been proposed in the past. We compare them in the State of the art, Chapter 2. We have chosen Valiant's solution [149] because of its elegance: it seems to be a simple model which is precise and close enough to typical and modern HPC machines; in short the best bridging model from our point of view.

1.4.2 The Multi-BSP model of computation

Modern HPC architectures are made of hundreds of interconnected nodes each, with thousand of cores. As we have just seen, the BSP model is not truly adapted for these architectures that breaks the bridging model. An updated of the BSP model that is able to handle such architectures is needed. This model needs to be defined with the same level of abstractions and must bridge the architecture as the original BSP model. Furthermore, it must be adapted to *clusters of multi-cores* architectures. Valiant made a proposition of such a model and he called it the MULTI-BSP model [149]. The goal of Valiant is: *with the comparison with the previous literature, our goal here is that of finding a bridging model that isolates the most fundamental issues of multi-core computing and allows them to be usefully studied in some detail.*

1.4.2.1 The Multi-BSP architecture

The MULTI-BSP model introduces a representation where a *hierarchical architecture* is a *tree* structure of *nested components* (*sub-machines*) where the lowest stage (*leaves*) are processors and every other stage (*nodes*) only contains memory. The tree is of depth **d** and is assumed balanced and homogeneous.

In the following, we use a particular vocabulary to describe the MULTI-BSP architecture. A

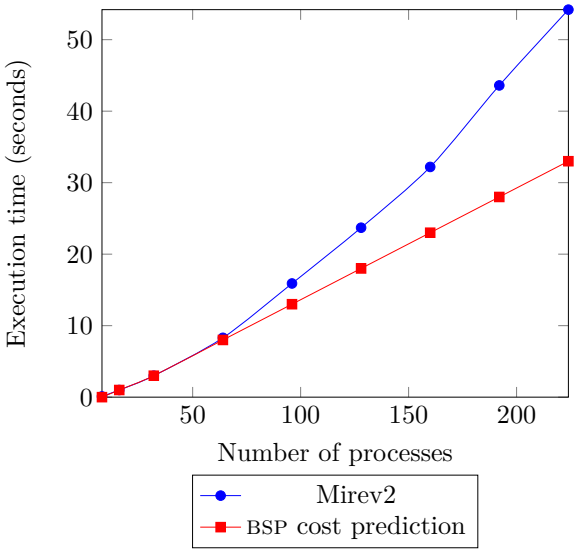


Figure 1.6: Times of a total exchange on Mirev2.

component stands for an entity of the MULTI-BSP architecture, which can be a node or a leaf. A *stage* designates a given depth of the MULTI-BSP architecture.

At the ith stage there are a specified number of components, each of them manages a number of $i{-}1^{th}$ level components, as illustrated in Figure 1.7 (left). This number of sub-components only depends on the depth i of the MULTI-BSP architecture. Finally, for the stage of depth d: the leaves, we find the hardware threads (unit of computations) as illustrated in Figure 1.7 (right). Each processor can access a small memory (usually caches). Figure 1.8 shows the MULTI-BSP model of an architecture with its parameters (which are explained below).

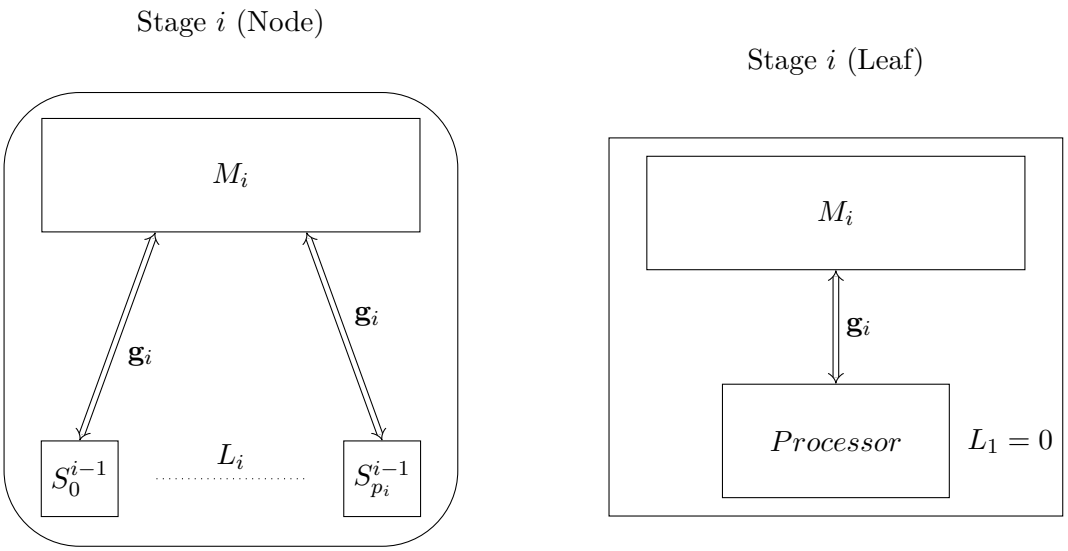


Figure 1.7: The MULTI-BSP components.

Note that a sub-component (child) can access the memory of its top component of stage i (parent). However, they cannot directly access to the memory of their "brothers" without doing an explicit copy through the memory of the component i.

The MULTI-BSP model is assumed to be homogeneous concerning the processors but not with the network capacities (bandwidth and latency) and memory sizes: they are identical for all the

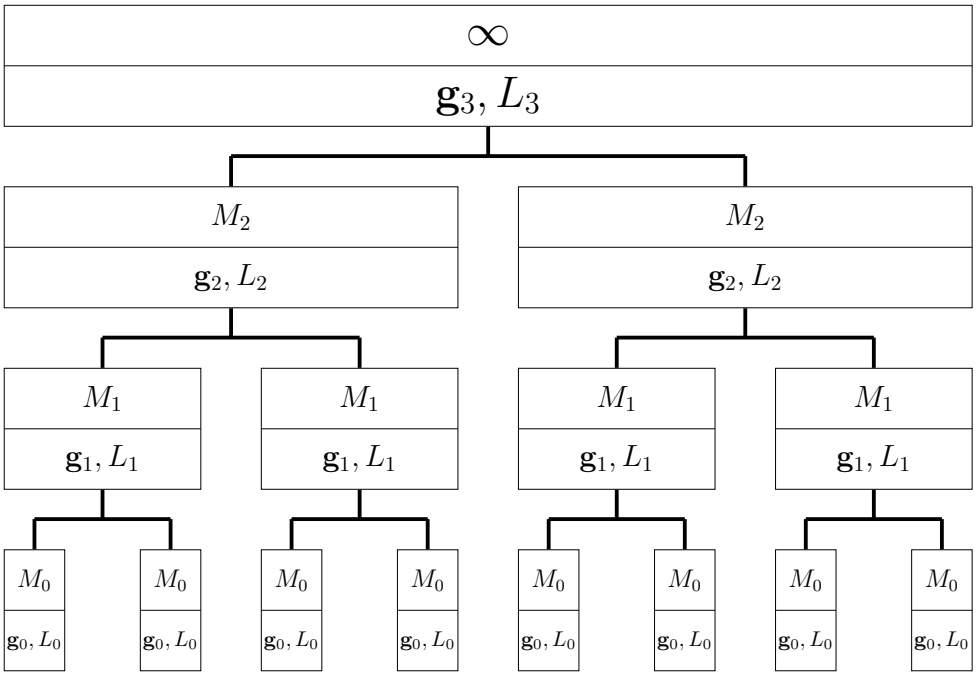


Figure 1.8: The MULTI-BSP computer model.

components of the same stage but could be different on different stages. Now, we have to specify the nature of the communication between a stage i component and the stage $i+1$ components.

We have made some minor modifications compared to the original MULTI-BSP model. Those modifications do not change the way to write or analyse the algorithms, they aim to simplify the approach and be more consistent with the rest of the thesis. In particular, we assume that the processors have access to a local memory whereas the original model assumes an additional abstract stage. Otherwise, the bandwidth parameter of the memory will concern the current stage, instead of the upper stage as described in the original version of MULTI-BSP (see [149]).

1.4.2.2 The execution model

The execution model of MULTI-BSP is similar to the BSP one. The notion of *superstep* is the key and those *supersteps* are hierarchically organised in order to manage sub-synchronisations.

At a stage i, a superstep is composed by the following steps:

- The sub-components of stage $i-1$ execute some code independently (until they reach a barrier)
- There are exchanges of informations with the m_i memory
- Synchronisation of the sub-components (stage $i-1$)

Then, a new superstep can start at stage i. Note that the definition requires synchronisation of the sub-components of a component, but no synchronisation across branches that are separated in the component hierarchy. The synchronisation of a superstep concerns all the nodes of the sub-components (stage $i-1$) and there is no sub-group synchronisations.

Figure 1.9 illustrates the sequence of local supersteps of the MULTI-BSP model. We can observe that the architecture is composed by a multi-core decomposed in two cores with respectively four threads per core. The black rectangles stand for the computations and the dotted lines symbolise the communications.

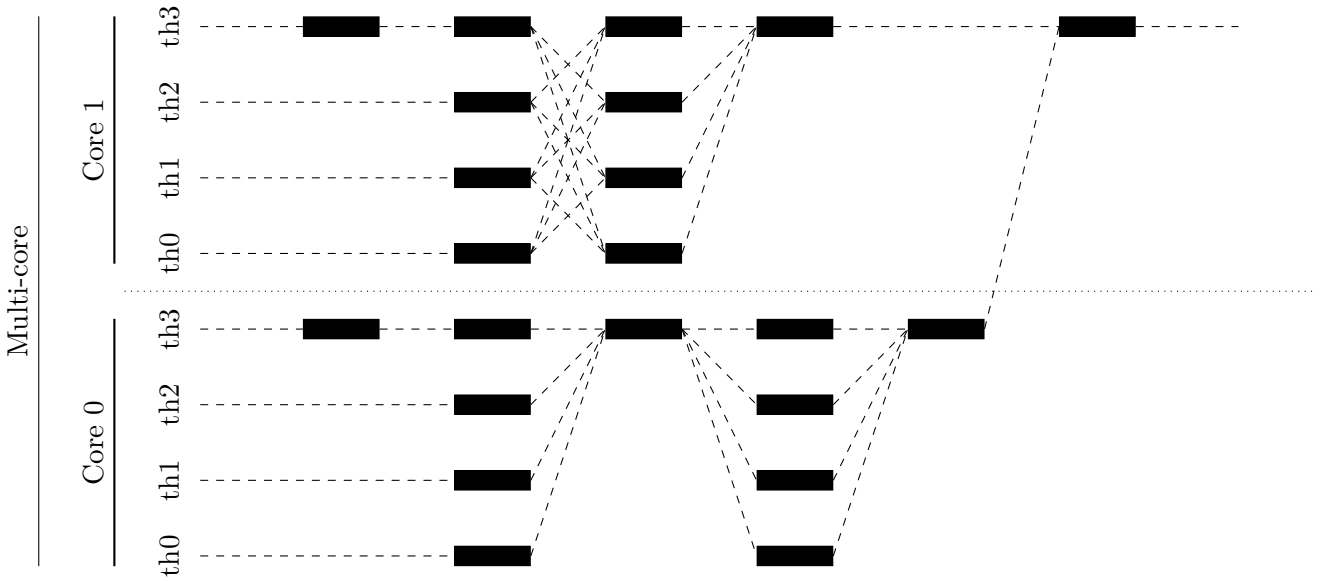


Figure 1.9: Sub-synchronisation capabilities of MULTI-BSP

Regarding the stages that are only composed by a memory capacity (that is to say nodes), all the memory manipulations is done by one of the processors of the branch (which is chosen depending of the implementation). The processor is charged to perform the data exchanges such as scattering or gathering data, *etc.* To do so, the most part of the workload must be done by the stage 1, that is to say, in the cache memory. Otherwise, the computations that are done in the top memories must focus on data management (such as scattering or gathering data).

The main observation done by Valiant in [149], is that certain recursive algorithms are well suited to computational models with multiple parameters. Programming such algorithms is an onerous task, however, the use of a bridging model enables one to make *one big effort*, once and for all, to write a program that is efficient for all inputs and all machines.

1.4.2.3 The cost model

An instance of a MULTI-BSP architecture is defined by the depth $\mathbf{d}$ of a tree and four parameters; four parameters for each *stage* i:

- $\mathbf{p_i}$ is the number of sub-components inside the $i-1$ stage.
- $\mathbf{g_i}$ is the *bandwidth* between stages i and $i-1$: the ratio of the number of operations to the number of words that can be transmitted in a second (illustrated in Figure 1.10). The difference is that, in the basic BSP model, direct communications are allowed horizontally, between units at the same stage. On the contrary, in MULTI-BSP, such communication must be simulated via a memory of a higher stage; For the sake of simplicity, horizontal communications are also allowed (as suggested by Valiant in [149]) with this same bandwidth parameter $\mathbf{g_i}$[4]
- $\mathbf{L_i}$ is the *synchronisation cost* of all sub-components of a component of $i-1$ stage.
- $\mathbf{m_i}$ is the amount of memory available at stage i for each component of this stage.

[4]In the worst case, an horizontal communication is just the succession of a write and a read operation in the upper memory of the component; the cost is thus $2 \times \mathbf{g_i}$.

Finally, there is the local processing speed $\mathbf{r}$ of each of the computing units (leaves).

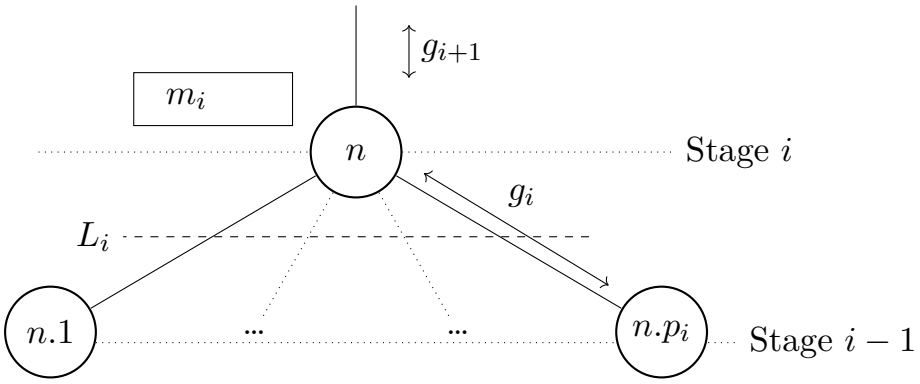


Figure 1.10: The MULTI-BSP parameters

Figure 1.10 illustrates the MULTI-BSP parameters on a MULTI-BSP architectures.

We consider $\mathbf{p}_0$ as a basic computing unit (a processor) where a step on a word is considered as the unit of time. Therefore, it does not share any memory with any other component and does not have sub-component making $\mathbf{p}_0$ equal to 1 and $\mathbf{L}_0$ equal to 0. On another side, the stage 1 memory is accessed by the level 0 processors, and thus the data rate (corresponding to the notional $\mathbf{g}_1$) has value 1. Figure 1.11 illustrates the difference between the BSP model and the MULTI-BSP one for a multi-core architecture with threads[5].

Notation:

In the following, when representing MULTI-BSP architecture, a *node* will be symbolised by a circle: ◯; a *leaf* by a square: □.

Compared to other models, the $d = 0$ case with $(\mathbf{p}_0 = 1, \mathbf{g}_0 = 1, \mathbf{L}_0 = 0, \mathbf{m}_0)$ (where $\mathbf{m}_0$ is the size of the RAM) gives the von Neumann model. The BSP model with parameters ($\mathbf{p}$, $\mathbf{g}$, $\mathbf{L}$) where the basic unit has memory $\mathbf{m}$ would be modeled with [149] $\mathbf{d} = 1$ and $(\mathbf{p}_0 = 1, \mathbf{g}_0 = 1, \mathbf{L}_0 = 0, \mathbf{m}_0 = \mathbf{m})$, $(\mathbf{p}_1 = \mathbf{p}, \mathbf{g}_1 = \mathbf{g}, \mathbf{L}_1 = \mathbf{L}, \mathbf{m}_1)$ where $\mathbf{m}_1$ is, for example, the size of a slower but bigger memory. Another value for $\mathbf{m}_1$ could be n, the size of the problem (implemented as a virtual memory with a distributed file system). Assuming a specific architecture with data streams as input, $\mathbf{m}_1$ could also be ∞ and $\mathbf{g}_1$ could rely on the acquisition stream data rate.

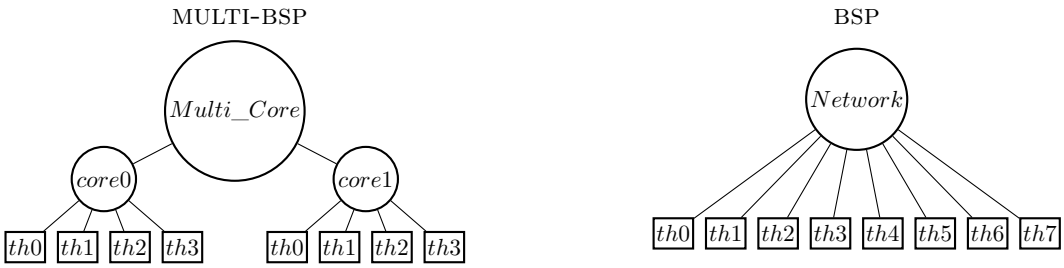


Figure 1.11: The difference between the MULTI-BSP and BSP models for a multi-core architecture.

We can relate MULTI-BSP parameters to existing multi-core architectures. For example, a cluster of p multi-processors of multi-cores with $\mathbf{d} = 3$ may have the following MULTI-BSP parameters[6]:

[5] The use of hyperthreading technology just adds an extra stage to specify physical threads.

[6] The measurements of $\mathbf{g}_i$ are hypothetical and aim to illustrate the slowdown of the bandwidth (in flops/words).

- Stage 0; one core has a basic computing unit plus L1/L2 caches: $(\mathbf{p}_0 = 1, \mathbf{g}_0 = 1, \mathbf{L}_0 = 0, \mathbf{m}_0 = 0)$
- Stage 1; one chip has 8 cores plus L3 caches: $(\mathbf{p}_1 = 8, \mathbf{g}_1 = 10, \mathbf{L}_1 = 40, \mathbf{m}_1 = 4MB)$
- Stage 2; one node has 4 chips plus RAM: $(\mathbf{p}_2 = 4, \mathbf{g}_2 = 100, \mathbf{L}_2 = 25, \mathbf{m}_2 = 128GB)$
- Stage 3: p nodes with network access (*e.g.* gigabit ethernet): $(\mathbf{p}_3 = \mathbf{p}, \mathbf{g}_3 = 1000, \mathbf{L}_3 = 200000, \mathbf{m}_3 = \infty)$

Figure 1.12 illustrates such a MULTI-BSP architecture.

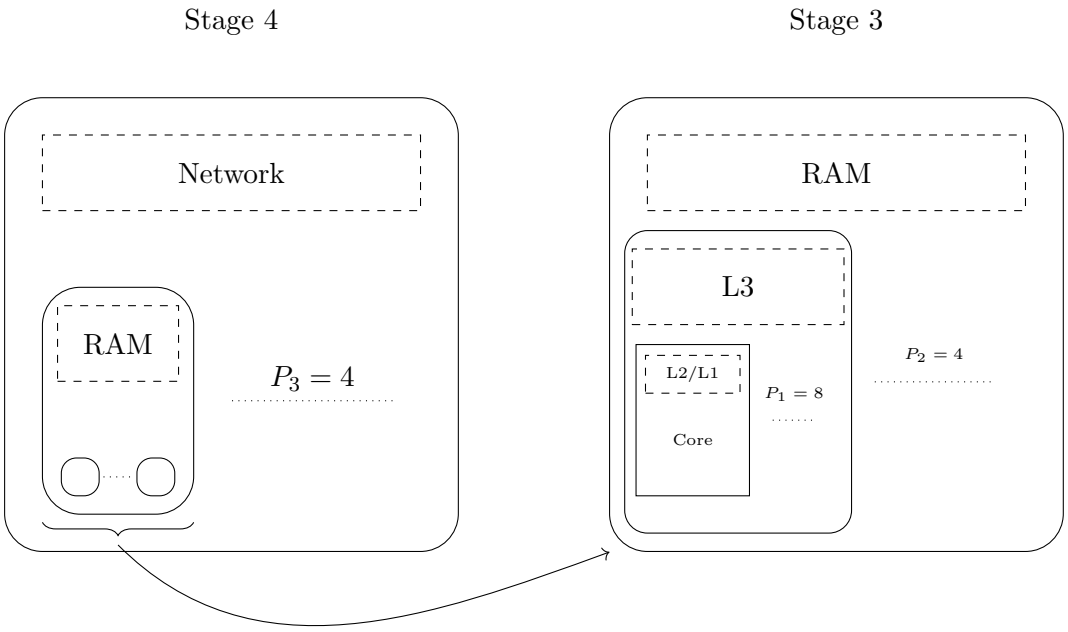


Figure 1.12: A cluster of multi-core modeled with MULTI-BSP

The MBSPDiscover tool of Alaniz *et al.* [1] is able to determine the MULTI-BSP parameters (for C codes) in order to get realistic values of the MULTI-BSP parameters. A case study of measurement of MULTI-BSP parameters can be found in [130].

We would like to add a few remarks on these parameters, proposed in [149]. First, the $\mathbf{L}_i$ parameters (except for the stage $\mathbf{d}$) are the latency parameters given by the chip manufacturer specifications, rather than the actual synchronisation costs. Second, the caches of the chip (L1 or L2 memories) are controlled internally by a specific protocol and cannot be explicitly used as usual, like other memories. Third, in a chip each physical computing unit runs threads, and groups of threads share a common arithmetic unit (or even a GPU). For all these reasons, while the relative magnitudes of the various $\mathbf{g}_i$ and $\mathbf{L}_i$ values shown may be meaningful, their absolute values may be hard to figure out, as shown in [1].

We saw that when a node executes some code on its nested components (*children*), it has to wait for the end of the sub-computation to do the communications and synchronise the children.

Consider the communication cost C_j^i of a superstep j at stage i as following:

$$C_j^i = h_j \times \mathbf{g}_i + \mathbf{L}_i$$

where h_j the maximum size of the exchanged messages at superstep j, $\mathbf{g}_i$ the communication bandwidth with stage i and $\mathbf{L}_i$ the synchronisation cost. We can express the cost T of a MULTI-BSP algorithms as following:

$$T = \sum_{i=0}^{d-1}(\sum_{j=0}^{N_i-1} w_j^i + C_j^i)$$

where d is the depth of the MULTI-BSP architecture, N_i is the number of supersteps at stage i, w_j^i is the maximum computational cost of the superstep j within stage i.

We can also *recursively* express the cost T of a MULTI-BSP algorithm as following:

$$T = \sum_{j=0}^{N_{root-1}} (T_{root-1} + C_j^{root} + w_j^{root})$$

with:

$$\begin{cases} T_0 = w_j^0 \\ \\ T_i = \sum\limits_{j=0}^{N_i-1} (T_{i-1} + C_j^i + w_j^i) \end{cases}$$

where T_i is the MULTI-BSP cost at stage i and *root* is the root node of the MULTI-BSP architecture such that $root = d - 1$.

Finally, we have three main observations about the MULTI-BSP architecture.

First, and unlike the BSP model, the MULTI-BSP model controls the storage explicitly. Nevertheless, it is not clearly specified how the various cache protocols are supported by the model.

Second, we only need that machines support the features of this model; no constraint is implied about what else they might support. Thus there is no objection to machines being able to have additional mechanisms such as external disks, GPU, *etc.* Valiant proposes a new bridging model: a minimal description on which architects, algorithm designers and programmers can come together.

Third, the MULTI-BSP model does not propose a way to handle heterogeneous and unbalanced architectures. However, it is known that programming homogeneous architecture is already a challenging task and adding another level of difficulty may lead to a non-programmable model.

1.5 Outline

Chapter 2 is dedicated to the state of the art. We compare our language with other approaches and previously published works.

In Chapter 3, we introduce the MULTI-ML language: an extension of ML for programming MULTI-BSP algorithms, as BSML is for BSP algorithms. We present the main primitives and illustrate them with simple examples. We also explain the underlying ML types needed to contain the specific information added to manage the multi-levels of execution. We also detail the arbitrary choices made in the design of our language. Thus, we extract the essence of our language in order to propose it as a "nutshell".

In Chapter 4, we first present a core language of MULTI-ML which will be used for the formal work of this thesis. As traditionally, we then present the formal operational semantics using

a natural semantics (big-steps) for both terminating and diverging programs. Now that the dynamic part have been presented, we show the static analysis of our core-language, which is the type system, and how we infer types. To do so, we first present the grammar of the types (how ML types has been extended) to explain how and why they are used; then the rules of the type system are given. Secondly, we show how the type inference works, as a traditional constraints resolution. We finally give some results, and the proof of the technical lemmas finishes this chapter. During this chapter, we illustrate the rules and the definitions with some simple examples.

In Chapter 5, we give an overview of the current implementation of the MULTI-ML language. To do so, we describe an abstract compilation scheme from a MULTI-ML parallel code to an OCAML sequential version with low level communication primitives. This scheme allows us to propose a generic and efficient implementation. We then explain how to use this model to make a modular implementation. We finally give the benchmarks of some algorithms.

Chapter 6 concludes this thesis and gives an overview of the results; some possible (near) future works and what could be done in a more expectative way.

2

State of the Art

This chapter provides an overview of the parallel programming methods that are available in the literature. We introduce the parallel programming models that aim to structure the way of programming parallel algorithms and some of their implementations. We will present the advantages and drawbacks of the ones that are the most important for our concern.

2.1 Parallel programming models

In this section we present a non-exhaustive list of parallel computing models that aim to describe various parallel architectures. We try to describe how the models evolve to match new architectures and new expectations.

2.1.1 The PRAM family

The PRAM family is the oldest way of structuring parallel algorithms. Nevertheless, it is still an accurate way of studying the intrinsic parallelism of algorithmic problems. In the following, we describe the different evolutions of the PRAM model.

PRAM

The PRAM model [56], first of its name, was introduced to conceive algorithms for parallel machines and measure the degree of parallelism of parallel algorithms. The Parallel Random Access Machine is made up by an unbounded number of **p** processors that can access to a shared memory. Using this shared memory is the only way to communicate between processors. At each evaluation step, each processor can read a value from the global memory, write a value to the global memory or compute in its own registers. The PRAM model allow the definition of a policy concerning simultaneous access to the global memory. These restrictions are: EREW (Exclusive Read Exclusive Write), CREW (Concurrent Read Exclusive Write) and CRCW (Concurrent Read Concurrent Write). The PRAM model does not provide a way to handle data locality and synchronisation to ease writing algorithms. With this model, it is possible to estimate the work performed by the computation units only. Nevertheless, there is no way to consider latency and communication bandwidth, which are the main concerns using HPC architectures.

APRAM

This is an asynchronous variant of the PRAM model called APRAM [69]. Its goal is to be close to the MIMD architectures. In the APRAM model, each processor has its own local memory and can access a shared global memory. Unlike PRAM, the processors can run asynchronous computations. Nevertheless, a memory access policy is used on top of synchronisation steps. The model is composed by four types of instructions: (1) Read the global memory to store data in local memory; (2) Local computations where data and results are stored into the local memory; (3) Write data from the local memory to the global memory; (4) Synchronisation steps. A synchronisation step consists of a logical point in a computation where each processor of a set waits for the other. Processors can read and write to shared memory asynchronously. Nevertheless, a processor cannot read a memory location that was written by another processor before a synchronisation step involving those processors. There exist variants of APRAM that introduce the notion of synchronisation latency costs [112]. This approach is still incomplete regarding current hierarchical architectures.

HPRAM

The Hierarchical PRAM [84] is proposed to allow distinct asynchronous PRAM computations to be synchronised within a single algorithm. HPRAM can build a disjoint subset of PRAM processors. Each subset receives different instructions that are executed asynchronously. Synchronisations are done independently, within the subsets. As expected, the partitioning instruction can be executed inside each subset. The HPRAM model proposes two variants. First, private HPRAM divides the memory among processors. Each subset has its own private memory block from the global memory. Secondly, shared HPRAM does not divide the shared memory. Every processor subset can access the global memory. HPRAM introduces latency and synchronisation costs. The latency cost is proportional to the size of the processor subsets. Concerning the synchronisations, it distinguishes synchronisations between the subsets and within the subsets. This model is closer to the hierarchical machines that are today standard, however such a *dynamic* hierarchy does not take into account the actual architecture. Thus it is not possible to benefit from the advantages of fast hierarchical memories.

2.1.2 The LogP family

Close to BSP, these models are, most of the time, used to study network capabilities and low-level libraries such as MPI.

LogP

The LOGP model [34] is an asynchronous model that describes point-to-point communications for distributed computers with distributed memories. It aims to reflect the critical technology trends underlying parallel computers and networks. It was intended to serve as a basis for developing fast, portable parallel algorithms and to offer guidelines to machine designers. The model provides a way to handle communication costs using various parameters: (l, o, g, P). The communication's latency upper bound is described by L. The incompressible send and receive cost is denoted by o. The parameter g represent the minimum gap between two consecutive messages, it can be interpreted as the inverse of the bandwidth per processors. Finally, P is the number of processors.

The LOGP model does not provide a way to handle costs induced by hierarchical architecture.

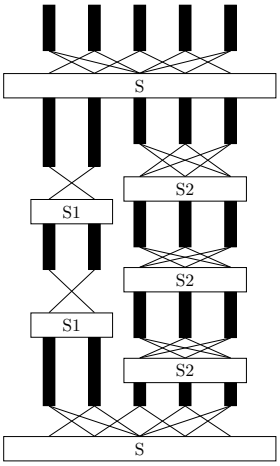


Figure 2.1: The sub-synchronisation scheme.

LogGP

The LOGGP [2] model is an extension of the LOGP model for parallel computation which abstracts the communication of fixed-sized short messages through the use of four parameters. Like in LOGP, those parameters are L : the communication latency, o the overhead, g the bandwidth and P the number of processors. The LOGP model can predict communication cost of short messages accurately. Nevertheless, as many parallel machines have special capabilities concerning bandwidth optimisation of long messages, LOGGP has an additional parameter, G, which considers the bandwidth for long messages. This simple extension can quite accurately predict communication performances for both short and long messages.

Those additional parameters are not able to handle different bandwidth of today's architectures. Indeed, HPC architectures are mainly hierarchical and require a model that manages multi-levels of parameters.

LoPC

The LOPC model [58] aims to extend the LOGP model to deal with contention between nodes. The authors claim that the costs of network latency of both fine-grain message passing algorithms and shared memory are dominated by the network contention. LOPC takes the L, o and P parameters directly from the LOGP model and uses them to predict the cost of contention C. The LOPC model makes the assumption that the hardware message buffers are infinite and do not induce interconnection contention. The main goal of the LOPC model is to write algorithms that minimises contention at runtime, with a limited set of parameters, and give a more accurate prediction of cost. A computation consists in a sequence of *supersteps*, with synchronisation taking place between supersteps.

The LOPC model is close to the BSP model concerning its supersteps execution and cost model. Nevertheless, the point-to-point communication of the model is not structured enough and can lead to complicated communication schemes.

2.1.3 The sub-synchronous family

Even if sub-synchronisations can introduce problems [72], it brings a lot of benefits, especially in terms of performance concerning clusters of multi-cores. Indeed, as shown in Figure 2.1, sub-synchronisation allow to split the machine (S) in order to synchronise, independently, different

groups of processors ($S1$ and $S2$). The extension of the BSP model with sub-synchronisation is thus logical. We will introduce some of the well-know models.

E-BSP

The E-BSP model [95] (standing for *Extended* BSP) extends the BSP model in two ways. First, it aims to deal with unbalanced communication patterns where the amount of data sent or received by each processor is different. Secondly, it adds the notion of general locality to the BSP model by supplying a non-linear delay when accessing a remote memory. Like a BSP computer, the E-BSP model consists of several attributes: a set of processors/memory nodes, a point-to-point message router and a facility to synchronise the nodes. The only difference between BSP and E-BSP is the cost of communication within a superstep. The h-relation denotes the communication pattern in which each processor sends and receives at most h messages. In BSP, the cost of a super step is $g * h + l$ (where g is the bandwidth and l the synchronisation cost). As the BSP model assumes that each processor receives, exactly, h messages, this estimation is overestimated with unbalanced communications: corresponding to the worst-case estimation. The E-BSP model proposes a data routing related to data localities with a better routing distribution (treated as a routing problem) to give a more accurate h-relation.

The E-BSP model trades simplicity for efficiency and complicates the design and analysis of parallel algorithms more than the BSP model. However, it does not handle multi-level architecture, that is the standard of today's architectures, and is not enough structured with the architectures itself.

*d*BSP

The *d*BSP model [147], for *decomposable*-BSP, extends the BSP model to take advantage of sub-machine locality. We denote *d*BSP $(p, g(p), l(p))$ a *d*BSP computer with p processors and particular router performance functions $g(p)$ and $l(p)$. During a superstep, all the processors can execute a *split* instruction to partition the computer into sub-machines (clusters). All the sub-machines compute independently as separate computers in the subsequent superstep. All the processors of all sub-machines must execute the *join* instruction to re-synchronise. After, a join, the computation of the sub-machine is suspended until a join is performed in all the sub-machines. Then, all the processors work together as in BSP. It is important to note that no partial synchronisation of the sub-machines are allowed.

As expected, the cost of a *d*BSP computation which is not decomposed is the same as the BSP one. Otherwise, the time between a pair of corresponding spilt and join is a maximum of the time complexities of computations performed by sub-machines. Assuming that the number of supersteps performed by the sub-machine i is S_i, we have $T = max_i\{T_i\} = max_i\{\sum_{k=1}^{S_i}(w_{k,i} + h_{k,i}g(p_i) + l(p_i))\}$.

A *d*BSP computer is, basically, a BSP computer where independent sub-group synchronisations are allowed. The cost model is thus extended with a few parameters to handle sub-synchronisations. The cost analysis of parallel algorithms is more accurate but also more complicated due to the $g(p)$ parameters that are difficult to measure. Furthermore, the *d*BSP model does not provide a way to manage hierarchical architectures with its own physical specifications.

It is interesting to note that the MULTI-BSP model is, in a way, a sub-synchronisation model. Nevertheless, it is more structured and does not allow dynamic group creation: it is based on the architectures itself.

2.1.4 The hierarchical family

HiHCoHP

The **Hi**erarchical **H**yper **C**lusters **o**f **H**eterogeneous **P**rocessors (HiHCoHP) [18, 20] claims to be the successor of the homogeneous LogP model in a hierarchical way. The aim of HiHCoHP is to propose a detailed enough architecture with an abstraction layer, in order to make the model algorithmically tractable. The HiHCoHP model characterises a hypercluster via parameters that reflect its tri-axial heterogeneity. There is the processor's heterogeneity: the individual processor's message-processing and memory access times; the bandwidth heterogeneity due to the interconnected clusters of a hierarchy of networks; and size heterogeneity of each level of clusters. The computing power of a HiHCoHP resides in N heterogeneous computing nodes composed by a pair of processor and memory. The P_i computing nodes intercommunicate via an l-level hierarchy of networks of successively higher latencies, lower bandwidth and higher capacities. The nodes of a HiHCoHP are aggregated into disjoint $level-1$ clusters. Two nodes communicate over the lowest-level network that they "share". The communication within a HiHCoHP is done by successive message processing and message transmissions between the P_i nodes. This message passing encounters three sources of delay: network latency, network bandwidth and network capacity (maximum number of packets that can be handle at a time).

The HiHCoHP model proposes a very detailed cost model that distinguishes, for each level k and processor a: the communication initialisation cost σ_a^k; the packet emission cost on the network π_a^k; the maximum capacity of the network κ^k; the network latency λ^k and the packet transmission cost β^k. As we have a hierarchy of clusters, we have $\forall a, \sigma_a^k \leq \sigma_a^{k+1}, \pi_a^k \leq \pi_a^{k+1}, \lambda_a^k \leq \lambda_a^{k+1}, \beta_a^k \leq \beta_a^{k+1}$. Thus, the total cost for sending p packets from P_a to P_b is: $(\sigma_a^k + \sigma_b^k) + (\pi_a^k + \pi_b^k) \times p + \lambda^k + \Delta(p)$ where $\Delta(p) = (p-1)/\beta^k$.

The numerous of parameters of the HiHCoHP model make the conception of algorithms difficult. Compared to Multi-BSP, the execution model of HiHCoHP is not structured and can lead to deadlocks.

H-BSP

The H-BSP model [23] aims to add a hierarchical concept to BSP. An H-BSP program consists of a number of BSP groups which are dynamically created at runtime and executed in a hierarchical fashion. Each group behaves as an independent BSP system and they communicate asynchronously. The working mechanism of H-BSP is similar to the process tree concept of Unix, in the sense that *fork* and *join* functionalities are available. Nevertheless, the group creation is done on existing processors and there are no process *spawn*. Thus, H-BSP provides a group-based programming paradigm and naturally supports divide-and-conquer algorithms.

The overall cost of a H-BSP algorithm comes from the BSP root, when the machine is not split. Hence, the cost of a H-BSP algorithm is obtained by combining the cost of the BSP root algorithm plus the sum of the costs of every group split. A group split reflects the time from the split to the join and it depends on the most costly group. Let p be the number of processors at root, f the number of groups at root, G_i the numbers of BSP groups created by the i^{th} split ($1 \leq i \leq f$), gid the created group's id ($0 \leq gid \leq G_i - 1$) and $GT(i, gid)$ the cost of the gid-th group resulting from the i^{th} group split. Then, the cost of the i^{th} group split is $FT_i = max\{GT(i, gid)\ where\ 0 \leq gid \leq G_i - 1\} + S_H(G_i, p)$. Where $S_H(G, p_G)$ is the synchronisation cost depending the number of groups G and the number of processors p_G in the parent group. Thus, the total cost of a split at root is $\sum_{i=1}^{f} FT_i$. Finally, the overall cost of an H-BSP algorithm A is $T_{\text{H-BSP}}(A) = N_{comp}^{root} + N_{comm}^{root} \times g + S^{root} \times l + \sum_{i=1}^{f} FT_i$.

H-BSP inherits the BSP principles but its hierarchical structure reflects the locality of processor communication and synchronisation. H-BSP also provides cost analysis and the authors claim

that the model predictions are better than the BSP model, thanks to its locality-preserving nature. Unlike the MULTI-BSP model, the H-BSP cost model is not fine enough to take into account the various bandwidth and latencies that belong to hierarchical architectures. Moreover, the ability to create groups dynamically makes the algorithmic analysis harder.

SGL

The SGL [105], standing for Scatter-Gather Language, aims to propose a programming language (as a set of primitives) adapted to multi-level and heterogeneous architectures. It is based on the idea of *scattering* data, which corresponds to a phase of data and computation distribution, followed by a *gathering* step, to collect all the computation results. SGL is very close to the MULTI-BSP model and the MULTI-ML language in its recursive BSP vision.

The SGL model defines a set of processors composed by both a processing unit and a memory. The processors are arranged in a tree structure with the root node called "master" and its children that are either masters themselves or "leaf-workers". The number of children is not limited so that the BSP/PRAM concept of flat p-vector of processors is easily simulated in SGL. A SGL machine can be a sequential machine: only one worker without master; a BSP computer: master with several workers; a hierarchical machine of any shape. Nevertheless, a SGL computer must have only one *root*-master, a master coordinates its children through communication, a worker is controlled by only one master and communications is always between master and children. The execution model is composed by a sequence of supersteps composed of 4 phases: (1) A scatter from the master; (2) An asynchronous computation performed by the children (this phase can also be a nested sequence of SGL supersteps); (3) A gather communication phase centred on the master; (4) A local computation phase on the master.

Like all the BSP-inspired systems, the cost T of an SGL algorithm is the sum of the costs of its supersteps S. The cost of a single superstep is divided into two independent terms: computation cost and communication cost. Thus $T = \sum S$ where $S = Comm_{scatter} + maxComp_{children} + Comm_{gather} + Comp_{master}$, $Comm_{scatter} = k_{scatter} \times g_{scatter} + l$, $Comm_{gather} = k_{gather} \times g_{gather} + l$ and k is the number of words, g the bandwidth and l the synchronisation cost. We can express a more detailed cost formula by defining $g \downarrow$ as the time interval for transmitting one word from master to its children and $g \uparrow$ for children to master, $k \downarrow$ the number of words that the master scatters to children and $k \uparrow$ the number of words that master gather from children, c the computation speed of processors (c_0 for the master and $c_i (i = 1...\mathbf{p})$ on children) and $\mathbf{p}$ the number of children processors that master has. Then, $Cost_{master} = \underset{i=1...p}{Cost_{child_i}} + w_0 \times c_0 + k \downarrow \times g \downarrow + k \uparrow \times g \uparrow + 2l$ and $Cost_{worker} = w_i \times c_i$. In case of symmetric communication, we have $Cost_{superstep} = w \times c + (\underset{i=1...p}{max}(Cost_{child_i}) + (k \downarrow + k \uparrow) \times g + 2l)$.

The SGL model proposes a simple execution model with a lightweight cost model to program multi-level architectures. However, there is no programming language implemented over this model. As the SGL communication scheme resides in scatter and gather operations, it is not conceivable to express point-to-point communications. The MULTI-BSP cost model is simpler, as there are fewer parameters, but it is accurate enough for HPC architectures. Regarding its approach and behaviour, SGL is really close to algorithmic skeletons.

2.1.5 The BSP like models

BSP2

The BSP2 [113] model was proposed to reduce the difference between real hardware and the BSP model. The observation states that 1) Local memory access times are variable and are affected by the usual hierarchy of memory, cache and registers. 2) Access times to remote

memory may vary because of non-uniformity within the interconnection network. Difference 1 may be absorbed into the BSP model by allowing to set the processor speed as an application-specific value. The difference 2 could also be concealed using random placement of processes to processors and random message routing. It involves diverting all messages via arbitrary intermediate locations but it will removes any possibility of performance improvements using local communications. To avoid those inefficient solutions, the authors propose an extension of the BSP model which allows two levels of synchronisation: the BSP2 model.

A BSP2 computer consists of a number of BSP units interconnected by a communication network. Each BSP unit consists of a fixed number of processor/memory pairs, connected by a local network. Each processor has a fast access to its local memory and a uniformly slow access to other memories, through the network. Each BSP unit has, also, a uniformly slower access to the memory within other BSP units. The execution of a BSP2 program resides in super-superstep, separated by global synchronisation. During each super-superstep, each BSP unit performs a complete BSP computation and/or communicates data with other BSP units.

The cost model of a BSP2 computer is modelled by seven parameters : (s, p, q, l, L, g, G). Where s is the processor speed, p the number of processors in each BSP unit, q the number of BSP units (total processor count is $p \times q$), l represents the latency for a BSP unit, L is the latency for all BSP units, g is the bandwidth within a unit and G is the bandwidth between two units. Thus, the cost of a super-superstep is $C_l + L + n_c G$ where C_l is the max cost of BSP computation performed by any unit, n_c the max number of values communicated by all the processors of a BSP unit to/from other units.

To conclude, the authors state that the BSP2 model does not show any significant benefit from its approach, even when local communication is several orders of magnitude faster than the remote communication. The BSP2 model fails to provide any major performance gain and encourages the usage of *flat* BSP on architectures with a good global communication network.

Dynamic BSP

The Dynamic BSP model [114] was born on the assumption that BSP was not adapted to grid computing. Dynamic BSP aims to adapt BSP to deal with heterogeneity issues introduced by grid computing. It is a significant extension of BSPGRID [151], a grid-based model for parallel algorithms. The Dynamic BSP is a flexible model allowing to spawn, when it is required, additional processes (virtual BSP processes) within a superstep. It allows to deal with heterogeneity issues as well as fault-tolerance. To do so, the model consists in: (1) a master processor (the task server), (2) worker processes and (3) a data server. At each superstep, a set of virtual processors is executed on the physical processors available on the architecture. The master processor is responsible for task scheduling and memory management. Each virtual process has the responsibility to retrieve local data from the data server, perform the required computations, write back the modified data, and then inform the master processor that it has finished the task. The master processor maintains a queue of pending virtual processors and dynamically assigns them to waiting physical processors. As the virtual processes need to access data, the data server must be implemented over a global shared memory. The principal barrier to scalability of Dynamic BSP is related to the bottleneck induced by such a global memory.

A cost model is also available in the Dynamic BSP model. However, as the g (bandwidth) and l (latency) parameters may vary significantly between grid nodes, the cost model difficult to handle.

Multi-bab and Multi-dac

In [129] and [131] the authors proposes two approaches that are, respectively, dedicated to branch-and-bound (bab) and divide-and-conquer (dac) algorithms. These approaches aim to program architectures that are based on a tree hierarchy of memories.

The MULTI-BAB model [129] maps branch-and-bound algorithms on a cache hierarchy of multi-cores. The model provides a cost model to predict the upper bounds for communication and synchronisation.

The MULTI-DAC model [131] deals with divide-and-conquer algorithms. It also proposes a cost model dedicated to this particular type of algorithms.

Currently, those models are not implemented over a programming language.

MBSP

The MBSP model [65], or multi-memory BSP, is an extension of BSP that abstracts and models parallelism in the presence of multiple memory hierarchies and multiple cores. The aim of MBSP is to retain the properties of BSP and, in addition, to abstract not only traditional external memory-supported parallelism (that uses another level of slower memory) but also multi-level cache-based memory hierarchies such as those present in multi-core systems. To do so, the model is parameterised by the septuplet (p, l, g, m, L, G, M). In addition to the BSP parameters (p, l, g), the collection of core/processor components has m alternative memory units distributed in an arbitrary way. The size of the *fast memory* is composed by M words. It is important to note that the model makes no assumption whether an alternative memory unit resides on every processor/core or if a unit is controlled by one of the p processors or not. Whatever the case is, the cost of memory unit-related accesses is modeled by the pair (L, G). L and G are, respectively, similar to the BSP parameters l and g. That is to say, G expresses the unit transfer bandwidth and L a denominate the memory latency time.

With the MBSP model, the cost of a superstep is $max(l, L, w) + gh + Gb$, where w is the maximum computational work of any process, h is the maximum size of sent or received messages by any processor in a superstep and b is the maximum amount of information read or written into an alternative memory-unit.

The MBSP model could be viewed as a two-level version of a MULTI-BSP instance. However, the model is not able to finely handle several levels of memory, latencies or bandwidths.

2.2 High-level parallel programming

Today, popular parallel and distributed programming methodologies are dominated by low-level techniques. High-level approaches offer many possible advantages by abstracting implantation details. Such structured approaches have a key role in scalable, portable and ubiquitous parallelism. We now present the main paradigms and try to list the pros and cons compared to our approach.

2.2.1 Algorithmic skeletons

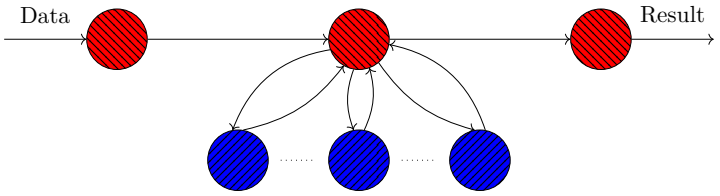
Proposed in [31], algorithmic skeletons are high-level parallel patterns that are build to solve a particular problem. They aim to be seen, by the programmer, as high order functions that are part of a collection of algorithmic tools from which a problem specific program must be fashioned. Meanwhile, the system implementor sees each skeleton as a generic computational pattern for which an efficient parallel code must be defined. An algorithmic skeleton must hide all the implementation and optimisation details in order to provide a simple way to use it.

Even if many parameters may be required, using an algorithmic skeleton allows to avoid low-level considerations, such as data decomposition, computation scheduling and communications. They also aim to be reusable in various domains and applications while being highly optimised for parallel computations. We can easily distinguish *general purpose skeletons*, that appear in a wide range of applications from different domains, and *domain specific skeletons*, that are specialised to particular domains. The aim of this section is to present the most common skeletons (we do not intend to be exhaustive) and some of the well-known libraries.

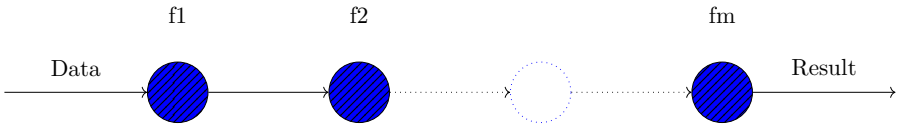
We can classify skeletons into four principal categories.

1. First, we have *task parallel skeletons*. Here, the parallelism is expressed by different "tasks" that are available as an input stream. All the tasks are distributed and scheduled to the different workers. The most common task parallel skeletons are the following:

   - *Farm* (Figure 2.2a) is a basic task distribution that is done on all the available computing units. When the tasks are completely independent, we talk about *pure farms*.
   - *Pipeline* (Figure 2.2b) is an algorithmic skeleton that splits a complex computation process into many simple steps. Each step is attributed to a stage S, that is a computation unit. The result of the stage S_i is the input of the stage S_{i+1}.



(a) Farm skeleton.



(b) Pipeline skeleton.

Figure 2.2: Data parallel skeletons.

2. Then, we have *data parallel skeletons* that compute a single task, in parallel, on distributed pieces of data. Those skeletons are often adapted to various data structures [96] such as lists, arrays, matrices, trees, *etc.* The principal skeletons are:

   - *Map* is a well known pattern that consists in applying a function f on a distributed collection of data $x_1, ..., x_n$. The resulting data collection $y_1, ..., y_n$ is obtained by $y_i = f(x_i)$.
   - *Reduce* is also a common operator that "sums up" all the items of a data collection $x_0, ..., x_n$ given a commutative operator.

   Figure 2.3 shows how the map-reduce skeleton works. From a distributed set of data, we apply an operator *op* on every value. Then, a resulting value is generated using a reduce function. The map-reduce concept is well known for a long time, especially in functional programming. The map-reduce skeleton should not be confused with the Map-reduce frameworks *proposed* by GOOGLE [40], as explained in the following section.

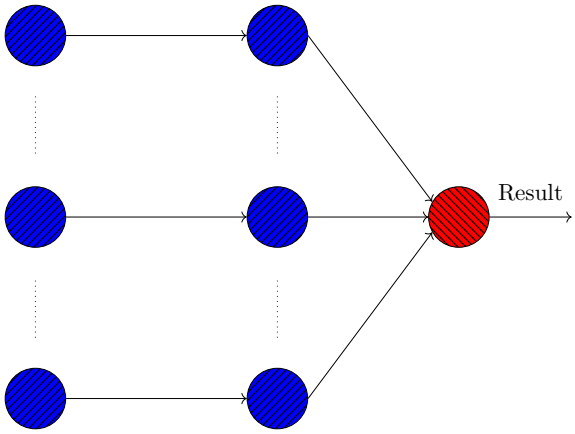


Figure 2.3: The mapreduce skeleton.

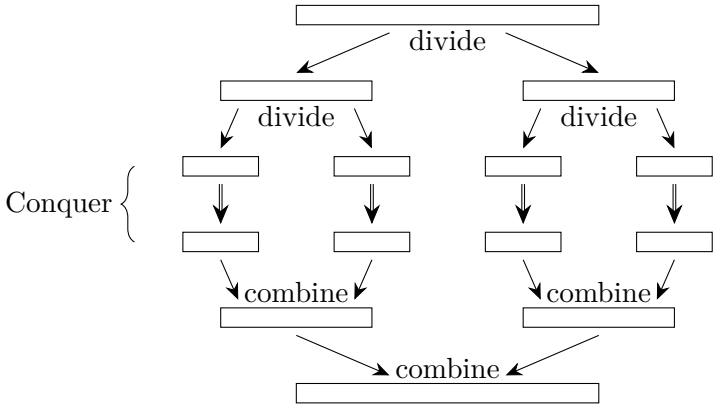


Figure 2.4: The divide and conquer skeleton.

3. The *control (service) parallel skeletons* are used to control a flow of data using parallel operators. For example:

   - *Divide & Conquer* (see Figure 2.4) which starts by *dividing* tasks into sub-tasks recursively until the sub-tasks are easily solved (*conquer* phase). Then, the partial results are recursively combined to compute the final result.
   - *While* is a simple skeleton that represents a conditional iteration where a given skeleton is executed until a condition is valid.

4. Finally, *specialised parallel skeletons* are dedicated to a specific task and are mainly used in particular domains. For example:

   - *Stencil* maps a function computing $y_i = f(x_i, \mathcal{N}(x_i))$ for some neighbourhood function $\mathcal{N}$, as shown in Figure 2.5.

9	2	8	7
7	5	6	6
5	7	4	8
3	9	2	1

$\times$

2	1	1
1	0	-1
-1	-1	-2

$=$

	7	−9	
	10	8	

Figure 2.5: The stencil skeleton.

- *Wavefront* [4] is a pattern close to the stencil where each element of a grid computes a value that depends on the computation of a set of previous elements. The computation typically flows from one region to another.
- *Orbit-skeleton* [92] is a pattern close to the state-space. The finite state-space construction problem is computing the explicit graph representation (also known as Kripke structure) of a given model from the implicit one. This graph is constructed by exploring all the states reachable through a successor function succ (which returns a set of states) from an initial state $s0$. Generally, during this operation, all the explored states must be kept in memory in order to avoid multiple explorations of a same state.

There are many algorithmic skeleton frameworks and libraries. They mainly differ in their skeleton set and their implementation language such as C, C++, JAVA, *etc.* A nice classification could be found on the dedicated wikipedia page: [[10]]. It is important to note that most of the skeleton libraries do not provide a way of nesting computations; that is to say nesting of algorithmic skeletons. Furthermore, the type safety is not always available either.

One can say that algorithmic skeletons are an easier way of programming parallel algorithms. Nevertheless, as algorithmic skeletons are derived from a set of parallel patterns [32], it is not always possible to find patterns that meets our needs. In this case, it is possible to extend the pattern set, but it requires a lot of work for the library implementer and complicates the user side by adding new skeletons. Otherwise, skeleton libraries mostly provide a special skeleton in which the programmer is free to use whats he wants (mostly C,C++, JAVA and MPI) with the drawback of adding unsafe features. It also requires a deep knowledge of how the skeleton work to be able to interact with them.

We now introduce some skeleton frameworks and libraries that are common to the domain. We also describe them and compare their capabilities.

SKLML

The SKLML [[11]], formerly called OCAMLP3L, is a functional parallel skeleton compiler and programming system for OCAML programs. This framework embeds an innovative compositional skeleton algebra into the OCAML language. Thanks to its skeleton algebra, SKLML provides two ways of program evaluation: (1) a regular sequential evaluation (merely used for prototyping and debugging) and (2) a parallel evaluation obtained via a recompilation of the same source program in parallel mode. SKLML was specifically designed to show that the sequential and parallel evaluation regimes coincide. To do so, the framework proposes several skeletons such as farm, pipeline (denoted `|||`), product (denoted `***`), sum (denoted `+++`), *etc.* The SKLML allows to write elegant functional codes. For example, to compute $f(x) = g(x) + h(x)$ with the `***` skeleton, we can write the following code [[12]]:

```
let split_skl = skl () -> fun x -> (x, x) in
let merge_skl = skl () -> fun (x, y) -> x + y in
let f_skl = (split_skl ()) ||| (g_skl *** h_skl) ||| (merge_skl ()) in
```

The `skl ()` syntactic construction allows the programmer to *transfer* a regular OCAML function into the *universe of skeletons*. Then, `g_skl` and `h_skl` stands for the computational functions which are used with the `***` skeleton, surrounded by the `split_skl` and `merge_skl` phases.

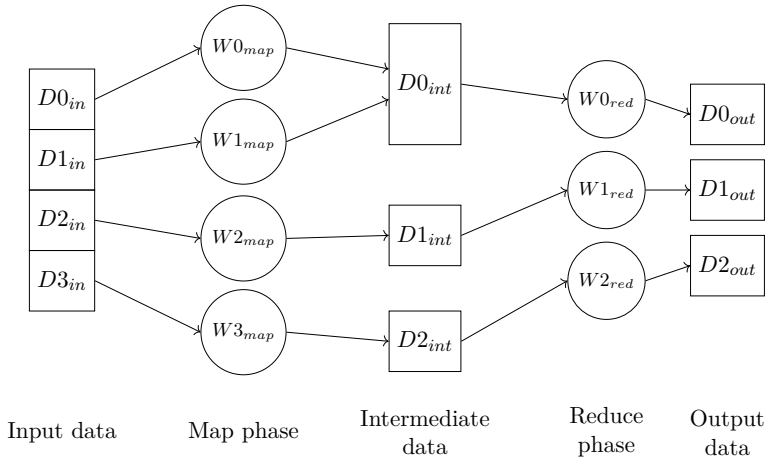


Figure 2.6: The *Google*'s mapreduce skeleton.

The MapReduce framework

The MapReduce parallel computation framework was introduced in [40] and widely inspired by the BSP model and algorithmic skeletons. The authors describe a programming model with an associated implementation for processing and generating large datasets (the famous BigData) that is amenable to a broad variety of real-world tasks. The user must specify a *map* and a *reduce* function first. Then, the runtime system will automatically run parallel computation across the available computation units: the *workers*. To do so, the computation takes a set of input (key/value pairs), and produces a set of output (key/value pairs). The *map* function takes an input pair and produces a set of intermediate key/value pairs. This set of intermediate values, identified with the key I, is then given to the *reduce* function. As expected, the *reduce* function takes an intermediate key I and a set of values corresponding to that key. It merges these values together to form a possibly smaller set of values. Figure 2.6 summarises the behaviour of the MapReduce framework, where square nodes stand for data, circles for the workers and the communications (merges) are symbolised by directed arrows. Many implementation of this framework are available, such as the HADOOP framework [153].

For example, we consider the problem of counting the number of occurrences of each word in a large collection of documents. The pseudo-code of the map and reduce function that the user would write is the following code [40] (Figure 2.7):

The `map` function emits, for each word, a pair identified by the word and a count of one. The `reduce` function sums all counts emitted for each words. At runtime, all the elements produced by the *map* function are *given* to the *reduce* function in order to compute the final list of counts.

The MapReduce framework is easy to use, even for programmers who are not familiar with parallel and distributed programming. A large variety of problems are easily expressible as MapReduce computations; an in-depth comparison with BSP is available in [121]. However, there is no way to exploit data locality in order to optimise computations. The performance mostly relies on the implementation of the framework and the communication scheme is limited. The BSP cost and scalability analysis for the MapReduce framework is proposed in [133].

FastFlow

FASTFLOW [37] is a C++ parallel programming framework advocating high-level, pattern-based parallel programming. FASTFLOW proposes a skeletal parallel programming framework specialised on streaming and data-parallel applications. It chiefly supports streaming and data

```
//key : document name, value : document content
map(string key, string value){
  for each word w in value
    emmitIntermediate(w,"1");
}

//key: a word,values: a list of counts
reduce(string key, Iterator values){
  int result = 0;
  for each v in values
    result +=parseInt(v);
  emit(asString(result));
  }
```

Figure 2.7: A MapReduce example.

parallelism, targeting heterogeneous platforms composed of clusters of shared-memory platforms, possibly equipped with computing accelerators. The main design philosophy of FASTFLOW is to provide applications to designers with key features for parallel programming (e.g. time-to-market, portability, efficiency and performance portability) via suitable parallel programming abstractions and a carefully designed run-time support. The FASTFLOW run-time support uses several techniques to efficiently support fine grain parallelism (and very high frequency streaming), such as non-blocking multi-threading with lock-less synchronisations; Zero-copy network messaging; Asynchronous data feeding for accelerator offloading.

For example, the code [[[13]]] in Figure 2.8 shows how to construct a 3-level pipeline using FASTFLOW.

The `firstStage` produces the data stream. The `secondStage` is the task that processes the data stream. Then, the `thirdStage` finalises the pipeline computation. Finally, we call the `ff_Pipe` skeleton with the three stages to execute the pipeline. The pipeline is created and evaluated within the FASTFLOW runtime, transparently.

Muesli

The Muenster Skeleton Library [29, 30], called MUESLI, is a C++ template library enabling fluent programming of multi-node, multi-core cluster computers by implementing the concept of algorithmic skeletons. MUESLI supports most of the task and data parallel skeletons. It proposes high scalability across nodes and cores by simultaneously using MPI and OPENMP. The user can also build complex skeletal topologies by arbitrary nesting both task and data parallel skeletons.

For example, the code [[[14]]] of Figure 2.9 shows how to construct a 3-level pipeline using the MUESLI library.

As expected, `init` produces the input data of the pipeline. The `compute` function is the computation stage of the pipeline and `fin` is the finalisation function that compute the resulting value. To execute the pipeline, we must construct the `Pipe` skeleton with the previously defined stages and start the execution using `pipe.start()`.

```
#include <iostream>
#include <ff/pipeline.hpp>
using namespace ff;

typedef long fftask_t;

struct firstStage: ff_node_t<fftask_t> {
  fftask_t *svc(fftask_t *t) {
    for(long i=0;i<10;++i)
      ff_send_out(new fftask_t(i));
    return EOS; // End-Of-Stream
    }
};

fftask_t* secondStage(fftask_t *t,ff_node*const node) {
  std::cout << "Hello I'm stage" << node->get_my_id() << "\n";
  return t;
}

struct thirdStage: ff_node_t<fftask_t> { fftask_t *svc(fftask_t *t) {
    std::cout << "stage" << get_my_id() << " received " << *t <<"\n";
    delete t;
    return GO_ON;
  }
};

int main() {
  ff_Pipe<> pipe(make_unique<firstStage>(), make_unique<ff_node_F<fftask_t>
  ↪ >(secondStage), make_unique<thirdStage>() );
  if (pipe.run_and_wait_end()<0)
  error("running pipe");
  return 0;
}
```

Figure 2.8: A FASTFLOW example.

```
#include "Muesli.h"
#include <iostream>
using namespace std; int

current = 10;

int* init(Empty){
  int* i = (int*) malloc(1*sizeof(int));
  *i = current;
  current--;
  if(current < 0) return NULL;
  cout << "IN is sending: " << *i << endl;
  return i;
}

int* compute (int* input){
  cout << "Compute received: " << *input;
  int total = 1;
  for(int i=0; i<*input;i++) total *= 2;
  *input = total;
  cout << " - Compute is sending: " << *input << endl;
  return input;
}

void fin(int* input){
  cout<< "OUT received: "<< *input << endl;
  free(input);
}

int main(int argc, char** argv){
  InitSkeletons(argc,argv);

  Initial<int> in(init); Atomic<int, int> atomic(compute,1);
  Final<int> out(fin); Pipe pipe(in, atomic, out);

  pipe.start();

  TerminateSkeletons();
  return 0;
}
```

Figure 2.9: A MUESLI example.

Skepu

SKEPU [47] is a skeleton programming framework for multi-core CPUs and multi-GPU systems. It is a C++ template library with six data-parallel and one task-parallel skeleton, two generic container types, and support for execution on multi-GPU systems both with CUDA and OPENCL. It also supports hybrid execution, dynamic scheduling and data load balancing thanks to the STARPU runtime system. It also provides a modern C++ syntax and an efficient implementation.

The following code[[[15]]] (Figure 2.10) shows how to program a dot product using the MapReduce skeleton in SKEPU.

```
float add(float a, float b){ return a + b; }

float mult(float a, float b){ return a * b; }

float dot_product(Vector<float> &v1, Vector<float> &v2){
  auto dotprod = MapReduce<2>(mult, add);
  return dotprod(v1, v2);
}
```

Figure 2.10: A SKEPU example.

The *map* sequence is defined by the `add` function while the *reduce* phase is defined by the `mult` function. The skeleton `MapReduce` is instantiated using the *map* and *reduce* functions. Finally, the skeleton istance is used as a regular function. The evaluation of the skeletons is transparently handled by the SKEPU runtime.

Quaff

QUAFF [48] is a skeleton-based parallel programming library. Its main originality is to rely on C++ template meta-programming techniques to achieve high efficiency. It performs most of the skeleton instantiation and optimisation at compile-time, and keeps the overhead traditionally associated with object-oriented implementation of skeleton-based parallel programming libraries very small.

QUAFF provides an implementation of the main skeletons, such as *pipeline*, *farm* and *pardo*, on both shared and distributed memories. Apart from these parallel skeletons, it also provides constructs allowing tasks or skeletons to be composed sequentially or conditionally in applications. Furthermore, skeleton nesting is naturally supported in QUAFF. Indeed, once instantiated, all skeletons are valid functors and consequently can be used as arguments to other skeleton templates. For example, the following code [48] (Figure 2.11) shows how nested skeletons can be used:

```
typedef task< CGetImg , none_t , image > get_img;
typedef task< CSobel , image , image > sobel;
typedef task< CSaveImg , image , none_t > save_img;

typedef farm< worker< repeat<sobel,4> > > inner;
typedef pipeline< stage<get_img, inner, save_img> > pipe;
typedef application< pipe > main_app;
```

Figure 2.11: A QUAFF example.

In this example, we intend to read an image, apply a Sobel filter and write the output image. The program is defined as a pipeline. The second stage is a farm. The 3 first definitions contain the specific sequential tasks. The farm skeleton is defined by giving the tasks to be computed by workers and the amount of workers. The pipeline uses the farm as its second stage to, respectively, read, compute and write the resulting image.

The parallel libraries which we just introduced aim to make parallel programming easier. To do so, they propose skeletons that have been designed to execute specific parallel patterns. The programmer does not have to worry about optimisations, everything is done within the skeleton. However, some algorithms are too complex to be modeled by a *simple* skeleton. In such cases, some programmers want to be able to write a specific optimised code that is structured enough, thanks to an expressive parallel programming model.

2.2.2 BSP programming

The Message Passing Interface

The **M**essage **P**assing Interface (MPI for short) library [142] is a standard API for communication in distributed-memory parallel applications. There are numerous implementations of this API, both commercial and open source (*e.g.* open-mpi, mpich, *etc.*). MPI aims to provide a tool for high-performance, scalability and portability. MPI is the de facto standard for high-performance computing today. It is the main library that is used for implementing parallel languages. The MPI standard defines the informal semantics of routines that are useful to a wide range of users writing both shared memory and distributed parallel programs in FORTRAN and C. However, there are many stubs codes for languages such as JAVA, OCAML, *etc.*

As MPI contains a huge number of routines, it is not possible to present an exhaustive list. We thus only present the routines that are close to BSP such as collective operations and operations for the communicators manipulation. That is to say subgroup creation and, thus, subgroup synchronisation. We also present some of the primitives that are used in the MULTI-ML implementation, see Chapter 5.

After the initialisation of MPI it is possible to access the initial communicator `MPI_COMM_WORLD` which contains all the processes participating to the computation. A collective communication is defined as communication that involves a group of processes, *i.e.* a communicator.

All processes in a group identified by a communicator must call the same collective routine. We do not give explanation of all the arguments but resume the most important ones for our purpose. The details concerning several MPI routines and its arguments could be found in Table 2.1. In many cases, collective communication can occur "in place" for communicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, `MPI_IN_PLACE`, instead of the send buffer or the receive buffer argument, depending on the performed operation.

In MPI, using point-to-point communications is a usual practice. To do so, the routine `MPI_Send` can be used to send data to another process. To receive the data, the routine `MPI_Recv` must *catch* the sent value. It is important to note that those two routines perform a blocking send/receive. MPI also provides a non blocking send: `MPI_Isend`, a synchronous one: `MPI_Ssend` and many other optimised alternatives.

The collective operations are for broadcasting, scattering, gathering and all-to-all exchanges. The main arguments are the buffer to send (respectively, to receive the data) with the number of data elements and their types (`MPI_INT`, `MPI_DOUBLE`, *etc.*) to ensure portable programs. As many C routines, they return an integer that indicates if the operation succeeded. The MPI primitives are often annotated with a suffix to specify its behaviour. For example, `MPI_Alltoall` is the all-to-all communication primitive. It sends a data of a fixed size to all concerned processes. `MPI_Alltoallv` stands for a all-to-all communication primitive that can send data with various sizes. Finally, `MPI_Alltoallw` is the most general form of all-to-all, it allows the communication of various types of data of different sizes. For example, by making all processes have `recvcounts[i] = 0`, the primitive achieves a `MPI_scatterw`, which perform a data distribution on processes. This is a powerful routine but it is hard to use. The `sendrecv` routine involves only two processes which exchange data in a synchronous way as a barrier. In fact, this is a subgroup of only two processors that are communicating synchronously. Collective operations are not really BSP communications since they are not synchronous. For example, using a broadcast, if the emitter changes the buffer to send after the call of the routine, the value received by other processors is unspecified. A call to a `MPI_barrier()` can be used to avoid such an undesirable behaviour. In practice, most MPI libraries implement collective operations in a synchronous way but this is not standard.

There are many routines for managing communicators (we count 35 in MPI-2). However, the main routine is `MPI_comm_split` which creates a new communicator by partitioning the group into disjoint sub-groups using a set of *colours*. Each subgroup contains all processes with the same colour. Within each sub-group, the processes are ranked in the order defined by the value of an argument key, with ties broken according to their rank in the old group. This is a collective call, but each process is permitted to provide different values for colour and key. This mechanism is very powerful for dividing a single communicating group of processes into k sub-groups and many routines, such as `MPI_comm_create`, are similar to a specific call to `MPI_comm_split`. Unlike the PUB (see thereafter) function, this MPI routine requires communications between the processors of the group. In fact, exchanging the colours can be seen as a call to the `MPI_Allgather` collective operation. Communicators in MPI are really flexible but they are complex. Unlike in PUB, there is almost no restriction to the way groups are formed. Processes can be in more than one group, as they have an unique rank within each group. A traditional example of use of such communicators would be in linear algebra, where it is often useful to split processors by rows and by columns.

MPI also provides routines for asynchronous DRMA accesses. Thus a programmer can manage a *window*. It is a synchronous operation that every processor performs. The, processors can write or read on remote processors using that window. There is a specific barrier, the `fence`, for processors that have been registered to the window. Using the `Win_complete`, we can force a processor to wait until all remote readings and writings have been performed on a window. Furthermore, MPI obviously provides asynchronous sends/receives of data. Such primitives are intentionally ignored in this thesis.

MPI can be seen as the *assembly* of the communication libraries. Most programming languages using communication are based on MPI, or could rely on it. MPI is very powerful, however, it is quite complicated to handle and may lead to complex code and deadlocks.

Collective communicating operations:	
`int MPI_Barrier(MPI_Comm comm)`	
`int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`	
`int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`	
`int source, int recvtag, MPI_Comm comm, MPI_Status *status)`	
`int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`	
`int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`	
`int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`	
`int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`	
`int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`	
Collective reducing operations:	
`int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`	
`int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`	
`int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`	
Communicator manipulation:	
`int MPI_Comm_dup(MPI_Comm comm,MPI_Comm *newcomm)`	Duplicates an existing communicator
`int MPI_Comm_free(MPI_Comm *comm)`	Deallocation of the communicator
`int MPI_Comm_split(MPI_Comm comm,int color,int key,MPI_Comm *newcomm)`	Creates new communicators based on colors and keys
`MPI_Comm peer_comm,int remote_leader,int tag,MPI_Comm *newintercomm)`	Creates a communicator from two others
Windows for DRMA:	
`int MPI_Win_create(void *base,MPI_Aint size,int disp_unit,MPI_Info info,MPI_Comm comm,MPI_Win *win)`	Creation of a window
`int MPI_Win_fence(int assert,MPI_Win win)`	Fence synchronisation
`int MPI_Put(void *origin_addr,int origin_count,MPI_Datatype origin_datatype,int target_rank,` `MPI_Aint target_disp,int target_count,MPI_Datatype target_datatype,MPI_Win win)`	Put data into a memory window on a remote process
`int MPI_Get(void *origin_addr,int origin_count,MPI_Datatype origin_datatype,int target_rank,` `MPI_Aint target_disp,int target_count,MPI_Datatype target_datatype,MPI_Win win)`	Get data from a memory window on a remote process

Table 2.1: The MPI primitives.

The BSPlib

The BSPLIB [86] is a C-library of communication routines. It aims to support the development of parallel algorithms based on the BSP model. It offers routines for both message passing (BSMP) and remote memory access (DRMA). The library offer two implementations over TCP/IP and MPI. Its implementation aims at distributed-memory super-computing and thus explicitly starts separate processes on supercomputer nodes. The main routines of the BSPLIB are (1) Core; (2) Synchronisation; (3) Message passing (4) and DRMA routines. It is interesting to note that about 20 routines are available, compared to the more than 200 that composed MPI. The BSPLIB is thus easier to understand and to use, without lack of expressivity for programming HPC architectures. As the BSPLIB can be seen as an *extension* of the PUB, the main primitives are jointly detailed in Table 2.2. We now introduce the main routines:

Core Routines. As in standard MPI, we first need to initialise our parallel computation, using the function `bsp_init`. Then, we can have different BSP computations, each beginning with `bsp_begin` and terminating with `bsp_end` . Within a BSP computation, we can query some information about the machine: `bsp_nprocs` returns the number of processors $\mathbf{p}$ and `bsp_pid` returns the processor identifier which belongs to $0, ..., \mathbf{p} - 1$.

Synchronisation. According to the BSP model, all messages are received during the synchronisation barrier and cannot be read before. The barrier is done using `bsp_sync` which blocks the node until all other nodes have called `bsp_sync` and all messages sent to it, in the current superstep, have been received.

Message Passing. With BSMP, a non-blocking send operation is provided to deliver (a packet of) messages to a system buffer associated with the destination process. It guarantees that the message will be available in the destination buffer at the beginning of the subsequent superstep. Furthermore, the message can be accessed by the destination process only during that superstep. If the message is not accessed within the superstep it is removed from the buffer. With the BSPLIB, the destination buffer of a processor may be viewed as a queue where incoming messages are enqueued in an arbitrary order.

Sending a packet (in a buffering mode) is done using `bsp_send`. The routine is based on the idea of a two-part message. A fixed-length part carries tag information that will help the receiver to interpret the message; a variable-length part contains the main data payload. The length of the tag is required to be fixed during any particular superstep, but can vary between supersteps.

Choosing a tag is done using `bsp_set_tagsize` where the user specifies the size of the fixed-length portion of every message in the subsequent supersteps. It allows the user to set the tag size to enable the use of tags that are appropriate for the communication requirements of each superstep. This is particularly useful in the development of subroutines either in user programs or in libraries. The procedure must be called collectively by all processes. A change in tag size takes effect in the following superstep; it then becomes valid. The programmer can know the number of received messages as well as the total size of received data (in bytes) using the routine `bsp_qsize` . This routine works on the queue of received messages. To receive a message, the user should use the procedures `bsp_get_tag` and `bsp_move`. `bsp_get_tag` returns the tag of the first message in the queue and the size of the corresponding payload (status is -1 if the queue is empty). `bsp_move` copies the payload of the first message of the system queue, *i.e.* the buffer call payload, and removes it from the queue. Then, the system will advance to the next message.

DRMA **routines.** Registering or deleting a variable from global access is done using: `void bsp_push_reg(ident,size)` and `bsp_pop_reg(ident)` (see Table 2.2). Due to the SPMD structure of BSP programs, if p instances share the same name, they will not, in general, have the same physical address. To allow BSP programs to be executed correctly, the BSPLIB provides a mechanism for relating these various addresses by creating associations called registrations. A registration is created when each process calls `void bsp_push_reg`. If different variables have to be registered/unregistered, all processors must call the functions in the same order. Both the address and the extent of a local area of memory must be provided. The registration takes effect at the next synchronisation barrier and newer registrations replace older ones. This scheme does not impose a strict nesting of push-pop pairs. For example:

Processor 0	Processor 1
`void x[5],y[5];`	`void x[5],y[5];`
`bsp_push_reg(x,5);`	`bsp_push_reg(y,5);`
`bsp_push_reg(y,5);`	`bsp_push_reg(x,5);`

The variable `x` on processor 0 would be associated with `y` on processor 1. Note that this example is clearly not a good way of programming in BSP. In the same manner, a registration association is destroyed when each process calls `bsp_popregister` and provides the address of its local area participating in that registration. A run-time error will be raised if these addresses (*i.e.* one address per process) do not refer to the same registration association. Un-registration takes effect at the next synchronisation barrier.

The two DRMA operations (Figure 2.12) are the following:

- `bsp_get(int pid,const void *src, int offset,void *dst,int nbytes)` stands for global reading access. It copies n bytes to the local memory address `dst` from the variable `src` at offset `offset` of the remote processor pid ;
- `bsp_put(int pid,const void *src,void *dst,int offset,int nbytes)` stands for global writing access. It copies n bytes from local memory `src` to `dst` at offset `offset` on remote processor pid .

It is important to note that the `get` and `put` operations are executed during the synchronisation step and all `get` are served before a `put` overwrites a value.

It is interesting to note that the BSPLIB also provides high-performance variants of the DRMA and BSMP primitives. These are `bsp_hpput`, `bsp_hpget`, and `bsp_hpsend`, and allow for communication to occur somewhere between the call to the *hp*-primitive and the end of the next

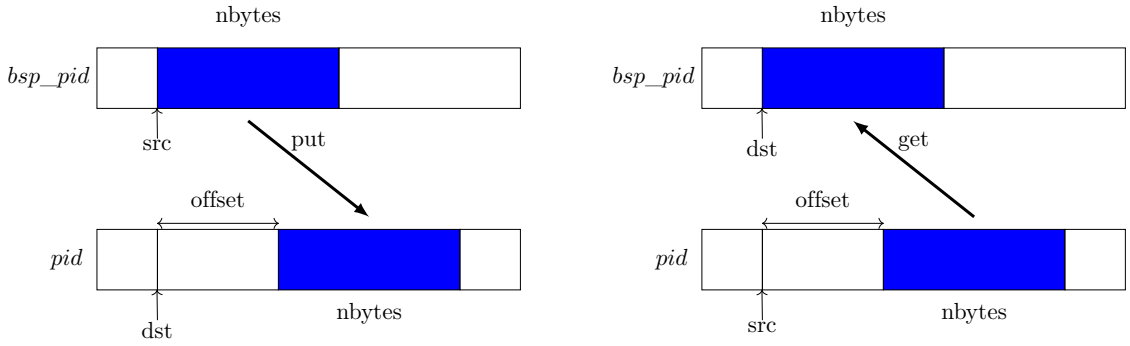


Figure 2.12: DRMA BSP operations: “put” and “get”.

Tools:	
`int bsp_nprocs(t_bsp* bsp)`	Returns the number of processors in the group
`int bsp_pid(t_bsp* bsp)`	Returns own processor-id in the group
`void bsp_sync(t_bsp* bsp)`	BSP synchronization of the group
BSMP primitives:	
`void bsp_send(t_bsp* bsp, int dest, void* buffer, int size)`	Bulk sending of a buffer
`void bsp_sendmsg(t_bsp* bsp, int dest, t_bspmsg* msg, int size)`	Bulk sending of a message
`t_bspmsg* bsp_createmsg(t_bsp* bsp, int size)`	Create a message
`int bsp_nmsgs(t_bsp* bsp)`	The number of received messages or buffers
`t_bspmsg* bsp_findmsg(t_bsp* bsp, int proc_id, int index)`	Find a message in the queue
`void* bspmsg_data(t_bspmsg* msg)`	Returns a pointer to the data of a message
`int bspmsg_size(t_bspmsg* msg)`	Returns the size of the message contain
`int bspmsg_src(t_bspmsg* msg)`	Returns the node-id of the sender
DRMA primitives:	
`void bsp_push_reg(t_bsp* bsp, void* ident, int size)`	Register a variable for remote access
`void bsp_pop_reg(t_bsp* bsp, void* ident)`	Delete the registration of a variable
`void bsp_put(t_bsp* bsp, int destPID, void* src, void* dest, int offset, int nbytes)`	Remote writing to another processor
`void bsp_get(t_bsp* bsp, int srcPID, void* src, int offset, void* dest, int nbytes)`	Remote reading from another processor
Subgroup routines:	
`void bsp_dup(t_bsp* bsp, t_bsp* dup)`	A new group as a copy of the group
`void bsp_partition(t_bsp* bsp, t_bsp* sub, int nr, int* partition)`	Creates a new subgroup
`void bsp_done(t_bsp* bsp)`	Destroy a subgroup

Table 2.2: The PUB/BSPLIB primitives.

`bsp_sync`. The usage of such an *hp*-primitive is twofold. Firstly, *hp*-primitives allow to overlap the computation and communication whenever it is possible. Secondly, it avoids the inefficiency of buffering communication. When those primitives are used, the *user* must guarantee that the source and destination memory areas remain unchanged until the end of the current superstep. Non-deterministic behaviour can easily be introduced using such *hp*-primitives.

The Paderborn University BSP library

The **P**aderborn **U**niversity BSP (PUB) library [14] is a C communication library based on the BSP model. The PUB's routines are close to the BSPLIB ones. The main differences remains in the availability of subgroup synchronisation features. The way that values are stored with concurrent access is also particular using the PUB. The main PUB routines are available in Table 2.2. In the following, we describe the main differences between the BSPLIB and the PUB. The DRMA routines are comparable except their subgroup capabilities.

Message Passing. Sending a single message can be done using `bsp_send` or `bsp_sendmsg`. After calling one of these routines the buffer (or the message) may be overwritten or freed. The `bsp_send` routine is recommended with a code with a small amount of communication during a superstep. On the contrary, with a lot of data with separated memory locations, it is better to use the `bsp_sendmsg` routine. As for DRMA operations, these routines work in the scope of a subgroup.

Subgroup Primitives. The main functions involved in the sub-grouping process are `bsp_dup`, `bsp_partition` and `bsp_done`. The `bsp_dup` function creates a subgroup that is a duplicate of the current BSP computer. This is useful to organise algorithms in a compositional manner, since a code working on the duplicate will not be affected by previous messages that were already in the queue, waiting to be sent. The synchronisation on the duplicate subgroup will only complete the communications that were requested with that subgroup. It is thus possible to define a function for a sub-algorithm, starting with a call to `bsp_dup` so that it can be called from anywhere in a parallel program, without interfering.

The `bsp_partition` function is the way to create proper subgroups. The subgroups are described as a partition of the current BSP computer in contiguous subsets. The starting indexes of these subsets are given as argument in the partition integer array. To be a correct partition, the array has to be sorted. Finally, the `bsp_done` function indicates that we have finished the work with the subgroup. In the PUB library, it is impossible to synchronise or create other subgroups from the parent object until the current subgroup is released with the routine `bsp_done`. However, it is possible to create different new subgroups of the current subgroup. The organisation of subgroups is similar to a stack, with only the lowest subgroup being allowed to create new subgroups. Here is an example of a program, in the PUB library, with subgroup synchronisation:

```
t_bsp subbsp;
int part[2];
part[0] = 2;
part[1] = bsp_nprocs(bsp);
bsp_partition (bsp, &subbsp, 2, part);
if(my_pid<2)
  {...;bsp_sync(&subbsp);...}
else {...}
bsp_done(&subbsp);
```

Communicators are thus present in both MPI and the PUB library as group which are called BSP objects. Although they play a similar role, there are notable differences between both. The PUB library allows to partition the set of processors into pairwise disjoint subsets. Each subset acts like an independent BSP computer. Subgroups are created by a call to the `bsp_partition` function. A call to the function creates a new `t_bsp` object, that can be used in any BSP call. It is also still possible to use the parent BSP object. A call to `bsp_partition` does not require communications between the processors, since each processor has to explicitly give the whole partition table. However, there are limitations. It is not possible to create other subgroups of the parent BSP object, until the current subgroup `subbsp` has been disposed of using the `bsp_done` function. The subgroups work like a stack: it is only possible to create subgroups of the lowest subgroup. Another limitation comes from the way the subgroups are defined. The partition has to be made of subgroups of consecutive processors. It is not possible to combine the processors 1 and 3 in a group, and 2 and 4 in another one, for a parallel machine of four computers.

The PUB and the BSPLIB can introduce synchronisation problems such as deadlocks. For example, the code `if pid = 0 then sync() else {}` makes the code unsafe. Incomplete synchronisation of groups of processes also leads to unwanted behaviours. It is thus necessary to use sophisticated tools [53] to avoid such problems.

MulticoreBSP

The MulticoreBSP for C programming interface is directly derived from the BSPLIB. It adds two new high-performance primitives and updates the interface of existing ones. Whereas BSPLIB focuses on distributed architectures, the MulticoreBSP targets shared-memory computing specifically and thus employs thread-based parallelism. As expected, MulticoreBSP provides a set of functions that are used to control the SPMD sections in BSP programs. The BSPLIB like DRMA and BSMP functions are available, as well. Furthermore, MulticoreBSP also provides the high-performance variants of the DRMA and BSMP primitives. In addition to the existing *hp*-primitives, MulticoreBSP introduced the `bsp_direct_get` primitive. Its semantics is exactly that of the `bsp_hpget`, except that it guarantees that communication is done after the call to the primitive has finished. However, the user must also ensure that the data remains unchanged during the current superstep.

Like the PUB library, MulticoreBSP supports hierarchical execution of BSP programs. This means that BSP processes may call the `bsp_init` and `bsp_begin` within a SPMD section. When a BSP process does so, it is considered as the initialising process for the upcoming nested BSP run. It is ruled by the same rules as the original BSP run. It is important to note that, after the spawn of a nested BSP run, the processes have no knowledge of the BSP processes that spawned them and previous variable registrations are no longer valid. It is also the case in BSML as there is no free variable inside the scope of parallel vectors (`<< e >>`).

The last version of MulticoreBSP improved support for nested BSP runs, which benefits code explicitly following the MULTI-BSP model. The full support of the MULTI-BSP model is the next logical step of the evolution of MulticoreBSP.

The MulticoreBSP programming model assumes a down-top execution. Such an evaluation model is less secure than a top-down approach as the synchronisations are not implicitly managed and must be handled by the programmer. The new MULTI-BSP (synchronous) primitive of MulticoreBSP are the following: `bsp_up()` is used to move upward in the MULTI-BSP tree; `bsp_down()` to move downward in the MULTI-BSP tree; `bsp_sync()` is used to explicitly synchronise. To inspect the MULTI-BSP tree information, MulticoreBSP proposes `bsp_lid()` to get the leaf identifier of this SPMD program; `bsp_leaf()` to know if the SPMD program runs on a leaf; `bsp_sleaves()` returns the number of leaf nodes in this subtree; `bsp_nleaves()` gives the total number of leaf nodes in the full tree.

To illustrate the way of programming MulticoreBSP algorithms, we propose the (pseudo-) code [[[16]]] Figure 2.13 that broadcast a value.

MigBSP

The MIGBSP framework [125] aims to control processes rescheduling in BSP applications. The main idea of MIGBSP is to adjust the process location in order to reduce the super steps times, while waiting for the slowest process. To do so, MIGBSP allows to create more BSP processes than the total amount of BSP processors. MIGBSP manages load balancing issues, where the load grain is represented by a BSP process. The model considers memory migration costs in order to evaluate the process transferring viability. The migration candidates are evaluated thanks to the non-trivial computation of a scalar metric PM (Potential Migration). The architecture is assembled with Sets: different sites. Set Managers are responsible for scheduling, capturing data from a specific Set and exchanging it among other managers. MIGBSP considers data from both processes and resources to generate the scalar metric PM. Its evaluation is available on both regular (same amount of data and same computation pattern for each BSP process) an irregular (potentially different amounts of data and computation patterns) applications. The

```
if val != 0 then
  set source = true
else
  set source = false
while not on the Multi-BSP root do
  if source and bsp_pid() != 0 then
    send val to PID 0
  move upwards in the Multi-BSP tree
while not on a leaf node do
  if bsp_pid() = 0 then
  for k = 1 to bsp_nprocs() do
    send val to PID k
  move downwards in the Multi-BSP tree
return val
```

Figure 2.13: A MulticoreBSP example.

BSML	BSP++
`'a par`	`par<T>`
`proj : 'a par -> int -> 'a`	`result_of::proj<T> proj (par<T> &)`
`put :(int -> 'a) par -> (int -> 'a) par`	`result_of:: put<function<T(int)> >`
	`put (par<function<T(int)> >& )`
`<< f $v$ >>`	`f(v)`

Table 2.3: Comparison between BSP++ and BSML primitives.

model applies adaptation on rescheduling launching in order to reduce its impact on application runtime.

BSP++

BSP++ [76] is a meta-programming library that aims to benefit from the different levels of parallelism of parallel architectures. It provides a high level interface that is simple to apprehend. To guarantee simplicity and abstraction, the BSP++ library is based on the BSML interface and uses almost identical primitives, as shown in Table 2.3 . Furthermore, it natively handles several widespread high-performance architectures. To take advantage of hierarchical architectures, and to avoid global synchronisation of a machine with a large amount of processes, BSP++ is based on a hierarchical model. Likewise *d*BSP, it is possible to decompose the machine into sub-machines. Furthermore, BSP++ exploits the localities of the machine. Thus, it is possible to benefit from shared memory available in cores. The BSP++ implementation uses OPENMP to communicate inside a shared memory and MPI for inter-nodes communications. For example, the code [75] of Figure 2.14 gives the implementation of the inner product using BSP++.

The computation of the inner product starts at line 5. As the data is distributed using a parallel vector, the inner product is computed on the local values of each processors. Then, the BSML-like `proj` primitive is used to *gather* the local results. Finally, in line 11, the inner product is finalised.

The BSP++ programming library is highly optimised and has been designed to obtain the maximum performance on very large clusters counting hundreds of cores. In [77] and [74] the

```
int bsp_main(argc, argv){
  par<vector<double> > v;
  par<double> r;
  // step 1
  *r=std::inner_product(v->begin(),v->end(), v->begin(), 0.0);
  result_of:: proj<double>::type fw;
  fw=proj(r);
  // step 2
  *r=std::accumulate(fw.begin(), fw.end(),0.0);
}
```

Figure 2.14: A BSP++ code example.

authors describe the implementation and benchmarks of some algorithms on very large clusters. However, there is no support of the MULTI-BSP model in BSP++. It is possible to *nest* parallel vectors on 2 levels, because of the implementation that uses MPI and OPENMP, but the library is not based on multi-level abilities. As there is no typing system, the nesting of parallel vectors which exceeds 2 levels leads to an error. Furthermore, there is no mechanism to ensure valid sub-synchronisations.

BSP Python

BSP PYTHON [87] is a way to program BSP algorithms in PYTHON. It is based on the idea that it is advantageous to use interpreted high-level languages in scientific programming. To obtain the maximum efficiency, the time-critical code is therefore written using an optimised compiled language. BSP PYTHON is close to the BSML language on many points. A BSP PYTHON program has two levels: local (any one process) and global (all processes). As expected, local objects (or values) are standard PYTHON objects that exist in a single process. Global objects are available by the parallel machine, as a whole. The simplest way of creating a global value is using the `ParConstant` routine which represents a constant that is available on all processors. In the same way, `ParData` can be used to create a global value that are computed on every processor.

According to the BSP model, communication takes place at the end of a superstep. To do so, BSP PYTHON proposes a set of communication patterns: (1) `put(pid_list)` sends the local value to all processors in `pid_list` and returns a global object whose local value is a list of all the values received from other processors in an unspecified order; (2) `fullExchange()` sends the local value of each processor to all other processors and returns a global object whose local value is a list of all the received values in a unspecified order; (3) `accumulate(operator,zero)` performs an accumulation where the result is a global object where the local value is the reduction of the values from all processors with lower or equal identifier.

The Figure 2.15 shows how to sort a distributed list using the PSRS [135] algorithm.

BSGP

BSGP [90] is a BSP programming language for GPU. It extends the C language with primitives for spawning threads and performing barriers. The code within barriers is automatically deduced as a superstep and is translated into a GPU kernel by the BSGP compiler. As BSGP variables are visible and shared across supersteps, data dependencies are implicitly defined. The main ambition of BSGP is, as for GPU, stream processing and especially streams of pixels for image processing. However the BSGP programming model does not directly match the GPU 's stream processing

```
def pssr( lists ):
lists.sort()
first_sample=select(nprocs,lists)
for i in xranges(nprocs):
  send.add((i,first_sample))

rcv=ParMessages(send).exchange().value
second_sample=select(nprocs,rcv)
send=intervalles(nprocs,second_sample,lists)

rcv=ParMessages(send).exchange().value
lists.empty()

for x in rcv:
  lists .add(x)
```

Figure 2.15: A BSP PYTHON example.

architecture, and the compiler must convert BSGP programs into efficient stream programs. First, the compiler automatically adds synchronisation barriers that are based on registers and thread usages. Secondly, due to the presence of shared variables across the supersteps, the compiler uses a dependency graph to automatically allocate and minimise temporary streams, used to store such values. BSGP also allows to create and destruct threads. It is also possible to manage load balancing and thread reassignment across the kernels. Remote variable access intrinsics for efficient communications between threads and collective primitives are also available.

The BSGP compiler generates a C code with CUDA calls. A current limitation is the inability to handle control flow across barriers, such as loops or if statements.

Pregel

PREGEL [110] is a powerful C++ (proprietary) parallel language dedicated to graph algorithms that are intended to be run efficiently on big clusters. A free version is available via the GIRAPH project: [[17]]. The approach centers around computations on the vertices of the graph and the computation are expressed using specific BSP primitives. One can see PREGEL as a BSP specialisation for graphs. The implementation of PREGEL relies on low-level Google libraries with checkpoint at each barrier to enable fault tolerance. Each vertex of the graph has a unique "id", an associated value and a list of outgoing weighted edges. The computation is organised using a master/slave architecture and the input data is arbitrarily partitioned and stored on a distributed storage system. The PREGEL system maintains the vertices and edges stored on the node that will perform the computation. Nevertheless, the computation is migrated to where the data are stored.

On each superstep, each worker node invokes a procedure `Compute()` for each active vertex that is under its control. This procedure is responsible for the execution of the algorithm and is allowed, among other actions, to invoke other methods, compute new values for the vertex, add or remove vertices and edges, and send messages to other vertices. These messages are exchanged directly among the vertices, even if the vertices are being executed on different machines of the platform. The messages are sent asynchronously in order to allow the overlapping of computation and communication, but are delivered to the destination vertex only on the beginning of the next superstep. If a vertex declares that all its processing was done, it sends a message informing all

```
class PageRankVertex: public Vertex<double, void, double> {
  public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
      sum += msgs->Value();
      *MutableValue() =0.15 / NumVertices() + 0.85 *sum;
    }
    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    }
    else {VoteToHalt();}
  }
}
```

Figure 2.16: A PREGEL code example.

the other nodes and deactivates itself. The master node stops the execution of the application after receiving this message from all the participants. An important feature required by graph algorithms is the ability to change the topology of the graph. For example, a clustering algorithm might replace each cluster with a single vertex, and a minimum spanning tree algorithm might keep only the tree edges. In order to avoid conflicting changes on the topology (for instance, multiple vertices could issue a command to create a vertex V with different initial values), PREGEL uses two deterministic mechanisms: (1) a partial ordering: removals are performed first, with edge removal before vertex removal and additions follow removals, with vertex addition before edge addition; (2) a handler for user-defined conflict-resolution policies. Any change on the topology is only performed on the next superstep, before the invocation of the Compute procedure.

PREGEL is considered as the main reference of graph applications on cloud and BigData processing on large clusters. It is used by Google for their own applications. Due to its proprietary nature, it is not possible to obtain the source code nor the full API. For example, a PREGEL implementation [110] of Google's page-rank algorithm can be found in Figure 2.16.

The example works as following: The graph is initialised so that in superstep 0, the value of each vertex is `1/NumVertices()`. In each of the first supersteps, each vertex sends, along each outgoing edge, its page-rank proposition divided by the number of outgoing edges. Each vertex sums up the values arriving on messages into sum and sets its own page-rank proposition. Note that, a complete page-rank algorithm would run until a convergence was found.

NestStep

NESTSTEP [99] is a parallel programming language for the BSP model. It extends the classical BSP model and supports dynamic nested parallelism with nested supersteps and introduces a hierarchical processor group concept. It also adds a virtual shared memory where the memory consistency is relaxed (but deterministic) during supersteps. It is also supports shared arrays. A prototype of NESTSTEP was designed as an extension of JAVA [99] and C [98] but it also possible to implement it over C++ or FORTRAN. The NESTSTEP-C implementation consist in a source-to-source compiler but it does not support nested supersteps yet. The NESTSTEP-

C run-time system is available through two versions: (1) Cluster-NESTSTEP-C using MPI for inter-processor communication [143]; (2) CELL-NESTSTEP-C running on the CELL BE [94](which is a heterogeneous multi-core processor). It is interesting to note that BLOCKLIB is a skeleton programming library of data-parallel skeletons working on CELL-NESTSTEP-C distributed arrays.

The NESTSTEP processes are organised in groups. The main method of a program is executed by all available processors of the partition of the parallel computer. A *step statement* represent both a step statement and a neststep statement. A step statement denotes supersteps that are executed by an entire group of processors in a BSP way. The step statements control the shared memory consistency within the current group. When a superstep is nested, we talk about neststep statements. A neststep statements splits the current group into disjoint subgroups. Each subgroup executes a statement, independently of the others, as a supersteps. The parent group has the ability to resume when all subgroups finish the execution of a neststep. NESTSTEP also offers a mechanism for load balancing the computation between groups. The main primitives of NESTSTEP are the following:

- `step` : executes a statement in parallel;
- `neststep(k)` : splits the current group (of size p) in k subgroups;
- `neststep(k,weight)` : here *weight* is a shared replicated array of k non-negative floats where the sum is equal to one. The k sub-groups created get a fraction of the available processors as close as possible as the fraction in $weight[i]$ (at least 1).
- `neststep(k,@=intexpr)` : creates k sub-groups where *intexpr* is evaluated on each processors. The processor joins the group with the id *intexpr* (if it ranges $[0...k-1]$, it skips the statement otherwise);
- `seq` : the group leader only is concerned by this execution. No synchronisation is implied;

The NESTSTEP syntax also introduces several symbols to manage parallelism: `#` is the number of processors in the current group; `$` is the processor rank in the current group. `@` is the id of the group the processor is part of.

To illustrate a NESTSTEP computation, the code [99] in Figure 2.17 describes a way to compute a parallel prefix sum using the NESTSTEP-C implementation:

NESTSTEP gives the possibility to nest BSP computation in a hierarchical way. Nevertheless, this hierarchy is not directly related to the architecture of a given machine. Such a limitation does not allow to fully take advantage of the hierarchical characteristics of a machine.

GRID-NESTSTEP [73] proposes to adapt the BSP model with the NESTSTEP language to enable grid computing. To do so, a precompiler translates the NESTSTEP constructs in order to explicitly decompose the program into supersteps. Each supersteps is a *virtual* BSP computation that matches the grid environment. As supersteps are synchronised, all grid processors work on the same supersteps. Following the hierarchical structure of grids, the computation work of a superstep is split into *workpackages* of suitable sizes. The workload is managed by a scheduler which manages the grid environment. An implementation of this extension is available via the GRID-MODELICA project [59].

The GRID-NESTSTEP approach is closer to the architecture and is adapted to HPC computing. Nevertheless, as the load-balancing is managed by a scheduler, it is difficult to determine a cost for such programs. It is also difficult to design an algorithm based on the cost analysis, as those costs are *hidden* by the run-time implementation.

```
void parprefix( sh int a[]</> )
{ int *pre;      // local prefix array
  int Ndp = N/p; // assuming p divides N for simplicity
  int myoffset;  // prefix offset for this processor
  sh int sum;    // automatically initialized to 0
  int i, j = 0;
  step {
    pre = new_Array( Ndp );
    sum = 0;
    forall ( i, a ) { // locally counts upward
      pre[j++] = sum;
      sum += a[i];
    }
  } combine( sum<+:myoffset> );
  j = 0;
  step
  forall ( i, a )
  a[i] = pre[j++] + myoffset;
}
```

Figure 2.17: A NESTSTEP example.

2.2.3 Other functional parallel languages

There exist a lot of parallel languages or parallel extensions of sequential languages. In this section, we try to point out functional parallel languages that were the most important in our view. Notice that there is a lack of comparisons between parallel languages. As efficiency, scalability, expressiveness, *etc.* must be taken into account, it is rather hard to compare them.

Two nice introductions (with many references) to parallel functional programming can be found in [79] and [81]. They have been used as a basis for the following classification. The authors explain the three main reasons to use functional languages in parallel programming. First, they ease the partitioning of parallel programs (task or data decomposition). Second, most of them are deadlock free. As a value is computed independently of the evaluation order, a sequential run will deliver the same value when run in parallel. Third, they have a straightforward semantics that is similar in both sequential and parallel evaluation. This is very useful for testing and debugging.

Data-parallel Languages

To our knowledge, the first important *data-parallel* functional language was NESL [10]. This language allows to create specific arrays and nested computations within these arrays. The distribution of both the data and the computations over the available computing units is done at compile time. The code [[[18]]] in Figure 2.18 shows how the nesting of parallelism is expressed with NESL, on the quick sort algorithm.

The quick sort algorithm written in NESL is really easy to understand. Assuming that `#a` returns the size of the list `a`, the algorithm proceeds has follows. First it returns `a` if the list consists of one element. Otherwise, it determines the `pivot`, builds the list of elements `lesser`, `equal` and `greater` that contain the elements related to their names, compared to the pivot. Then it calls, in parallel, the algorithm on the lesser an greater elements. Finally, the result

```
function quicksort(a) =
if (#a < 2) then a
else
  let pivot   = a[#a/2];
      lesser  = {e in a| e < pivot};
      equal   = {e in a| e == pivot};
      greater = {e in a| e > pivot};
      result  = {quicksort(v): v in [lesser,greater]};
  in result[0] ++ equal ++ result[1];
```

Figure 2.18: A NESL code example.

consist in the concatenation of sorted element that where computed in parallel. This example shows that the nested parallelism in NESL is not controlled as it is in MULTI-BSP. The parallel execution is not related to a specific memory or computing unit, everything is abstracted by the syntax.

Two extensions of NESL for ML programming are NEPAL [24] and MANTICORE [51]; the latter is clearly a mix between NESL and Concurrent ML [50]. Concurrent ML is a language of creation of asynchronous threads and send/receive messages in ML.

SAC (Single Assignment C) [71] is a lazy functional language (with a syntax close to C) for array processing. The language provides high-order operations on multi-dimensional arrays and the compiler is responsible for the generation of efficient parallel code.

An extension of the famous lazy functional language HASKELL is Data Parallel HASKELL [25]. It allows to create data arrays that are distributed across the processors. And some specific operations permit to manipulate them.

The main drawback of these languages is that cost analysis (for comparing algorithms) is hard to do since the system is responsible for the data distribution.

Data-flow and Coordination languages

Coordination languages [64] (and data-flow languages, which are similar to them) add primitives to a sequential language in order to describe how to coordinate processes over a parallel architecture. To do so, they mainly generate a data flow graph. CALIBAN [97] is on of the first language that allows declarative description of a processes network and the mapping of processes to processors. The compiler determines statically code for all processes and their interconnections. CALIBAN is an extension of a purely functional languages.

One of the most popular coordination languages is SISAL (Streams and Iteration in a Single Assignment Language) [49]. It is close to most statement-driven languages, but variables should be assigned once. This allows the compiler to identify easily the inputs and outputs. The two main approaches are LINDA [22] and PCN (Program Composition Notation) [57], which have been implemented in many programming languages. These languages are inserted as notations in sequential languages, such as C,C++, LISP or JAVA, to generate and manage parallelism. A complex runtime system needs to adequately synchronise concurrent memory access of the processes.

Explicit process creation

In this category we have two extensions of HASKELL: EDEN [108] and GPH (Glasgow Parallel HASKELL) [78, 107]. They both extend HASKELL by adding a small set of syntactic constructs for explicit process creation. Those extensions allow fine-grain parallelism while providing enough

control to implement efficient parallel algorithms. The usage of communications based on lazy shared data also frees the programmer from the tedious task of managing low-level communication details. Threads are automatically managed by a sophisticated runtime system for both shared or distributed memories. It is interesting to note that algorithmic skeletons are also proposed in both languages [7]. As explained previously, cost analysis of programs written with those languages is hard. Indeed, as the runtime handles data distribution [132], it may introduce too much communication and, thus, a lack of scalability.

The HUME language [80] is another distributed language that proposes a cost analysis for real-time purpose programs. Nevertheless, allow such an analysis the language expressiveness is arbitrarily limited.

Distributed functional languages

In addition to parallel functional languages, there are many concurrent extensions of functional languages such as ERLANG, CONCURRENT CLEAN or JOCAML [111]. JOCAML is a concurrent extension of CAML based on the join-calculus. It adds explicit generation of parallel processes and distribution directives across processes using "resources". Processes synchronisation is done using specific patterns called join-patterns.

The communication with *futures* is proposed in ALICE ML [127]. A future is a placeholder for an undetermined result of a concurrent computation. When the computation delivers a result, the associated future is eliminated by globally replacing it by the result value. The concept of *promises* that explicitly handle futures is also available.

MULTIMLTON [138] is a multi-core aware runtime for standard ML, which is an extension of the MLTON compiler. It effectively manages composable and asynchronous events using, in particular, *safe-futures* [157]. It provides an analysis to automatically claim the resulting value of a future `F` by avoiding manual calls to the `touch(F)` operation, which blocks until the execution of `F` is complete.

SCALA is a functional extension of JAVA which provides concurrency, using the actor model. OZ-MOZART [141] is a multi-paradigm language (originally functional) with explicit creation of threads and communications are performed using "ports". It also provides data-flow computation over lazy lists, but the generated code is not very efficient.

Such languages are able to simulate or implement the MULTI-ML language. Nevertheless, they are not adapted to HPC computations.

3

Multi-ML in a Nutshell

The MULTI-ML language is based on the idea of executing a BSML-like code on every stage of the MULTI-BSP architecture, that is on every sub-machine. It is therefore possible to program MULTI-BSP algorithms while maintaining the BSML programming convenience. MULTI-ML consist in a lightweight specific syntax added to ML allowing to express multi-level parallelism. In this chapter, we talk about the philosophy of the MULTI-ML language and its syntax. Using several examples, we give some intuitions of its semantics, which will be developed in the next chapters.

This chapter is dedicated to a language that is an extension of OCAML. Thus, we will use some OCAML features such as pairs, operation on lists, labels, *etc.* The reader unfamiliar with OCAML can find the required information in [[19]] and [[20]].

3.1 The Multi-ML concepts

3.1.1 What Multi-ML is not

MULTI-ML aims to program HPC architectures, that is to say clusters of multi-cores with, potentially, both shared and distributed memories. This language does not intend to be generalist, with multi-paradigm programming features. MULTI-ML has been designed to program specific architecture that are hierarchically organised and it is intrinsically dedicated to it. The programming model does not assume that the code must be written for a specific architecture, it is rather disconnected from the runtime support. As MULTI-ML proposes a high level of abstraction, it is not possible to spawn or manage threads manually. All the parallelism is controlled by the language syntax, which is directly related to the MULTI-BSP model.

3.1.2 The execution levels

As MULTI-ML deals with MULTI-BSP architectures, we can distinguish several levels of execution. An expression is thus evaluated at a specific stage of the MULTI-BSP architecture, depending on the level of parallelism expressed. Those levels are useful to identify where the computations take place, but also where a data is stored. In MULTI-ML, an expression is always annotated with a locality representing where the data belongs to. With this locality information, the programmer can easily know if a value can be communicated to another level or if it is already available in another memory. This notion of locality is embedded in the MULTI-ML

type system to guarantee safe communications through the MULTI-BSP levels (see Chapter 4 for more details) and optimised memory managements. The memory localities of MULTI-ML are the following:

- `m`: The *multi*, or global, memory of the multi-level architecture. At this level, all the nodes and leaves of the MULTI-BSP architecture are concerned by any execution. A value defined at level `m` is accessible "anywhere" on the machine. Some restrictions concerning parallel vectors will be discussed later on.

- `b`: The BSP memory of a node. At this level, a BSML code can be executed in order to manage communications and parallel computations. It is possible to *point toward* the underlying memory via the parallel vectors. A value defined at level `b` is only accessible in the concerned node and can be communicated with the respect of certain limitations.

- `l`: The *local* memory of a sub-node (inside a parallel vector). This is the underlying memory that can be managed inside a parallel vector from the upper node . This memory can contain *pointers* to the upper memory via a specific syntax explained below. It is important to note that the expressions of this level cannot be communicated because they are specific to a node and may have no sense anywhere else.

- `c`: Like `l`, this memory is managed inside parallel vectors but it is free of references to the upper level. It is possible to transfer such *communicable* values through the MULTI-BSP levels.

- `s`: The *sequential* memory of a leaf. It can contain specific values such as non-deterministic expressions. At this level, there is only sequential (traditional ML) code and the execution takes place in the lower level of the memory. A value defined at level `s` can be communicated under certain constraints.

It is important to note that the evaluation of a function that is specific to the locality `s`, such as random functions, is not possible inside a parallel vector. Like in BSML, the MULTI-ML implementation could be distributed. In this case, the values generated by a non-deterministic function on different components of the architecture may differ and then break the replicated coherency. Such a behaviour could introduce deadlocks, something we want to avoid. Of course, one can imagine a random operator that generates a random value that is coherent in a distributed implementation. An in-depth discussion on this topic is available in Chapter 5.

Figure 3.1 shows how the different localities are mapped on the MULTI-BSP memory hierarchy. Such an architecture can correspond to multi-core with two physical threads per core. Inside the nodes (at BSP level), we can see the notation `<< >>`. It denotes the scope of a parallel vector. As explained previously, this section represent the underlying memories of a node, which are divided in two categories: `c` and `l` (respectively *communicable* and purely *local*).

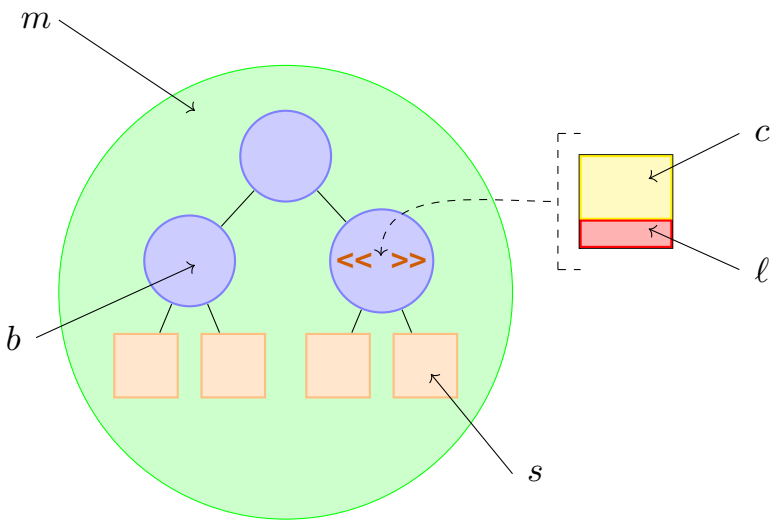


Figure 3.1: The MULTI-ML levels.

3.1.3 A short introduction to the syntax of Multi-ML types

We saw that the MULTI-ML language manipulates expressions that belongs to a specific locality. This locality information is *attached* to every expression of a MULTI-ML program. To keep track of those locality informations, we choose to extend the regular OCAML type system by adding locality annotations to types. Thus, the MULTI-ML types are annotated or tagged by their definition localities, representing the memory level where they belongs to.

This section is an overview of the MULTI-ML types. The typing system is widely discussed and detailed in Chapter 4.

For example, the execution of `let x = 1 in x` at the MULTI-BSP level returns a value of type `int_m`. As the MULTI-BSP level of execution deals with the whole architecture, the value `x` is computed and available in all the memories. Then, it is annotated with the locality `m`.

This type annotation is also attached to various type constructions such as pairs: `(int_m * string_m)_m` meaning that the pair and its elements are defined at the *multi* level.

Function types are also annotated by their level of definition. Moreover, as functions can hide parallelism inside their definitions, we must add an extra annotation called a latent effect. A latent effect [145] is a locality annotation that gives information on the nature[1] of the code that will be evaluated during the function application. This information is particularly useful to avoid malformed expressions, such as manipulating a parallel vector at the MULTI-BSP level.

For example, the evaluation of the code `let f = fun () -> << 42 >>` at the *multi* level will produce an expression typed `(unit_`z -(b)-> int_c par_b)_m with [`z <| b]`. The function `f` is thus defined at the MULTI-BSP level, it takes a value of type `unit_`z` (where `` `z `` refers to a polymorphic locality) and returns a parallel vector `int_c par_b`. The annotated arrow `-(b)->` stands for the latent effect of the function, that is to say the locality where the function must be evaluated. In this example, as a parallel vector is build inside the function, its locality must refer to the BSP level. Finally, the `with [`z <| b]` element concerns the remaining constraints that must be respected. Here, we specify that the locality `` `z `` (of the first argument) must be valid (or *acceptable*) in locality `b`. The constraint operator $\lhd$ checks that a locality makes sense in another locality. This notion of *acceptability* is widely discussed in Chapter 4.

[1] In this context, it stands for the level of parallelism.

Note that the MULTI-ML type system allows the definition of BSP codes at the *multi* level. This is particularly interesting for the design and use of BSML libraries.

Localities are not always fixed. For example, the identity function cannot be typed with a fixed locality. Indeed, the code `let id = fun x -> x in id << id 1 >>` is perfectly valid. The application of the function `id` is valid regardless of its locality. Therefore, we introduce a notion of locality polymorphism using the ` (backquote) notation, analogue to the ' (quote) notation for type polymorphism in OCAML. The identity function is thus typed: `id : 'a_`z -(`z)-> 'a_`z`. Note that the latent effect of the arrow is `` `z ``, as both input and output localities. Indeed, the locality of the values manipulated by a function must appear in its latent effect. For example, the application of the identity function on a parallel vector must have the latent effect `b` as it manipulates parallel data.

To ease the understanding of types, a convention is used for the polymorphic types and localities: they are alphabetically, respectively reversed alphabetically, enumerated from a set of letters deprived of `m`, `b`, `c`, `l` and `s`.

3.2 The Multi-ML features

In the following section, we choose to illustrate the MULTI-ML language using code examples. To ease the *pen and paper execution* of codes, we define a small MULTI-BSP architecture that will be recurrent through all the examples. This architecture is composed by 3 levels and every inner node has 2 sub-nodes. To identify each component, we use a regular tree based numbering. Therefore, this 3-leveled ($d = 2$) architecture is composed by one root node (0), two nodes (0.0 and 0.1) and four leaves (0.0.0, 0.0.1, 0.1.0 and 0.1.1). Figure 3.2 shows a representation of this small MULTI-BSP architecture where, as before, circles and squares respectively stand for nodes and leaves.

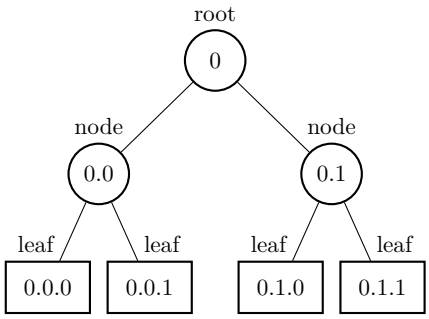


Figure 3.2: The small architecture used as example.

3.2.1 The multi-functions definitions

The main functionality added by MULTI-ML, compared to BSML, is the ability to define multi-functions. A multi-function is a particular kind of recursive function that is adapted to the recursion through a MULTI-BSP tree. This kind of function is used to execute some codes all over the architecture thanks to a simple mechanism based on both parallel vectors and recursive calls. The multi-function is identified by the keyword `multi` and the body declaration of this function is split in two specific parts: the node code and the leaf code. The declaration structure of a multi-function is the following:

```
1 let multi mf () =
2  where node = (*node code*)
3   "hello node"
4  where leaf =
5   "hello leaf" (*leaf code*)
```

As expected, the node code on lines $2-3$ is going to be executed on the nodes[2]. Within this section, the programmer must write a BSML code to manage parallel computations and communications. The leaf code on lines $4-5$ is the sequential code executed on the leaves. The type of such a multi-function is `(unit_`z -(m)-> string_m)_m with [`z <|m]`. The multi-function `mf` takes an `unit_`z` as argument and returns a `string` of location `m`. The annotated arrow `-(m)->` implies that the evaluation of the multi-function must be done at the MULTI-BSP level. It is important to note that the definition of a multi-function is possible at the MULTI-BSP level only. As the multi-function body code must be known by the whole architecture, a definition during a recursion through the MULTI-BSP architecture is not conceivable. It must be defined in *all* memories (depending on the implementation: see Chapter 5 for details).

One can notice that there is a slight difference between the MULTI-BSP model and the MULTI-ML implementation. Indeed, MULTI-BSP describes an execution model where the computations take place on the leaves only and nodes are memories, without computing capabilities. In MULTI-ML, it is possible to express computation on nodes to manage parallel computations. However, the underlying implementation can propose an execution in a MULTI-BSP fashion, where node computations are simulated by leaves. To guarantee the efficiency of a MULTI-ML program, it is important to concentrate the intensive computing tasks to leaves and favour lightweight management code for nodes.

3.2.2 The recursive multi-functions calls

Multi-functions are very powerful to recurse through a tree and are easy to handle. In MULTI-ML, the code that is executed inside a parallel vector of a node is actually evaluated on the sub-nodes. Based on this simple idea, the evaluation of a multi-function inside a parallel vector will launch a multi-function recursion on the sub-nodes. That is to say: *recurse through the* MULTI-BSP *architecture.*

```
1 let multi mf () =
2  where node =
3   let rec_call = << mf () >> in
4   "hello node"
5  where leaf =
6   "hello leaf"
```

When a multi-function is applied to a value at the MULTI-BSP level, the evaluation is initiated on the root node of the architecture. In this example, the multi-function `mf` is called inside a parallel vector (line 3) to initiate the recursion on the sub-nodes. As the evaluation of a multi-function starts from the root node, the recursion will spread from the root toward the leaves. The Figure 3.3 shows how the multi-function is distributed on the sub-nodes using the parallel vector construction.

[2]Note that this code does not *actually* lead to a multi-level execution. As explained in the next section, we must recursively call a multi-function to do so. In practice, this code is just executed on the node code and then, returns the final value.

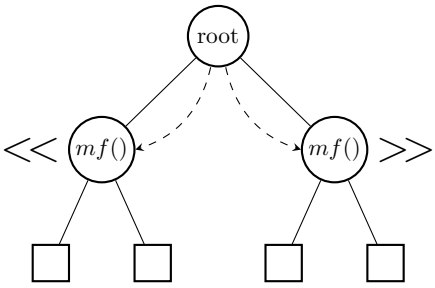


Figure 3.3: The multi-function recursion.

Thanks to a typing mechanism that will be detailed in Chapter 4, the parallel vector `rec_call` is consequently typed `string_c par_b` (as the string is fully communicable and the parallel vector is restricted to the level `b`). The Figure 3.4 shows how the values transit through the different levels of the MULTI-BSP architecture. It is important to note that the values *returned* by both the node and the leaf code (symbolised by arrows in Figure 3.4) are not copied in the upper memory, they are just available through the parallel vector. It is possible to manipulate the parallel vector `rec_call` as a regular one, using the BSML primitives[3]. For example, the BSML primitive `proj` can be used to extract, to the BSP level of a node, the values of this parallel vector.

When the evaluation is done, the final value computed by the root node is given as the *result* of the multi-function. This value is pushed to the MULTI-BSP level. It is then available for every unit of the architecture. Concerning our example, the multi-function recursion will transit from the root node to the leaves, and then, up from the leaves to the root node. The string `"hello node"` is therefore the result of the multi-function application.

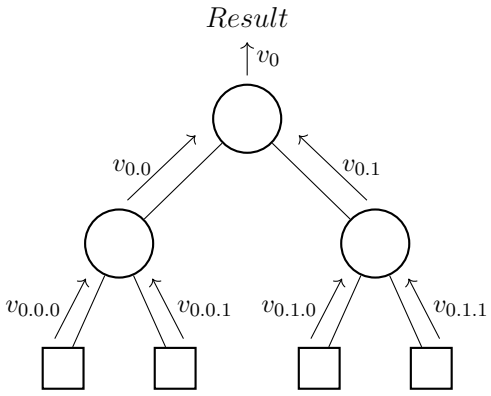


Figure 3.4: The multi-function evaluation.

The recursive multi-function calls are only accepted within the scope of a parallel vector. Calling a multi-function at the BSP level would introduce a behaviour that is difficult to handle in both the typing system and the implementation. The main difficulty concerns the introduction of tree structures at BSP level dues to multitree-functions (see Section 3.2.4).

3.2.3 The tree structure

As MULTI-ML deals with MULTI-BSP architectures, we choose to create a data type that represents values that are distributed all over the architecture. This particular type of data is called a *tree*. As the values of a tree are available in all the memories, they can be easily referenced by

[3]In this example, the `rec_call` value is ignored.

a single identifier. Such a data structure allows to keep previously computed values in the concerned memories, to avoid memory transfers. The easiest way to create a *tree* is using the `mktree` primitive. This primitive takes an expression e and evaluates it on every level of the architecture.

```
let t = mktree "hello tree"
```

This code builds a tree `t` that contains the string `"hello tree"` at each level of the MULTI-BSP architecture. In accordance to the MULTI-ML type system, the value `t` is typed `string_c tree_m`. The type `string_c` means that the string contained in the tree has no restrictions concerning its communication. As expected, the tree is annotated with the locality `m`, as its definition takes place at the MULTI-BSP level. One can say that the *tree* concept is similar to the parallel vectors of BSML, in a MULTI-BSP way.

3.2.4 The multitree-functions

Unlike the multi-function that only generates a communicable sequential value at the MULTI-BSP level, a multitree-function aims to build a tree. However, the multitree-functions are much more powerful than the `mktree` primitive in the sense that it is possible to execute a code according to the execution level to generate a specific value.

Let us have a look to the following example:

```
1 let multi tree mtf () =
2  where node =
3   let rec_call = << mtf () >> in
4   finally rec_call "node"
5  where leaf =
6   "leaf"
```

The multitree-function `mtf` build a tree that contains the value `"node"` on every node of the MULTI-BSP architecture (lines $2-4$) and `"leaf"` on every leaf (lines $5-6$). This simple example shows that, in multitree-functions, the node code must return a value constructed by the operator named `finally`. This operator is a constructor of a value of type `Final` and is typed `finally : ~sub:'a_c tree_l par_b -(b)-> ~head:'a_`z -(b)-> Final ('a_c tree_l par_b, 'a_`z)_m`.

This constructor highlights the fact that particular operations are going to be executed during a tree construction. As expected, the first argument, `sub`, is a parallel vector of trees. It represents the sub-trees or branches that we want to use in order to construct the resulting tree. Indeed, as the type of the multitree-function `mtf` is `(unit_`a -(m)-> int_c tree_m)_m with [`a<|m]`, the result of the recursive call of the multitree-function inside a vector is typed `int_c tree_l par_b`. It is important to note that the location of the tree inside the vector is `l` because we do not want to communicate such a value. Thus, a tree can be annotated by the localities `m` or `l`. The value `rec_call` contains the sub-trees or branches that have been constructed on the sub-levels. The second argument, `head`, is the value that is kept, at the current node. It is the top node of all the sub-branches. Figure 3.5 shows how the construction is done, by successively combining all the branches of the levels. Of course, the trees are not copied between the levels, they are only accessible through the parallel vector structure.

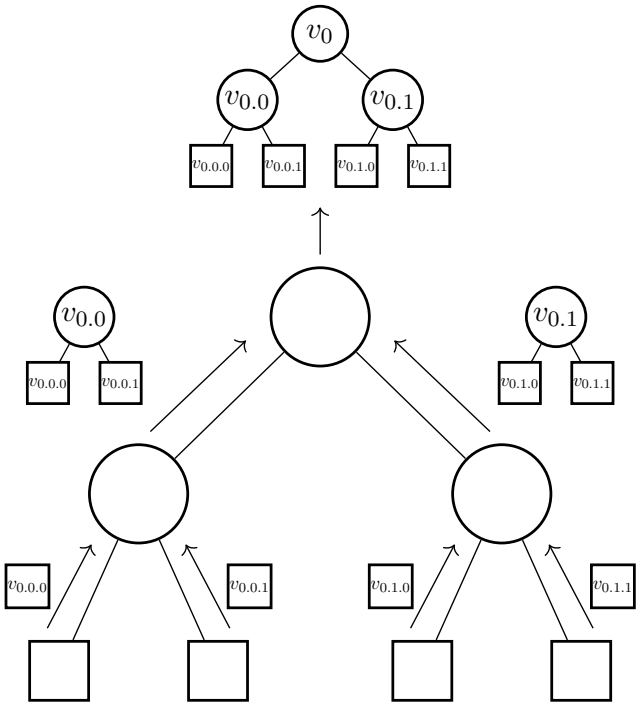


Figure 3.5: Tree construction.

With the combination of multitree-functions and tree recursion it is possible to take advantage of the MULTI-BSP architecture with a small syntax overhead.

3.2.5 Tree manipulations

We saw that both multitree-functions and the `mktree` primitive are useful to build a tree. Such a parallel data structure can be accessed using a particular operator in order to use, one by one, the values distributed at every level of the MULTI-BSP architecture. We propose the primitive `at` to perform this operation. `at` can be executed at different levels and its locality of execution determines its behaviour. Currently, we have one `at` operator for each level: `at_b`, `at_c`, `at_l` and `at_s`. This operator gives an access to its own memory, that is to say, values of trees that are relative to the current level. It is not possible to read the memory of an other level without explicit communication. Furthermore, there is no `at` operator defined for level `m` as it does not make sense to access to the value of a tree at this level. Figure 3.6 shows that the application of `at_b` on a tree `t` at level 0.1 returns the value contained at level 0.1 in `t`: $t_{0.1}$.

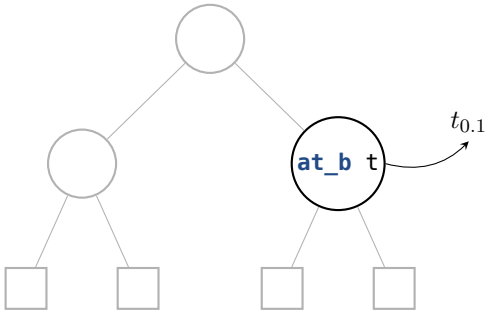


Figure 3.6: The `at` operator.

To illustrate the `at` primitive, we consider the following example:

```
1  let sum_all t =
2    let multi sum_t () =
3     where node =
4      let rec_call = << sum_t () >> in
5      let p = proj rec_call in
6      let res =
7       List.fold_left (fun x y -> (p x) + y ) 0 procs
8      in res + (at_b t)
9     where leaf =
10     at_s t
11 in sum_t ()
```

In this code, the multi-function `sum_t` computes the sum of all the elements contained in a tree given as argument. Concerning the node code, at line 4 the parallel vector `rec_call` contains all the sub-sums computed during the recursion. Using the `proj` primitive (line 5) in combination of a list operator (lines $6-7$), we can compute the sum of the elements (where `procs` is a list containing the processors ids $[0, ..., \mathbf{p}-1]$). Finally, the result of the node code is the sub-sum of the sub-nodes plus the value contained in the tree at this level (line 8). The leaf code is pretty simple, we just need to return the value of the given tree (line 10).

It is interesting to note that the `at` primitive can also be applied on a tree contained in a parallel vector. For example, if `v : int_c tree_l par_b`, `<< at_l $v$ >>` is valid. In this case, the resulting value is the local value of `v` that is stored on the sub-node. As this value is in the sub-memory, it cannot be manipulated outside the parallel vector without explicit communication.

Concerning tree composition, the MULTI-ML languages impose several restrictions. For example, let A be a built tree. During the construction of a tree B, one can imagine taking a whole branch of A to build B. This is not allowed as the tree manipulations are highly restricted. Nevertheless, building B with the values of A using the `at` operator is possible. To guarantee safe executions, the programmer is not allowed to *manually* nest trees.

3.2.6 A global identifier

In BSML, the keyword `this` (also known as `pid`) is used to get the identifier of a processor, inside a parallel vector. It is basically a built-in parallel vector containing the values $0, ..., \mathbf{p}-1$. As MULTI-ML deals with trees, we propose a new type of identifier adapted to this structure. The `gid` (***G**lobal **Id**entifier*) is a built-in tree that contains those identifiers. As shows in Figure 3.7, the positions are encoded by a top-down path of positions. For example, 0 stands for the root node, 0.0 and 0.1 for, respectively, its first and second child, *etc.* For the ith component of a vector at node n, the identifier is $n.i$.

3.3 Communications

3.3.1 Vertical communications

In BSML, and therefore in MULTI-ML, we can express horizontal communication using the `put` primitive. In both BSML and MULTI-ML, vertical communications can be performed using the `proj` primitive. In this case, the communication is upwards: from the leaves to the node.

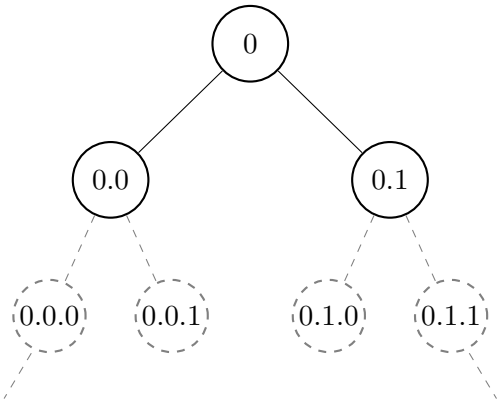


Figure 3.7: The `gid` tree.

In MULTI-ML we propose a notation to explicitly communicate data downwards: from the node to the leaves, directly inside the scope of a parallel vector. As the BSML implementation is replicated on every process of the architecture, a value used inside a parallel vector is implicitly available in every memory. Nevertheless, in MULTI-ML, the memories are all distinct. Thus, a value defined at an arbitrary level is not available in the sub-nodes memories: we need to explicitly distribute it in order to make it available in the sub-memories. By annotating a value with `#`, we explicitly communicate a value to the sub-memory, in order to be able to use it inside the scope of a parallel vector.

For example, the following code is valid:

```
1 ...
2 where node =
3  let x = 42 in
4  let v = << #x# >> in
5 ...
```

We explicitly specify that the value `x`, defined in the node memory line 3, must be communicated downwards using the notation `#x#` (line 4). Therefore, `x` is available inside the parallel vector. As the code is known by all the processes of the architecture, one can say that the value of `x` is implicitly known at runtime. Nevertheless, the following code is not valid:

```
1 ...
2 where node =
3  let (cond : bool) = ... in
4  let x = if cond then 42 else 43 in
5  let v = << x >> in
6 ...
```

Indeed, the evaluation of `x` on the sub-nodes will lead to an *unbound value* because `x` is only bound in the current scope: the node environment. Furthermore, as the boolean `cond` (line 3) is only known in the node memory, the sub-node cannot know the actual value of `x`. Thus, we cannot know the value of `x` in the sub-nodes. Such a code justifies the usage of this annotation to get reliable communications.

3.3.2 Communication friendly vector construction

As explained previously, it is mandatory to communicate data to the sub-nodes to use it inside parallel vectors. One can imagine that splitting a list through the sub-nodes is inef-

ficient if we need to copy the whole list everywhere and, then, split it. We assume that, omitting the locality annotation that is not significant in this example, we have a function `split : int -> int -> int list -> int list` that takes an integer n, an integer i, a list of integers and returns the ith list of the n split lists. When we want to split the list `lst` using the vector notation, we could write the following code:

```
...
where node =
 let (lst : int list) in
 let n_lst = << split #n# #lst# >> in
 ...
```

Thanks to the `#` operator, this code is perfectly valid and its execution leads to a valid list split. However, it implies an unnecessary amount of communication: the whole list is, first, distributed (because of the `#` operator) and, then, split. To avoid superfluous communications, we choose to redefine the `mkpar` primitive that exists in BSML. Its aim remains the same: distribute values to the sub-nodes, but its execution is different. By omitting the type annotations, the OCAML type of `mkpar` is: `(int -> 'a) -> 'a par`. Its usage is similar to the BSML one. However, the function provided to `mkpar` to build the parallel vector is going to be executed sequentially on the node. And then, the results will be distributed on the leaves. Figure 3.8 shows the execution of the code `mkpar (fun i -> f i)`. The first step is the sequential computation: the evaluation of `f 0` and `f 1` produces respectively `v0` and `v1`. The second step is the communication of `v0` and `v1` to the sub-nodes.

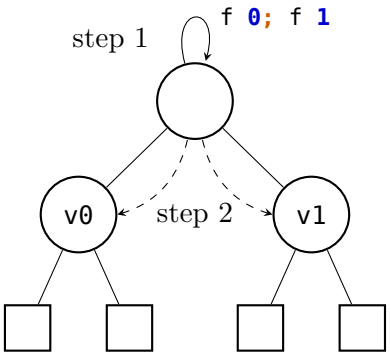


Figure 3.8: `mkpar` execution steps.

The evaluation of the `mkpar` primitive is computationally more intensive for the node, compared to the regular vector construction, but it is more efficient in terms of communication. For example, using the `mkpar` primitive and the previously defined `split` function, we can write the following code:

```
...
where node =
 ...
 let i_lst = mkpar (fun i -> split n i lst) in
 ...
```

In this example, the list `lst` is split on the node and only the required sub-lists are communicated. At level j, the estimated cost of `mkpar (fun i -> split n i lst)` is: $\sum_{k=0}^{\mathbf{p_{j-1}}} (w_k) + n \times g_j + L_j$;

Where $\mathbf{p_{j-1}}$ is the number of processes of the sub-node of j, n is the size of list `lst`, g_j is the communication bandwidth, L_j the synchronisation cost and w_{j-1} the work done by level $j-1$. Otherwise, the cost of `<< split #n# #lst# >>` can be estimated with the following formula: $\sum_{k=0}^{\mathbf{p_{j-1}}}(n) \times g_j + L_j + w_{j-1}$.

Compared to the code without `mkpar` that communicates $\mathbf{p_{j-1}}$ times the whole list, here, the whole list is communicated only once. If the computations implied by the evaluation are not intensive and the amount of data is important, the usage of `mkpar` is more efficient. Such a behaviour can be dynamically chosen during the execution of an algorithm. For example, given the MULTI-BSP parameters and the data size, one can choose to favour the usage of `mkpar` when the amount of data is important and the bandwidth is low: at the top of the architecture; Otherwise, when the amount of data is limited by the memory size and the bandwidth is high: at the bottom of the architecture (next to the cache memories), it seems legitimate to use the `<< >>` primitive.

3.4 A complete example

In this section, we are going to analyse a small code example that uses most of the MULTI-ML features presented above. Let us say that we have a big list available in a file and we want to sum all its elements and keep the sub-sums at all levels of the MULTI-BSP architecture. As this list is huge, it can only fit in the memory of the root node and it must be split to fit the sub-memories. To simplify the algorithm, we assume that when the list is split in a node in $\mathbf{p}$ sub-lists, the element will fit the sub-memories. This assumption avoids to check the memory size of the sub-nodes and iterates through a set of well calibrated lists. To simplify the illustration of the example, we chose the list `[0;1;2;3;4;5;6;7]` as input value. We also have, at our disposal, the `split: int -> int -> int list -> int list` function that, given a number of slices, an element `i` and a list, returns the i^{th} slice of the list; `nprocs` is the number of sub-processes; a `flatten: 'a par -> 'a list` function that simply transforms a parallel vector into a sequential list; and a `sum_seq: int list -> int` function that computes the sum of all the elements of a list, sequentially.

```
1  (*List summing with intermediate sums *)
2  let multi tree sum_list lst =
3    where node =
4      let l =
5        if (at_b gid) = 0 then
6          list_of_file "./big_list.data"
7        else
8          lst
9      in
10     let v = mkpar (fun i -> split nprocs i l) in
11     let rc = << sum_list $v$ >> in
12     let s = sum_seq (flatten << at_l $rc$ >>) in
13     finally rc s
14   where leaf =
15     sum_seq l
```

The type of the `sum_list` multitree-function is `sum_list: (int_`z list_`y -(m)-> int_c tree_m)_m`. As expected, this function takes a list of integers that does not require a specific locality. It

returns a tree of integers that are communicable. The latent effect `m` annotates the arrow in order to indicate the MULTI-BSP nature of the value.

The first step of this multi-function node code, lines $5-6$, is to read the list from the file if the current level is the root (`gid` is 0). In the other cases we take, as input, the list given as argument: line 8. Note that the `list_of_file` function is defined in accordance to the node memory limits. The next important step is the distribution of the list, line 10. One can note that we choose to use the `mkpar` primitive to avoid useless communications. Therefore, `v` is a parallel vector that contains all the slices of the given list `l`. Now, we need to initiate the recursion of the multitree-function on the sub-nodes by calling `sum_list` on the vector `v`, line 11. Just before the execution of the leaf code, the history of the distribution of the list through the memories is illustrated in Figure 3.9. Afterwards, we are going to compute the sum of all the sub-sums that are available thanks to the recursive call of the multi-function, line 12. They are available on the tree `rc` and the sub-sums of the sub-nodes can be accessed using `at_l`. `<< at_l $rc$ >>` is therefore a vector containing the sub-sums of the sub-nodes. We just need to `flatten` it and sum it to get the sum of the sub-sums. Finally, line 13, we returns the vector of branches (to construct the tree of sub-sums) with the value `s`, freshly computed, to conclude the node computation. Concerning the leaf code (line 15), it is pretty simple: we just need to compute the sequential sum of the given list and return it as a result.

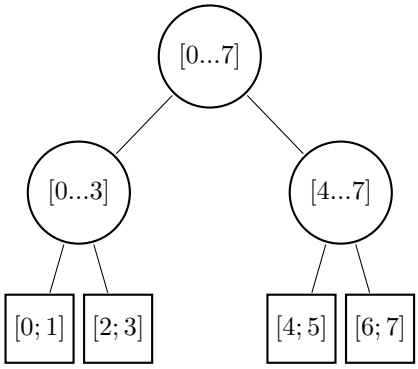


Figure 3.9: End of the distribution.

The result of the execution of the multi-function `sum_list` on the list `[0;1;2;3;4;5;6;7]` is thus, a tree containing the sum of all the elements on the root node and the sub-sums of the sub-lists on the other nodes. The Figure 3.10 shows the values of the resulting tree.

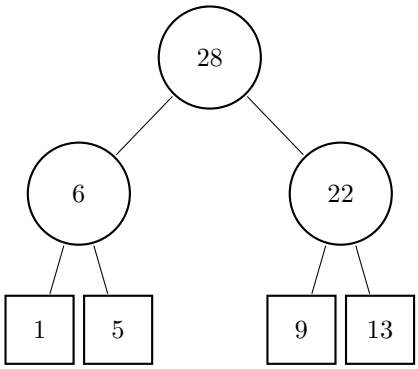


Figure 3.10: Resulting tree.

4

Type system

As it is said in [116]: *"Well-typed programs cannot go wrong"*. The typing system that we will define for MULTI-ML ensures that a program is well-formed. In this context of MULTI-BSP programming we add a degree of complexity, compared to the OCAML type system. We now need to handle new problems such as memory coherency, deadlocks, safe data communications, *etc.*

To ease the understanding of the MULTI-ML typing system, we choose to split it in several parts that are highly related. After introducing the main problematic issues of MULTI-BSP program safety in general and how it is related in BSML and MULTI-ML, we present first, a semantics and then, a type system for a core version of MULTI-ML. Then, we introduce the management of references and exceptions. The last section of this chapter is dedicated to the type inference system, which is an algorithm that finds the (most general) type for a given expression. The proofs of lemmas are available in the Annex 6.2.6 at the end of the chapter.

4.1 Safety in parallel and distributed programming

In this section we are going to talk about the pathological cases raised by the parallel capabilities of the MULTI-ML language.

4.1.1 Replicated coherency

In MULTI-ML, values can be defined at various *levels*. In addition to designating a component of a given depth, a *level* adds a notion of parallelism. A *level* is associated to a *locality* which stands for the *quantity of parallelism* within a component of the MULTI-BSP architecture. For example, the operational semantics of the language says that a value declared at the MULTI-BSP level is accessible from every process. This value is thus stored inside a memory that can be accessed from each process of the architecture. However, according to the implementation of the language, this value is replicated in the memory of every process. In this case, we need to keep all these value instances (potentially stored in different memories) to the same value to preserve the replicated coherency. Otherwise, it is possible to introduce incoherency and unpredictable behaviour. For example, if a value `v` defined at level `m` is different in some memories, a code of the form `if v = … then (*multi-function call*) else (*sequential code*)` may lead to an incoherent state where only a sub-part of the machine will execute a multi-function. They can have

unpredictable results, especially when global barriers are performed without all the processors.

We also need to prevent the execution of random operators[1]. For example, let `random_bool` be an operator that returns a random boolean. When executed inside a node, the following code is not valid:

```
if random_bool () then
  (proj v)
else
  (fun x -> x)
```

Here, this code clearly leads to an undefined behaviour: some processors would execute the `proj` synchronisation primitive and the others will not. Depending of the implementation (*e.g.* with different communication libraries), we could face a deadlock or an erroneous message signal.

4.1.2 Level compatibility

Some values are intended to be used at a specific level. For example, the parallel vector construction (`<< (*code*) >>`), cannot be executed outside a node. As a parallel vector manages the memory of sub-nodes, it only makes sense at the BSP level. In our model it is absurd to execute this primitive on the leaves (sequential level) because there are no sub-processes. The evaluation of a parallel vector at the MULTI-BSP level does not make sense either; the MULTI-BSP level cannot express parallelism in this way, it only represents a replicated execution of a sequential code or the call of a multi-function. Furthermore, such a code inside a parallel vector (at local level) would lead to parallel vector nesting: something we want to avoid, like in BSML (see next section). We want to avoid the execution of BSP code outside the scope of a node. As the access to the sub-component of a level is possible from a node only, it is meaningless to express BSP parallelism anywhere else.

Because of those reasons, we need to define rules concerning the *accessibility* (where the data can be accessed) and *definability* (where a data can be declared) of values, to check whether or not a value makes sense at a given level. For example, the definition of a BSP code outside the scope of a node is possible. However, such a value will be usable on a node only.

4.1.3 Parallel structure nesting

At first, we want to avoid parallel vector nesting. Like in BSML, we do not want to accept a code such as `let v = << $this$ >> in << v >>` which leads to type `int par par`. We can imagine that this vector nesting could stand for the manipulation, as a global structure, of the sub-nodes, directly inside a parallel vector. But such a structure is not convenient in terms of programming and also introduces implicit communication cost through levels, as we manipulate several levels of memory. Therefore, the vector nesting is prohibited by the typing system.

Some languages allow this kind of nested parallelism, such as NESL [10], but this implies difficulties concerning implementation, portability and predictions (due to a more complicated cost model). Note that experiences on parallel vector nesting (within two levels) were proposed in the Departmental Metacomputing ML library (DMML) [63], but this approach was too complex to handle.

It is also necessary to prohibit trees nesting. Indeed the type `'a tree tree` is not acceptable because it means that a complete tree (representing the amount of values distributed all over the nodes and leaves of the architecture) is available at every level of the architecture. In other terms,

[1] We can imagine a pseudo-random operator, available in the standard library, initialised with the same seed in order to keep the replicated coherency using communications, or not. However, it is not the topic here.

we can say that a complete tree has been serialised and distributed all over the architecture. Such a structure is unreasonable in MULTI-ML and is not acceptable.

Then, we need to manage cross-nesting such as `'a par tree` and `'a tree par`.

The first nesting is obviously a nonsense. A type `'a par tree` means that the value stored at each node and leaf is a parallel vector. However, it is impossible (and inconceivable in the MULTI-ML language) to have a parallel vector on leaves. This type is therefore rejected by the MULTI-ML typing system.

Concerning the type `'a tree par`, it is a bit different. Having trees inside parallel vectors seems to be a bad idea, but it makes sense under certain conditions. During the construction of a tree, we need to make a recursive call of a multi-function inside a parallel vector. This execution will lead to a parallel vector containing all the sub-trees of the current level: a vector of *branches*. In other terms, we have a parallel vector that contains the *pointers* to branches (sub-trees) that are going to be used during the construction of the final tree. Such a type is not a problem in our type system because its construction is highly restricted (by the multi-tree-function mechanism) and cannot be communicated outside the level where it belongs to. This is the reason why this kind of nesting is permitted in the MULTI-ML language, during the construction of trees only.

4.2 Operational semantics

To ensure that a program *"cannot go wrong"*, we first need to define how a program *"goes"*, which consists in defining the operational semantics. This formal definition of the MULTI-ML language will be used to prove several properties relative to parallelism, such as determinism. To agree with the typing system's proof, we introduce the natural (also called big step) semantics. To do so, and to avoid being overwhelmed by the details of a complete language, we describe the *core-language* and the definitions that will be used for the semantics. Then, we propose the set of evaluation rules that describe the behaviour of the MULTI-ML language.

To formalise the MULTI-ML semantics, OTT [134], [[21]] has been used. OTT is a tool for writing definitions of programming languages and calculi. It takes as input a definition of a language syntax and semantics, in a concise and readable ASCII notation that is close to what one would write in informal mathematics. It generates many outputs, such as a COQ version of the definition, which is very useful to formally prove the correctness of the semantics.

4.2.1 A core language

We chose to select a subset MULTI-ML and form the μMULTI-ML language, to ease the understanding and analysis of the semantics. This *core-language* relies on a minimal set of OCAML constructions. This small amount of programming features is sufficient enough to express all the parallel behaviour that is used in MULTI-ML. Thus, features such as records, modules, pattern matching, sum types are excluded from μMULTI-ML.

The grammar of the μMULTI-ML expressions is defined in Figure 4.1.

In this grammar, x and f range over an infinite set of *identifiers*. We also find the typical ML-like language constructors such as **let** for bindings and also **fun** and **rec** for, respectively, function and recursive functions. As expected, the application is denoted $(e\ e)$. For the sake of readability, we take the liberty to use the familiar infix notation for binary and ternary operators, as well as the usual precedence and associativity rules. When the context is clear, we can avoid the usage of parentheses. The multi-function definition is written with the keyword **multi** and

e	::=		
	|	x	*Variable*
	|	**op**	*Operator*
	|	**cst**	*Constant*
	|	**let** $x = e$ **in** e	*Let binding*
	|	**fun** $x \to e$	*Function*
	|	**rec** $f\ x \to e$	*Recursive function*
	|	**multi** $f\ x \to e \dagger e$	*Multi-function*
	|	$(e\ e)$	*Application*
	|	**if** e **then** e **else** e	*Conditional*
	|	(e, e)	*Pair*
	|	**mkpar** e	*Parallel primitives*
	|	**proj** e | **put** e	*Synchronous parallel primitives*
	|	$<< e >>$ | $\#x\#$ | $\$x\$$	*Parallel syntactic sugar*
	|	**replicate** (**fun** $_ \to e$)	*Core parallel primitives*
	|	**down** x | **apply** $e\ e$	*Core parallel primitives*
v	::=		*Values*
	|	**op**	*Operator*
	|	**cst**	*Constant*
	|	$\overline{(\textbf{fun}\ x \to e\ )[\mathcal{E}]}$	*Closure*
	|	$\overline{(\textbf{rec}\ f\ x \to e\ )[\mathcal{E}]}$	*Recursive closure*
	|	$\overline{(\textbf{multi}\ f\ x \to e \dagger e)[\mathcal{E}]}$	*Multi-function closure*
	|	(v, v)	*Pair*
	|	$< v, ..., v >$	*Parallel vector*
op	::=		*Operators*
	|	$+, -, *, /,$ **fst**, **snd**, ...	*Basic operators*
$\mathcal{E}$	::=	$\{x_1 \mapsto v_1, ...x_n \mapsto x_n\}$	*Environment*

Figure 4.1: The grammar of the μMULTI-ML expressions.

takes two arguments separated by the † symbol. Then, we have the parallel primitive **mkpar** followed by the synchronous parallel primitives used for communications: **proj** and **put**. The grammar also contains the syntactic sugar which eases the way of manipulating parallel vectors and the core parallel primitives which are used in the μMULTI-ML languages.

Then, the values `v` stands for the standard operators, such as common computations on integers or the **fst** and **snd** projections, and also constants (which might refer to integer values, booleans, characters, strings, *etc.*), functional, recursive and multi-function closures, pairs and also parallel vectors.

An environment $\mathcal{E}$ is interpreted as a partial mapping (informally called "finite mapping") with finite domain from identifiers to values. The mapping associates v_i to x_i, for all i from 1 to n, and is undefined on other identifiers. The empty mapping is written $\{\emptyset\}$. We also write $Dom(\mathcal{E})$ for the domain of $\mathcal{E}$, that is $\{x_1, \ldots, x_n\}$, and $CoDom(\mathcal{E})$ for its range, that is $\{v_1, \ldots, v_n\}$. If x belongs to $Dom(\mathcal{E})$, $\{x \mapsto v\} \in \mathcal{E}$ denotes the variable v associated to x in $\mathcal{E}$. Finally, we define the extension of $\mathcal{E}$ by v in x, written $\mathcal{E} \oplus \{x \mapsto v\}$, by:

$$\begin{array}{rcl} \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\} \oplus \{x \mapsto v\} & = & \{x_1 \mapsto v_1, ..., x_n \mapsto v_n, x \mapsto v\} \\ & & \text{if } x \notin \{x_1, ..., x_n\} \\ \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\} \oplus \{x \mapsto v\} & = & \{..., x_{i-1} \mapsto v_{i-1}, \boldsymbol{x \mapsto v}, x_{i+1} \mapsto v_{i+1}, ...\} \\ & & \text{if } x = x_i \end{array}$$

Hence, we have as expected:

$$\begin{array}{rcl} Dom(\mathcal{E} \oplus \{x \mapsto v\}) & = & Dom(\mathcal{E}) \cup \{x\} \\ \{x \mapsto v\} \in (\mathcal{E} \oplus \{x \mapsto v\}) & = & v \\ \{y \mapsto v\} \in (\mathcal{E} \oplus \{x \mapsto v\}) & = & \{y \mapsto v\} \in \mathcal{E} \text{ for all } y \in Dom(\mathcal{E}), y \neq x \end{array}$$

The set of expressions available in the MULTI-ML language is pretty straightforward and reflects the constructions available in the language. However, it is important to distinguish the *parallel syntactic sugar* (which can be used to program a MULTI-ML algorithm) and the *Core parallel primitives* (which are only used in the semantics). Indeed, the syntactic sugar eases the way of programming but it is not suitable for the semantics as it introduces implicit assumptions. Thus, we must transform and abstract the syntactic sugar proposed using the *core parallel primitives*. This transformation is trivial and produces an equivalent expression.

The informal description of this *new* set of primitives is the following:

- **apply** e e is similar to the original BSML primitive. It takes a two values: a parallel vector of functions and a parallel vector of values, and produces a parallel vector consisting in the application of functions on values. **apply** is useful to apply a function on each element of a parallel vector. Its informal semantics is:

$$\textbf{apply} < f_0, ..., f_{\mathbf{p}-1} >< v_0, ..., v_{\mathbf{p}-1} > \mapsto < (f_0\ v_0), ..., (f_{\mathbf{p}-1}\ v_{\mathbf{p}-1}) >$$

- **replicate** (**fun** _ $\rightarrow$ e) is used to replicate a value on every component of a sub-node. To do so, it takes a function (**fun** _ $\rightarrow$ e), which takes any arguments and returns e. Then, the function is evaluated on a *unit* argument, on each sub-component. Its informal semantics is:

$$\textbf{replicate}\ (\textbf{fun}\ _ \rightarrow e\ ) \mapsto < (\textbf{fun}\ _ \rightarrow e\ )(), ..., (\textbf{fun}\ _ \rightarrow e\ )() >$$

- **down** x simply takes a value `x` and returns a parallel vector containing `x` on each compo-

nents. Its informal semantics is:

$$\textbf{down}\ \mathsf{x} \mapsto < \mathsf{x}, \ldots, \mathsf{x} >$$

replicate, **down** and **apply** are *core primitives* which have been introduced in the semantics to perform the transformation. Thus, those *keywords* are not available in the syntax of the MULTI-ML language. The transformation applied to switch from the syntactic sugar to the core parallel primitives is straightforward.

The parallel vector scope, denoted $<< e >>$, is transformed using the **replicate** core primitive. Thus, $<< e >>$ is transformed into **replicate** (**fun** _ $\rightarrow e$).

The **open** syntax ($) is transformed using the **apply** primitive. Like in BSML, the transformation is simple and does not require a complicated expression analysis. To do so, we build a vector of functions that takes, as argument, the dollar's annotated value. Using the **apply** primitive, we can *apply* this vector of functions on the vector of values. For example, the expression $<< e\ \$x\$ >>$ is transformed into **apply** (**replicate** (**fun** _ $x \rightarrow e\ x$)) x.

The **copy** syntax (#) stands for a communication of a value to the sub-nodes. The primitive **down** is used to perform such a transfer. As the **down** primitive aims to distribute a value to all the subnodes, the transformation is obvious. For example, the expression $<< e\ \#x\# >>$ is transformed into **apply** (**replicate** (**fun** _ $x \rightarrow e\ x$))(**down** x). As the sharp annotated value is given as argument of the vector of functions, there are no redundant copies. The expression $<< \#x\# + \#x\# >>$ is tranformed into a code that copy x to the sub-nodes, only once.

To sum up, the informal description of the primitives and their (ML) types are available in Table 4.1. It is important to note that **p**, which stands for the number of processes of a BSP architecture, describes, here in MULTI-BSP, the number of sub-nodes of a node.

primitive	type	informal description
`down`	`down : 'a -> 'a par`	**down** $\mathsf{x} \mapsto \langle \mathsf{x}, \ldots, \mathsf{x} \rangle$
`replicate`	`replicate : (unit -> 'a) -> 'a par`	**replicate** (**fun** _ $\rightarrow e$) $\mapsto \langle$(**fun** _ $\rightarrow e$)(), ...$\rangle$
`apply`	`apply : ('a -> 'b) par -> 'a par -> 'b par`	**apply**$\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle \mapsto \langle (f_0\ v_0), \ldots, (f_{\mathbf{p}-1}\ v_{\mathbf{p}-1}) \rangle$

Table 4.1: The informal description of μMULTI-ML primitives.

In this core language we do not propose the grammar for trees. The multi-tree-function definition, its application and the usage of trees are thus not available in both the semantics and the typing system. Indeed, the usage of the tree structure introduces a large number of annotations resulting in technical and rather unreadable rules. For these reasons, we do not introduce them in this thesis and we postpone their presentation to future work.

However, we will informally introduce the types of the tree primitives, as they are available in the current MULTI-ML implementation.

4.2.2 Semantics rules

The big-step semantics (or natural semantics) is used to describe the evaluation of an expression to its final value using inductive rules. On the contrary, the small-steps semantics apply one-step reductions on terms. Even if big-step semantics may be more convenient in various cases (especially to prove some properties by induction), small-step semantics is widely used because of its expressiveness concerning both terminating and non-terminating programs. Thanks to [104], it is also possible to describe both terminating and diverging evaluations with a big-step semantics. Thus, we choose to use this approach to describe the MULTI-ML semantics. Unlike

the method chosen in [55], here, there is no need to handle processors interleaving. As a consequence, it is easier to model the semantics using COQ. First, we will propose the terminating semantics for terms and then give rules for diverging ones. In the context of ML programming, we only consider eager (or call-by-value) evaluation strategy. Thus, we do not consider lazy (or call-by-need) evaluations.

4.2.3 Terminating semantics rules

The inference rules are written:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v} \text{ and } \frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty}$$

On the left-hand side, we have the inductive inference rule. It can be read: under the premises (or antecedents) $\mathcal{P}$, the inference rule concludes that, on the component p of locality $\mathcal{L}$, within a multi-environment $\mathcal{M}$, the expression e is evaluated into a value v. This rule leads to a finite derivation tree. On the right-hand side, the co-inductive inference rule is written similarly, but using a *double-bar*. It stands for a case where e diverges (∞). Otherwise, it is an error: no applicable rules are defined.

The environment of evaluation is denoted by $\mathcal{M}$ and is called a *multi-environment*. It is composed by two elements: (1) a MULTI-BSP environment and (2) a tree of environments.

The first element represents the MULTI-BSP *memory*. This memory contains values that are accessible by any processes. Depending on the implementation and the architecture, it can be simulated by a virtual unified memory[2] or by a physical global memory, for example, when simulating an execution in the toplevel. The virtual memory is not available in the current implementation of MULTI-ML, we leave it for future work. In practice, the memory is replicated on each component of the MULTI-BSP architecture. We choose to create a single environment that is used during MULTI-BSP evaluations. An evaluation taking place at the MULTI-BSP level, denoted by *multi*, is then evaluated within the environment $||\mathcal{M}||@multi$. The notation $||\mathcal{M}||$ denotes the extraction of an environment from the memory $\mathcal{M}$ and the $@multi$ refers to the targeted memory. As expected, every unit of the architecture can access this MULTI-BSP environment. It is possible to lookup in it at any time, from any level p, that is to say, from each component of the MULTI-BSP architecture. Nevertheless, the modifications of this environment are possible at the MULTI-BSP level only.

The second element represents all the environments of the architecture. It takes the shape of a tree which maps all the environments of the different levels of the architecture, that is to say all the nodes and leaves. Referring to the environment of a particular component p of this tree is denoted $||\mathcal{M}||@p$. We recall that the components are identified using a regular tree numbering where 0 stands for the *root* node, and 0.0 to 0.n identify its n sub-components. Thus, it contains its environment only, deprived of the MULTI-BSP memory. As the MULTI-BSP memory is always available, the notation $||\mathcal{M}||_p$ stands for the environment composed by $||\mathcal{M}||@multi \cup ||\mathcal{M}||@p$, where the subscript p stands for the targeted memory.

Now, we introduce the symbol of rule evaluation. The piece of notation $\Downarrow_p^{\mathcal{L}}$ denotes an evaluation taking place at a specific level $\mathcal{L}$, also parametrised by the position p. The evaluation level $\mathcal{L}$ can take the values `m`, `b`, `l` or `s`, to specify on which level the evaluation must occur. The locality `c`, which is proposed in the MULTI-ML type system and denotes values that are communicable to any contexts, is not used here. This locality is important and makes sense in the type system

[2] A memory which is updated and unified globally by a synchronous mechanism.

$$
\begin{array}{rcl}
\mathcal{C}omm_s(\textbf{mkpar}\ e) & = & \bot \\
\mathcal{C}omm_s(\textbf{proj}\ e) & = & \bot \\
\mathcal{C}omm_s(\textbf{put}\ e) & = & \bot \\
\mathcal{C}omm_s(\textbf{apply}\ e_1\ e_2) & = & \bot \\
\mathcal{C}omm_s(\textbf{down}\ x) & = & \bot \\
\mathcal{C}omm_s(\textbf{replicate}\ e) & = & \bot \\
\mathcal{C}omm_s(<\ldots>) & = & \bot \\
\mathcal{C}omm_s(\textbf{multi}\ f\ x \rightarrow e_1 \dagger e_2) & = & \bot \\
\mathcal{C}omm_s(\overline{(\textbf{multi}\ f\ x \rightarrow e_1 \dagger e_2)[\mathcal{E}]}) & = & \bot \\
\mathcal{C}omm_s((e_1, e_2)) & = & \mathcal{C}omm_s(e_1) \wedge \mathcal{C}omm_s(e_2) \\
\mathcal{C}omm_s(\overline{(\textbf{fun}\ x \rightarrow e\ )[\mathcal{E}]}) & = & \mathcal{C}omm_{\mathcal{E}}(e) \\
\mathcal{C}omm_s(\overline{(\textbf{rec}\ f\ x \rightarrow e\ )[\mathcal{E}]}) & = & \mathcal{C}omm_{\mathcal{E}}(e) \\
\mathcal{C}omm_s(\textbf{fun}\ x \rightarrow e\ ) & = & \mathcal{C}omm_s(e) \\
\mathcal{C}omm_s(\textbf{rec}\ f\ x \rightarrow e\ ) & = & \mathcal{C}omm_s(e) \\
\mathcal{C}omm_s(e_1\ e_2) & = & \mathcal{C}omm_s(e_1) \wedge \mathcal{C}omm_s(e_2) \\
\mathcal{C}omm_s(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) & = & \mathcal{C}omm_s(e_1) \wedge \mathcal{C}omm_s(e_2) \\
\mathcal{C}omm_s(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) & = & \mathcal{C}omm_s(e_1) \wedge \mathcal{C}omm_s(e_2) \wedge \mathcal{C}omm_s(e_3) \\
\mathcal{C}omm_s(x) & = & \top, \textbf{if}\ x \notin s \\
 & & \mathcal{C}omm_s(v), \textbf{if}\ \{x \mapsto v\} \in s \\
\mathcal{C}omm_s(\textbf{cst}) & = & \top \\
\mathcal{C}omm_s(\textbf{op}) & = & \top
\end{array}
$$

Figure 4.2: The communication predicate.

but, it is not necessary in the semantics. We use the locality `l` (*local*) to refer to the scope of a parallel vector, and we do not pay attention to its communication ability through an explicit annotation. Nevertheless, the communicability of a value is still important in the semantics. Thus, we propose a predicate to make this verification: $\mathcal{C}omm_s(v)$. By abuse of the notation, this predicate is $\top$ if the value v is communicable, $\bot$ otherwise, within environment s. The definition of the predicate can be found in Figure 4.2. The subtlety of this notation comes with the communicability of x. We assume that, if $x \notin s$, then the value x is communicable. Indeed, if x does not belong to s, it means that x is a free variable which comes from the application of a closure, as the environment used for a closure is ε, which is the environment of the closed term. Thus, during the evaluation of an application $(e_1\ e_2)$, we assume that x is communicable in e_1 as its communicability will be checked with e_2.

The position p stands for the identifier of the component where an evaluation takes place. We use the notation $p.i$ to refer to the ith sub-unit of a component that is accessible through a parallel vector. The identifier alias *root* stands for the root node and, as explained previously, *multi* is used when the whole architecture is concerned by an evaluation.

As the MULTI-ML language proposes different levels associated with different behaviours, we propose several rules that are either generic or specific to `m`, `b`, `l`. In Figure 4.3, we define all the evaluation rules which are applicable independently of the evaluation level. Those generic rules are valid everywhere and represent the functional kernel of the language. These rules are pretty straightforward and are really close to the ML semantics. However, some of them introduce new notations that will be explained along the rules description.

The VALUES and OP_EVAL rules a straightforward.

The VAR rule introduces the predicate $lookup(x, \mathcal{M}, p, \mathcal{L})$, which is defined in Figure 4.7. As

$$\text{VALUES} \quad \frac{}{\mathcal{M} \vdash \mathbf{cst} \Downarrow_p^{\mathcal{L}} \mathbf{cst}}$$

$$\text{OP__EVAL} \quad \frac{}{\mathcal{M} \vdash \mathbf{op} \Downarrow_p^{\mathcal{L}} \mathbf{op}}$$

$$\text{VAR} \quad \frac{\{x \mapsto v\} \in lookup(x, \mathcal{M}, p, \mathcal{L})}{\mathcal{M} \vdash x \Downarrow_p^{\mathcal{L}} v}$$

$$\text{CLOSURE} \quad \frac{\begin{array}{l}\mathcal{E} = select(\mathcal{M}, \mathcal{F}(\mathbf{fun}\ \ x \to v\ \ ), p, \mathcal{L}) \\ v \equiv \overline{(\mathbf{fun}\ \ x \to e\ \ )[\mathcal{E}]}\end{array}}{\mathcal{M} \vdash \mathbf{fun}\ \ x \to e\ \ \Downarrow_p^{\mathcal{L}} v}$$

$$\text{REC__CLOSURE} \quad \frac{\begin{array}{l}\mathcal{E} = select(\mathcal{M}, \mathcal{F}(\mathbf{rec}\ \ f\ \ x \to v\ \ ), p, \mathcal{L}) \\ v \equiv \overline{(\mathbf{rec}\ \ f\ \ x \to e\ \ )[\mathcal{E}]}\end{array}}{\mathcal{M} \vdash \mathbf{rec}\ \ f\ \ x \to e\ \ \Downarrow_p^{\mathcal{L}} v}$$

$$\text{FUN__APP} \quad \frac{\begin{array}{ll}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\mathbf{fun}\ \ x \to e\ \ )[\mathcal{E}]} & \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v \\ \mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x \mapsto v\} \vdash e \Downarrow_p^{\mathcal{L}} v' & \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} v'}$$

$$\text{REC__FUN__APP} \quad \frac{\begin{array}{l}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\mathbf{rec}\ \ f\ \ x \to e\ \ )[\mathcal{E}]} \\ \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v \qquad \mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x \mapsto v\} \oplus_p \overline{(\mathbf{rec}\ \ f\ \ x \to e\ \ )[\mathcal{E}]} \vdash e \Downarrow_p^{\mathcal{L}} v'\end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} v'}$$

$$\text{OP__APP} \quad \frac{\begin{array}{ll}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \mathbf{op} & \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v \\ \overline{\mathbf{op}\ \ v} \equiv v' & \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} v'}$$

$$\text{LET__IN} \quad \frac{\begin{array}{l}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} v_1 \\ \mathcal{M} \oplus_p \{x \mapsto v_1\} \vdash e_2 \Downarrow_p^{\mathcal{L}} v_2\end{array}}{\mathcal{M} \vdash \mathbf{let}\ \ x = e_1\ \ \mathbf{in}\ \ e_2 \Downarrow_p^{\mathcal{L}} v_2}$$

Figure 4.3: Generic evaluation rules.

$$\text{REPLICATE}\quad \frac{\begin{array}{l}\forall i \in p \quad \mathcal{M} \oplus_{p_i} \{f \mapsto (\textbf{fun } _ \rightarrow e\ )\} \vdash f\ \ () \Downarrow^{\mathsf{l}}_{p_i} v_i \\ \mathcal{C}omm_{||\mathcal{M}||_{p_i}}(\textbf{fun } _ \rightarrow e\ )\end{array}}{\mathcal{M} \vdash \textbf{replicate } (\textbf{fun } _ \rightarrow e\ ) \Downarrow^{\mathsf{b}}_p < v_0, ..., v_n >}$$

$$\text{DOWN}\quad \frac{\begin{array}{l}\mathcal{M} \vdash x \Downarrow^{\mathsf{b}}_p v \\ \mathcal{C}omm_{||\mathcal{M}||_p}(v)\end{array}}{\mathcal{M} \vdash \textbf{down } x \Downarrow^{\mathsf{b}}_p < v >}$$

$$\text{MKPAR}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e \Downarrow^{\mathsf{b}}_p \overline{(e')[\mathcal{E}']} \\ \forall i \in p \quad \mathcal{M} \oplus_p \mathcal{E}' \vdash e'\ \ i \Downarrow^{\mathsf{b}}_p v_i \\ \mathcal{C}omm_{||\mathcal{M}||_p}(v_i)\end{array}}{\mathcal{M} \vdash \textbf{mkpar } e \Downarrow^{\mathsf{b}}_p < v_0, ..., v_n >}$$

$$\text{APPLY}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e_1 \Downarrow^{\mathsf{b}}_p < \overline{(e_i)[\mathcal{E}_i]} > \qquad \mathcal{M} \vdash e_2 \Downarrow^{\mathsf{b}}_p < v_0, ..., v_n > \\ \forall i \in p \quad \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}\} \vdash f_i\ \ i \Downarrow^{\mathsf{l}}_{p.i} v'_i\end{array}}{\mathcal{M} \vdash \textbf{apply } e_1\ \ e_2 \Downarrow^{\mathsf{b}}_p < v'_0, ..., v'_n >}$$

$$\text{PROJ}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e \Downarrow^{\mathsf{b}}_p < v_0, ..., v_n > \\ \forall i \in p \quad \mathcal{M} \oplus_{p.i} \{f \mapsto \overline{(e')[\mathcal{E}']}\} \vdash f\ \ i \Downarrow^{\mathsf{b}}_{p.i} v_i \qquad \mathcal{C}omm_{||\mathcal{M}||_p}(v_i)\end{array}}{\mathcal{M} \vdash \textbf{proj } e \Downarrow^{\mathsf{b}}_p \overline{(e')[\mathcal{E}']}}$$

$$\text{PUT}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e \Downarrow^{\mathsf{b}}_p < \overline{(e_0)[\mathcal{E}_0]}, ..., \overline{(e_n)[\mathcal{E}_n]} > \\ \forall i, j \in p \\ \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}\} \vdash f_i\ \ j \Downarrow^{\mathsf{l}}_{p.i} v_{ij} \\ \mathcal{M} \oplus_{p.j} \{f'_j \mapsto \overline{(e'_j)[\mathcal{E}'_j]}\} \vdash f'_j\ \ i \Downarrow^{\mathsf{l}}_{p.j} v_{ij}\end{array}}{\mathcal{M} \vdash \textbf{put } e \Downarrow^{\mathsf{b}}_p < \overline{(e'_0)[\mathcal{E}'_0]}, ..., \overline{(e'_n)[\mathcal{E}'_n]} > f}$$

Figure 4.4: BSP evaluation rules.

$$\textsc{multi__node}\quad \frac{\begin{array}{l} isNode(p) \\ \mathcal{M} \vdash e_1 \Downarrow_p^{\mathtt{l}} \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathtt{l}} v \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \oplus_p \{f \mapsto \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']}\} \vdash e_1' \Downarrow_p^{\mathtt{b}} v' \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathtt{l}} v'}$$

$$\textsc{multi__leaf}\quad \frac{\begin{array}{l} isLeaf(p) \\ \mathcal{M} \vdash e_1 \Downarrow_p^{\mathtt{l}} \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathtt{l}} v \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \vdash e_2' \Downarrow_p^{\mathtt{s}} v' \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathtt{l}} v'}$$

Figure 4.5: Local evaluation rules.

$$\textsc{multi__def}\quad \frac{\begin{array}{l} \mathcal{M}' = select(||\mathcal{M}||_{multi}, \mathcal{F}(\textbf{multi}\ \ f\ \ x \to e_1 \dagger e_2)) \\ v \equiv \overline{(\textbf{multi}\ \ f\ \ x \to e_1 \dagger e_2)[\mathcal{M}']} \qquad \mathcal{C}omm_{||\mathcal{M}||_{root}}(v) \end{array}}{\mathcal{M} \vdash (\textbf{multi}\ \ f\ \ x \to e_1 \dagger e_2) \Downarrow_{multi}^{\mathtt{m}} v}$$

$$\textsc{multi__call}\quad \frac{\begin{array}{l} \mathcal{M} \vdash e_1 \Downarrow_{multi}^{\mathtt{m}} \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow_{multi}^{\mathtt{m}} v \\ \mathcal{M} \oplus_{root} \{x \mapsto v\} \oplus_{root} \{f \mapsto \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']}\} \vdash e_1' \Downarrow_{root}^{\mathtt{b}} v' \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_{multi}^{\mathtt{m}} v'}$$

Figure 4.6: MULTI-BSP evaluation rules.

$$
\begin{array}{lcll}
lookup(x, \mathcal{M}, p, \mathcal{L}) & = & \textbf{if } \{x \mapsto v\} \in ||\mathcal{M}||@p \textbf{ then } v & \textit{Value at } p \\
 & & \textbf{if } \{x \mapsto v\} \in ||\mathcal{M}||@multi \textbf{ then} & \\
 & & \quad \textbf{if } \mathcal{L} \equiv \mathtt{l} \textbf{ then } \mathtt{error} & \textit{Out of reach} \\
 & & \quad \textbf{else } v & \textit{Value at multi} \\
 & & \textbf{else } \mathtt{error} & \textit{Unbound value}
\end{array}
$$

Figure 4.7: The lookup predicate.

$$
\begin{array}{lcl}
select(\mathcal{M}, \mathcal{V}, p, \mathcal{L}) & = & let\ \mathcal{E}' = \emptyset \\
 & & \forall \alpha \in \mathcal{V}, lookup(\alpha, \mathcal{M}, p, \mathcal{L}) \oplus \mathcal{E}' \\
 & & where\ \mathcal{V} \equiv \alpha_0, ..., \alpha_n \\
 & & return\ \mathcal{E}'
\end{array}
$$

Figure 4.8: The select predicate.

the VAR rule gives the value corresponding to a binding, we must *lookup* for it in a particular way. The value x must be seek, first, in $||\mathcal{M}||@p$, which is the memory of the evaluation at level p. Then, if the value is not found there, we must look for it in the MULTI-BSP environment: $||\mathcal{M}||_{multi}$. If the value is not found anyway, the identifier is unbound in the current expression, which is an evaluation error (*"unbound value"*). As expected, if the evaluation takes place at the MULTI-BSP level (*multi*), the variable x is only searched for in the environment $||\mathcal{M}||_{multi}$. Furthermore, we add the locality of the evaluation as an input of the *lookup*() predicate to ensure that the values defined in the `m` memory won't be accessed within the scope of a parallel vector (of locality `l`).

The CLOSURE and REC_CLOSURE rules are used to enclose all the needed variables that are available in $\mathcal{M}$. To do so, we use the predicate $select(\mathcal{M}, \mathcal{V}, p, \mathcal{L})$ which is defined in Figure 4.8. This predicate *selects* all the variables of the set of variables $\mathcal{V}$ from the multi-environment $\mathcal{M}$. As the predicate uses *lookup*(), we must provide the current level (p) and its locality ($\mathcal{L}$). The set of variables $\mathcal{V}$ is determined using $\mathcal{F}(e)$, which stands for the set of free variables of the expression. The set of free variables $\mathcal{F}(e)$ is defined in Figure 4.9.

Definition 4.1 (*Free variables*)

A free variable stands for a variable which is referenced in a context but which is not bound within the context.

A term e is referenced as closed *if it does not contain any free variable: $\mathcal{F}(e) = \emptyset$.*

The final and minimal environment $\mathcal{E}$ produced by *select*() is then attached to the final closure.

The FUN_APP is used to evaluate functional constructions on values. The rule introduces the notation $\oplus_p$, which is a concatenation operator allowing to enrich the environment of p with bindings. Here, we add the closure environment $\mathcal{E}$ and the binding $\{x \mapsto v\}$ to $||\mathcal{M}||_p$ to the environment of p. The REC_FUN_APP rule is similar to FUN_APP but it also adds the associated recursive closure into the environment. The OP_APP is also close to the FUN_APP rule as it deals with basic operators instead of closures.

The LET_IN rule is necessary to bind values within a context. The rule is straightforward as it only adds $\{x \mapsto v_1\}$ to the environment of p.

$$
\begin{array}{lrcl}
\mathcal{F}(e) ::= & \mathcal{F}(\mathbf{op}) & = & \emptyset \\
& \mathcal{F}(\mathbf{cst}) & = & \emptyset \\
& \mathcal{F}(x) & = & x \\
& \mathcal{F}(\mathbf{fun}\ x \to e\ ) & = & \mathcal{F}(e)\backslash\{x\} \\
& \mathcal{F}(\mathbf{rec}\ f\ x \to e\ ) & = & \mathcal{F}(e)\backslash\{x\} \\
& \mathcal{F}(\mathbf{multi}\ f\ x \to e_1 \dagger e_2) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2)\backslash\{x\} \\
& \mathcal{F}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2)\backslash\{x\} \\
& \mathcal{F}(e_1\ e_2) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \\
& \mathcal{F}((e_1, e_2)) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2)
\end{array}
$$

Figure 4.9: The free variable predicate.

In Figure 4.4 we present all the rules that can be evaluated at the BSP level.

The REPLICATE rule is used to create parallel vectors. We recall that this primitive is introduced by the code transformation and is not available in the syntax of the MULTI-ML language. The REPLICATE primitive requires, syntactically, a function definition as an argument. This function takes *any* argument, denoted with _ (similarly to the OCAML syntax), which is unbound in the enclosed term. Such function definition allows to delay the evaluation of the expression e when REPLICATE is evaluated. This is necessary for the MULTI-ML runtime system, as explained in Section 5.1. The notation $\forall i \in p$ allows to reference the i sub-components of the current node p, from 0 to n. Thus, the closure is added to the environment of each leaf of the current node, identified by p_i, if the term is communicable. Then, the function is evaluated on (), which is a unit term, on each sub-components. As the evaluation takes place within the scope of a parallel vector, its evaluation locality is `l`. The resulting value is, as expected, a parallel vector which contains the results of the evaluation of e on each sub-component.

The DOWN rule is used to distribute data to the sub-nodes. From a variable x, it builds a parallel vector containing the corresponding value v, if v is communicable. The distributed values are identical on the sub-nodes, thus the resulting parallel vector is $< v >$.

MKPAR also aims to build a parallel vector. Nevertheless, the evaluation of the ith elements of the resulting vector takes place at level p, sequentially. This evaluation is done within a single environment that is common to every evaluation. Thus, one evaluation could impact the environment of another one with a code using side effects, even if there is no way of doing any side effects in the current semantics. It is important to note that the big-step semantics does not allow to describe precisely the evaluation order of the i evaluations. We arbitrarily choose to do it in accordance to the ascending order: from 0 to n. Thus, the expression e' is applied on i, which corresponds to the processor identifier (pid) of the i^{st} component. It is important to check that every value v_i is communicable. Indeed, we need to communicate the values $v_0, .., v_n$ to the sub-nodes of the level p and we must check the validity of such a transfer.

As expected, the APPLY rule applies a parallel vector of functions (e_i) on a parallel vector of data (v_i) and returns a parallel vector of values. Here, we compute, in parallel, the evaluations of $f_i\ \ i$ within the environment of each sub-component i.

The PROJ rule must be applied on a parallel vector available at the current level p. The resulting value is a closure which contains the content of the given vector. As expected, the values are accessible via their original identifiers. It is also necessary to check whether or not the values of the given vector are communicable in order to build a communicable closure.

The PUT rule works as expected. It takes a parallel vector of communication functions, referenced as f_i within environment p_i. Then, the received values are accessible via f'_j. Here, both i and j stand for the sub-components of the current node p. They are denoted as distinct variables to show the *all-to-all* exchange.

The rules presented in Figure 4.5 are specific to the locality `l`. We recall that this locality corresponds to the scope of a parallel vector. With the core language, only the multi-functions recursive calls are particular within the local context.

The MULTI_NODE and MULTI_LEAF rules are relative to the recursive multi-function calls, which corresponds to the application of multi-function function on a value within the scope of a parallel vector. MULTI_NODE and MULTI_LEAF are applied regarding to the type of the current component. The operators $isNode(p)$ and $isLeaf(p)$ are used to make such a distinction. Informally speaking, $isNode(p)$ returns an error if p has no sub-components. On the contrary, $isLeaf(p)$ returns an error if p has sub-components. Concerning the MULTI_NODE rule, the environment of evaluation of e'_1 is enriched by the argument given to the multi-function and the closure of the multi-function. Indeed, the sub-node must be able to recall the multi-function. We can observe that, after enriching the environment, the evaluation of e'_1 goes from `l` to `b`. This locality change is due to the fact that, when a recursive call of a multi-function (done within the scope of a parallel vector) is evaluated, we must initiate the recursion on the sub-nodes: which corresponds to evaluate e'_1 at level `b`. On the contrary, the MULTI_LEAF rule is not concerned by this environment enrichment because of its status of leaf. A similar behaviour can be observed in the MULTI_LEAF rule, where we go from `l` to `s`.

Finally, in Figure 4.6, we propose the rules that are specific to the MULTI-BSP level. It mainly deals with the multi-functions.

The MULTI_DEF rule aims to create the closure of a multi-function and make it available in the MULTI-BSP environment. To do so, we simply call the predicates $select(||\mathcal{M}||_{multi}, \mathcal{F}(\textbf{multi}\ f\ x \rightarrow e_1 \dagger e_2))$ in order to built the environment $\mathcal{M}'$ from the MULTI-BSP environment and the free variables of the multi-function itself. As the closure v aims to be communicated to the *root* node in order to initiate the recursion, we must check that v is communicable.

The MULTI_CALL rule is used to initiate the multi-function evaluation. As expected, the multi-function evaluation starts on the *root* node, with an environment enriched by the given argument and the multi-function closure. Thus, the evaluation of e'_1, corresponding to the node code, is evaluated at level `b`, on the *root* node. As the resulting value of the evaluation of the multi-function is going to be transferred to the MULTI-BSP level, we also need to check if v' is communicable. Indeed, the final value must be available in the $||\mathcal{M}_{multi}||$ environment. To do so, in case of a distributed MULTI-BSP memory, we must *broadcast* v' from the *root* node to all the components of the MULTI-BSP architecture.

We recall that, for now, we do not describe the usage of multi-tree-functions nor trees. To do so, we must define a new kind of environment which is, basically, a tree of environments. Thus, at each evaluation step, it produces a new tree of environment with a resulting value. Such a definition introduces a large amount of notation which makes the semantics rules barely readable.

4.2.4 Terminating semantics is deterministic

In this section, we give the proof of the terminating semantics rules. To do so, we prove that the evaluation of an expression following the proposed rules is deterministic.

Lemma 4.2 (*Evaluation or not*)

Let e be an expression, $\mathcal{M}$ be an evaluation environment, $\mathcal{L}$ an evaluation locality and p a position (referencing to a particular unit). Then, using classical logic with the excluded middle:

- *It is impossible to obtain a value v such that $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$*
- **or** *there exists a value v such that $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$*

Proof: (**Using classical logic**). The excluded middle holds over $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$. ■

Lemma 4.3 (*Evaluation is deterministic*)

Let e be a program, let $\mathcal{M}$ be an environment, $\mathcal{L}$ an evaluation locality, p a position and v_1 and v_2 be values. If $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_1$ and $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_2$ then $v_1 = v_2$.

Proof: See Appendix A.1.1. ■

4.2.5 Diverging semantics rules

Following [104], we define the divergent rules (with infinite evaluations) by the coinductive interpretation of the previous inference rules. The relation of divergence is written $e \Downarrow_p^{\mathcal{L}} \infty$ where e is an expression which diverges, when evaluated at level p of locality $\mathcal{L}$.

In Figures 4.10, 4.11, 4.12, 4.13 and 4.14 we describe the diverging rules, written directly from the terminating semantics rules defined in Section 4.2.3. They, respectively, describe the coinductive rules of generic, BSP (divided in two figures), local and MULTI-BSP. As the principle of the rules is straightforward, we do not describe the rules one by one. Informally, the main idea of the diverging rules is to describe all the *states* where the evaluation of an expression can diverge. The rules where an application $e_1\ e_2$ takes place and the diverging term is e_1 (corresponding to rules FUN_APP, REC_FUN_APP, MULTI_NODE, MULTI_LEAF and MULTI_CALL) were fused into a single rule: APP-L.

4.2.6 Diverging semantics properties

In this section, we use the properties of the co-inductive semantics to prove that an evaluation produces a value or diverges.

Lemma 4.4 (*Evaluation and divergence exclusivity*)

Let e be a program, let $\mathcal{M}$ be an environment, $\mathcal{L}$ an evaluation locality, p a position and v a value. If $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$ and $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty$ then there is a contradiction.

Proof: See Appendix A.1.2. ■

Note that this lemma does not prove that any program can be evaluated. For example, this lemma is clearly false when "random" operators are allowed: one can write a code like `if randomBool then finiteComputation else infiniteComputation`, which neither diverge nor

$$\text{APP-L}\ \frac{\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \infty}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} \infty}$$

$$\text{FUN__APP-R}\ \frac{\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{fun}\ \ x \to e\ \ )[\mathcal{E}]} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} \infty}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} \infty}$$

$$\text{FUN__APP-E}\ \frac{\begin{array}{c}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{fun}\ \ x \to e\ \ )[\mathcal{E}]} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v \\ \mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x \mapsto v\} \vdash e \Downarrow_p^{\mathcal{L}} \infty\end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} \infty}$$

$$\text{REC__FUN__APP-R}\ \frac{\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{rec}\ \ f\ \ x \to e\ \ )[\mathcal{E}]} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} \infty}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} \infty}$$

$$\text{REC__FUN__APP-E}\ \frac{\begin{array}{c}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{rec}\ \ f\ \ x \to e\ \ )[\mathcal{E}]} \\ \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v \qquad \mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x \mapsto v\} \oplus_p \overline{(\textbf{rec}\ \ f\ \ x \to e\ \ )[\mathcal{E}]} \vdash e \Downarrow_p^{\mathcal{L}} \infty\end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} \infty}$$

$$\text{LET__IN}\ \frac{\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \infty}{\mathcal{M} \vdash \textbf{let}\ \ x = e_1\ \ \textbf{in}\ \ e_2 \Downarrow_p^{\mathcal{L}} \infty}$$

$$\text{LET__IN}\ \frac{\begin{array}{c}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} v_1 \\ \mathcal{M} \oplus_p \{x \mapsto v_1\} \vdash e_2 \Downarrow_p^{\mathcal{L}} \infty\end{array}}{\mathcal{M} \vdash \textbf{let}\ \ x = e_1\ \ \textbf{in}\ \ e_2 \Downarrow_p^{\mathcal{L}} \infty}$$

Figure 4.10: Generic coinductive evaluation rules.

$$\text{REPLICATE}\ \frac{\exists i \in p \quad \mathcal{M} \oplus_{p_i} \{f \mapsto (\textbf{fun}\ _ \to e\ )\} \vdash f\ ()\Downarrow^{\mathsf{l}}_{p_i} \infty}{\mathcal{M} \vdash \textbf{replicate}\ (\textbf{fun}\ _ \to e\ ) \Downarrow^{\mathsf{b}}_{p} \infty}$$

$$\text{MKPAR-E}\ \frac{\mathcal{M} \vdash e \Downarrow^{\mathsf{b}}_{p} \infty}{\mathcal{M} \vdash \textbf{mkpar}\ e \Downarrow^{\mathsf{b}}_{p} \infty}$$

$$\text{MKPAR-V}\ \frac{\mathcal{M} \vdash e \Downarrow^{\mathsf{b}}_{p} \overline{(e')[\mathcal{E}']} \quad \exists i \in p \quad \mathcal{M} \oplus_p \mathcal{E}' \vdash e'\ i \Downarrow^{\mathsf{b}}_{p} \infty}{\mathcal{M} \vdash \textbf{mkpar}\ e \Downarrow^{\mathsf{b}}_{p} \infty}$$

$$\text{APPLY-L}\ \frac{\mathcal{M} \vdash e_1 \Downarrow^{\mathsf{b}}_{p} \infty}{\mathcal{M} \vdash \textbf{apply}\ e_1\ e_2 \Downarrow^{\mathsf{b}}_{p} \infty}$$

$$\text{APPLY-R}\ \frac{\mathcal{M} \vdash e_1 \Downarrow^{\mathsf{b}}_{p} < \overline{(e_i)[\mathcal{E}_i]} > \qquad \mathcal{M} \vdash e_2 \Downarrow^{\mathsf{b}}_{p} \infty}{\mathcal{M} \vdash \textbf{apply}\ e_1\ e_2 \Downarrow^{\mathsf{b}}_{p} \infty}$$

$$\text{APPLY-F}\ \frac{\mathcal{M} \vdash e_1 \Downarrow^{\mathsf{b}}_{p} < \overline{(e_i)[\mathcal{E}_i]} > \qquad \mathcal{M} \vdash e_2 \Downarrow^{\mathsf{b}}_{p} < v_0, ..., v_n > \quad \forall i \in p \quad \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}\} \vdash f_i\ i \Downarrow^{\mathsf{l}}_{p.i} \infty}{\mathcal{M} \vdash \textbf{apply}\ e_1\ e_2 \Downarrow^{\mathsf{b}}_{p} \infty}$$

$$\text{PROJ-E}\ \frac{\mathcal{M} \vdash e \Downarrow^{\mathsf{b}}_{p} \infty}{\mathcal{M} \vdash \textbf{proj}\ e \Downarrow^{\mathsf{b}}_{p} \infty}$$

$$\text{PROJ-F}\ \frac{\mathcal{M} \vdash e \Downarrow^{\mathsf{b}}_{p} < v_0, ..., v_n > \quad \exists i \in p \quad \mathcal{M} \oplus_{p.i} \{f \mapsto \overline{(e')[\mathcal{E}']}\} \vdash f\ i \Downarrow^{\mathsf{b}}_{p.i} \infty}{\mathcal{M} \vdash \textbf{proj}\ e \Downarrow^{\mathsf{b}}_{p} \infty}$$

Figure 4.11: BSP coinductive evaluation rules - part 1.

$$\text{PUT-E}\ \frac{\mathcal{M} \vdash e \Downarrow_p^{\mathsf{b}} \infty}{\mathcal{M} \vdash \textbf{put}\ \ e \Downarrow_p^{\mathsf{b}} \infty}$$

$$\text{PUT-IJ}\ \frac{\mathcal{M} \vdash e \Downarrow_p^{\mathsf{b}} < \overline{(e_0)[\mathcal{E}_0]}, ..., \overline{(e_n)[\mathcal{E}_n]} > \quad \forall i \in p \qquad \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}\} \vdash f_i\ \ j \Downarrow_{p.i}^{\mathsf{l}} \infty}{\mathcal{M} \vdash \textbf{put}\ \ e \Downarrow_p^{\mathsf{b}} \infty}$$

$$\text{PUT-JI}\ \frac{\begin{array}{c}\mathcal{M} \vdash e \Downarrow_p^{\mathsf{b}} < \overline{(e_0)[\mathcal{E}_0]}, ..., \overline{(e_n)[\mathcal{E}_n]} > \\ \forall i, j \in p \\ \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}\} \vdash f_i\ \ j \Downarrow_{p.i}^{\mathsf{l}} v_{ij} \\ \mathcal{M} \oplus_{p.j} \{f'_j \mapsto \overline{(e'_j)[\mathcal{E}'_j]}\} \vdash f'_j\ \ i \Downarrow_{p.j}^{\mathsf{l}} \infty\end{array}}{\mathcal{M} \vdash \textbf{put}\ \ e \Downarrow_p^{\mathsf{b}} \infty}$$

Figure 4.12: BSP coinductive evaluation rules - part 2.

$$\text{MULTI__NODE-R}\ \frac{isNode(p) \qquad \mathcal{M} \vdash e_1 \Downarrow_p^{\mathsf{l}} \overline{(\textbf{multi}\ \ f\ \ x \to e'_1 \dagger e'_2)[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathsf{l}} \infty}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathsf{l}} \infty}$$

$$\text{MULTI__NODE-B}\ \frac{\begin{array}{c}isNode(p) \qquad \mathcal{M} \vdash e_1 \Downarrow_p^{\mathsf{l}} \overline{(\textbf{multi}\ \ f\ \ x \to e'_1 \dagger e'_2)[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathsf{l}} v \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \oplus_p \{f \mapsto \overline{(\textbf{multi}\ \ f\ \ x \to e'_1 \dagger e'_2)[\mathcal{M}']}\} \vdash e'_1 \Downarrow_p^{\mathsf{b}} \infty\end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathsf{l}} \infty}$$

$$\text{MULTI__LEAF-R}\ \frac{isLeaf(p) \qquad \mathcal{M} \vdash e_1 \Downarrow_p^{\mathsf{l}} \overline{(\textbf{multi}\ \ f\ \ x \to e'_1 \dagger e'_2)[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathsf{l}} \infty}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathsf{l}} \infty}$$

$$\text{MULTI__LEAF-S}\ \frac{\begin{array}{c}isLeaf(p) \qquad \mathcal{M} \vdash e_1 \Downarrow_p^{\mathsf{l}} \overline{(\textbf{multi}\ \ f\ \ x \to e'_1 \dagger e'_2)[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathsf{l}} v \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \vdash e'_2 \Downarrow_p^{\mathsf{s}} \infty\end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathsf{l}} \infty}$$

Figure 4.13: Local coinductive evaluation rules.

$$\text{MULTI__CALL-R}\ \frac{\mathcal{M} \vdash e_1 \Downarrow^{\mathtt{m}}_{multi} \overline{(\textbf{multi}\ \ f\ \ x \to e'_1 \dagger e'_2)[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow^{\mathtt{m}}_{multi} \infty}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow^{\mathtt{m}}_{multi} \infty}$$

$$\text{MULTI__CALL-B}\ \frac{\begin{array}{c}\mathcal{M} \vdash e_1 \Downarrow^{\mathtt{m}}_{multi} \overline{(\textbf{multi}\ \ f\ \ x \to e'_1 \dagger e'_2)[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow^{\mathtt{m}}_{multi} v \\ \mathcal{M} \oplus_{root} \{x \mapsto v\} \oplus_{root} \{f \mapsto \overline{(\textbf{multi}\ \ f\ \ x \to e'_1 \dagger e'_2)[\mathcal{M}']}\} \vdash e'_1 \Downarrow^{\mathtt{b}}_{root} \infty\end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow^{\mathtt{m}}_{multi} \infty}$$

Figure 4.14: MULTI-BSP coinductive evaluation rules.

evaluate in general. Without "random" operators, programs that neither evaluate nor diverge according to the rules above are said to *"go wrong"*. For instance, the program `0 0` (*zero* applied on *zero*) goes wrong since, for any environment $\mathcal{E}$ and value v, neither $\mathcal{E} \vdash (0\ \ 0) \Downarrow v$ nor $\mathcal{E} \vdash (0\ \ 0) \Downarrow_\infty$ holds.

In the following, we propose a system of typing rules that allows to identify such programs. A type system is an *over-approximation* which forbid programs that *"go wrong"*.

4.3 A typing system for μMULTI-ML

In this section, we propose a typing system for the core language μMULTI-ML. This mini-language offers a subset of the OCAML features that are necessary to program a MULTI-ML algorithm. We choose to reduce the expressivity of this toy language for conciseness reasons.

4.3.1 Type definitions

A lot of effect based type systems or flow analyses use type annotations [136]. Considering a *regular* type system, annotating types with an effect allows to represent the level of information carried in addition to the structure of the values. These informations embedded with the type are controlled by relations defined, most of the time, in lattices. As a type describes the value that an expression may return, effects describe the side-effect that an expression may have, that is to say its *locality of evaluation*.

In MULTI-ML, a type annotated by its locality information is denoted: τ_π. This means that a traditional ML type τ is complemented with the locality of use: π. The locality of a type is, basically, an effect associated to a type that keeps its location information. The evaluation of a lambda term may also produce an effect. We talk about a *latent effect* (usually symbolised with ε instead of π) and we denote it $\xrightarrow{\varepsilon}$. This annotation announces the presence of a particular effect ε inside the closure of the lambda term. This effect is *emitted* during the evaluation of the function and we need to deal with it according to the current evaluation level. When an expression e is evaluated, it also emits an effect which represents the locality requirement of e. We write τ_π/ε when the evaluation of an expression leads to a value typed τ_π with an emitted effect ε.

Type variables are denoted by α and β (and its derivatives $\alpha_1, ..., \alpha', ..., \beta_1, ..., \beta'$). The locality variable is usually denoted by π (and its derivatives $\pi_1, ..., \pi', ...$). When representing a locality, λ denotes a variable effect and ε refers to a latent effect.

The definition of the grammar of annotated types is given in Figure 4.15.

$$
\begin{array}{rcll}
\tau & ::= & \alpha_\pi & \textit{type variable} \\
 & | & \texttt{Base}_\pi & \textit{Basic type } (\texttt{int}\ ,\ \texttt{Bool},\ \ldots) \\
 & | & (\tau \xrightarrow{\pi} \tau)_\pi & \textit{arrow type} \\
 & | & (\tau \times \tau)_\pi & \textit{product type} \\
 & | & (\tau\ \texttt{par})_\pi & \textit{parallel vector} \\
 & | & (\tau\ \texttt{tree})_\pi & \textit{tree}
\end{array}
$$

Figure 4.15: Annotated types definition.

The locality of a type is an important information that is carried in the type annotation. It refers to the locality of the definition and evaluation of a value and it is used to control where such a value is acceptable, or not. The definition of localities is given in Figure 4.16.

$$
\begin{array}{rcll}
\pi & ::= & \lambda & \textit{variable locality} \\
 & | & \texttt{m} & \text{MULTI-BSP } \textit{locality} \\
 & | & \texttt{b} & \text{BSP } \textit{locality (on node)} \\
 & | & \texttt{c} & \textit{communicable locality (inside parallel vectors)} \\
 & | & \texttt{l} & \textit{local locality (inside parallel vectors)} \\
 & | & \texttt{s} & \textit{purely sequential locality (on leaf)}
\end{array}
$$

Figure 4.16: Locality definition.

The meaning of the different localities can be expressed as follow:

- `m` : The MULTI-BSP level. An expression evaluated at this level concerns every stage of the MULTI-BSP architecture. Thus, the generated value is available for every process of the machine. This level is useful to run some code that is going to be executed through the whole architecture and can handle particular constructions such as multi-functions and trees. Depending on the implementation and the architecture, this memory can be a specific memory accessible from every level; it also could be simulated by the replication of the values in every memory.

- `b` : The BSP memory of a node. At this level, we can execute BSML code, manage multi-function recursion through parallel vectors and also execute communications and synchronisations. It is also possible to *point towards* the underlying memory of a node via the parallel vector construction.

- `c` : The *communicable* memory of the underlying memory which is managed through a parallel vector. This locality is manly used inside the scope of a parallel vector to denote a value that makes sense in any context. Thus, this memory is free of references to the upper level and context sensitive constructions.

- `l` : The *local* memory of the underlying memory that is managed inside a parallel vector. Like `c`, this is the underlying memory that can be managed inside a parallel vector from the upper node. This memory can contain *references* to the upper memory via a specific syntax and context sensitive constructions. This is the reason why it makes no sense to communicate values of such locality to another stage of the architecture.

- `s` : The *sequential* memory of a leaf. This memory is used to execute sequential computations. As it is purely sequential, it can contain specific values, such as random values.

To sum up, each locality is related to a memory level which refers to a level of execution. These levels are defined by the MULTI-BSP architecture and are distinguished by their parallel capabilities. We have `m` for MULTI-BSP, `b` for BSP, `s` for sequential, `c` for communicable and `l` for local.

We recall that, as effects are not displayed in standard OCAML, we need to update the OCAML type syntax in order to show the locality information on types. For example, the annotated type $int_{\mathtt{m}}$ is naturally displayed `int_m`. Concerning functional values, the latent effect $\xrightarrow{\mathtt{l}}$ is represented `-(l)->`. Finally, to denote polymorphism on localities we choose the *backquote* ‘: `` `z ``.

As the MULTI-ML type system deals with polymorphic types, we need to add several definitions. Here, as the definition concerns type polymorphism, the type annotations are not required. Thus, we do not consider them in the following definitions, for conciseness purposes.

Let $\mathcal{F}(\tau)$ denote the set of types variables that are free (also known as *Free Type Variables*) in the type τ which is defined by :

$$\begin{array}{lcl}
\mathcal{F}(\mathtt{Base}_\pi) & = & \emptyset \\
\mathcal{F}(\alpha_\pi) & = & \{\alpha\} \\
\mathcal{F}(\tau_1 \to \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) \\
\mathcal{F}(\tau_1 \times \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2)
\end{array}$$

Here and elsewhere, an environment Γ is interpreted as a partial mapping (improperly called finite mapping) with finite domain from identifiers to values: the mapping that associates v_i to x_i, for all i from 1 to n, and that is undefined on the other identifiers. The empty mapping is written $\{\}$. We write $Dom(\Gamma)$ for the domain of Γ, that is $\{x_1, \ldots, x_n\}$, and $CoDom(\Gamma)$ for its range, that is $\{v_1, \ldots, v_n\}$. If x belongs to $Dom(\Gamma)$, $\{x \mapsto v\} \in \Gamma$ denotes the type v associated to x in Γ. Finally, we define the extension of Γ by v in x, written $\Gamma \oplus \{x \mapsto v\}$, by:

$$\begin{array}{lcl}
\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \oplus \{x \mapsto v\} & = & \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, x \mapsto v\} \\
 & & \text{if } x \notin \{x_1, \ldots, x_n\} \\
\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \oplus \{x \mapsto v\} & = & \{\ldots, x_{i-1} \mapsto v_{i-1}, \boldsymbol{x \mapsto v}, x_{i+1} \mapsto v_{i+1}, \ldots\} \\
 & & \text{if } x = x_i
\end{array}$$

As expected, we have:

$$\begin{array}{rcl}
Dom(\Gamma \oplus \{x \mapsto v\}) & = & Dom(\Gamma) \cup \{x\} \\
\{x \mapsto v\} \in (\Gamma \oplus \{x \mapsto v\}) & = & v \\
\{y \mapsto v\} \in (\Gamma \oplus \{x \mapsto v\}) & = & \{y \mapsto v\} \in \Gamma \text{ for all } y \in Dom(\Gamma), y \neq x
\end{array}$$

The substitutions considered here are finite mappings from type variables to type expressions. They are denoted φ, ψ:

$$\varphi, \psi \ ::= \ [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \to \tau_n]$$

A substitution φ is defined as follows for a type expression:

$$\begin{array}{lcl}
\varphi(\mathtt{Base}) & = & \mathtt{Base} \\
\varphi(\alpha) & = & \tau' \text{ if } [\alpha \mapsto \tau'] \in \varphi \\
\varphi(\tau_1 \to \tau_2) & = & \varphi(\tau_1) \to \varphi(\tau_2) \\
\varphi(\tau_1 \times \tau_2) & = & \varphi(\tau_1) \times \varphi(\tau_2)
\end{array}$$

A substitution φ is also naturally defined on a set of variables. The following property shows the effect of a subscript over the free variables of a type:

Proposition 4.5

For a type τ and a substitution φ, we have: $\mathcal{F}(\varphi(\tau)) = \bigcup_{\alpha \in \mathcal{F}(\tau)} \mathcal{F}(\varphi(\alpha))$

Proof: By structural induction over τ:

- Case $\tau = \texttt{Base}$. $\mathcal{F}(\varphi(\texttt{Base})) = \emptyset$ and there is no $\alpha \in \mathcal{F}(\tau)$, hence the result.
- Case $\tau = \alpha$. If $\alpha \notin Dom(\varphi)$ then $\varphi(\alpha) = \alpha$ and $\varphi(\tau) = \tau$ and hence the result. Otherwise, there exists β such that $\beta = \varphi(\alpha)$. Thus, $\mathcal{F}(\varphi(\tau)) = \beta$ and only $\alpha \in \mathcal{F}(\tau)$; hence the result.
- Case $\tau = \tau_1 \to \tau_2$ (resps. $\tau_1 \times \tau_2$). Both by straightforward applications of the inductive hypothesis. ■

To handle polymorphism, we now define the set *type schemes*, with the typical element σ, by the following grammar:

$$\sigma ::= \forall \alpha_1, \ldots, \alpha_n.\tau$$

In this syntax, the quantified variables $\alpha_1, \ldots, \alpha_n$ are considered as a set of variables: their relative order is not significant and they are assumed to be distinct. When the context is obvious, we do not distinguish the type τ and the trivial type scheme $\forall.\tau$. Thus, we write τ for both. We identify two kinds of type schemes that differ modulo the renaming of the variables bound by the $\forall$ quantifier (α-conversion), and by the introduction, or deletion, of quantification variables that are not free in the type. More precisely, the type schemes are equated as following:

$$\begin{array}{lcl} \forall \alpha_1, \ldots, \alpha_n.\tau & = & \forall \beta_1, \ldots, \beta_n.[\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau) \\ \forall \alpha \alpha_1, \ldots, \alpha_n.\tau & = & \forall \alpha_1, \ldots, \alpha_n.\tau \text{ if } \alpha \notin \mathcal{F}(\tau) \end{array}$$

The free variables in a type scheme are:

$$\mathcal{F}(\forall \alpha_1, \ldots, \alpha_n.\tau) = \mathcal{F}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}$$

This definition is naturally compatible with the two equations over type schemes. Then, we extend substitutions to type schemes as following:

$$\varphi(\forall \alpha_1, \ldots, \alpha_n.\tau) = \forall \alpha_1, \ldots, \alpha_n.\varphi(\tau) \text{ if } \{\alpha_1, \ldots, \alpha_n\} \not\subseteq \mathcal{F}(\tau)$$

Thus, assuming that, after renaming the α_i, there are *out of reach* of φ which are: none of the α_i is in the domain of φ; and none of the α_i is free in one of the types in the range of φ. That is: (1) $\varphi(\alpha) = \alpha$ (the substitution does not modify α); (2) if α is not free in τ then α is not free in $\varphi(\tau)$ (φ does not introduce α in its result). We can easily check that this definition is compatible with the two equations over schemes. The Proposition 4.5 also holds with the type τ replaced by a scheme σ:

Proposition 4.6

For a type σ and a substitution φ, we have: $\mathcal{F}(\varphi(\sigma)) = \bigcup_{\alpha \in \mathcal{F}(\sigma)} \mathcal{F}(\varphi(\alpha))$

Proof: Take $\sigma = \forall \alpha_1, \ldots, \alpha_n.\tau$ such that $\forall i$ the α_i are out of reach of φ. By definition, $\mathcal{F}(\sigma) = \mathcal{F}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}$ and $\varphi(\sigma) = \forall \alpha_1, \ldots, \alpha_n.\varphi(\tau)$. Then $\mathcal{F}(\varphi(\sigma)) = \mathcal{F}(\varphi(\tau))$. Since the α_i are out of reach of φ then $\mathcal{F}(\sigma) = \mathcal{F}(\tau)$. Thus $\bigcup_{\alpha \in \mathcal{F}(\sigma)} \mathcal{F}(\varphi(\alpha)) = \bigcup_{\alpha \in \mathcal{F}(\tau)} \mathcal{F}(\varphi(\alpha))$ and then by application of the Proposition 4.5, we get the result. ■

Proposition 4.7

For a substitution φ and an environment E, we have: $\mathcal{F}(\varphi(E)) = \bigcup_{\alpha \in \mathcal{F}(E)} \mathcal{F}(\varphi(\alpha))$

Proof: By induction on E and applying the Proposition 4.6 for each $\beta \in Dom(E)$. ■

A typing environment, written E, is a finite mapping from identifiers to type schemes:

$$E ::= \{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\}$$

The mapping definitions (Dom, $CoDom$, $\in$, $\oplus$) are also naturally defined for a typing environment. The image $\varphi(E)$ of an environment E by a substitution φ is naturally defined as following:

$$\varphi(\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\}) = \{x_1 \mapsto \varphi(\sigma_1), \ldots, x_n \mapsto \varphi(\sigma_n)\}$$

The operator $\mathcal{F}$ is extended to the typing environment, by assuming that a type variable α is free in E if and only if for some x, $\{x \mapsto \sigma\} \in E$, α is free in σ:

$$\mathcal{F}(E) = \bigcup_{x \in Dom(E)} \mathcal{F}(\sigma) \text{ if } \{x \mapsto \sigma\} \in E$$

Finally, we say that the type τ is an *instance* of the type scheme $\sigma = \alpha_1, \ldots, \alpha_n.\tau'$ and we write $\tau \leq \sigma$, if and only if there exists a substitution φ whose domain is a subset of $\{\alpha_1, \ldots, \alpha_n\}$ such that τ is equal to $\varphi(\tau')$:

$$\tau \leq \forall\alpha_1, \ldots, \alpha_n.\tau' \text{ if and only if } \exists\tau_1, \ldots, \tau_n \text{ such as } \tau = [\alpha \leftarrow \tau_1, \cdots \alpha \leftarrow \tau_n](\tau')$$

The relation $\tau \leq \sigma$ is obviously compatible with the two equations over schemes (α-conversion and useless quantified variable elimination).

For constants and primitives, we assume a function TC that assign type schemes to constants and operators. For instance, we propose the following list of non exhaustive assignments:

$TC(i)$	=	`int`	(if i is an integer)
$TC(b)$	=	`Bool`	(if b equals `True` or `False`)
$TC(+)$	=	`int` $\rightarrow$ `int` $\rightarrow$ `int`	(if $+$ is addition on integer)
$TC($`IfThenElse`$)$	=	$\forall\alpha$.`Bool` $\rightarrow \alpha \times \alpha \rightarrow \alpha$	

4.3.2 Type constraints definition

As our type system is highly inspired by the OCAML one, and is basically an extension of it, we are faithful to the standard Damas and Milner principles. In this section, our type system is related to the Hindley-Milner system relying on a X constraint system: HM(X) [119]. In our context of parallel programming, the X constraint system will handle all the MULTI-ML constructions. The type inference system will rely on PCB(X). As explained in [124], HM(X) and PCB(X) are really close in their concepts and definitions.

To manage effects dues to the parallel features of the MULTI-ML language, we choose to introduce a set of constraints. The grammar of the constraints can be found in Figure 4.17.

c	$::=$		*constraints*
	$\mid$	`True`	*truth*
	$\mid$	`False`	*falsity*
	$\mid$	$c \wedge c$	*conjunction*
	$\mid$	$\tau = \tau$	*type equality*
	$\mid$	$\lambda = \lambda$	*locality equality*
	$\mid$	$\lambda \lhd \lambda$	*accessibility of locations*
	$\mid$	$\lambda \blacktriangleleft \lambda$	*definability of locations*
	$\mid$	$\lambda = \mathbf{Propgt}(\varepsilon, \varepsilon')$	*effect propagation*
	$\mid$	$\tau = \mathbf{Seria}_{\Lambda}(\alpha_{\pi})$	*type serialisation*

Figure 4.17: Constraints definition.

As expected, a constraint can be `True` or `False` and is equated using regular conjunctions. The constraint language allows to equate both types and localities. It also defines several predicates (**Propgt**, **Seria**) and relations ($\lhd$,$\blacktriangleleft$) that are necessary to handle the annotated types associated with the particular MULTI-ML hierarchical architecture. We describe them in the following.

4.3.2.1 Accessibility

As the locality refers to the evaluation level of a value, all the localities are not *accessible* everywhere. Indeed, the locality of a value stands for, in a way, its "*quantity* of parallelism". Thus, all localities are not compatible with each other. In a MULTI-ML *way of speaking*, we can say that the notion of *accessibility* is similar to an access to a level by reading in its memory. In practice, it is basically how it works. For example, declaring a parallel vector in the scope of a leaf must be forbidden. As a leaf is a purely sequential unit, it is a nonsense to use a parallel data structure at this level. Accessing a `b` value at level `s` must also be blocked by the accessibility relation.

Definition 4.8 (*Locality* accessibility*:* $\lhd$)

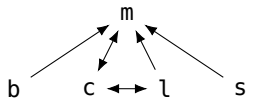
> *The piece of notation* $\lambda 1 \lhd \lambda 2$ *means that the locality* $\lambda 1$ *is* accessible *(by read access) in locality* $\lambda 2$*, that is to say «locality* $\lambda 1$ *can be read from locality* $\lambda 2$*».*

Therefore, we need to define which read accesses are acceptable between two locality levels. The valid accessibility relations are described in Figure 4.18a. As expected, a variable annotated with the locality `m` is *accessible* from everywhere. On the contrary, level `b` and `s` are *accessible* in their own context only. Finally, `l` and `c` are mutually accessible in the body of a parallel vector.

The Figure 4.18b show the relation between all the levels using directed arrows, from λ_1 to λ_2, to denote the notion of *« λ_1 can read in memory λ_2 »*.

`m, c`	$\lhd$	`m`
`m, b`	$\lhd$	`b`
`m, l, c`	$\lhd$	`c`
`m, l, c`	$\lhd$	`l`
`m, s`	$\lhd$	`s`

(a) Relations of *accessibility*.



(b) Relations of *accessibility*.

Figure 4.18: The *accessibility* relation.

4.3.2.2 Definability

Values that deal with special localities cannot be *defined* at every level of the architecture. The definition of a value that is not usable in its context and which cannot be communicated outside its context shall not be permitted. For example, it is important to reject the definition of a multi-function within a level which is not the MULTI-BSP (`m`) one. As we said previously, the multi-function is a particular construction which concerns the whole machine. It is thus senseless to define it the context of a single unit. In a sense, it is important to avoid the definition of values that are pointless.

Nevertheless, it is important to be able to define constructions that are not usable in their definition context, but that will be afterwards. For example, the definition of a function that manages parallel vectors at the MULTI-BSP level is not usable in its definition context. As this code deals with the BSP level, it is suitable on nodes, for the design of BSP algorithms. Such a *cross-definition* is particularly useful for MULTI-ML libraries and functions managing parallel vectors. We can interpret the notion of *definability* as a write access from a locality to another [3].

Definition 4.9 (*Locality* definability*:* ◀)

The notation $\lambda_1 \blacktriangleleft \lambda_2$ means that it is valid to define a value of locality λ_1 within a context of locality λ_2; in other word, λ_1 can "write" in memory λ_2.

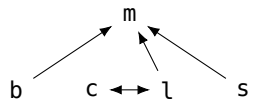
We also need to define which "write accesses" are conceivable between two memory levels. The valid definability relations are described in Figure 4.19a. As expected, `s` and `b` are valid within `m`. The locality `s` is restricted to its own context and both `c` and `l` are inter-compatible. One can note that the definability relations `c` ◀ `m` and `c` ◀ `b` are not defined. Indeed, as the `c` locality comes from the scope of a parallel vector, it is not possible to explicitly declare a value with such a locality in `m` and `b`.

A OCAML code would have an unbound locality (‘z for instance), which may be bound during the application.

The Figure 4.19b shows the relations between localities using directed arrows, from λ_1 to λ_2, to specify that *« λ_1 is definable in λ_2 »*. In order to handle side effects, that is to say references, we need to add several constraints to guarantee safe executions. Those considerations are detailed in Section 4.4.1.

s, b, m	◀	m
b	◀	b
l, c	◀	c
l, c	◀	l
s	◀	s

(a) Relations of *definability*.



(b) Relations of *definability*.

Figure 4.19: The *definability* relation.

4.3.2.3 Propagation

To handle the multiple effects that can be generated by the evaluation of an expression, we need to have a relation of *propagation*. This relation allows to select the prevailing effect among several of the emitted effects of an evaluation. For example, the rule APP (Figure 4.24) may generate different effects. Without going into the details of this rule, it is obvious that the

[3] This interpretation makes perfect sens with references (see Section 4.4.1).

evaluation of $e_1\ e_2$ (*i.e.* e_1 applied to e_2) may emit two different effects; respectively ε_1 and ε_2 for the evaluation of e_1 and e_2. In order to determine the effect of the whole expression, it is necessary to choose the prevailing effect between ε_1 and ε_2.

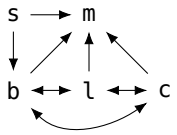
Definition 4.10 (Propagation : Propgt($\varepsilon_1, \varepsilon_2$))

This relation returns the prevailing effect amongst ε_1 and ε_2.

The Figure 4.20 shows both the table (Figure 4.20a) and the lattice (Figure 4.20b) of the relations of *propagation*, between all localities. We can easily see a relation of superiority between levels. Some combinations are marked with a $\perp$ when the relation is not acceptable. For example, **Propgt**(`s`, `l`) means that we are going to mix up what have been detected to be a sequential code with local code: which is not conceivable.

Propgt	`m`	`b`	`l`	`c`	`s`
`m`	`m`	`m`	`m`	`m`	`m`
`b`	`m`	`b`	`b`	`b`	`b`
`l`	`m`	`b`	`l`	`l`	$\perp$
`c`	`m`	`b`	`l`	`c`	$\perp$
`s`	`m`	`b`	$\perp$	$\perp$	`s`

(a) The table of *propagation*.



(b) The lattice of *propagation*.

Figure 4.20: The *propagation* predicate.

4.3.2.4 Serialisation

Serialisation is a safeguard assuring that all the values that are going to be communicated will be safe in any context. If we refer to the implementation, we must check that the serialisation mechanism of OCAML (`Marshal.to_bytes`) is suitable for each communicated value. Thus, it is necessary to use it with every communication primitive and also with several MULTI-ML constructions.

Definition 4.11 (Serialisation: Seria$_\Lambda(\tau_\pi)$)

This relation takes an annotated type τ_π and returns a potentially transformed type which is safe to communicate to the given level Λ. Otherwise, it fails with an error.

The *serialisation* procedure is quite easy to understand: when a value of type τ_π need to be transferred from a level to another, its type and locality must be inspected to approve such a communication. This verification is used to forbid values that are not communicable by essence, but also values that have no sense in a locality Λ. The serialisation also allows to transform values in order to make them acceptable in another locality. For example, an integer declared in a leaf will be typed `int_s` and seems to be noncommunicable. Thanks to the serialisation, and to the harmlessness of such a value, it will be *transformed* into $\texttt{int}_\Lambda$. That is to say, a perfectly communicable value.

The *serialisation* relation **Seria** is defined in Figure 4.21.

It is important to note that the serialisation rules are ordered. It is important to respect the order to be able to communicate harmless values. For example, the predicate **Seria**$_{\texttt{c}}(\texttt{int}_{\texttt{s}})$ is called when we need to communicate a integer value from a leaf to its upper node (see Section 4.3.3 for details concerning the typing rules). In such a case, as the type `int` can

$$
\begin{array}{lllll}
(1) & \mathbf{Seria}_\Lambda(\mathtt{Base}_\pi) & = & \mathtt{Base}_\Lambda \ if \ \mathtt{Base} = \mathtt{int}, \mathtt{string}, \mathtt{float}, \mathtt{Bool}, \ldots \\
(2) & \mathbf{Seria}_\Lambda(\mathtt{Base}_\pi) & = & \mathtt{Fail} \ if \ \mathtt{Base} = \mathtt{i/o}, \ldots \\
(3) & \mathbf{Seria}_\Lambda(\tau_\pi) & = & \begin{cases} \tau_\Lambda, & if \ \pi \lhd \Lambda \\ \mathtt{Fail}, & otherwise \end{cases} \\
(4) & \mathbf{Seria}_\Lambda(\tau_\pi \ \mathtt{par_b}) & = & \mathtt{Fail} \\
(5) & \mathbf{Seria}_\Lambda(\tau_\pi \ \mathtt{tree}_{\pi'}) & = & \mathtt{Fail} \\
(6) & \mathbf{Seria}_\Lambda(_ \xrightarrow{\mathtt{l}} _) & = & \mathtt{Fail} \\
(7) & \mathbf{Seria}_\Lambda(_ \xrightarrow{\mathtt{b}} _) & = & \mathtt{Fail} \\
(8) & \mathbf{Seria}_\Lambda(_ \xrightarrow{\mathtt{m}} _) & = & \mathtt{Fail} \\
(9) & \mathbf{Seria}_\Lambda(_ \xrightarrow{\mathtt{s}} _) & = & \mathtt{Fail} \\
(10) & \mathbf{Seria}_\Lambda((\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_\delta) & = & \begin{cases} (\tau'^1_{\pi'_1} \xrightarrow{\varepsilon} \tau'^2_{\pi'_2})_\Lambda, & if \ \varepsilon \notin \{\mathtt{m}, \mathtt{b}, \mathtt{s}, \mathtt{l}\} \\ & and \ \tau'^1_{\pi'_1} = \mathbf{Seria}_\Lambda(\tau^1_{\pi_1}) \neq \mathtt{Fail} \\ & and \ \tau'^2_{\pi'_2} = \mathbf{Seria}_\Lambda(\tau^2_{\pi_2}) \neq \mathtt{Fail} \\ \mathtt{Fail}, & otherwise \end{cases} \\
(11) & \mathbf{Seria}_\Lambda((\tau^1_{\pi_1} \times \tau^2_{\pi_2})_\delta) & = & \begin{cases} (\tau'^1_{\pi'_1} \times \tau'^2_{\pi'_2})_\Lambda, & and \ \tau'^1_{\pi'_1} = \mathbf{Seria}_\Lambda(\tau^1_{\pi_1}) \neq \mathtt{Fail} \\ & and \ \tau'^2_{\pi'_2} = \mathbf{Seria}_\Lambda(\tau^2_{\pi_2}) \neq \mathtt{Fail} \\ \mathtt{Fail}, & otherwise \end{cases}
\end{array}
$$

Figure 4.21: Serialisation rules.

be communicated to any level without any worries, the rule saying that: *a base type can be serialised to any locality* prevails. Here, the notion of *base* type stands for a basic or primitive type provided by OCAML as a built-in type. In the first rule, we give a non exhaustive list of the so called base types that are harmless. On the contrary, the second rules avoids the serialisation of context sensitive base types, such as input/output channels. To complete the first two rules, the third one restricts, recursively, the communications using the accessibility relation between the locality of the serialised type π and the destination locality Λ. The rules (4) and (5) are useful to avoid the communication of parallel data structures: parallel vectors and trees. From rule (6) to (9), we forbid the communication of functional values using emits context sensitives effect ($\mathtt{l}, \mathtt{b}, \mathtt{m}$ and $\mathtt{s}$). The tenth rule is used to serialise a functional type. To do so, we need call recursively the serialisation predicate on the left and right hand side of the functional value. If and only if each side is recursively valid, the value can be communicated. Otherwise, it fails. Finally, rule (11) is used to serialise pairs and works similarly to the previous rule.

As the *serialisation* is used to restrict the communication of harmful values, some safe code may be rejected as well.

4.3.2.5 Weakening

We need an overlay of *generalisation* to handle harmless values. To do so, we propose the notion of *weakening*. In addition to the standard type generalisation, this predicate is used to *erase* the harmless effects of an expression. Otherwise, the effect is kept. First, we recall the definition of a type scheme, extended with effects:

Definition 4.12 (*Type scheme*)

A type scheme σ is the generalisation of a type in a polymorphic type system. We write $\sigma = \forall\alpha_1 \ldots \alpha_n \lambda_1 \ldots \lambda_m[c].\tau_\pi$ to denote the set the types obtained by the instantiation of $\alpha_1 \ldots \alpha_n \lambda_1 \ldots \lambda_m$

by types and localities respecting the constraint c. Where $\alpha_1 ... \alpha_n$ are type variables and $\lambda_1 ... \lambda_m$ are effect variables. We write τ for a scheme $\forall \emptyset[\texttt{True}].\tau$.

Definition 4.13 (Weakening: Weak(τ_π, ε))

The weakening *of the type τ_π and the effect ε is performed according to the the nature of the effect in order to erase it, if it is harmless. Otherwise, the locality π of the type is kept and the effect of the type is set to π.*

According the specificities of the localities, the *Weakening* rules are defined in Figure 4.22.

$$
\begin{array}{rcl}
\textbf{Weak}(\tau^1 \xrightarrow{l} \tau^2, \varepsilon) & = & \tau^1 \xrightarrow{l} \tau^2/\texttt{l} \\
\textbf{Weak}(\tau^1 \xrightarrow{s} \tau^2, \varepsilon) & = & \tau^1 \xrightarrow{s} \tau^2/\texttt{s} \\
\textbf{Weak}(\tau^1 \xrightarrow{b} \tau^2, \varepsilon) & = & \tau^1 \xrightarrow{b} \tau^2/\texttt{b} \\
\textbf{Weak}(\tau^1 \xrightarrow{m} \tau^2, \varepsilon) & = & \tau^1 \xrightarrow{m} \tau^2/\texttt{m} \\
\textbf{Weak}(\tau^1 \xrightarrow{\varepsilon} \tau^2, \varepsilon') & = & \tau'^1 \xrightarrow{\varepsilon} \tau'^2/\textbf{Propgt}(\varepsilon_1, \varepsilon_2) \\
 & & \textit{where } \tau^{1'}/\varepsilon_1 = \textbf{Weak}(\tau^1, \varepsilon') \\
 & & \textit{and } \tau^{2'}/\varepsilon_2 = \textbf{Weak}(\tau^2, \varepsilon') \\
\textbf{Weak}((\tau\ \texttt{tree})_\pi, \texttt{m}) & = & (\tau\ \texttt{tree})_\pi/\texttt{m} \\
\textbf{Weak}((\tau\ \texttt{par})_\texttt{b}, \texttt{b}) & = & (\tau\ \texttt{par})_\texttt{b}/\texttt{b} \\
\textbf{Weak}(\tau, \varepsilon) & = & \tau/\texttt{c}
\end{array}
$$

Figure 4.22: *Weakening* rules.

We can see that the *weakening* of latent effects of locality `l`, `s`, `b` or `m` are spread to keep track of them.

For example, the code `<< let x = #y# in fun _ -> x >>` is perfectly valid in MULTI-ML and the value is communicable. Let say that `y` is a value defined previously and typed `y` : $\texttt{int}_\texttt{b}$. Even if `#y#` is going to emit an effect of locality `l` that is going to be spread to the binding of `x`. `x` is no longer dangerous inside the `fun _ -> x` because it is available in the environment so the closure can store the value of `x`. Here, the *weakening* is going to erase the effect of `x`. The type of this expression is thus $('\texttt{a}_\texttt{z} \xrightarrow{\texttt{c}} \texttt{int}_\texttt{c})\ \texttt{par}_\texttt{b}$.

On the contrary, the code `<< fun _ -> let x = #y# in x >>` is also valid in MULTI-ML but the value is not communicable. In this case, `x` has the effect `l`, because of `#y#`, and will be free of effect inside the right part of the *let binding*. Nevertheless, the effect `l` is spread inside the closure in order to specify that this value is not communicable. The type of this expression is thus $('\texttt{a}_\texttt{z} \xrightarrow{\texttt{l}} \texttt{int}_\texttt{l})\ \texttt{par}_\texttt{b}$.

The communication of closures is widely discussed in Section 4.3.5.

4.3.3 Typing rules

The typing rules describe how our type system attributes types and effects to a given expression. In this section we define and explain all the rules that are used in the MULTI-ML type system.

Definition 4.14 (*Typing context (environment)*)

The typing context Γ is a function that associate a type schemes and an effect to variables. $\Gamma(x)$ denote the type scheme of x in Γ. $\varnothing$ denotes an empty environment and ; is used to

$$
\begin{array}{lcl}
\Gamma(\textbf{proj}) & = & \forall(\tau,\tau',\pi,\pi')[\tau'_{\pi'} = \mathbf{Seria}_{\mathtt{b}}(\tau_\pi)].(\tau_\pi\ \mathtt{par_b} \xrightarrow{b} (\mathtt{int_c} \xrightarrow{\mathtt{c}} \tau'_{\pi'})_{\mathtt{b}})_{\mathtt{m}}/\mathtt{b} \\
\Gamma(\textbf{put}) & = & \forall(\pi,\pi_1,\pi_2,\pi_3,\tau,\tau',\varepsilon)[\pi \lhd \mathtt{c} \wedge \varepsilon \lhd \mathtt{c} \wedge \pi_1 \lhd \mathtt{c} \wedge \tau'_{\mathtt{c}} = \mathbf{Seria}_{\mathtt{c}}(\tau_{\pi_1}) \wedge \pi_3 \lhd \mathtt{c} \wedge \pi_2 \lhd \mathtt{c}]. \\
 & & ((\mathtt{int}_\pi \xrightarrow{\varepsilon} \tau_{\pi_1})_{\pi_3}\ \mathtt{par_b} \xrightarrow{\mathtt{b}} (\mathtt{int}_{\pi_2} \xrightarrow{\mathtt{c}} \tau'_{\mathtt{c}})_{\mathtt{c}}\ \mathtt{par_b})_{\mathtt{m}}/\mathtt{b} \\
\Gamma(\textbf{mkpar}) & = & \forall(\pi,\pi_1,\pi_2,\tau,\tau',\varepsilon).[\pi \lhd \mathtt{b} \wedge \varepsilon \lhd \mathtt{b} \wedge \pi_1 \lhd \mathtt{b} \wedge \pi_2 \lhd \mathtt{b} \wedge \tau'_{\mathtt{c}} = \mathbf{Seria}_{\mathtt{c}}(\tau_{\pi_1})] \\
 & & ((\mathtt{int}_\pi \xrightarrow{\varepsilon} \tau_{\pi_1})_{\pi_2} \xrightarrow{\mathtt{b}} \tau_{\mathtt{c}}\ \mathtt{par_b})_{\mathtt{m}}/\mathtt{b} \\
\Gamma(\textbf{fst}) & = & \forall(\alpha,\beta,\pi,\pi_1,\pi_2)[\ ].((\alpha_\pi \times \beta_{\pi_1})_{\pi_3} \xrightarrow{\pi_3} \alpha_\pi)_{\mathtt{m}}/\pi_3 \\
\Gamma(\textbf{fst}) & = & \forall(\alpha,\beta,\pi,\pi_1,\pi_2)[\ ].((\alpha_\pi \times \beta_{\pi_1})_{\pi_3} \xrightarrow{\pi_3} \beta_{\pi_1})_{\mathtt{m}}/\pi_3 \\
\Gamma(+) & = & \forall(\pi,\pi_1,\pi_2)[\ ].((\mathtt{int}_\pi \xrightarrow{\mathtt{c}} \mathtt{int}_{\pi_1})_{\mathtt{m}} \xrightarrow{\mathtt{c}} \mathtt{int}_{\pi_2})_{\mathtt{m}}/\mathtt{c}] \\
 & \ldots &
\end{array}
$$

Figure 4.23: Types of primitives and basic operators.

concatenate environments.

Definition 4.15 (*Typing Judgement*)

A judgement $\Gamma, \Lambda \vdash e : \tau_\pi/\varepsilon[c]$ denotes that, within an environment Γ, when evaluated at a level Λ, the expression e is typed τ_π under the constraints c and emit the effect ε.

The types of the MULTI-ML primitives and basic operators that are needed in the environment are defined in Figure 4.23.

The application of **proj** now emits an effect of locality b and generates a function that associates, for each i ($i \in \mathbf{p_i}$), the value contained in the given vector. This output value needs to be transformed by the predicate **Seria** as the value is going to be transferred from the body of a parallel vector to the scope of a node. This *validation* is necessary, for example, to avoid the projection of parallel vectors containing branches; that is to say vectors of trees.

Likewise, the output value of the **put** primitive must be transformed by the predicate **Seria** to ensure the communication of the value between the different member of the parallel vector. This transformation is particularly important with references, see Section 4.4.1. As the values are used inside the scope of a parallel vector, localities $\varepsilon, \pi, \pi_1, \pi_2$ and π_3 must be acceptable in c. Finally, as the function that describes the received data is generated through a parallel vector, its latent effect is c.

The execution of the **mkpar** primitive is different compared to BSML but its type behaviour is almost similar. We recall that, in MULTI-ML, the function given to the **mkpar** primitive is executed sequentially on a node, then the computed values are distributed to the sub-nodes. Thus, the evaluation of the construction function is evaluated within locality b. The locality of the effects of the first argument must be acceptable in b. The results of the sequential executions

are going to be serialised in order to be communicated. The output value is thus a parallel vector with the type τ'_c.

The behaviour of the **fst** and **snd** operators is pretty straightforward: the latent effect is directly related to the locality of the given pair (constrained by the typing rules).

To illustrate common operators, we describes the (+) operator used for integer additions. Here, the latent effects are arbitrarily set to `c` as the operator is harmless. The localities of the integers given to the function are left as free variables as they will be constrained during the evaluation of the application (APP) rule.

Now, we need to define the typing rules that are used to guarantee the safe evaluation of a MULTI-ML expression. The typing rules are defined in Figure 4.24, 4.25 and 4.26. The figures are, respectivelly, corresponding to generic rule (rules that can be evaluated everywhere such as let-bindings, application, constants, *etc.*), BSP rules (evaluation takes place at level `b` or `c/l`) and MULTI-BSP rules (corresponding to the `m` level).

First, we describe the generic rules given in Figure 4.24.

The VAR rule allows to type a value already bound in the environment. The locality π of the variable must be acceptable in the current environment Λ as there is no *re-binding* of the type τ_π to the current environment.

The CST and OP rules works in a similar way.

The PAIR and IF_THEN_ELSE rules are pretty straightforward. The PAIR rule introduces a constraint that is used to determine the locality of the final type. To do so, we use the **Propgt** predicate to get the prevailing locality. The IF_THEN_ELSE rule works in a similar way, the **Propgt** is used to determine the effect of the expression using both the effect produce by the condition e_1 and the expression e_2 and e_3. As expected, the types and locality of e_2 and e_3 must be identical.

LET IN is used to generalise the value e_1 inside e_2. To do so, we call the predicate **Weak** to generalise the type and to try to erase harmless effects, or spread harmful ones. As explained in the Definition 4.13, this manipulation is used to relax the let binding rule and to avoid zealous restrictions. However, the effects emitted by the evaluation of e_1 and e_2 are propagated with Ψ.

LET REC is the standard rule for recursive bindings. This rules behave like the regular LET IN except that it adds the type of f in the environment of e_2, to be able to make the recursive call. Likewise LET IN, we try to weaken the type of f to relax the binding.

DEFFUN is going to escape the current locality Λ by evaluating e in a free locality: Λ'. This locality bypassing allows to declare function that is not applicable in its definition context. As said previously, it is important to be able to declare BSP code outside a node code (level `b`), to declare, for example, a library of BSP communication functions. However, the function definition is constrainted by $\varepsilon \blacktriangleleft \Lambda$ to avoid inconsistent definition such as declaring a multi-function inside a parallel vector. The output arrow type is annotated by the effect generated by the evaluation of the body of the function and the value is bound to the current context: Λ.

The APP rule generate a lot of constraints in favor of the respect of the localities of the operands. We need to assure that the locality of the given argument is valid regarding the type of

$$\text{VAR}\quad \frac{\Gamma(x) = \tau_\pi/\varepsilon\ [c] \quad c_1 = [\pi \lhd \Lambda, c]}{\Lambda, \Gamma \vdash x : \tau_\pi/\varepsilon\ [c_1]} \qquad \text{CST}\quad \frac{\vdash cst\ : \tau_\pi/\varepsilon\ [c] \quad c_1 = [\pi \lhd \Lambda, c]}{\Lambda, \Gamma \vdash cst : \tau_\pi/\varepsilon\ [c_1]} \qquad \text{OP}\quad \frac{\Gamma(\mathbf{op}) = \tau_\pi/\varepsilon\ [c] \quad c_1 = [\pi \lhd \Lambda, c]}{\Lambda, \Gamma \vdash \mathbf{op} : \tau_\pi/\varepsilon\ [c_1]}$$

$$\text{PAIR}\quad \frac{\begin{array}{l}\Lambda, \Gamma \vdash e_1 : \tau^1_{\pi_1}/\varepsilon_1\ [c_1] \\ \Lambda, \Gamma \vdash e_2 : \tau^2_{\pi_2}/\varepsilon_2\ [c_2] \\ c_3 = [\Psi = \mathbf{Propgt}(\varepsilon_1, \varepsilon_2), c_1, c_2]\end{array}}{\Lambda, \Gamma \vdash (e_1, e_2) : (\tau^1_{\pi_1} \times \tau^2_{\pi_2})_\Psi/\Psi\ [c_3]}$$

$$\text{IF THEN ELSE}\quad \frac{\begin{array}{l}\Lambda, \Gamma \vdash e_1 : \mathtt{Bool}_{\pi_1}/\varepsilon_1\ [c_1] \\ \Lambda, \Gamma \vdash e_2 : \tau_{\pi_2}/\varepsilon_2\ [c_2] \\ \Lambda, \Gamma \vdash e_3 : \tau_{\pi_2}/\varepsilon_2\ [c_3] \\ c_4 = [\Psi = \mathbf{Propgt}(\varepsilon_1, \varepsilon_2), c_1, c_2, c_3]\end{array}}{\Lambda, \Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ : \tau_{\pi_2}/\Psi\ [c_4]}$$

$$\text{LET IN}\quad \frac{\begin{array}{l}\Lambda, \Gamma \vdash e_1 : \tau^1_{\pi_1}/\varepsilon_1\ [c_1] \\ \Lambda, \Gamma; x : \mathbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1) \vdash e_2 : \tau^2_{\pi_2}/\varepsilon_2\ [c_2] \\ c_3 \equiv [\Psi = \mathbf{Propgt}(\varepsilon_1, \varepsilon_2), c_1, c_2]\end{array}}{\Lambda, \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau^2_{\pi_2}/\Psi\ [c_3]}$$

$$\text{LET REC}\quad \frac{\begin{array}{l}\Lambda', \Gamma; f : (\tau^1_{\pi_1} \xrightarrow{\varepsilon_1} \tau^2_{\pi_2})_{\Lambda'}/\varepsilon_2; x : \tau^1_{\pi_1}/\varepsilon_3 \vdash e_1 : \tau^2_{\pi_2}/\varepsilon_1\ [c_1] \\ \Lambda, \Gamma; f : \mathbf{Weak}((\tau^1_{\pi_1} \xrightarrow{\varepsilon_1} \tau^2_{\pi_2})_{\Lambda'}, \varepsilon_2) \vdash e_2 : \tau^3_{\pi_3}/\varepsilon_4\ [c_2] \\ c_3 = [\Psi = \mathbf{Propgt}(\varepsilon_2, \varepsilon_4), c_1, c_2]\end{array}}{\Lambda, \Gamma \vdash \mathbf{let\ rec}\ f\ x = e_1\ \mathbf{in}\ e2 : \tau^3_{\pi_3}/\Psi\ [c_3]}$$

$$\text{DEFFUN}\quad \frac{\begin{array}{l}\Lambda', \Gamma; x : \tau^1_{\pi_1}/\varepsilon_1 \vdash e : \tau^2_{\pi_2}/\varepsilon_2\ [c_1] \\ c_2 = [\varepsilon_2 \blacktriangleleft \Lambda, \pi_1 \lhd \varepsilon_2, c_1]\end{array}}{\Lambda, \Gamma \vdash \mathbf{fun}\ x\ \ \rightarrow\ \ e : (\tau^1_{\pi_1} \xrightarrow{\varepsilon_2} \tau^2_{\pi_2})_\Lambda/\varepsilon_2\ [c_2]}$$

$$\text{APP}\quad \frac{\begin{array}{l}\Lambda, \Gamma \vdash f : (\tau^1_{\pi_1} \xrightarrow{\varepsilon_1} \tau^2_{\pi_2})_{\Lambda'}/\varepsilon_2\ [c_1] \\ \Lambda, \Gamma \vdash e : \tau^1_{\pi_3}/\varepsilon_3\ [c_2] \\ c_3 = [\Psi = \mathbf{Propgt}(\varepsilon_1, \mathbf{Propgt}(\varepsilon_2, \varepsilon_3)), \pi_3 \lhd \pi_1, \varepsilon_1 \lhd \Lambda, \pi_2 \lhd \Lambda, \varepsilon_1 \blacktriangleleft \Lambda, c_1, c_2]\end{array}}{\Lambda, \Gamma \vdash f\ \ e : \tau^2_\Lambda/\Psi\ [c_3]}$$

Figure 4.24: Inductive rules - Generic level.

$$\text{VECTOR}\quad \frac{\begin{array}{c}\mathtt{c},\Gamma \vdash e : \tau_\pi/\varepsilon \;\; [c_1] \\ c_2 = [\varepsilon \lhd \mathtt{c}, \pi \lhd \mathtt{c}, c_1]\end{array}}{\mathtt{b},\Gamma \vdash << e >> : \tau_\pi \; \mathtt{par_b}/\mathtt{b} \;\; [c_2]}$$

$$\text{COPY}\quad \frac{\begin{array}{c}\Gamma(x) = \tau_\mathtt{b}/\varepsilon[c_1] \\ c_2 = [\tau_\pi = \mathbf{Seria}_\mathtt{c}(\tau_\mathtt{b}), c_1]\end{array}}{\mathtt{c},\Gamma \vdash \#x\# : \tau_\pi/\mathtt{l} \;\; [c_2]}$$

$$\text{OPEN}\quad \frac{\Gamma(x) = \tau_\pi \; \mathtt{par_b}/\varepsilon[c_1]}{\mathtt{c},\Gamma \vdash \$x\$: \tau_\pi/\mathtt{l} \;\; [c_1]}$$

$$\text{OP'}\quad \frac{\begin{array}{c}MmlPrimitives(\mathbf{op'}) \quad \Gamma(\mathbf{op'}) = (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_{\Lambda'}/\mathtt{b} \;\; [c_1] \\ \mathtt{b},\Gamma \vdash e : \tau^1_{\pi_3}/\varepsilon_2 \;\; [c_2] \quad c_3 = [\pi_3 \lhd \mathtt{b}, \varepsilon \lhd \mathtt{b}, \pi_2 \lhd \mathtt{b}, c_1, c_2]\end{array}}{\mathtt{b},\Gamma \vdash \mathbf{op'} \;\; e : \tau^2_{\pi_2}/\mathtt{b} \;\; [c_3]}$$

$$\text{REPLICATE}\quad \frac{\begin{array}{c}\mathtt{b},\Gamma \vdash f : (\mathtt{unit}_{\pi_1} \xrightarrow{\varepsilon_1} \tau^2_{\pi_2})_{\Lambda'}/\varepsilon_1 \;\; [c_1] \\ c_2 = [\varepsilon_1 \lhd \mathtt{c}, \pi_2 \lhd \mathtt{c}, c_1]\end{array}}{\mathtt{b},\Gamma \vdash \mathbf{replicate} \;\; f : \tau^2_{\pi_2} \; \mathtt{par_b}/\mathtt{b} \;\; [c_2]}$$

$$\text{DOWN}\quad \frac{\begin{array}{c}\Gamma(x) = \tau_\mathtt{b}/\varepsilon[c_1] \\ c_2 = [\tau_\pi = \mathbf{Seria}_\mathtt{c}(\tau_\mathtt{b}), c_1]\end{array}}{\mathtt{c},\Gamma \vdash \mathbf{down} \;\; x : \tau_\pi \; \mathtt{par_b}/\mathtt{l} \;\; [c_2]}$$

$$\text{APPLY}\quad \frac{\begin{array}{c}\mathtt{b},\Gamma \vdash e_1 : (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_\mathtt{b}/\mathtt{b} \;\; [c_1] \\ \mathtt{b},\Gamma \vdash e_2 : \tau^1_{\pi_3} \; \mathtt{par_b}/\mathtt{b} \;\; [c_2] \\ c_3 = [\pi_3 \lhd \pi_1, \varepsilon \lhd \mathtt{c}, \pi_2 \lhd \mathtt{c}, c_1, c_2]\end{array}}{\mathtt{b},\Gamma \vdash \mathbf{apply} \;\; e_1 \;\; e_2 : \tau^2_{\pi_2} \; \mathtt{par_b}/\mathtt{l} \;\; [c_3]}$$

Figure 4.25: Inductive rules - BSP level.

$$\text{MULTI FUN}\quad \frac{\begin{array}{c}\mathtt{b},\Gamma; x : \tau^1_\mathtt{b}/\varepsilon; f : (\tau^1_{\pi_1} \xrightarrow{\mathtt{l}} \tau^2_\mathtt{c})_\mathtt{l}/\mathtt{l} \vdash e_1 : \tau^3_{\pi_3}/\varepsilon_1 \;\; [c_1] \\ \mathtt{s},\Gamma; x : \tau^1_\mathtt{s}/\varepsilon \vdash e_2 : \tau^3_{\pi_4}/\varepsilon_2 \;\; [c_2] \\ c_3 = [\tau^3_\mathtt{m} = \mathbf{Seria}_\mathtt{m}(\tau^3_{\pi_3}), \tau^3_\mathtt{c} = \mathbf{Seria}_\mathtt{c}(\tau^3_{\pi_4}), c1, c2]\end{array}}{\mathtt{m},\Gamma \vdash \mathbf{multi} \;\; f \;\; x \to e_1 \dagger e_2 : (\tau^1_\mathtt{m} \xrightarrow{\mathtt{m}} \tau^3_\mathtt{m})_\mathtt{m}/\mathtt{m} \;\; [c_3]}$$

Figure 4.26: Inductive rules - MULTI-BSP level.

f using $\pi_3 \lhd \pi_1$. The effect thrown by the application must be acceptable in the current locality ($\varepsilon_1 \lhd \Lambda$). This constraint is very important and rejects, for example, the application of BSP primitives outside the scope of a node. Obviously, we also check that the locality π_2 of the type resulting in the application of the function is acceptable in the current locality Λ. The constraint $\varepsilon_1 \blacktriangleleft \Lambda$ is, in particular, used to check if the application of the function is possible. For example, it avoid to evaluate a multi-function (annotated `-(m)->`) in a leaf, as $m \blacktriangleleft b$ is `False`. This definability constraint is also necessary with references as detailed in Section 4.4.1.1. Finally, Ψ is the propagation of the effects emitted by the evaluation of e, f and its arrow effect. It is also important to note that the locality of the outgoing type is bound to the current locality Λ.

Then, we describe the BSP rules given in Figure 4.25.

The VECTOR rule is necessary to generate well formed vectors. We need to check that the locality of τ and its effect are acceptable inside the vector (*i.e.* `c`). As parallel vectors can only take place at a BSP level (`b`), the rule forces the evaluation locality to be `b`. As expected, the output type is a parallel vector of type τ_π $\texttt{par}_\texttt{b}$ where τ_π is directly obtained by the evaluation of e.

COPY is the operator related to the syntactic sugar $\#x\#$. It allows to access, within the scope of a parallel vector (`c`), a value which is defined in the memory of the upper-node. As expected, the value annotated by the sharps must be of locality `b`. As the value x is communicated from `b` to `c`, we must serialise it using the **Seria** predicate.

Similarly, OPEN is related to the $\$x\$$ piece of notation. It is used to access, within the scope of a parallel vector, to an element of a parallel vector. Thus, x must be of type τ_π $\texttt{par}_\texttt{b}$.

Both COPY and OPEN emit an effect of locality `l` to restrict their communication.

The OP' rule is close to the OP rule but is dedicated to the MULTI-ML primitives. It aims to reduce the number of typing rules, by introducing a generic one, dedicated to MULTI-ML primitives. The $MmlPrimitives(\mathbf{op}')$ match the set of synchronous and asynchronous parallel primitives: **`proj`**, **`put`** and **`mkpar`**. The definition of $MmlPrimitives(e)$ can be found in Figure 4.27. As those primitives are defined in the initial environment, we just need to get their type to check whether or not the type of e is valid.

$$MmlPrimitives(e) = \begin{cases} \texttt{True}, & if\ e \equiv \textbf{put} \mid \textbf{proj} \mid \textbf{mkpar} \\ \texttt{False}, & otherwise \end{cases}$$

Figure 4.27: The MmlPrimitives(e) predicate.

The REPLICATE, DOWN and APPLY rules are part of the core primitives of the MULTI-ML language. Those rules are necessary to prove the coherency of our typing system. We recall that those rules are not available in the MULTI-ML syntax and thus, cannot be used by the programmer.

The REPLICATE rule is the core primitives which builds a parallel vector from an expression f of the form (**fun** _ $\rightarrow$ e). We recall that this particular expression is generated by the syntactic sugar transformation. As the function f aims to be evaluated inside the scope of a parallel vector, it must be acceptable within `c`. As expected, the output type is a parallel vector typed: $\tau^2_{\pi_2}$ $\texttt{par}_\texttt{b}$.

The DOWN rule is the core primitives which builds a parallel vector from a variable x. It aims to "copy" x inside the memory of all the sub-nodes of the current node. As expected, it works similarly to the COPY rule.

The APPLY rule is the core primitives which aims to manipulate the construction of parallel vectors using both **`replicate`** and **`down`**. Like in BSML, **`apply`** takes e_1 and e_2, where e_1 is a parallel vector of functions which is applied on a parallel vector of values: e_2.

$$\text{MULTI TREE FUN}\quad \frac{\begin{array}{l}\mathtt{b},\Gamma; x:\tau^1_{\mathtt{b}}/\varepsilon; f:(\tau^1_{\pi_1} \xrightarrow{\mathtt{l}} \tau^2_{\mathtt{c}}\ \mathtt{tree}_{\mathtt{l}})_{\mathtt{l}}/\mathtt{l} \vdash e_1 : \tau^2_{\pi_2}\ \mathtt{tree}_{\mathtt{l}}\ \mathtt{par}_{\mathtt{b}} \times \tau^2_{\pi_3}/\varepsilon_2\ \ [c_1]\\ \mathtt{s},\Gamma; x:\tau^1_{\mathtt{s}}/\varepsilon \vdash e_2 : \tau^2_{\pi_4}/\varepsilon_3\ \ [c_2]\\ c_3 = [\tau^2_{\mathtt{c}} = \mathbf{Seria}_{\mathtt{c}}(\tau^2_{\pi_4}), c1, c2]\end{array}}{\mathtt{m},\Gamma \vdash \mathbf{multi\ \ tree}\ \ f\ \ x \to e_1 \dagger e_2 : (\tau^1_{\mathtt{m}} \xrightarrow{\mathtt{m}} \tau^2_{\mathtt{c}}\ \mathtt{tree}_{\mathtt{m}})_{\mathtt{m}}/\mathtt{m}\ \ [c_3]}$$

Figure 4.28: The MULTI TREE FUN inference rule.

Finally, we describe the MULTI-BSP rule given in Figure 4.26, which is basically the multi-function definition.

MULTI FUN is the rule that defines a multi-function. It behaves like the declaration of a recursive function: by injecting the type of the function inside the environment of the node code. This recursive multi-function has the latent effect `l` to prevent communications and to force the application to be within the scope of a parallel vector. Its type is similar to the type of the multi-function itself, expect the value of the latent effect and the locality of the types. As expected, the level of evaluation of e_1 (*i.e.* the node code) is `b` (BSP) and e_2 (*i.e.* the leaf code) is `s` (sequential). The rule constraints the evaluation of the multi-function to be at level **`multi`**, in order to restrict the definition context of multi-functions. Concerning the values returned by both nodes and leaves, we need to transform them using the predicate **Seria**. Indeed, it is important to verify that the values that are going to be given as a result are communicable to the upper level. Namely, from `s` to `c` (when the value is communicated from a leaf to the body of the vector defined in the upper node), from `b` to `c` (when the value is communicated from a node to the body of the vector defined in the upper node, *i.e.* during the recursive call of the multi-function) and from `b` to `m` (when the value is communicated from the root node to the multi level).

As said previously, the definition of the rules handling trees are not detailed. However, we give the intuition on the rule used to build a tree, the MULTI TREE FUN, as the MULTI-ML language currently implements the tree construction. The inference rule can be found in Figure 4.28.

The MULTI TREE FUN rule allows to define multi-tree-functions, and is similar to the MULTI FUN rule. The principal difference remains in its capacity of building a tree. To do so, the node type (e_1) must the be a pair containing the sub-tree ($\tau^2_{\pi_2}\ \mathtt{tree}_{\mathtt{l}}\ \mathtt{par}_{\mathtt{b}}$) and the value of the current node ($\tau^2_{\mathtt{b}}$). As expected, the resulting value is a function that build a tree, typed: $(\tau^1_{\mathtt{m}} \xrightarrow{\mathtt{m}} \tau^2_{\mathtt{c}}\ \mathtt{tree}_{\mathtt{m}})_{\mathtt{m}}$.

4.3.4 Examples of rule derivation

In this section, we propose several rule derivation to show how the inference rules behave. To do so, we chose some typical expressions which illustrate the specificites of the MULTI-ML language. To lighten the rules, we assume that the unsolved constraints of the premises are implicitly kept into the conclusion's constraints. In the final conclusion, we inject the unsolved constraints of the premises to get the complete type representation.

First, we derive an expression which aims to define a function dealing with parallel vectors within the MULTI-BSP level. We recall that such a feature is essential to define BSP libraries that can be used on node codes. The MULTI-ML typing system allows to write such *cross-definitions* under some constraints. In Figure 4.29, we propose the derivation of the expression `let f = fun x -> << 1 >> in f`. The type inference of the expression takes place, at level `m`. As expected, the type inference holds as the expression is valid. The type of the given expression is thus : $(\tau^1_{\pi_1} \xrightarrow{\mathtt{b}} \mathtt{int}_{\mathtt{c}}\ \mathtt{par}_{\mathtt{b}})_{\mathtt{m}}/\mathtt{b}\ \ [\pi_1 \lhd \mathtt{b}]$ where $\pi_1 \lhd \mathtt{b}$ stands for an unsolved constraint which is *attached* to the type, until the application of the function raise a value for π_1. This derivation

illustrates how the *definability* predicate works.

Then, we propose the derivation of the syntactic sugar introduced in the MULTI-ML syntax: the $ notation. We recall that, within the scope of a parallel vector, this notation aims to *copy* a value defined in the memory of a node into the memory of its sub-nodes. The derivation of the expression `<< #x# >>` can be found in Figure 4.30. To reduce the size of the derivation, we assume that the expression is directly inferred within the locality of a node (b) with an environment containing $\{x : \texttt{int}_\texttt{b}/\texttt{b}\}$. We observe that the $\mathbf{Seria}_\texttt{c}(\texttt{int}_\texttt{b})$ is used to *serialise* an $\texttt{int}_\texttt{b}$ to an $\texttt{int}_\texttt{c}$. The type infered by the rules is, as expected, $\texttt{int}_\texttt{c}\ \texttt{par}_\texttt{b}/\texttt{b}$.

Finally, a more complex derivation is proposed in Figure 4.31 to show how the attempt to communicate a non-communicable value is rejected by the typing system. To do so, we try to find a type for the expression `proj << fun x -> #y# >>`. As explained in Section 4.3.5, the expression `fun x -> #y#` is not communicable. Thus, the application of the **proj** primitive on such an expression must be forbidden. As previously, we begin the evaluation within the scope of a node with an initial environment containing $\{y : \texttt{int}_\texttt{b}/\texttt{b}\}$. On the left hand part of the rule, we yield the type of the **proj** operator and its constraints. On the right hand side of the rule, we derive the expression `<< fun x -> #y# >>`, which is similar to the Figure 4.30, expect from the environment. We can see that the expression is rejected during the application of **proj** on the parallel vector, because of the constraint of **proj** which consists in $\mathbf{Seria}_\texttt{b}((\tau^3_{\pi_3} \xrightarrow{\texttt{l}} \texttt{int}_\texttt{c})_\texttt{c})$. Here we try to *serialise* a functional value with a latent effect `l` to the `b` level: something prohibited. Indeed, the latent effect `l` was introduced to prevent the communication of such values. The type inference fails and produce typing error.

4.3.5 Discussions on closure communication

A closure (or lexical closure) refers to an expression where free variables have been bound in the environment, resulting to a closed expression. The communication of such a term forces to serialise all the values needed by the closure, that is to say: copy the minimal environment inside the closed term. Even if it is quite complex to implement, there are no problem to use this kind of techniques on *regular* values. On the contrary, with parallel values, it is not the same thing.

For example, the code `<< fun x -> #y# >>` where `y` is a well defined value in the node memory. Here, we have the case of a closure that uses a value, `y`, which is not defined at the same level. The mechanism of closure could work by binding `y` inside the closure. As `y` is not at the same level of the function declaration, the binding the value of `y` implies a communication: from the memory of the node to the memory of the sub-node. In terms of implementation, there is no problem for doing such a thing. However, it adds *hidden* communications which are disturbing for the programmer and it is not satisfying regarding the cost system. Furthermore, as the code is not executed where the closure is made but where it is evaluated; such a code could be ambiguous concerning the evaluation. The closure of this expression explicitly says that the COPY of a value can be executed somewhere else, regardless of its locality. As it is ambiguous, we choose to refuse the communication of COPY (and also OPEN) by adding the effect `l` when the primitive is evaluated.

4.3.6 Type soundness

To respect the well known slogan: *well-typed programs cannot "go wrong"* [116], we must prove that programs containing a type error are not typable. That is to say, if an expression e is irreducible due to a type error, the type system must not be able to assign a type to such an

$$\text{LET}\ \frac{\text{FUN}\ \dfrac{\text{VECTOR}\ \dfrac{\text{CST}\ \dfrac{\vdash 1 : \mathtt{int_c}/\mathtt{c}\ [\ ]}{\mathtt{c}, \{x : \tau^1_{\pi_1}/\varepsilon_1\} \vdash 1 : \mathtt{int_c}/\mathtt{c}\ [\mathtt{c} \lhd \mathtt{c}]}}{\mathtt{b}, \{x : \tau^1_{\pi_1}/\varepsilon_1\} \vdash \ll 1 \gg : \mathtt{int_c}\ \mathtt{par_b}/\mathtt{b}\ [c \lhd c, c \lhd c]}}{\mathtt{m}, \{\} \vdash \mathbf{fun}\ x \to \ll 1 \gg\ : (\tau^1_{\pi_1} \xrightarrow{\mathtt{b}} \mathtt{int_c}\ \mathtt{par_b})_{\mathtt{m}}/\mathtt{b}\ [\mathtt{b} \blacktriangleleft \mathtt{m}, \pi_1 \lhd \mathtt{b}]}\quad \text{VAR}\ \frac{\Gamma(f) \equiv (\tau^1_{\pi_1} \xrightarrow{\mathtt{b}} \mathtt{int_c}\ \mathtt{par_b})_{\mathtt{m}}/\mathtt{b}\ [\ ]}{\mathtt{m}, \{f : \mathbf{Weak}((\tau^1_{\pi_1} \xrightarrow{\mathtt{b}} \mathtt{int_c}\ \mathtt{par_b})_{\mathtt{m}}, \mathtt{b})\} \vdash f : (\tau^1_{\pi_1} \xrightarrow{\mathtt{b}} \mathtt{int_c}\ \mathtt{par_b})_{\mathtt{m}}/\mathtt{b}\ [\mathtt{m} \lhd \mathtt{m}]}}{\mathtt{m}, \{\} \vdash \mathbf{let}\ f = \mathbf{fun}\ x \to \ll 1 \gg\ \mathbf{in}\ f : (\tau^1_{\pi_1} \xrightarrow{\mathtt{b}} \mathtt{int_c}\ \mathtt{par_b})_{\mathtt{m}}/\Psi\ [\Psi = \mathbf{Propgt}(\mathtt{b}, \mathtt{b}), \pi_1 \lhd \mathtt{b}]}$$

Figure 4.29: BSP function derivation.

$$\text{VECTOR}\ \frac{\text{COPY}\ \dfrac{\Gamma(x) \equiv \mathtt{int_b}/\mathtt{b}\ [\ ]}{\mathtt{b}, \{x : \mathtt{int_b}/\mathtt{b}\} \vdash \#x\# : \tau^1_{\pi_1}/\mathtt{l}\ [\tau^1_{\pi_1} = \mathbf{Seria}_{\mathtt{c}}(\mathtt{int_b})]}}{\mathtt{b}, \{x : \mathtt{int_b}/\mathtt{b}\} \vdash \ll \#x\# \gg : \mathtt{int_c}\ \mathtt{par_b}/\mathtt{b}\ [\mathtt{l} \lhd \mathtt{c}, \mathtt{c} \lhd \mathtt{c}]}$$

Figure 4.30: Parallel vector derivation.

$$\text{OP'}\ \frac{\Gamma(\mathbf{proj}) = (\tau^1_{\pi_1}\ \mathtt{par_b} \xrightarrow{b} (\mathtt{int_c} \xrightarrow{\mathtt{c}} \tau^2_{\pi 2})_{\mathtt{b}})_{\mathtt{m}}/\mathtt{b}\ [\tau^2_{\pi_2} = \mathbf{Seria}_{\mathtt{b}}(\tau^1_{\pi_1})]\quad \text{VECTOR}\ \dfrac{\text{FUN}\ \dfrac{\text{COPY}\ \dfrac{\Gamma(y) \equiv \mathtt{int_b}/\mathtt{b}\ [\ ]}{\mathtt{c}, \{y : \mathtt{int_b}/\mathtt{b}; x : \tau^3_{\pi_3}/\varepsilon_3\} \vdash \#y\# : \tau^3_{\pi_3}/\mathtt{l}\ [\tau^3_{\pi_3} = \mathbf{Seria}_{\mathtt{c}}(\mathtt{int_b})]}}{\mathtt{c}, \{y : \mathtt{int_b}/\mathtt{b}\} \vdash \mathbf{fun}\ x \to \#y\#\ : (\tau^3_{\pi_3} \xrightarrow{\mathtt{l}} \mathtt{int_c})_{\mathtt{c}}/\mathtt{l}\ [\mathtt{l} \blacktriangleleft \mathtt{c}, \pi_3 \lhd \mathtt{l}]}}{\mathtt{b}, \{y : \mathtt{int_b}/\mathtt{b}\} \vdash \ll \mathbf{fun}\ x \to \#y\#\ \gg : (\tau^3_{\pi_3} \xrightarrow{\mathtt{l}} \mathtt{int_c})_{\mathtt{c}}\ \mathtt{par_b}/\mathtt{b}\ [\mathtt{l} \lhd \mathtt{c}, \mathtt{c} \lhd \mathtt{c}]}}{\mathtt{b}, \{y : \mathtt{int_b}/\mathtt{b}\} \vdash \mathbf{proj} \ll \mathbf{fun}\ x \to \#y\#\ \gg : \mathtt{unsound}/\mathtt{b}\ [\pi_3 \lhd \mathtt{l}, \mathtt{c} \lhd \mathtt{b}, \mathtt{b} \lhd \mathtt{b}, \mathtt{b} \lhd \mathtt{b}, \tau^2_{\pi_2} = \mathbf{Seria}_{\mathtt{b}}((\tau^3_{\pi_3} \xrightarrow{\mathtt{l}} \mathtt{int_c})_{\mathtt{c}})] = \mathtt{Fail}}$$

Figure 4.31: Derivation of a non-communicable vector projection.

expression. We choose to follow the *syntactic approach* proposed in [154] to establish the type soundness.

In the following, we propose the proof body of the type soundness. It is important to note that the proof was *not* mechanised *nor* formally proved using the COQ proof assistant. It is an ongoing work. First, we prove the validity of expression within their context, which is necessary to ensure safe locality manipulation and avoid unsound parallel structure nesting, for example.

Lemma 4.16 (*Validity of localities within the evaluation context*)

If $\Lambda, \Gamma \vdash e : \tau_\pi / \varepsilon \ \ [c]$ *then* $[c]$ *holds.*

Proof: See Appendix A.2.1. ■

The aim of the type system of a programming language is not to ensure termination, but to guarantee the structural conformity of data. We are therefore going to show that the evaluation of a term with type τ_π, if it terminates, returns a value that, semantically, does belong to the type τ_π. We must therefore define precisely what does it means for a value to *semantically belong to a type*. To do so, we define the following predicates:

(1) $\models v : \tau_\pi$ the value v belongs to the type τ_π,

(2) $\models v : \sigma$ the value v belongs to the scheme σ,

(3) $\models \mathcal{M} : \Gamma$ all the values contained in the evaluation environment $\mathcal{M}$ belong to the corresponding type schemes in Γ.

These predicates are defined as follows:

- $\models const : \texttt{Base}$ and $TC(const) = \texttt{Base}$
- $\models op : \sigma$ and $\sigma = TC(op)$
- $\models v : \sigma$ if for all types τ_π such that $\tau_\pi \leq \sigma$, we have $\models v : \tau_\pi$
- $\models \mathcal{M} : \Gamma$ if $Dom(\mathcal{M}) = Dom(\Gamma)$ and for all $x \in Dom(\mathcal{M})$ we have $\models \mathcal{M}(x) : \Gamma(x)$
- $\models \overline{(\textbf{fun}\ \ x \to e\ )[\mathcal{M}]} : (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_\Lambda$ if there exists a typing environment Γ such that $\models \mathcal{M} : \Gamma$ and $\Lambda, \Gamma \vdash (\textbf{fun}\ \ x \to e\ ) : (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_\Lambda$
- $\models \overline{(\textbf{multi}\ \ f\ \ x \to e_1 \dagger e_2)[\mathcal{M}]} : (\tau^1_{\mathtt{m}} \xrightarrow{\mathtt{m}} \tau^2_{\mathtt{m}})_{\mathtt{m}}$ if there exists a typing environment Γ such that $\models \mathcal{M} : \Gamma$ and $\mathtt{m}, \Gamma \vdash (\textbf{multi}\ \ f\ \ x \to e_1 \dagger e_2) : (\tau^1_{\mathtt{m}} \xrightarrow{\mathtt{m}} \tau^2_{\mathtt{m}})_{\mathtt{m}}$
- $\models \ \ << e, ..., e >> : \tau_\pi\ \mathtt{par}_\mathtt{b}$ if there exists a typing environment Γ such that $\models \mathcal{M} : \Gamma$ and $\mathtt{b}, \Gamma \vdash << e, ..., e >> : \tau_\pi\ \mathtt{par}_\mathtt{b}$

In the following, to ease the reasoning on type scheme, we use the notation $\forall \overrightarrow{\alpha}\ \overrightarrow{\lambda}[c].\tau_\pi$ to stand that, for all sets of type α and for all sets of locality λ, the type τ_π respects the constraint c. Furthermore, the notation $\exists \overrightarrow{\alpha}\ \overrightarrow{\lambda}.c$ denotes the existence of a constraint c based on $\overrightarrow{\alpha}\ \overrightarrow{\lambda}$. Those constructions are not available in the grammar of the constraints, they just aims to make the proofs more readable.

Hypothesis 4.17 (*Well typed operators and constants*)

H0 *For all operators* **op** *within locality* Λ, $TC(\mathbf{op})$ *is of the form* $\forall \overrightarrow{\alpha}\,\overrightarrow{\lambda}.(\tau_\pi \xrightarrow{\varepsilon} \tau'_{\pi'})_{\Lambda'}$ *with* $\mathcal{F}(TC(\mathbf{op})) = \emptyset$*; For all constants* **cst** *and for all localities,* $TC(\mathbf{cst})$*, is a valid type of* `Base`

H1 *If* $\Lambda, \Gamma \vdash v : \tau_\pi/\varepsilon$ *and, within locality* Λ, $TC(op) = \forall \overrightarrow{\alpha}\,\overrightarrow{\lambda}.(\tau^1_{\pi_1} \xrightarrow{\varepsilon_1} \tau^2_{\pi_2})_{\Lambda'}$, $\tau_\pi \leq \forall \overrightarrow{\alpha}\,\overrightarrow{\lambda}.\tau^1_{\pi_1}$, $\varepsilon_1 \lhd \Lambda$ *and* $\Lambda' \lhd \Lambda$ *then there exists* $v' \equiv \overline{op\ v}$ *such that* $\models v : \forall \overrightarrow{\alpha}\,\overrightarrow{\lambda}.\tau^2_{\pi_2}/\varepsilon_2$

Lemma 4.18 (*Typing is stable under substitution*)

Under the following hypothesis which defines the substitution:

- $\Lambda, \Gamma; x : \forall \overrightarrow{\alpha}\,\overrightarrow{\lambda}[c_1].\tau^1_{\pi_1} \vdash e : \tau^2_{\pi_2}/\varepsilon \ \ [c_2; \exists \overrightarrow{\alpha}\,\overrightarrow{\lambda}.c_1]$
- $\Lambda, \Gamma \vdash v : \tau^1_{\pi_1}/\varepsilon \ \ [c_1; c_2]$
- $\overrightarrow{\alpha}\,\overrightarrow{\lambda} \cap \mathcal{F}(c_2) \cap \mathcal{F}(\Gamma) = \emptyset$

We have: $\Lambda, \Gamma \vdash \{v/x\}e : \tau^2_{\pi_2}/\varepsilon \ \ [\exists \overrightarrow{\alpha}\,\overrightarrow{\lambda}.c_1; c_2]$ *where* $\{v/x\}e$ *stands for the substitution of* x *by* v *in expression* e*.*

Proof: See Appendix A.2.1. ■

Proposition 4.19 (*Semantic generalisation*)

Let v *be a value and* τ_π *be a type such that* $\models v : \tau_\pi$*. Then, for all substitution* φ*, we have* $\models v : \varphi(\tau_\pi)$*.*

Proof: See Appendix A.2.1. ■

We now need to prove that an expression is well-formed after each evaluation steps. Thus, after a step, the previous set of constraint of a typing judgement is simplified. By definition, as $\Lambda, \Gamma \vdash e : \tau_\pi/\varepsilon \ \ [c]$ implies that $[c]$ holds at the current evaluation step, we do not need to handle the constraints. If a set of unsolved constraints remains (not enough information on types or localities) they are considered holding, as there they are not unsatisfied.

Lemma 4.20 (*Evaluation progress*)

Let e *be an expression,* τ_π *be a type,* Γ *be a typing environment,* $\mathcal{M}$ *be an evaluation environment such that* $\Lambda, \Gamma \vdash e : \forall \overrightarrow{\alpha}\,\overrightarrow{\lambda}[c].\tau_\pi$ *and* $\models \mathcal{M} : \Gamma$*. If there exists a value* v *such that* $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p v$ *then* $\models v : \tau_\pi$

Proof: See Appendix A.2.1. ■

Lemma 4.21 (*Co-evaluation progress*)

Let e *be an expression,* τ_π *be a type,* Γ *be a typing environment,* $\mathcal{M}$ *be an evaluation environment such that* $\Lambda, \Gamma \vdash e : \tau_\pi$ *and* $\models \mathcal{M} : \Gamma$*. Let* v *be a value; if it is impossible to obtain* $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p v$*; Then* $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p \infty$*.*

Proof: See Appendix A.2.1. ■

To provide a more pleasant way to characterise *"programs that do not go wrong"*, we now use the relation $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \mathcal{S}afe$:

$$\frac{\exists v \ \textit{such that} \quad \mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \mathcal{S}afe} \qquad \frac{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \mathcal{S}afe}$$

Theorem 4.22 (*Type soundness*)

Let e be an expression, τ_π be a type, Γ be a typing environment, $\mathcal{M}$ be an evaluation environment such that $\Lambda, \Gamma \vdash e : \tau_\pi$ and $\models \mathcal{M} : \Gamma$. Then:

$$\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \mathcal{S}afe$$

Proof: By application of Lemma 4.2, we have two cases: (1) there exists a value v as a result; (2) there is no possible resulting value. By application of Lemma 4.20 with the resulting value v, we prove the first case. Otherwise, by application of Lemma 4.21, we obtain an infinite evaluation and thus prove the second case. ■

Finally, using Lemma 4.4, we can also prove that the two cases are mutually exclusive that is: any well-typed program can evaluate to a value that semantically belongs to the type or diverge; so without *"going wrong"*. Note that we do not have any information on ill-typed programs: some of them may diverge or evaluate well.

4.4 Extensions to imperative features and exceptions

In this section we propose two extensions of the μ**multi**-ML typing system. We propose, first, a way to handle a very useful imperative feature: references; and then, a way of managing exceptions.

Note that no proofs of the typing system's extensions has been written. As those features are standard to ML, we believe that the proofs are almost direct and mainly deals with technical issues.

4.4.1 A typing system for μ**multi**-ML^{ref}

In pure functional programming, it is not allowed to perform side effects. It is thus not possible to change the state of a value. However, in OCAML, the concept of *reference* is one of the most used imperative (or *impure*) features. Basically, a *reference* is a *pointer*. It is possible to define a *reference* to a value, and change this value along the program execution.

To declare a reference, we use the keyword **ref** which takes any type α and produce a value of type α **ref**. To modify the value of a given reference, we use the **set** operator, usually written `:=` in OCAML. The access to a reference is done using the **get** operator, denoted by `!` in OCAML. For example, we can write the following code, using the OCAML toplevel:

```
#let r = ref 1;;
val r : int ref = {contents = 1}

#let f = fun x -> r:=x in f 42;;
- : unit = ()

# !r;;
- : int = 42
```

As expected, references are not safe regarding to the different levels of locality of MULTI-ML. Indeed, changing the state of a value shared by many processes may cause unexpected behaviours without strong limitations. Thus, we need to handle references in the context of both parallel programming and multi-level definitions.

A reference cannot be modified regardless of its locality and the locality of the modification request. For example, the following code:

```
let rm = ref 1 in
let multi f _ =
 where node =
   let v = << rm := $pid$ >> in
   ...
 where leaf =
   ...
 in f ()
```

In this code, we are going to change the value of the reference `rm` (declared at level `m`) from a parallel vector (level `l`). Such a reference modification is not valid. In this case we are losing the replicated coherency and the value `rm` is corrupted because it is impossible to know, without global communications and synchronisations, the value written in `rm`. To avoid dealing with such concerns, we propose some restrictions regarding references management. The same problem is also existing in the BSML language and was study and detailed, with code examples, in [66].

4.4.1.1 Definitions

As said previously, we need to prohibit the modification requests of a reference bound in a level different from the request locality. To keep track of the references declarations localities, we choose to add a locality annotation to the reference itself. It is necessary to keep this locality declaration unchanged during the management of the references. Otherwise, we cannot keep in mind where the reference can be read or written. Thus, a reference is now written: τ_π $\mathbf{ref}_\Lambda$, where Λ stands for the locality of definition of the reference which *point* to a value of type τ_π.

The types of the operators that manipulate references are defined in Figure 4.32.

$$\Gamma(\mathbf{ref}) = \forall(\alpha, \pi, \Lambda).(\alpha_\pi \xrightarrow{\Lambda} \alpha_\pi\ \mathbf{ref}_\Lambda)_{\mathtt{m}}\ [\pi \lhd \Lambda]$$

$$\Gamma(\mathbf{get}) = \forall(\alpha, \pi, \Lambda_1, \Lambda_2).(\alpha_\pi\ \mathbf{ref}_{\Lambda_1} \xrightarrow{\Lambda_2} \alpha_{\Lambda_2})_{\mathtt{m}}\ [\pi \lhd \Lambda_2]$$

$$\Gamma(\mathbf{set}) = \forall(\alpha, \pi_1, \pi_2, \pi_3, \Lambda_1, \Lambda_2).(\alpha_{\pi_1}\ \mathbf{ref}_{\Lambda_1} \xrightarrow{\Lambda_2} (\alpha_{\pi_2} \xrightarrow{\Lambda_1} \mathtt{unit}_{\pi_3})_{\Lambda_2})_{\mathtt{m}}\ [\pi_1 \lhd \Lambda_2, \pi_2 \lhd \pi_1]$$

Figure 4.32: Operators of references.

The references *constructor* **ref** takes any type α annotated with locality π and produces a reference, with the same type, which is bound at level Λ, *i.e.* the current locality. The constraint forces the given argument to be acceptable in the current location.

Operator **get** reads the value contained in a reference $\alpha_{\pi}\mathbf{ref}_{\Lambda_1}$ and returns the pointed value, bound to the current level. As previously, the constraint forces the given argument to be acceptable in the current location.

The **set** operator is a little more complicated. We need to check if the locality of the reference is accessible from the level where the operator is applied. For example, let `rm` be a reference on an integer value at the MULTI-BSP level: `rm : int_m ref_m`. This reference cannot be modified from a level which is not the MULTI-BSP level, otherwise the replicated coherency will not be respected. To preserve the coherency, the **set** rule unifies the locality declaration of the reference with the pending effect of the function that effectively sets a value to the given reference. In other words, when applied to a reference, the **set** operator returns a function that takes a value and *set* it. The latent effect of this function is equal to the locality of the reference given in argument. Thus, this function must be applied at an acceptable locality only (thanks to the APP rule). As expected, the constraints force that the locality of the value sets in the reference to be acceptable: $\pi_2 \lhd \pi$.

It is important to note that the **set** operator endorses the usage of the *definability* predicate, written $\lambda_1 \blacktriangleleft \lambda_2$, used in the type constraints. In the context of typing with references, this predicate makes perfect sense. For example, in the APP rule (seeFigure 4.24), the constraint $\varepsilon \blacktriangleleft \Lambda$ is used to avoid an incoherent behaviour that could be raised by the **set** operator. Let `rm` be of type `int_m ref_m`, if we apply the **set** operator to the reference `rm` inside a node (level `b`): `(:=) rm`, we obtain a value of type `(int_`z -(m)-> unit_`y)_b` with the constraints `([m <| b, `z <| m])`. If we want to apply this function on a value on the node, we need to respect the constraint of acceptability of the pending effect of the function, which is `m` $\lhd$ *b*; here, the constraint is valid. Thus, if we do not add another constraint, we have the ability to modify, from a node, a reference defined at the MULTI-BSP level: something unacceptable. To avoid such a case, the constraint of definability of the APP rule checks if *the function could have been declared, regularly, at this level.* In this case, the definition of a functional value with a latent effect of locality `m` within the BSP level is rejected: the predicate $m \blacktriangleleft b$ is `False`.

4.4.1.2 Generalisation and references

It is necessary to avoid the generalisation of the type of an expression which is able to generate references. This problem has been widely studied and is not specific to MULTI-ML.

With OCAML, a typical code exposing the problem of polymorphic references is the following:

```
1 let r = ref (fun x -> x) in
2 r := (fun x -> +(x,1));
3 (!r) true
```

We can easily see that the generalisation of the type of `r` leads to inconsistencies. The standard rule will generalise the type of `r` to `('a -> 'a) ref`. On line 2, `r := (fun x -> +(x,1))` is well typed: `r : (int -> int) ref`. On the third line, `(!r) true` is also well typed: `(bool -> bool) ref`. The whole code is well typed, but the reduction of the last expression leads to `+(true,1)`, which is not a valid reduction.

To make it simple, we choose to use a standard solution which involves limiting the generalisation to non-expansive expression only. The definition of a non-expansive expressions (denoted e_{ne}) is defined in the grammar of Figure 4.33. Less restrictive solution exists, as explained in [60, 155], but they generate a important modification of the typing rules. In addition to the non-expansive original LET-IN rule, we only need to add the LET-IN-REF rule for expansive expression. This rule is similar to the LET-IN but does not generalise the type. The *updated* rules are defined in Figure 4.34.

$$
\begin{array}{lll}
e_{ne} ::= & & \textit{non-expansive expression} \\
\quad | & x & \textit{identifier} \\
\quad | & c & \textit{constant} \\
\quad | & op & \textit{operators} \\
\quad | & \textbf{fun } x \ \rightarrow e & \textit{functions} \\
\quad | & (e'_{ne}, e''_{ne}) & \textit{non-expansive pairs} \\
\quad | & op(e_{ne}) & \textit{if } op \neq \texttt{ref} \\
\quad | & \textbf{let } x = e'_{ne} \textbf{ in } e''_{ne} & \textit{let binding}
\end{array}
$$

Figure 4.33: Non-expansive expressions grammar.

$$
\text{LET-IN} \quad \frac{\begin{array}{l}\Lambda, \Gamma \vdash e_1 : \tau^1_{\pi_1}/\varepsilon_1 \ [c_1] \\ \Lambda, \Gamma; x : \textbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1) \vdash e_2 : \tau^2_{\pi_2}/\varepsilon_2 \ [c_2] \\ c_3 \equiv [\Psi = \textbf{Propgt}(\varepsilon_1, \varepsilon_2), c_1, c_2] \quad \text{when } e_1 \text{ is non-expansive}\end{array}}{\Lambda, \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau^2_{\pi_2}/\Psi \ [c_3]}
$$

$$
\text{LET-IN-REF} \quad \frac{\begin{array}{l}\Lambda, \Gamma \vdash e_1 : \tau^1_{\pi_1}/\varepsilon_1 \ [c_1] \\ \Lambda, \Gamma; x : \tau^1_{\pi_1} \vdash e_2 : \tau^2_{\pi_2}/\varepsilon_2 \ [c_2] \\ c_3 \equiv [\Psi = \textbf{Propgt}(\varepsilon_1, \varepsilon_2), c_1, c_2] \quad \text{when } e_1 \text{ is expansive}\end{array}}{\Lambda, \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau^2_{\pi_2}/\Psi \ [c_3]}
$$

Figure 4.34: Typing rules for references.

The application of the **ref** operator is excluded from the non-expansive expression. It is also the case for function application because we don't know if they are going to create new references (without using complex mechanisms).

Concerning the previous example, as `(fun x -> x)` is not non-expansive, the code is rejected. First, the value `r` is typed `'a -> 'a ref` is `'a` not generalised. Then, thanks to the line 2, `'a` is unified with `int`. Finally, at third line, the application of `(!r) true` is not valid: an expression was expected of type `int`, not `bool`.

4.4.1.3 References and communications

As references are strongly linked to a locality, their communication must be controlled meticulously. We need to extend the definition of the predicate **Seria** in order to re-instantiate a reference after its communication. In Figure 4.35, we can found the additional rules that need to be added to the previous definition of the **Seria** predicate (Figure 4.21).

$$
\begin{array}{rcl}
 & \ldots & \\
\mathbf{Seria}_{\mathtt{c}}(\tau_\pi\ \mathbf{ref}_{\mathtt{c}}) & = & \mathbf{Seria}_{\mathtt{c}}(\tau_\pi)\ \mathbf{ref}_{\mathtt{c}} \\
\mathbf{Seria}_{\mathtt{c}}(\tau_\pi\ \mathbf{ref}_{\mathtt{b}}) & = & \mathbf{Seria}_{\mathtt{c}}(\tau_\pi)\ \mathbf{ref}_{\mathtt{c}} \\
\mathbf{Seria}_{\mathtt{b}}(\tau_\pi\ \mathbf{ref}_{\mathtt{c}}) & = & \mathbf{Seria}_{\mathtt{b}}(\tau_\pi)\ \mathbf{ref}_{\mathtt{b}} \\
\mathbf{Seria}_{\mathtt{m}}(\tau_\pi\ \mathbf{ref}_{\mathtt{b}}) & = & \mathbf{Seria}_{\mathtt{c}}(\tau_\pi)\ \mathbf{ref}_{\mathtt{m}} \\
\mathbf{Seria}_{\mathtt{c}}(\tau_\pi\ \mathbf{ref}_{\mathtt{s}}) & = & \mathbf{Seria}_{\mathtt{c}}(\tau_\pi)\ \mathbf{ref}_{\mathtt{c}} \\
\text{otherwise} & = & \mathtt{Fail}
\end{array}
$$

Figure 4.35: Serialisation with references.

The communication of a references lead to its re-instantiation in a potentially different locality. Thus, we need to create a new memory zone that contains the serialised value, in the new locality, and make the reference point to it. This transformation is necessary to keep the replicated coherency concerning references. The data pointed by the reference remains unchanged but its locality is *re-located*. For example, at the end of a multi-function, the value returned by the root node is going to be communicate from the node (level `b`) to the level multi (level `m`). If this value is identified as a reference, it is important to re-locate it from the level `b` to level `m`.

4.4.2 A typing system for μmulti-ML^{exn}

In OCAML, exceptional situations are handled with a system of *exceptions*. For example, the evaluation of `1/0` leads to an exception: `Exception: Division_by_zero`. In the world of sequential programming, it is easy to catch exception with a simple system of `try … with …`. In parallel computing, if one processor triggers an exception during a computation, we need to deal with it to prevent a crash of the whole system. Some work has been done with BSML in [67]. In this subsection, we propose an sketchy approach of MULTI-ML with embedded exceptions.

The main issue introduced by exceptions is when an exception escapes the scope of a parallel vector. Indeed, when an exception is thrown from the local level by, at least, one processor, the execution flow continues on other processors. When a synchronisation barrier occurs, a global barrier is missed by the processors in an exceptional state. Such a situation can quickly lead to a *dead-lock*. To prevent this kind of situation, a specific handler was introduce in [67]. It consist of a scope that captures exceptions, as usual in OCAML. The parallel exception handler is written `trypar ... withpar ...`. Within this particular structure, all exceptions are communicated between the processes during the synchronisation barrier. Then, a decision regarding the next execution step is taken, in accordance, by all the concerned processes.

Furthermore, any access to a parallel vector that is in an incoherent state, that is to say with a processor in an exceptional state, will raise the exception, repeatedly. Thus, a set of processes containing at least one process in an exceptional state, remains in that state while the exception is not handled.

In MULTI-ML, as the execution structure is shaped like a tree, an uncaught exception at any stage must be propagated through the architecture. Any uncaught exception thrown at a node of the MULTI-BSP tree is propagated to the upper level, that is to say, toward the root node during the synchronisation barrier. If an exception is not handled correctly during the evaluation of a multi-function, the exception is propagated to the MULTI-BSP level and all the architecture is impacted. When a node, or a leaf, is left aside in such an exceptional state, any access to this tree leads to an exception. Indeed, we want to avoid *partially evaluated trees*. For example, the code `<< if random () then raise Error else f () >>`, where `f` is a multi-function, leads to

$$
\begin{array}{rcll}
C, D & ::= & & \textit{constraints:} \\
 & & \texttt{True} & \textit{truth} \\
 & & \texttt{False} & \textit{falsity} \\
 & & C \wedge C & \textit{conjunction} \\
 & & \exists \bar{X}.C & \textit{existential quantification} \\
 & & \textbf{let } x : \sigma \textbf{ in } C & \textit{type scheme introduction} \\
 & & x \preceq \tau & \textit{type scheme instantiation} \\
\sigma & ::= & & \textit{type scheme:} \\
 & & \forall \bar{X}[C].\tau &
\end{array}
$$

Figure 4.36: Constraints and type scheme syntax.

a tree where some branches (sub-trees) are never instantiated. Accessing a partially evaluated tree immediately leads to a exceptional case, as for the parallel vectors.

4.5 A purely constraint-based type system for μMULTI-ML

Constraint based type systems have been widely studied: from the Damas and Milner (DM) [36] which introduces a standard style were typing judgements are defined inductively by a collection of typing rules; to the well known Hindley and Milner's parametrised type system (HM(X)) [119], a constraint based extension of DM. Here, we concentrate on PCB(X) (**P**urely **C**onstrained **B**ased with **X**), which is a way to reduce type inference problems to constraint solving problems. Type inference is therefore done in two steps: constraint generation and constraint solving. This partitioning allows us to add new features to the language by only adding new constraint generation rules. As the constraint solver is independent, there is no need to modify it after adding new constructions. It is also possible to call an external solver to take care of the constraint solving. We may have used the *Dalton* solver proposed in [137]. The current section is based on the remarkable chapter *The Essence of* ML *Type Inference*, written by François Pottier and Didier Rémy [124], from [123]. Most of their notations and proposals are used and adapted to match the needs of the MULTI-ML type system.

4.5.1 PCB(X)

As explained in [124], relating PCB(X) and HM(X) is pretty straightforward. Basically, PCB(X) is an alternative presentation of HM(X).

As the PCB(X) system is based on constraints, we now use the constraint language proposed in Figure 4.36.

4.5.2 Constraint generation

In the standard ML, a type inference problem consists of an expression e and a type τ. We want to determine if e is well-typed with the type τ, that is to say, if the constraint C is satisfiable; then $C \vdash e : \tau$ holds. A constraint solving problem must determine if the constraint C is satisfiable. To reduce a type inference problem to a constraint solving problem, we must produce a constraint C that is both *sufficient* and *necessary* for $C \vdash e : \tau$ to hold.

A constraint C, written $[\![\ \Gamma \vdash e : \tau\ ]\!]$, is *sufficient* if $[\![\ \Gamma \vdash e : \tau\ ]\!] \vdash e : \tau$, that is to say: the constraint is specific enough to guarantee that e has type τ. Therefore, the constraint generation is sound.

Definition 4.23

We write $C_1 \Vdash C_2$, and say that C_1 entails C_2, if and only if, any valid type assignment of C_1 implies C_2.

A Constraint is *necessary* if, for every constraints C, $C \vdash e : \tau$ implies $C \Vdash [\![\ \Gamma \vdash e : \tau\ ]\!]$. Then, the constraint guaranties that e has type τ is as least as specific as $[\![\ \Gamma \vdash e : \tau\ ]\!]$. We say the constraint generation is *complete*.

So, if $C \vdash e : \tau$ holds for a satisfiable constraint C, then, by completeness, $C \Vdash [\![\ \Gamma \vdash e : \tau\ ]\!]$ holds. Then $[\![\ \Gamma \vdash e : \tau\ ]\!]$ is satisfialbe; And conversely.

With μ-MULTI-ML typing system, a constraint is define as follow: $[\![\ \ \Gamma \vdash e : \tau_\pi/\varepsilon\ ]\!]^\Lambda$. Where e is the expression that we want to type, τ_π is the output annotated type, ε is the effect raised by the evaluation of e and Λ is the level where the evaluation takes place. The main constraint generation rules are described in Figure 4.37, and are *directly* obtained from the rules of Figure 4.24, Figure 4.25 and Figure 4.26.

The constraint generation rules are pretty straightforward and are similar in many points to the typing rules previously defined. Thus, we do not describe the rules in details. One can note that the LET construct introduces a principal type scheme for x constrained by the constraint generated by the examination of e_1. This is necessary to give the most general type for x.

4.5.3 Constraints solving

In this section we are going to talk about the unification algorithm and the constraint solver which are used in the MULTI-ML type inference. We need to define a unification algorithm on types and a unification algorithm on localities. In fact, the type and locality unifications are related (some localities are constrained by types) but are solved almost independently. Finally, we will present a constraint solver that is build on top of the unification algorithms.

4.5.3.1 Type unification

The unification algorithm that we present below is due to [91]. This algorithm is, basically, a (non-deterministic) system of constraint rewriting rules. Based on the *union-find* data structure, we update the syntax of constraints by replacing equation with multi-equations.

But first, we need to introduce several definitions and notations.

Definition 4.24

The notion of kind*, written κ, denotes an element of the set of type variables. Roughly speaking, it is a particular type among all types.*

Definition 4.25

For every kind κ, the n-ary *equality predicate $=^n_\kappa \tau_1 \ldots \tau_n$ is written $\tau_1 = \ldots = \tau_n$, and is called a multi-equation. A multi-equation ϵ of kind κ can be viewed as a finite multi-set of types of kind κ. The concatenation of ε and ε' is written $\varepsilon = \varepsilon'$.*

Definition 4.26

For every kind κ, $\mathcal{V}_\kappa$ is disjoint and enumerable set of type variables.

Definition 4.27

Let α, β range over the set $\mathcal{V}$ of all type variables. We write $\alpha\beta$ for the set $\alpha \cup \beta$ and α for

$[\![\ \Gamma \vdash x : \tau_\pi/\varepsilon\ ]\!]^\Lambda = \Gamma(x) = \tau_\pi/\varepsilon$

$[\![\ \Gamma \vdash \mathbf{cst} : \tau_\pi/\varepsilon\ ]\!]^\Lambda = \Gamma(\mathbf{cst}) = \tau_\pi/\varepsilon$

$[\![\ \Gamma \vdash \mathbf{op} : \tau_\pi/\varepsilon\ ]\!]^\Lambda = \Gamma(\mathbf{cst}) = \tau_\pi/\varepsilon$

$[\![\ \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau^2_{\pi_2}/\Psi\ ]\!]^\Lambda = \mathbf{let}\ x : \forall\tau^1[[\![\ \Gamma \vdash e_1 : \tau^1_{\pi_1}/\varepsilon\ ]\!]^\Lambda].\tau^1_{\pi_1}/\varepsilon\ \mathbf{in}\ [\![\ \Gamma \vdash e_2 : \tau^2_{\pi_2}/\varepsilon'\ ]\!]^\Lambda \wedge \Psi = \mathbf{Propgt}(\varepsilon, \varepsilon')$

$[\![\ \Gamma \vdash \mathbf{fun}\ x \to e\ : \tau_\Lambda/\varepsilon\ ]\!]^\Lambda = \mathbf{let}\ x : \tau^1_{\pi_1}/\varepsilon_1\ \mathbf{in}\ [\![\ \Gamma \vdash e : \tau^2_{\pi_2}/\varepsilon_2\ ]\!]^{\Lambda'} \wedge \varepsilon_2 \blacktriangleleft \Lambda \wedge \pi_1 \lhd \varepsilon_2 \wedge \tau_\Lambda = (\tau^1_{\pi_1} \xrightarrow{\varepsilon_2} \tau^2_{\pi_2})_\Lambda$

$[\![\ \Gamma \vdash e_1\ e_2 : \tau^2_\Lambda/\Psi\ ]\!]^\Lambda = [\![\ \Gamma \vdash e_1 : (\tau^1_{\pi_1} \xrightarrow{\varepsilon_1} \tau^2_{\pi_2})_{\Lambda'}/\varepsilon_2\ ]\!]^\Lambda \wedge [\![\ \Gamma \vdash e_2 : \tau^1_{\pi_3}/\varepsilon_3\ ]\!]^\Lambda \wedge \pi_3 \lhd \pi_1 \wedge \pi_2 \lhd \Lambda \wedge \varepsilon_1 \lhd \Lambda \wedge \varepsilon_1 \blacktriangleleft \Lambda \wedge \Psi = \mathbf{Propgt}(\varepsilon_1, \mathbf{Propgt}(\varepsilon_2, \varepsilon_3))$

$[\![\ \Gamma \vdash << e >> : \tau_\pi\ \mathtt{par_b}/\mathtt{b}\ ]\!]^\mathtt{b} = [\![\ \Gamma \vdash e : \tau_\pi/\varepsilon\ ]\!]^\mathtt{c} \wedge \varepsilon \lhd \mathtt{c} \wedge \pi \lhd \mathtt{c}$

$[\![\ \Gamma \vdash \#x\# : \tau_\pi/\mathtt{l}\ ]\!]^\mathtt{c} = \Gamma(x) = \tau_\mathtt{b}/\varepsilon \wedge \varepsilon \lhd \mathtt{c} \wedge \tau_\pi = \mathbf{Seria}_\mathtt{c}(\tau_\mathtt{b})$

$[\![\ \Gamma \vdash \$e\$: \tau_\pi/\mathtt{l}\ ]\!]^\mathtt{c} = \Gamma(x) = \tau_\pi\ \mathtt{par_b}/\varepsilon \wedge \varepsilon \lhd \mathtt{c}$

$[\![\ \Gamma \vdash \mathbf{op'}\ e : \tau^2_\Lambda/\mathtt{b}\ ]\!]^\mathtt{b} = [\![\ \Gamma \vdash e_1 : (\tau^1_{\pi_1} \xrightarrow{\varepsilon_1} \tau^2_{\pi_2})_{\Lambda'}/\varepsilon_2\ ]\!]^\mathtt{b} \wedge [\![\ \Gamma \vdash e_2 : \tau^1_{\pi_3}/\varepsilon_3\ ]\!]^\mathtt{b} \wedge \pi_3 \lhd \mathtt{b} \wedge \varepsilon \lhd \mathtt{b} \wedge \pi_2 \lhd \mathtt{b}$

$[\![\ \Gamma \vdash \mathbf{multi}\ f\ x \to e1 \dagger e2 : \tau_\mathtt{m}/\mathtt{m}\ ]\!]^\mathtt{m} = \mathbf{let}\ f : (\tau^1_{\pi_1} \xrightarrow{\mathtt{l}} \tau^2_{\pi_2})_\mathtt{l}/\mathtt{l}\ \mathbf{in}\ \mathbf{let}\ x : \tau^1_\mathtt{b}/\varepsilon\ \mathbf{in}\ [\![\ \Gamma \vdash e_1 : \tau^3_{\pi_3}/\varepsilon_1\ ]\!]^\mathtt{b} \wedge \mathbf{let}\ x : \tau^1_\mathtt{s}/\varepsilon\ \mathbf{in}\ [\![\ \Gamma \vdash e_2 : \tau^3_{\pi_4}/\varepsilon_2\ ]\!]^\mathtt{s} \wedge \tau^3_\mathtt{m} = \mathbf{Seria}_\mathtt{m}(\tau^3_{\pi_3}) \wedge \tau^3_\mathtt{c} = \mathbf{Seria}_\mathtt{c}(\tau^3_{\pi_4}) \wedge \tau_\mathtt{m} = (\tau^1_\mathtt{m} \xrightarrow{\mathtt{m}} \tau^3_\mathtt{m})_\mathtt{m}$

Figure 4.37: Constraints generation rules.

$$(\exists\overline{\alpha}.U_1) \wedge U_2 \rightarrow \exists\overline{\alpha}.(U_1 \wedge U_2) \quad \textbf{if } \overline{\alpha}\#ftv(U_2) \qquad \text{(S-ExAnd)}$$

$$\alpha = \epsilon \wedge \alpha = \epsilon' \rightarrow \alpha = \epsilon = \epsilon' \qquad \text{(S-Fuse)}$$

$$\alpha = \alpha = \epsilon \rightarrow \alpha = \epsilon \qquad \text{(S-Stutter)}$$

$$F\vec{\alpha} = F\vec{\tau} = \epsilon \rightarrow \vec{\alpha} = \vec{\tau} \wedge F\vec{\alpha} = \epsilon \qquad \text{(S-Decompose)}$$

$$F\tau_1 \ldots \tau_i \ldots \tau_n = \epsilon \rightarrow \exists\alpha.(\alpha = \tau_i \wedge F\tau_1 \ldots \alpha \ldots \tau_n = \epsilon) \quad \textbf{if } \tau_i \notin \mathcal{V} \wedge \alpha \notin ftv(\tau_1,\ldots,\tau_n,\epsilon) \qquad \text{(S-Name-1)}$$

$$F\vec{\tau} = F'\vec{\tau'} = \epsilon \rightarrow \texttt{False} \quad \textbf{if } F \neq F' \qquad \text{(S-Clash)}$$

$$F\vec{\tau} \rightarrow \texttt{True} \quad \textbf{if } \tau \notin \mathcal{V} \qquad \text{(S-Single)}$$

$$U \wedge \texttt{True} \rightarrow U \qquad \text{(S-True)}$$

$$U \rightarrow \texttt{False} \quad \textbf{if } U \textit{ is cyclic} \qquad \text{(S-Cycle)}$$

$$\mathcal{U}[\texttt{False}] \rightarrow \texttt{False} \quad \textbf{if } \mathcal{U} \neq [] \qquad \text{(S-Fail)}$$

Figure 4.38: Type unification algorithm.

the singleton set $\{\alpha\}$.

Definition 4.28

Let $\overline{\alpha}$ *and* $\overline{\beta}$ *range over finite sets of type variables.*

Definition 4.29

Let $\vec{\alpha}$ *be a vector of distinct type variables.*

The constraint language used in this section is defined as follow:

$$U ::= \texttt{True} \mid \texttt{False} \mid \epsilon \mid U \wedge U \mid \exists\overline{\alpha}.U$$

We now introduce the unification algorithm in Figure 4.38. It is interesting to note that a multi-equation encode both state and control. Indeed, a multi-equation $\tau_1 = \tau_2 = \epsilon$ may be viewed as the equivalence class $\tau_1 = \tau_2$ with $\tau_2 = \epsilon$ as a pending request to solve. The specification of this algorithm is non deterministic and several rules can be applied at the same time. As the type unification and the locality unification are separated, the Figure 4.38 does not deal with localities.

L	$::=$		*constraints*
		`True`	*truth*
		`False`	*falsity*
		$c \wedge c$	*conjunction*
		$\lambda = \lambda$	*locality equality*
		$\lambda \lhd \lambda$	*accessibility of locations*
		$\lambda \blacktriangleleft \lambda$	*definability of locations*
		$\lambda = \mathbf{Propgt}(\varepsilon, \varepsilon')$	*effect propagation*

Figure 4.39: Constraints on localities.

S-ExAnd gather all the multi-equations into a single one. This rule makes the application of S-Fuse and S-Cycle possible. S-Fuse takes two multi-equations with the same variable α and fuses them into a single one. S-Stutter wipes out all the redundant variables. S-Decompose decomposes an equation between type variables with two terms where head symbols are matching. It produces a conjunction of equation between their sub-terms: $\vec{\alpha} = \vec{\beta}$. One of the two terms remain in the original multi-equation. S-Decompose is not applicable when both terms at hand have a non-variable sub-term. S-Name-1 remedies this problem by introducing a fresh variable standing for such a sub-term. The repeated application of S-Name-1 yields a unification problem composed of *small-terms*. S-Clash handles equations with different head symbols. This kind of expression leads to a failure. S-Single and S-True suppress multi-equations that are tautologies. S-Cycle is the *occurs check* that signals a failure if the constraint is cyclic. S-Fail propagates the failure where $\mathcal{U}$ is a unification constraint context.

4.5.3.2 Locality unification

To deal with localities constraints generated during the constraint generation phase and type unification, we need a dedicated unification algorithm on localities.

Considering an environment L that contains constraints on localities, defined in Figure 4.39, the unification algorithm will simply tell whether or not the constraint is coherent. To do so, we need to resolve all the locality equalities and propagate effects. Then we have to check if the location accessibility and definability are valid. The unification holds only if the constraint is valid. There is no need to have an algorithm as complex as the type unification one, in the sense that localities are bounded. Indeed, polymorphism on types must deal with any *kind* of types. On the contrary, locality unification must deal with a finite set of localities, namely `m`, `b`, `c`, `l`, `s`. It is important to note that, in some cases, a pending locality is not unified with non-variable locality and the solver admits that there is not enough information to solve it. The variable locality is returned as a result and the constraints concerning the locality are kept. Thereby, the annotated type is given with a constraint that indicates the nature of the accepted locality.

The principle of the unification algorithm on localities is pretty straightforward. L-Fuse takes two multi-equations with the same locality and fuses them into a single one. L-Stutter wipes out redundant variables. L-Clash raises a failure when different localities are equated. L-Loc-1 and L-Loc-2 check whether or not a locality is acceptable in another one. It raises an error when the localities are not compatible. L-Def-1 and L-Def-2 process similarly using the definability check. L-Prop computes the propagation of two localities and injects it as a new constraint.

$$\pi = \epsilon \wedge \pi = \epsilon' \rightarrow \pi = \epsilon = \epsilon' \qquad \text{(L-Fuse)}$$

$$\pi = \pi = \epsilon \rightarrow \pi = \epsilon \qquad \text{(L-Stutter)}$$

$$\pi = \pi_1 = \epsilon \rightarrow \underset{\textbf{if } \pi \neq \pi_1}{\texttt{False}} \qquad \text{(L-Clash)}$$

$$\lambda_1 \lhd \lambda_2 \wedge L \rightarrow \underset{\textbf{if } \lambda_1 \lhd \lambda_2 = \texttt{True}}{L} \qquad \text{(L-Loc-1)}$$

$$\lambda_1 \lhd \lambda_2 \wedge L \rightarrow \underset{\textbf{if } \lambda_1 \lhd \lambda_2 = \texttt{False}}{\texttt{False}} \qquad \text{(L-Loc-2)}$$

$$\lambda_1 \blacktriangleleft \lambda_2 \wedge L \rightarrow \underset{\textbf{if } \lambda_1 \blacktriangleleft \lambda_2 = \texttt{True}}{L} \qquad \text{(L-Def-1)}$$

$$\lambda_1 \blacktriangleleft \lambda_2 \wedge L \rightarrow \underset{\textbf{if } \lambda_1 \blacktriangleleft \lambda_2 = \texttt{False}}{\texttt{False}} \qquad \text{(L-Def-2)}$$

$$\lambda = \mathbf{Propgt}(\varepsilon_1, \varepsilon_2) \wedge \rightarrow \underset{Where\ \varepsilon = \mathbf{Propgt}(\varepsilon_1, \varepsilon_2)}{\lambda = \varepsilon} \qquad \text{(L-Prop)}$$

Figure 4.40: Locality unification.

4.5.3.3 A constraint solver

We now define a constraint solver that is independent from the unification algorithms. All the type or locality unification is explicitly triggered from the constraint solver. These algorithms are inter-dependant to build a full constraint solver and their implementations are not related to each other.

Definition 4.30

We assume that every type can be branded with a kind. For every kind κ, let $\mathcal{V}_\kappa$ be a disjoint, denumerable set of type variables. We write $ftv(o)$ for the set of free type variables *of an object o. The notation $A\#B$ stands for $A \cap B = \emptyset$. The subtyping predicate $\preceq$ is interpreted as equality in the following rules.*

We define the notion of *stack*, denoted S, to represent the information that is analysed by the constraint solver. A stack is basically a list of *frames* that are added and deleted at the inner end of a stack. The syntax of a stack is defined as following:

$$S ::= [] \mid S[[] \wedge C] \mid S[\exists \overline{\alpha}.[]] \mid S[\textbf{let } x : \forall \overline{\alpha}[[]].T \textbf{ in } C] \mid S[\textbf{let } x : \sigma \textbf{ in } []]$$

A state of the constraint solver is a quadruplet $S; U; L; C$ where S is the stack, U is a unification constraint on types, L is a unification constraint on localities and C is an external constraint. In other words, C is the constraint that the solver will solve up next. U and L contains all the information on types, respectively localities, that have been considered since the beginning of the solver execution. S is the head of the stack which is examined. The solver's initial state is $[]; \texttt{True}; \texttt{True}; C$ where C is the whole constraint.

$$S; U; L; C \to S; U'; L; C \quad \textbf{if } U \to U' \tag{S-Unify-T}$$

$$S; U; L; C \to S; U;' L; C \quad \textbf{if } L \to L' \tag{S-Unify-L}$$

$$S; U; L; \tau^1_\pi = \tau^2_{\pi'} \to S; U \wedge \tau^1_\pi = \tau^2_{\pi'}; L; \mathsf{True} \tag{S-Solve-Eq}$$

$$S; U; L; x_\pi \preceq \tau_{\pi'} \to S; U; L \wedge \pi = \pi'; S(x) = \tau \tag{S-Solve-Id}$$

$$S; U; L; C_1 \wedge C_2 \to S[[] \wedge C_2]; U; L; C_1 \tag{S-Solve-And}$$

$$S; U; L; \exists \bar{\alpha}.C \to S[\exists \bar{\alpha}.[]]; L; U; C \quad \textbf{if } \bar{\alpha} \# ftv(U) \tag{S-Solve-Ex}$$

$$S; U; L; \textbf{let } x : \forall \bar{\alpha}[D].\tau \textbf{ in } C \to S[\textbf{let } x : \forall \bar{\alpha}[[]].\tau \textbf{ in } C]; U; L; D \quad \textbf{if} \bar{\alpha} \# ftv(U) \tag{S-Solve-Let}$$

$$S[[] \wedge C]; U; L; \mathsf{True} \to S; U; L; C \tag{S-Pop-And}$$

$$S[\textbf{let } x : \forall \bar{\alpha}[[]].\alpha \textbf{ in } C]; U_1 \wedge U_2; L; \mathsf{True} \to S[\textbf{let } x : \forall \bar{\alpha}[U_2].\alpha \textbf{ in } []]; U_1; L; C \quad \textbf{if} \bar{\alpha} \# ftv(U_1) \wedge \exists \bar{\alpha}.U_2 \equiv \mathsf{True} \tag{S-Pop-Let}$$

$$S[\textbf{let } x : \sigma \textbf{ in } []]; U; L; \mathsf{True} \to S; U; L; \mathsf{True} \tag{S-Pop-Env}$$

Figure 4.41: The constraint solver.

$$
\begin{array}{rcll}
S[[] \wedge C](x) & = & S(x) & \\
S[\exists \bar{\alpha}.[]](x) & = & S(x) & \textbf{if } \bar{\alpha} \ \# \ ftv(S(x)) \\
S[\textbf{let } y : \forall \bar{\alpha}[[]].T \textbf{ in } C](x) & = & S(x) & \textbf{if } \bar{\alpha} \ \# \ ftv(S(x)) \\
S[\textbf{let } y : \sigma \textbf{ in } [](x) & = & S(x) & \textbf{if } x \neq y \\
S[\textbf{let } x : \sigma \textbf{ in } [](x)] & = & \sigma &
\end{array}
$$

Figure 4.42: Stack examination.

The solver is based on the proposal made in [124]. We choose to remove some of the rules that are not decisive in our context, for example recursive types. And we add the ability to process annotated types, a requirement for the MULTI-ML typing system. This solver is a non deterministic rewriting system performed modulo α-conversions. S-UNIFY-T, respectively S-UNIFY-L, makes the type unification, respectively locality unification, a component of the solver by allowing to invoke it at any time. S-SOLVE-EQ inject an equation in the typing environment U. S-SOLVE-ID match an instanciation constraint $x \preceq \tau$ and replace it with $\sigma \preceq \tau$, where $\sigma = S(x)$ is the type scheme carried in the nearest environment frame that defines x in the stack S. The stack examination is defined in Figure 4.42.

S-SOLVE-AND pushes, arbitrarily, the left element of the constraint on the stack. S-SOLVE-EX introduces a new existential frame in the stack. S-SOLVE-LET handle the *let binding* of the stack. First it examines the left-hand side to simplify the type scheme instanciation. Indeed, starting with the right-hand side involves a new frame where the type scheme is not simplified: $\forall \overline{\alpha}[D].T$. On the contrary, we can inject a simplified type scheme: $\forall \overline{\alpha}[U].\alpha$. S-POP-AND deals with the remaining conjunction that was partially examined by S-SOLVE-AND. S-POP-LET retrieve the type constraint of the *let binding* to introduce it in the type scheme and start to explore the right-hand side of the expression. S-POP-ENV remove the unnecessary frame composed by a type scheme that is not occurring in U any more and a solved right-hand side (turned into a constraint in U).

The current typing system of the MULTI-ML language is implemented on top of those algorithms.

5 Implementation

In this chapter we describe the elements relative to the implementation of the MULTI-ML language. In Section 5.1 we propose a formalisation of the compilation scheme of a MULTI-ML program. Then, in Section 5.2 we introduce a natural semantics with costs which aims to be used in order to predict the evaluation cost of a MULTI-ML program. Section 5.3 describes the compilation toolchain of MULTI-ML and describes the way of implementing each step. Finally, in Section 5.4, we propose some experimentation and benchmarks that were done, with MULTI-ML, on real architectures.

5.1 Formalisation of the compilation

Abstract machines [38] bridge the gap between the high level of a programming language and the low level of a real (*physical*) machine. They are thus very useful to provide an intermediate language for compilation. They are called "machines" because they allow step-by-step execution of programs; and they are "abstract" as they omit details of real architectures (hardware).

The design of a generic compilation scheme for a parallel abstract machine, based on *any* sequential machine, has already been used for the implementation of functional languages. Regarding hierarchical machines, it ensures a greater confidence for the implementation of the compiler and thus, for the safety of the language. Such a compilation scheme can be easily extended, or may be a sub-part of a more complex machine.

In the following, we describe the abstract compilation scheme dedicated to the MULTI-ML language.

5.1.1 Current implementation

The current implementation of the MULTI-ML language relies on MPI. The implementation works in a SPMD way and the compiler generates OCAML + MPI code from a source program written in MULTI-ML.

Currently, there is one MPI process for each node and leaf of the MULTI-BSP architecture, except for the root which executes the global part of the code. Each process can be seen as a "daemon", which is waiting for orders, to execute them. Basically, an order consists in a piece of code which must be evaluated by the concerned process. The communication between the daemons is done using asynchronous send and receive routines. The communicated values

$e ::=$ *Expressions*
core-ML
$\mid x \mid \textbf{cst} \mid \textbf{op} \mid (e, e) \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid (e\ e)$
$\mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid (\textbf{fun}\ x \rightarrow\ e) \mid (\textbf{rec}\ f\ x \rightarrow\ e)$
BSML-*like primitives*
$\mid$ **mkpar** $e \mid$ **gid** $\mid$ **proj** $e \mid$ **put** $e \mid$ **pid** $\mid (\textbf{copy}\, x)$
$\mid$ **replicate** $(\textbf{fun}\ f \rightarrow\ e) \mid \textbf{apply}\ e\ e$
multi-fun, without tree construction
$\mid (\textbf{multi}\ f\ x \rightarrow e_1 \dagger e_2)$
$P ::=$ *A program*
$\textbf{let}\ x = e \mid \textbf{let}\ x = e\ ;;\ P$

Figure 5.1: Syntax of core-MULTI-ML.

are composed by serialised closures which contain a code and a set of values (needed for the evaluation of the closure). The daemon un-serialises the closure and executes it with the appropriate arguments. For example, the following code `<< #x#+1 >>` is compiled into the closure (**fun** $x \rightarrow\ x + 1$). Both the closure and the value of `x` are sent to the nested nodes of the MULTI-BSP component.

This *"naive"* implementation has the advantage to be quite simple to implement and relatively efficient in most cases. However, it has several drawbacks: (1) The closure to be sent can be large if it contains several ML codes; in a SPMD model, one might want the necessary values to be sent only (here, the value of `x`); (2) In case of a tree with many levels, every node above the leaves (of the MULTI-BSP tree) is a process. In addition to an unnecessary scheduling, these processes are also executed by the computing units (the cores) as MPI processes. Thus, for nested processes executed of the same unit, an unnecessary sending is performed (the value is already available as they share the same computing unit).

The goals of the MULTI-ML virtual machine are: (1) having only one process on each computing unit, we will use continuations (threading) in place of multiple MPI processes; (2) minimise the number of daemons (and thus the extra processes); (3) abstract the communications (like in BSML) in order to have only a module of routines to implement in the future (using MPI or any low level library).

The main goal is to abstract the compiler. To do so, we will extend the traditional compilation scheme of ML with MULTI-ML routines. Thus, we will describe the compilation scheme of MULTI-ML codes on any abstract machine.

5.1.2 A core language

We propose, in Figure 5.1, a syntax which extends the popular core-ML.

A program is a list of expressions which are executed one after the other. Expressions contain variables, constants (integers, *etc.*), operators ($\leq$, $+$, *etc.*), pairing, **let**, **if**, **fun** statements as usual in ML, **rec** for recursive calls, the BSML primitives (**replicate**, **put**, **proj**, **mkpar**, **apply**), local copy (**copy**) of a parent's variable to its child, and **gid** the tree of identifiers.

Finally, we define "let-multi" as a particular function construction with codes for both the nodes and the leaves. Only the expressions without free variables in **replicate** (**fun** $g \rightarrow\ e$) are valid. Furthermore, only programs without free variables are valid. Free variables are defined in Figures 5.2. A **replicate** is well defined when the g in **replicate** (**fun** $g \rightarrow\ e$) corresponds to a

$$
\begin{array}{rcl}
\mathcal{F}(x) & = & \{x\} \\
\mathcal{F}(\mathbf{cst}) & = & \emptyset \\
\mathcal{F}(\mathbf{op}) & = & \emptyset \\
\mathcal{F}((e_1, e_2)) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \\
\mathcal{F}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) & = & \mathcal{F}(e_1) \cup (\mathcal{F}(e_2) \setminus \{x\}) \\
\mathcal{F}(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \cup \mathcal{F}(e_3) \\
\mathcal{F}((\mathbf{fun}\ x \rightarrow\ e)) & = & \mathcal{F}(e) \setminus \{x\} \\
\mathcal{F}((\mathbf{rec}\ f\ x \rightarrow\ e)) & = & \mathcal{F}(e) \setminus \{f, x\} \\
 & & \text{BSML-}\textit{like primitives} \\
\mathcal{F}(\mathbf{replicate}\ (\mathbf{fun}\ g \rightarrow\ e)) & = & \mathcal{F}(e) \setminus \{g\} \\
\mathcal{F}(\mathbf{copy}\ x) & = & \emptyset \\
\mathcal{F}(\mathbf{mkpar}\ e) & = & \mathcal{F}(e) \\
\mathcal{F}(\mathbf{put}\ e) & = & \mathcal{F}(e) \\
\mathcal{F}(\mathbf{apply}\ e_1\ e_2) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \\
\mathcal{F}(\mathbf{proj}\ e) & = & \mathcal{F}(e) \\
 & & \textit{multi-functions} \\
\mathcal{F}((\mathbf{multi}\ f\ x \rightarrow e_1 \dagger e_2)) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \\
 & & \textit{Full programs} \\
\mathcal{F}(\mathbf{let}\ x = e) & = & \mathcal{F}(e) \\
\mathcal{F}(\mathbf{let}\ x = e)\ ;;\ P & = & \mathcal{F}(e) \cup (\mathcal{F}(P) \setminus \{x\})
\end{array}
$$

Figure 5.2: Free variables of core-MULTI-ML.

multi-function. We note $\mathcal{WF}_(e)$ a well formed expression (induction definition in Figure 5.3) which uses $\mathcal{WF}_f(e)$ to check if **replicate** is well formed or not. This is used to make sure that g is a multi-function. For example, it forbids codes like `(fun f -> replicate (fun f -> e))` or the definition of `(replicate (fun f -> e))` outside the scope of multi-function (where it is impossible to communicate downward the identifiers of the multi-function).

We do not use the (common) syntactic sugars of BSML and MULTI-ML and we suppose that they have been transformed into primitives calls, first:

(1) $<\ e\ > \equiv$ **replicate** (**fun** _ $\rightarrow\ e$);

(2) every access $\$x\$$ (inside a vector) to the local value of a vector x has been replaced by a call of **apply**, *e.g.* $<\ \$x\$ + 1\ > \equiv$ **apply** (**replicate** (**fun** _ $x \rightarrow\ x + 1$)) x;

(3) in the same way, each local copy $\#x\#$ of a parent's variable x is replaced by a call to the **copy** primitive.

We recall that those primitives are available in the MULTI-ML-core syntax; Only the syntactic sugars are available to programmers, for sake of programming simplicity.

5.1.2.1 Compilation

To get a generic compilation of MULTI-ML, we need a compilation that does not depend on the compilation of the sequential parts of the language. Thus we are able to generate pure ML codes with some low level communication routines. We abstract such a compilation with the notation $[\![e]\!]_s$, corresponding to the compilation of the term e with the standard ML compiler. This compilation generates instructions of the standard virtual machine of ML (OCAML in our current implementation). In general, such instructions manipulate a stack and some

$$
\begin{array}{rcl}
\mathcal{WF}_f(x) & = & \textbf{true} \\
\mathcal{WF}_f(\textbf{cst}) & = & \textbf{true} \\
\mathcal{WF}_f(\textbf{op}) & = & \textbf{true} \\
\mathcal{WF}_f((e_1, e_2)) & = & \mathcal{WF}_f(e_1) \wedge \mathcal{WF}_f(e_2) \\
\mathcal{WF}_f(\textbf{let } x = e_1 \textbf{ in } e_2) & = & \mathcal{WF}_f(e_1) \wedge (\mathcal{WF}_(e_2)) \quad \text{if } (x \equiv f) \\
\mathcal{WF}_f(\textbf{let } x = e_1 \textbf{ in } e_2) & = & \mathcal{WF}_f(e_1) \wedge (\mathcal{WF}_f(e_2)) \quad \text{otherwise} \\
\mathcal{WF}_f(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) & = & \mathcal{WF}_f(e_1) \wedge \mathcal{WF}_f(e_2) \wedge \mathcal{WF}_f(e_3) \\
\mathcal{WF}_f((\textbf{fun } x \rightarrow e)) & = & \mathcal{WF}_(e) \quad \text{if } (x \equiv f) \\
\mathcal{WF}_f((\textbf{fun } x \rightarrow e)) & = & \mathcal{WF}_f(e) \quad \text{otherwise} \\
\mathcal{WF}_f((\textbf{rec } f\ x \rightarrow e)) & = & \mathcal{WF}_(e) \quad \text{if } (x \equiv f \vee g \equiv f) \\
\mathcal{WF}_f((\textbf{rec } f\ x \rightarrow e)) & = & \mathcal{WF}_f(e) \quad \text{otherwise} \\
 & & \text{BSML-}\textit{like primitives} \\
\mathcal{WF}_f(\textbf{replicate } (\textbf{fun } g \rightarrow e)) & = & \textbf{true} \quad \text{if } (g \equiv f) \\
\mathcal{WF}_f(\textbf{replicate } (\textbf{fun } g \rightarrow e)) & = & \textbf{false} \\
\mathcal{WF}_f(\textbf{copy } x) & = & \textbf{true} \\
\mathcal{WF}_f(\textbf{mkpar } e) & = & \mathcal{WF}_f(e) \\
\mathcal{WF}_f(\textbf{put } e) & = & \mathcal{WF}_f(e) \\
\mathcal{WF}_f(\textbf{proj } e) & = & \mathcal{WF}_f(e) \\
\mathcal{WF}_f(\textbf{apply } e_1\ e_2) & = & \mathcal{WF}_f(e_1) \wedge \mathcal{WF}_f(e_2) \\
 & & \textit{multi-functions} \\
\mathcal{WF}_((\textbf{multi } f\ x \rightarrow e_1 \dagger e_2)) & = & \mathcal{WF}_f(e_1) \wedge \mathcal{WF}_(e_2)
\end{array}
$$

Figure 5.3: Well formed terms of core-MULTI-ML.

environments (memories). We denote them $\mathcal{E}$ with the pid of each core of the machine (assuming an unique **pid** for each core of the machine) as subscript.

We note $[\![e]\!]_{\text{M}}^{-}$ the compilation of a MULTI-ML expression. This compilation may have called $[\![e]\!]_{\text{S}}$, if a sub-term is free of MULTI-ML primitives. Without considering the type system of MULTI-ML, it is hard to detect such expressions statically. In our case, and without lack of generality, we prefer to apply such a compilation to e after the pass of $[\![e]\!]_{\text{M}}^{-}$[1].

The compilation of a term e is thus a compilation which transforms all MULTI-ML features into standard ML ones and then, compile them using the standard ML compilation. The compilation of a program $P \equiv e_1;;\cdots;;e_n$ is the compilation of all the expressions, that is, $[\![P]\!]_{\text{M}} = [\![e_1]\!]_{\text{M}}^{-};;\cdots;;[\![e_n]\!]_{\text{M}}^{-}$. The compilation and execution can be expressed as following:

$$\langle\!\langle \{\mathcal{E}_0, P'\}, \ldots, \{\mathcal{E}_{\mathbf{p_c}}, P'\} \rangle\!\rangle$$

where $\mathbf{p_c}$ stands for the total number of cores of the MULTI-BSP machine and where $P' = [\![[\![P]\!]_{\text{M}}]\!]_{\text{S}}$.

The execution scheme of a program P is based on the following ideas:

1. The code is duplicated to be executed on each core; the hierarchical architecture is thus *"flattened"* with the hypothesis that on the same core, only nodes and leaves of the same branch may appear. Moreover, each node is managed by one leaf. Figure 5.4 illustrates the *"flattening"* of the MULTI-ML tree on cores, using two different methods. Depending on the flattening, we can observe variations of g_0, corresponding to the communication bandwidth between nodes 1,2 and 3. This is mainly due to the hardware capabilities of the machine;

2. The code outside multi-functions is executed sequentially, one time, for each core;

[1] We left the possibility of proof of optimisations for future work.

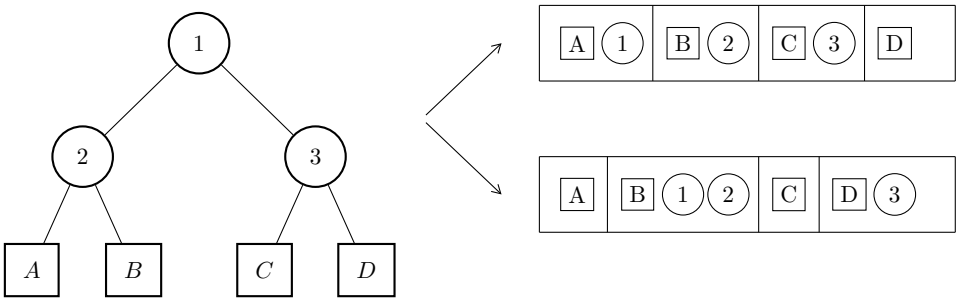


Figure 5.4: Flattening the MULTI-ML tree into cores.

3. A multi-function is a special function which initialises some data (basically, it is defined by an identifier corresponding to the codes of nodes and leaves, stored in a global hash table of multi-functions) and run a scheduler with the appropriate code of the root node;

4. Each scheduler is waiting for two kinds of instructions: spawning a daemon and continue the execution or terminate a multi-function execution; Daemons are thus independent, which allows to have different daemons running on the same physical core; each daemon corresponds to a node or a leaf during both the *"up"* and *"down"* phase of the recursive call of the multi-function;

5. Each daemon is waiting for instructions of its *parent* (upper-node):

   - Perform an asynchronous BSML routine such as a **`replicate`** or **`apply`**;
   - Perform a synchronous routine such as **`proj`**, **`mkpar`**, **`copy`** or **`put`**;
   - Complete the computation, since the parent has also completed the node code.

6. Leaf codes are running sequentially;

7. A multi-function recursive call is *"just"* the execution of the code corresponding to the identifier of a multi-function.

The main difficulty comes from parallel vectors. From the node point of view, a parallel vector contains a code which manages the memories of distant cores. To avoid the communication of serialised codes (which are known and shared by each core), the construction of a parallel vector is based on the following idea: we give a *static identifier* for each parallel vector in the program (syntactically, the code following **`replicate`**). When created, a parallel vector uses a global hash table that relates those identifiers to the corresponding codes. Then a *dynamic identifier* is shared between a node (parent) and its sub-nodes (children), allowing to reference a parallel vector between the two processes. It is thus possible to remove a parallel vector when it is detected as obsolete by the parent's node (using a garbage collection procedure).

A multi-function is not a simple function since it can be propagated through the whole hierarchical machine and can perform communications. A simple closure is thus not satisfying. The system must keep the trace of which the multi-function is currently running, using a global variable call `CurrentIdM`. We write `CodeId` for the identifier of each core of the machine.

A multi-function is thus compiled as following:

```
(* Multi function "where node = e1 where leaf = e2" *)
(fun mf x ->
  let idM= newIdM() in
  let f = (fun i x ->
    CurrentIdM:=idM;
    if node(i) then
    begin
      WakeUpChildren^i CoreId;
      let v=e1 in
      Signal^i EndNode;
      v
    end
    else e2) in
  HM[idM]:=f;
  if CoreId=root then run (WakeUpAll (f 0̃ x));
  let v = scheduler() in
  HM[idM]:=null;
v)
```

Notice that, due to the execution model of the MULTI-ML, all cores are running the exact same code and thus, all of them are registering the same multi-functions in order to execute them (in a SPMD mode). To do so, an identifier is first created. Then, we create a specific function with an argument `i` that will be the **`gid`** (global identifier) of the current level of execution. The `CurrentIdM` is then modified by the identifier of the multi-function called. If the process is a node, the node code of the multi-function is executed, after notifying the sub-nodes to wait for instructions. Then daemons will be used to manipulate the vectors (construction, projection, communication, *etc.*). Otherwise, when the process is a leaf, the leaf code is executed. When the multi-function starts, the root node is active and all other processes are awakened, waiting for orders, as daemons. Finally, the multi-function is un-registered (globally by all the cores) and we return the computed value.

The code of the scheduler is:

```
let scheduler () = match wake() with
  | Dmn i -> run daemon^i(); scheduler();
  | EndMulti v -> v
```

Where the `wake` is a *passive-wait* function which waits until an order is received. Note that each core has its own ordered queue of messages. That could be easily implemented using a library such as MPI.

Now, we have to handle parallel vector's manipulations. We recall that the idea is to have a *static identifier* which correspond to the code to execute and another identifier (*dynamic identifier*) which is used during the life-time of a parallel vector. As static identifiers are shared by all cores, the codes must be the same on each core (SPMD). On the contrary, dynamic identifiers will be shared by a node and its children (which could be nodes or leaves). The content of the vector (data and expressions) is handled by the child, using a hash table of vectors where the identifiers are the keys. From the parent point of view, a parallel vector is just a dynamic identifier. Here is the generated code for **`replicate`**:

```
let replicate = fun idS ->
  let idD= newIdD() in
  let idM = CurrentIdM in
   Signal^i Rpl(idD,idS,idM);
   idD
```

The parameter is the static identifier used to build the parallel vector. We will explain, later, how we obtain them. Here, two identifiers are necessary: (1) the dynamic identifier of the current vector and (2) the identifier of the current (running) multi-function.When a parallel vector construction order is received (`Rpl`), three identifiers must be communicated to the children that are, by construction, daemons which are waiting orders. The identifier of the multi-function is used to bind the call of the multi-function in the body of the **`replicate`**. As the **`replicate`** routine is asynchronous, the `Signal` routine is an asynchronous send which involves insignificant costs for modern architectures and many cores. Finally the generated dynamic identifier is returned to complete the **`replicate`**. The code of **`apply`** works in the same manner, except for the use of two vectors:

```
let apply = fun idD_1 idD_2 ->
  let idD_3= newIdD() in
   Signal^i App(idD_1,idD_2,idD_3);
   idD_3
```

The code for the **`proj`** is also close, except for the use of a synchronous routine to communicate the values of the children upwards:

```
  let proj = fun idD ->
    Signal^i Prj(idD);
  let tab=fromChildren^i() in
    (fun j -> tab[j])
```

The routine `fromChildren` generates an array containing all the received values of every process. The order `Prj` thus forces the children to communicate their values. The resulting value is thus, for each i, a function mapping of those values.

The code of the **`put`** primitive is as follow:

```
  let put = fun idD_i ->
  let idD_o= newIdD() in
  Signal^i Pt(idD_o,idD_i);
  idD_i
```

There are two parallel vectors: one for the input, another one for the output. The `Pt` order forces the children to synchronise and exchange their values (similarly to the BSML semantics). On the other hand, the parent just returns the identifier of the new parallel vector since it does not directly access to the values stored on its children.

The code for **`copy`** is:

```
let copy = fun x ->
  let idD= newIdD() in
  Signal^i Do(idD);
  toChildren^i x;
  idD
```

Basically, it simply signals, to the children, the construction of a new parallel vector and waits for the corresponding value. The generated code for **mkpar** is close, except for the sending of values to the children:

```
let mkpar = fun g ->
  let idD= newIdD() in
  Signal^i Mk(idD);
  toChildren^i (Array.create f p);
  idD
```

Finally, the code of a daemon consists in waiting for orders, executing them and waiting until the execution ends.

```
  let rec daemon^i() = match rcv^i with
  | Rpl(idD,idS,idM) -> VD[idD]:=(VS[idS] VM[idM]); daemon^i()
  | App(idD1,idD2,idD3) ->VD[idD3]:=(VD[idD1] VD[idD2]); daemon^i()
  | Prj(idD) -> up^i VD[idD]; daemon^i()
  | Pt(idDi,idDo) -> VD[idDo]:=(flatSend^i VD[idDi]); daemon^i()
  | Cpy (idD) -> VD[idD]:=down^i; daemon^i()
  | Mk (idD) -> VD[idD]:=down^i; daemon^i()
  | EndNode  -> ()
```

In this code, we have:

- $\mathcal{VD}$, which is the hash table of the dynamic identifier of parallel vectors. Respectively, $\mathcal{VS}$ is the shared hash table of static identifiers (from identifiers to codes) and $\mathcal{VM}$ the hash of multi-function identifiers;
- `up` which sends the value to the parent and thus synchronises the children;
- `flatSend` which works similarly to the BSML `Send` (also for the **put** primitive);
- **down** which allows to receive the values of the parent in a synchronous way.

The code works as follow. When the **replicate** order is received, we create a new parallel vector with the given information. The **apply** order works similarly. Regarding the **proj** order, we communicate upwards the value of the vector, by reading the hash table. For the **put** order, we perform the all-to-all exchanges. Finally, for the **copy** and **mkpar** orders, we receive the values sent by the parent in order to build, and update, the hash table of parallel vectors.

Note that using only one daemon for simulating the execution of upper nodes seems impossible. Indeed, if a process must manage two nodes, the daemon simulating the first node may be waiting for the results of its children, which could be itself. Moreover the second node could also wait for the results of its own children, making the core in a situation of deadlock if no more than one thread is active. To avoid such problems, we chose a solution where, on each core, only one thread is active during all the execution of the MULTI-ML code.

The compilation $[\![e]\!]_{\text{M}}^{-}$ of an expression e is defined in Figure 5.5 and it works as follows:

- Most sequential constructions are left as they are or are build over trivial inductions. This is the case of constants, operators, pairing, conditional, binding and functions;
- At the beginning, we are not within the scope of a multi-function binding. But after the declaration of a multi-function, we must select the appropriate identifier which is given to the compilation function;

$$
\begin{array}{rcl}
 & & \textit{Nothing to change for sequential computations} \\
[\![x]\!]^g_{\text{M}} & = & x \quad \text{if } x \neq y \\
[\![x]\!]^g_{\text{M}} & = & (\textbf{call } \mathit{CurrentidM}) \\
[\![\textbf{cst}]\!]^g_{\text{M}} & = & \textbf{cst} \\
[\![\textbf{op}]\!]^g_{\text{M}} & = & \textbf{op} \\
[\![(e_1, e_2)]\!]^g_{\text{M}} & = & ([\![e_2]\!]^g_{\text{M}}, [\![e_1]\!]^g_{\text{M}}) \\
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]^g_{\text{M}} & = & \textbf{let } x = [\![e_1]\!]^g_{\text{M}} \textbf{ in } [\![e_2]\!]^g_{\text{M}} \quad \text{if } x \neq y \\
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]^g_{\text{M}} & = & \textbf{let } x = [\![e_1]\!]^g_{\text{M}} \textbf{ in } [\![e_2]\!]^-_{\text{M}} \quad \text{otherwise} \\
[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]^g_{\text{M}} & = & \textbf{if } [\![e_1]\!]^g_{\text{M}} \textbf{ then } [\![e_2]\!]^g_{\text{M}} \textbf{ else } [\![e_3]\!]^g_{\text{M}} \\
[\![(\textbf{fun } x \rightarrow \ e)]\!]^g_{\text{M}} & = & (\textbf{fun } x \rightarrow \ [\![e]\!]^g_{\text{M}}) \quad \text{if } x \neq y \\
[\![(\textbf{fun } x \rightarrow \ e)]\!]^g_{\text{M}} & = & (\textbf{fun } x \rightarrow \ [\![e]\!]^-_{\text{M}}) \quad \text{otherwise} \\
 & & \text{BSML-}\textit{like primitives} \\
[\![\texttt{replicate } (\textbf{fun } g \rightarrow \ e)]\!]^g_{\text{M}} & = & \text{code given above with } \mathit{idS}, \mathcal{H} \oplus \{\mathit{idS} \Rightarrow (\textbf{fun } g \rightarrow \ [\![e]\!]^g_{\text{M}})\} \\
 & & \text{where } \mathit{idS} \text{ is a fresh value} \\
[\![\texttt{copy } x]\!]^g_{\text{M}} & = & \text{Code given above} \\
[\![\texttt{mkpar } e]\!]^g_{\text{M}} & = & \text{Code given above} \\
[\![\texttt{put } e]\!]^g_{\text{M}} & = & \text{Code given above} \\
[\![\texttt{apply } e]\!]^g_{\text{M}} & = & \text{Code given above} \\
[\![\texttt{proj } e]\!]^g_{\text{M}} & = & \text{Code given above} \\
 & & \textit{multi-functions} \\
[\![(\textbf{multi } f \ x \rightarrow e_1 \dagger e_2)]\!]^-_{\text{M}} & = & \text{Code given above where } e'1 = [\![e_1]\!]^f_{\text{M}} \text{ and } e'2 = [\![e_2]\!]^f_{\text{M}}
\end{array}
$$

Figure 5.5: Compilation scheme of core-MULTI-ML.

- When executing the `replicate` primitive, g stands for the current evaluated multi-function and thus, it is used to define where the call to the `CurrentIdM` must be done. Moreover, we update the hash table of vectors: $\mathcal{H}$.

Finally, the compilation of a program is performed with $\mathcal{H} \equiv \emptyset$.

Note that our core-language does not allow the definition of multiple multi-function at the same time. It is thus possible to declare one multi-function at a time. To exceed this limitation, an appropriate set of bindings must be given as arguments of `replicate`, that is all the potential multi-function calls. This feature is discussed in future works.

5.1.3 Execution and Correctness

The execution of a compiled program, as a virtual machine, is done using a small-steps semantics. It consists of a predicate between an expression and another expression defined by a set of rules called steps.

As usual, we have two kinds of reductions. One for the cores and one for the whole machine. We note $\Rightarrow_c$ the local reduction of one expression on a single core and $\Rightarrow$ the reduction of an expression for the whole machine. For both rules, $\Rightarrow^*$ is a reflexive and transitive closure (defined using an induction) and $\Rightarrow^\infty$ is used for infinite programs (defined using a co-induction). The reduction of a program $P = e_1;;\cdots;;e_n$ is the reduction e_1 followed by the reduction of e_2 *etc.*

For sake of simplicity, we do not give the reduction rules of OCAML expressions, but we focus one the reduction of communication routines needed for the implementation of MULTI-ML. Note that sequential codes do not modify the communication environments nor messages queues. An environment $\mathcal{E}$ contains at least two independent queues of orders, one (denoted $\mathcal{F}^s$) for the schedulers and another one (denoted $\mathcal{F}^d$) shared by the daemons. We write $\lhd e \rhd$ for a thread

$$
\begin{array}{lcll}
\Gamma^i & ::= & [] & \textit{head evaluation} \\
 & | & \Gamma^i e & \textit{application} \\
 & | & v\Gamma^i & \textit{application} \\
 & | & \textbf{let } x = \Gamma^i \textbf{ in } e & \textit{let} \\
 & | & (\Gamma^i, e) & \textit{left pair} \\
 & | & (v, \Gamma^i) & \textit{right pair} \\
 & | & \textbf{if } \Gamma^i \textbf{ then } e \textbf{ else } e & \textit{conditional}
\end{array}
$$

Figure 5.6: Abstract context syntax.

that runs the expression e. We note $|\overset{v}{\vdots}|$ for reading the message v at the top of a queue and $|\overset{\vdots}{v}|$ for adding a message v at the bottom of a queue.

It is easy to observe that we cannot always make a head reduction of an expression. We have to reduce, in depth, the sub-expressions. To define such a depth reduction, we define two kinds of contexts: (1) An expression Γ^i with a "hole" denoted [] which has the abstract syntax given in Figure 5.6 where i is the **gid** of the expression; (2) We then define Γ^i_j as the context of each thread where the reduction holds. For the core j at gid i, $\{\mathcal{E}_j, \Gamma^i_j[e]\}$ stands for $\{\mathcal{E}_j, \lhd e' \rhd, \ldots, \lhd \Gamma^i[e] \rhd, \ldots \lhd e'' \rhd\}$.

From the previously defined low level routines, we have the following rules:

fromChildren and up First, to communicate synchronously a value upward (from children to the parent), we have the following rule:

$$
\begin{array}{l}
\langle\!\langle \ldots, \{\mathcal{E}_0, \Gamma^i_0[\textbf{up } v_0]\}, \ldots, \{\mathcal{E}_{\mathbf{p}}, \Gamma^i_{\mathbf{p}}[\textbf{up } v_{\mathbf{p}}]\}, \{\mathcal{E}, \Gamma'^i_j[\textbf{fromChildren}()]\}, \ldots \rangle\!\rangle \\
\Rightarrow \\
\qquad \langle\!\langle \ldots, \{\mathcal{E}_0, \Gamma^i_0[()]\}, \ldots, \{\mathcal{E}_{\mathbf{p}}, \Gamma^i_{\mathbf{p}}[()]\}, \{\mathcal{E}, \Gamma'^i_j[[|v_0, \ldots, v_{\mathbf{p}}|]]\}, \ldots \rangle\!\rangle
\end{array}
$$

where $\{0, \ldots, \mathbf{p}\}$ are the **gid** i of the branch of the children of the parent on core j.

toChildren and down On the contrary, we also have a synchronous operation to communicate downward a value (from a parent to children) and thus, we have the following rule:

$$
\begin{array}{l}
\langle\!\langle \ldots, \{\mathcal{E}_0, \Gamma^i_0[\textbf{down}()]\}, \ldots, \{\mathcal{E}_{\mathbf{p}}, \Gamma^i_{\mathbf{p}}[\textbf{down } v]\}, \{\mathcal{E}, \Gamma'^i_j[\textbf{ToChildren } \mathbf{v}()]\}, \ldots \rangle\!\rangle \\
\Rightarrow \\
\qquad \langle\!\langle \ldots, \{\mathcal{E}_0, \Gamma^i_0[v]\}, \ldots, \{\mathcal{E}_{\mathbf{p}}, \Gamma^i_{\mathbf{p}}[v]\}, \{\mathcal{E}, \Gamma'^i_j[v]\}, \ldots \rangle\!\rangle
\end{array}
$$

where $\{0, \ldots, \mathbf{p}\}$ are the **gid** i of the branch of the children of the parent on core j.

flatSend This synchronous low level primitive is the one used for implementing the **put** primitive. **flatSend** takes an array of (functional) values to send and returns an array of received values:

$$
\begin{array}{l}
\langle\!\langle \ldots, \{\mathcal{E}_0, \Gamma^i_0[\textbf{flatSend } [|v^0_0, \ldots, v^0_k, \ldots, v^0_{\mathbf{p}}|]]\}, \ldots, \{\mathcal{E}_{\mathbf{p}}, \Gamma^i_{\mathbf{p}}[\textbf{flatSend } [|v^{\mathbf{p}}_0, \ldots, v^{\mathbf{p}}_k, \ldots, v^{\mathbf{p}}_{\mathbf{p}}|]]\}, \ldots \rangle\!\rangle \\
\Rightarrow \\
\qquad \langle\!\langle \ldots, \{\mathcal{E}_0, \Gamma^i_0[[|v^0_0, \ldots, v^k_0, \ldots, v^{\mathbf{p}}_0|]]\}, \ldots, \{\mathcal{E}_{\mathbf{p}}, \Gamma^i_{\mathbf{p}}[[|v^0_{\mathbf{p}}, \ldots, v^k_{\mathbf{p}}, \ldots, v^{\mathbf{p}}_{\mathbf{p}}|]]\}, \ldots \rangle\!\rangle
\end{array}
$$

The processor n sends the value v_n^m to processor m. After the communication, the processor m gets the value v_m^n from processor n.

Signal and rcv The **Signal** function is a low level asynchronous communication routine which aims to send a message to the queue $\mathcal{F}^d$ of another (distant or not) thread. Then the **rcv** routine can read it. We write $\mathcal{E} \oplus |\vdots|^i$ to highlight the use of the queue of daemons (allocated to **gid** i) without modifying the rest of the environment.

First we define the sending of a message:

$$\langle\!\langle \ldots, \{\mathcal{E}_0 \oplus |\vdots|^i, e_0^i\}, \ldots, \{\mathcal{E}_{\mathbf{p}} \oplus |\vdots|^i, e_{\mathbf{p}}^i\}, \{\mathcal{E}, \Gamma_j^i[\mathbf{Signal^i}\ v]\}, \ldots \rangle\!\rangle$$

$$\Rightarrow_c$$

$$\langle\!\langle \ldots, \{\mathcal{E}_0 \oplus |\vdots^{|v|}|^i, e_0^i\}, \ldots, \{\mathcal{E}_{\mathbf{p}} \oplus |\vdots^{|v|}|^i, e_{\mathbf{p}}^i\}, \{\mathcal{E}, \Gamma_j^i[()]\}, \ldots \rangle\!\rangle$$

Where v is one of the possible messages presented above and where $\{0, \ldots, \mathbf{p}\}$ are the **gid** i of the branch of the children of the parent on core j. Note the message could be processed later. For example, such a case may occur when a daemon received different messages manipulating a parallel vector:

```
let v1=replicate (fun g -> 1) in
let v2=replicate (fun g -> 2) in
let vplus=replicate (fun g -> (+) ) in
 (apply (apply vplus v1) v2)
```

As it is purely asynchronous, one daemon may be in a situation where it has received all the messages before executing the orders. Thus we define the rule of reception:

$$\langle\!\langle \ldots, \{\mathcal{E}_j \oplus |\vdots_v|^i \lhd \Gamma_j^{i'} \rhd \cdots \lhd \Gamma_j^i[\mathbf{rcv^i}()] \rhd \cdots, \ldots \rangle\!\rangle$$

$$\Rightarrow_c$$

$$\langle\!\langle \ldots, \{\mathcal{E}_j \oplus |\vdots|^i \lhd \Gamma_j^{i'} \rhd \cdots \lhd \Gamma_j^i[v] \rhd \cdots, \ldots \rangle\!\rangle$$

On core j, the daemon of **gid** i (in a thread) reads the message in its own queue. Note that those rules are not global since they only rely on one thread of one unique core instead of synchronous rules that need the whole machine to work.

WakeUpChildren, WakeUpAll and wake They are working in the same way, but where the queue $\mathcal{F}^s$ of the schedulers and **WakeUpAll** sends the message to all cores, **WakeUpChildren** sends the messages to its Children, similarly to **toChildren**. We do not give the rule for sake of conciseness.

run Finally, we have a routine to run threads (daemons):

$$\langle\!\langle \ldots, \lhd\Gamma_j^{i'} \rhd \cdots \lhd \Gamma_j^i[\mathbf{run}\ d^k] \rhd \cdots, \ldots \rangle\!\rangle$$

$$\Rightarrow_c$$

$$\langle\!\langle \ldots, \lhd\Gamma_j^{i'} \rhd \cdots \lhd \Gamma_j^i[()] \rhd \lhd d^k \rhd \cdots, \ldots \rangle\!\rangle$$

Where, on core j, for **gid** i, we run the daemon for **gid** k.

When a thread ends, it may have produced an *orphan* value. When such a value is not communicated upward using the **proj** primitive, there is no need to keep them. Thus:

$$\langle\!\langle \ldots, \lhd e \rhd \cdots \lhd v \rhd \cdots, \ldots \rangle\!\rangle$$

$$\Rightarrow_c$$

$$\langle\!\langle \ldots, \lhd e \rhd \cdots, \ldots \rangle\!\rangle$$

Finally, the asynchronous reduction $\Rightarrow_c$ (for standard OCAML expressions) can only work in the context of a call of the $\Rightarrow$ reduction on the whole machine, using the following rule:

$$\langle\!\langle \ldots, \{\mathcal{E}_j, \cdots \lhd \Gamma^i_j[e] \rhd \cdots\}, \ldots, \rangle\!\rangle$$

$$\Rightarrow$$

$$\langle\!\langle \ldots, \{\mathcal{E}_j, \cdots \lhd \Gamma^i_j[e'] \rhd \cdots\}, \ldots, \rangle\!\rangle$$

If $\Gamma[e] \Rightarrow_c \Gamma[e']$ uses a standard reduction rule of OCAML (matching, if-then-else, application, binding, pairing, *etc.*).

Note that all the low level communication routines only works when sending valid values, as in the operational semantics. We do not introduce a validity test for values to simplify the formalisation.

Now we have the following results that could be proved by induction and co-induction.

Theorem 5.1

For a program $P = e_1;;\cdots;;e_n$ and if $\mathcal{WF}_(e_i)$,

- *If $\mathcal{M} \vdash e_i \Downarrow^{\mathcal{L}}_p v_i$ for each e_i then:*
  $\langle\!\langle \{\mathcal{E}_0, \lhd [\![P]\!]_{\text{M}} \rhd\}, \ldots, \{\mathcal{E}_{\boldsymbol{p_c}}, \lhd [\![P]\!]_{\text{M}} \rhd\} \rangle\!\rangle \Rightarrow^* \langle\!\langle \{\mathcal{E}'_0, \lhd v_0 \rhd\}, \ldots, \{\mathcal{E}'_{\boldsymbol{p_c}}, \lhd v_{\boldsymbol{p_c}} \rhd\} \rangle\!\rangle$
- *If $\mathcal{M}, \vdash e_i \Downarrow^{\mathcal{L}}_p \infty$ for one e_i then:*
  $\langle\!\langle \{\mathcal{E}_0, \lhd [\![P]\!]_{\text{M}} \rhd\}, \ldots, \{\mathcal{E}_{\boldsymbol{p_c}}, \lhd [\![P]\!]_{\text{M}} \rhd\} \rangle\!\rangle \Rightarrow^\infty$

Then, we write $\langle\langle e, ..., e \rangle\rangle \Rightarrow_{safe}$, where $\Rightarrow_{safe} \equiv \Rightarrow^* \cup \Rightarrow^\infty$.

The generated code works similarly to the MULTI-ML's big step semantics. If all the expressions give values, then the code finally gives a final value. Otherwise, if one expression diverges, then the whole machine is considered to be diverging.

5.1.4 Distributed Garbage collector

Distributed garbage collection is known to be non trivial. In MULTI-ML, the main concern about garbage collection is related to the parallel vectors. Indeed, when a node builds a parallel vector, the values are actually stored inside the memory of the sub-nodes. Thus, after each recursion of the multi-functions, the hash table which contains the dynamic identifier (linked to values) are growing bigger. As the hash table cannot be automatically freed by the OCAML garbage collector, we must clean the table manually. To do so, the simpler solution may consist in a system which can determine the maximal life time of a parallel vector. In the context of tree construction (using multi-tree-function) we cannot say that a parallel vector's life is over when the node code that built it ends. Indeed, the final tree may need this particular value, at the end, to build the resulting tree. In this case, we have to wait until the end of the multi-tree-function itself to delete the values which were used to build a tree. The other parallel vectors can be deleted at the end of the execution of the node which built them. Then, all the

parallel vectors built using an identifier which is *"older"* than the last multi-tree-function call are declared as *garbage collectable*. During the recursion of the next multi-functions, we can imagine a mechanism, on each process, that updates and cleans its own memory. We could also use the `finalize` function of the OCAML garbage collector to warn the processes when a value is garbage collected.

As such a feature is considered as an implementation detail which aims to optimise the memory footprint of a MULTI-ML program, it is not yet implemented in the current MULTI-ML version.

5.1.5 Limitations of the implementation

Currently, the MULTI-ML implementation has several limitations. In the following, we present two of the principal restrictions.

To reference a value defined in the MULTI-BSP memory from the body of a parallel vector, we must declare it as following:

```
let pi=3.14;;

let multi f () =
where node = .... << 2+pi >>
where leaf = 2+pi
```

If the values are not declared as a *list of expressions*, it is not possible to reference them within the scope of a parallel vector. The value must be completely evaluated otherwise, it is not possible to determine which identifier corresponds to a particular value. Indeed, in the case of imbricated bindings, it is not possible to determine a *global* identifier to reference a value. It is thus impossible to tell to a process which value to use.

The multi-function nesting is not yet available. Indeed, as we said previously, a multi-function is not as simple as a function. It implies additional runtime systems to be executed all over the MULTI-BSP architecture. A multi-function needs an *initialisation* phase in order to be executed. Thus, the following code is not valid:

```
let multi g () = ...;;

let multi f () =
 where node = .... << f()+g() >>
 where leaf = ...
```

The multi-function `g` is not initialised in the expression `<< f()+g() >>`, as the expression is in the body of `f`. This multi-function composition has been studied and we propose, as future work in Section 6.2.6, solutions to be able to do it.

5.1.6 Modular Implementation

As an abstract machine is defined onto elementary blocks which aims to be dedicated to simple tasks, it is easy to have a modular implementation of the MULTI-ML language. Such an abstract and generic design is interesting for the evolution of the language. The development is easier and it is possible to propose different implementations of the language to be able to execute MULTI-ML programs on a wide variety of architectures. For example, as it is detailed in Section 5.3.4.1, it is possible to propose communication modules relying on different communication libraries.

5.2 Big step semantics with costs

Thanks to the MULTI-BSP cost model, it is possible to compute the evaluation cost of a given expression. This cost gives an estimation of the execution time of an expression according to the machine specifications. It is even possible to propose a static analysis of a program which gives an approximation of its execution time.

The combination of the big-step semantics and a simple cost algebra allow to make such a performance prediction. In this section we add costs to the big-step semantics introduced in Chapter 4. Thus, the rules behaves as previously.

An inference rule with costs annotations can be written as following:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} e'/C_p} \quad \text{and} \quad \frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathfrak{b}} e'/C_p/ < C_{p.i} >}$$

We can distinguish two ways of annotating rules by costs. First, the left hand side rule stands for evaluations which do not deal with BSP computation, that is to say: (1) generic rules (which can be executed a any level of the architecture); (2) local rules (which can be executed inside the scope of a parallel vector) and (3) MULTI-BSP rules (which are executed *globally*, at the MULTI-BSP level). Here, C_p describes the *amount of cost* a component itself, at level p. When dealing with rules which can be evaluated at level $\mathtt{m}$, we consider the whole machine as a single component where its cost is C_{p_multi}. This cost is the sum of all the costs of all the components of the MULTI-BSP architecture.

The rule on the right hand side handles BSP execution: nodes evaluations. As a node is a particular component where it is possible to express BSP parallelism, we must consider the asynchronous costs generated by BSP computations. To do so, in addition to the cost inherent to the node itself, we keep track of the sub-nodes costs (that were engenders by the evaluations taking place at level p) by using a parallel vector of costs. This vector, denoted $< C_{p.i} >$, lists all the cost of the i components of p: $< C_0, ..., C_n >$. We recall that, for readability, we use n to refer to the number of sub-nodes of a component at level p.

Apart from the cost annotations, the semantics rules behave in a way which is similar to the *standard* rules (defined in the Section 4.2 of Chapter 4)

To manipulate the costs within the rules, we use a simple cost algebra which is defined in Figure 5.7.

Definition 5.2 (*Cost algebra*)

The cost algebra is used to accumulate the cost of evaluation of the semantics rules. The cost algebra relies on integer values and uses simple operators.

The constants **g** and **l**, correspond to the standard MULTI-BSP parameters that are used to, respectively, quantify the bandwidth and the synchronisation cost. To lighten the rules, we assume that **g** and **l** are standing for the MULTI-BSP parameters corresponding to the level of evaluation of the rule. Thus, at level p, they correspond to $\mathbf{g}_p$ and $\mathbf{l}_p$. We also use the notation $\mathcal{S}_\varepsilon(v)$ to compute the size of a value v within environment ε. This function is useful when data transfers are required. A simple data size computation is given in Figure 5.8. We can imagine a more realistic version with allocation costs, garbage collections costs and where the closures costs are propotional to the size of the enclosed environment. It is important to note that, as closures are intended to be communicated, the operator is not defined on parallel primitives, parallel vectors nor trees. For sake of simplification, this function is an approximation of the real size of data. We also consider that the environments of closures are minimal environments, by

$$
\begin{array}{lll}
C & ::= & \\
 & | \ 1 & \textit{Arbitrary unit cost} \\
 & | \ \mathbf{g} & \textit{g parameter} \\
 & | \ \mathbf{l} & \textit{l parameter} \\
 & | \ C \oplus C & \textit{Addition} \\
 & | \ C \otimes C & \textit{Multiplication} \\
 & | \ \max_{i=0}^{n}(C_i) & \textit{Maximum} \\
 & | \ \sum_{i=0}^{n}(C_i) & \textit{Sum} \\
 & | \ \mathcal{S}_{\varepsilon}(v) & \textit{Data size}
\end{array}
$$

(a) Grammar.

$$
\begin{array}{lcl}
C_1 \oplus C_2 & \equiv & C_2 \oplus C_1 \\
C_1 \otimes C_2 & \equiv & C_2 \otimes C_1 \\
\max_{i=0}^{n}(C_i) & \equiv & \textit{maximum}(C_0, ..., C_n) \\
\sum_{i=0}^{n}(C_i) & \equiv & C_0 \oplus ... \oplus C_n
\end{array}
$$

(b) Rules equivalence.

Figure 5.7: Cost algebra definitions.

construction. For example, the environment $\mathcal{E}'$ of $(\overline{(\textbf{fun}\ \ x \rightarrow e)\ \ [\mathcal{E}']})$ only contains free variable of e.

$$
\begin{array}{rcl}
\mathcal{S}(c) & = & 1 \\
\mathcal{S}(op) & = & 1 \\
\mathcal{S}_{\mathcal{E}}(x) & = & \mathcal{S}(v)\ \ if\ \ \{x \mapsto v\} \in \mathcal{E} \\
\mathcal{S}_{\mathcal{E}}(e_1\ e_2) & = & \mathcal{S}_{\mathcal{E}}(e_1) \oplus \mathcal{S}_{\mathcal{E}}(e_2) \oplus 1 \\
\mathcal{S}_{\mathcal{E}}(e_1, e_2) & = & \mathcal{S}_{\mathcal{E}}(e_1) \oplus \mathcal{S}_{\mathcal{E}}(e_2) \oplus 1 \\
\mathcal{S}_{\mathcal{E}}([e_o, ..., e_p]) & = & \sum_{i=0}^{p}(\mathcal{S}_{\mathcal{E}}(e_i)) \\
\mathcal{S}_{\mathcal{E}}(\textbf{fun}\ \ x \rightarrow e\ \ ) & = & \mathcal{S}_{x \circ \mathcal{E}}(e) \oplus 1 \\
\mathcal{S}(\mathcal{E}) & = & \sum_{i=0}^{p}(\mathcal{S}(v)).\ \ \forall x \in \mathcal{E} : \{x \mapsto v\} \\
\mathcal{S}_{\mathcal{E}}(\overline{(\textbf{fun}\ \ x \rightarrow e)\ \ [\mathcal{E}']}) & = & \mathcal{S}_{x \circ \mathcal{E}'}(e) \oplus 1 \oplus \mathcal{S}(\mathcal{E}) \\
\mathcal{S}_{\mathcal{E}}(\textbf{multi}\ \ f\ \ x \rightarrow e_1 \dagger e_2) & = & \mathcal{S}_{x \circ f \circ \mathcal{E}}(e_1) \oplus \mathcal{S}_{x \circ \mathcal{E}}(e_2) \oplus 1 \\
\mathcal{S}_{\mathcal{E}}(\overline{(\textbf{multi}\ \ f\ \ x \rightarrow e_1 \dagger e_2)[\mathcal{E}']}) & = & \mathcal{S}_{x \circ f \circ \mathcal{E}'}(e_1) \oplus \mathcal{S}_{x \circ \mathcal{E}'}(e_2) \oplus 1 \\
\mathcal{S}_{\mathcal{E}}(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) & = & \mathcal{S}_{\mathcal{E}}(e_1) \oplus \mathcal{S}_{\mathcal{E}}(e_2) \oplus \mathcal{S}_{\mathcal{E}}(e_3) \oplus 1
\end{array}
$$

Figure 5.8: Computation of the size of data.

In Figure 5.9 we propose the generic rules that can be evaluated regardless of the locality.

The rules VALUES and OP_EVAL does not generate any additional cost as their evaluations are almost syntactical. With the VAR rule, we need to *lookup* for variables in the environment. The *lookup*() predicate is defined in Figure 4.7. For sake of conciseness, we choose to evaluate the cost of *lookup*() as an atomic cost of one.

The CLOSURE and REC_CLOSURE rules only rely on the size of the final closure. According to the size of the environment, we approximate the evaluation cost of the closure. One can imagine a finer approach taking into account the number of variables stored in the enclosed environment. In this semantic, such a coarse cost approximation is plenty enough.

The rest of the generic rules are simple to handle. We just compute the final cost using the cost of the premises. We add an extra cost when the rule needs an additional operation for the

$$\text{VALUES} \quad \frac{}{\mathcal{M} \vdash \mathbf{cst} \Downarrow_p^{\mathcal{L}} \mathbf{cst}/C_p}$$

$$\text{OP__EVAL} \quad \frac{}{\mathcal{M} \vdash \mathbf{op} \Downarrow_p^{\mathcal{L}} \mathbf{op}/C_p}$$

$$\text{VAR} \quad \frac{\{x \mapsto v\} \in lookup(x, \mathcal{M}, p, \mathcal{L})}{\mathcal{M} \vdash x \Downarrow_p^{\mathcal{L}} v/C_p \oplus 1}$$

$$\text{CLOSURE} \quad \frac{\begin{array}{l}\mathcal{E} = select(\mathcal{M}, \mathcal{F}(\mathbf{fun}\ \ x \to v\ \ ), p, \mathcal{L}) \\ v \equiv \overline{(\mathbf{fun}\ \ x \to e\ \ )[\mathcal{E}]}\end{array}}{\mathcal{M} \vdash \mathbf{fun}\ \ x \to e\ \ \Downarrow_p^{\mathcal{L}} v/C_p \oplus 1}$$

$$\text{REC__CLOSURE} \quad \frac{\begin{array}{l}\mathcal{E} = select(\mathcal{M}, \mathcal{F}(\mathbf{rec}\ \ f\ \ x \to v\ \ ), p, \mathcal{L}) \\ v \equiv \overline{(\mathbf{rec}\ \ f\ \ x \to e\ \ )[\mathcal{E}]}\end{array}}{\mathcal{M} \vdash \mathbf{rec}\ \ f\ \ x \to e\ \ \Downarrow_p^{\mathcal{L}} v/C_p \oplus 1}$$

$$\text{FUN__APP} \quad \frac{\begin{array}{ll}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\mathbf{fun}\ \ x \to e\ \ )[\mathcal{E}]}/C_1 & \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v/C_2 \\ \mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x \mapsto v\} \vdash e \Downarrow_p^{\mathcal{L}} v'/C_3 & \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} v'/C_p \oplus 1 \oplus C_1 \oplus C_2 \oplus C_3}$$

$$\text{REC__FUN__APP} \quad \frac{\begin{array}{l}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\mathbf{rec}\ \ f\ \ x \to e\ \ )[\mathcal{E}]}/C_1 \\ \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v/C_2 \qquad \mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x \mapsto v\} \oplus_p \overline{(\mathbf{rec}\ \ f\ \ x \to e\ \ )[\mathcal{E}]} \vdash e \Downarrow_p^{\mathcal{L}} v'/C_3\end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} v'/C_p \oplus 1 \oplus C_1 \oplus C_2 \oplus C_3}$$

$$\text{OP__APP} \quad \frac{\begin{array}{ll}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \mathbf{op}/C_1 & \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v/C_2 \\ \overline{\mathbf{op}\ \ v} \equiv v' & \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathcal{L}} v'/C_p \oplus 1 \oplus C_1 \oplus C_2}$$

$$\text{LET__IN} \quad \frac{\begin{array}{c}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} v_1/C_1 \\ \mathcal{M} \oplus_p \{x \mapsto v_1\} \vdash e_2 \Downarrow_p^{\mathcal{L}} v_2/C_2\end{array}}{\mathcal{M} \vdash \mathbf{let}\ \ x = e_1\ \ \mathbf{in}\ \ e_2 \Downarrow_p^{\mathcal{L}} v_2/C_p \oplus 1 \oplus C_1 \oplus C_2}$$

Figure 5.9: Generic evaluation rules with costs.

$$\text{REPLICATE}\quad \frac{\begin{array}{l}\forall i \in p \quad \mathcal{M} \oplus_{p_i} \{f \mapsto (\textbf{fun } _ \to e\ )\} \vdash f\ ()\Downarrow^{\mathtt{l}}_{p_i} v_i/C_i \\ \mathcal{C}omm_{||\mathcal{M}||_{p_i}}(\textbf{fun } _ \to e\ )\end{array}}{\mathcal{M} \vdash \textbf{replicate } (\textbf{fun } _ \to e\ ) \Downarrow^{\mathtt{b}}_{p} < v_0, ..., v_n > /C_p \oplus 1/ < C_{p.i} \oplus C_i >}$$

$$\text{DOWN}\quad \frac{\begin{array}{l}\mathcal{M} \vdash x \Downarrow^{\mathtt{b}}_{p} v/1/ < 0 > \\ \mathcal{C}omm_{||\mathcal{M}||_p}(v)\end{array}}{\mathcal{M} \vdash \textbf{down } x \Downarrow^{\mathtt{b}}_{p} < v > /C_p \oplus 1 \oplus \mathcal{S}_{\mathcal{M}}(v) \otimes \mathbf{g} \oplus \mathbf{l} \oplus \max_{i=0}^{n}(C_{p.i})/ < 0 >}$$

$$\text{MKPAR}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e \Downarrow^{\mathtt{b}}_{p} \overline{(e')[\mathcal{E}']}/C_1/ < 0 > \\ \forall i \in p \quad \mathcal{M} \oplus_p \mathcal{E}' \vdash e'\ i \Downarrow^{\mathtt{b}}_{p} v_i/C_i/ < 0 > \\ \mathcal{C}omm_{||\mathcal{M}||_p}(v_i)\end{array}}{\mathcal{M} \vdash \textbf{mkpar } e \Downarrow^{\mathtt{b}}_{p} < v_0, ..., v_n > /C_p \oplus C_1 \oplus 1 \oplus \sum_{i=0}^{n}(C_i) \oplus \sum_{i=0}^{n}(\mathcal{S}_{\mathcal{M}}(v_i)) \otimes \mathbf{g} \oplus \mathbf{l} \oplus \max_{i=0}^{n}(C_{p.i})/ < 0 >}$$

$$\text{APPLY}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e_1 \Downarrow^{\mathtt{b}}_{p} < \overline{(e_i)[\mathcal{E}_i]} > /C_1/ < C_{1.i} > \qquad \mathcal{M} \vdash e_2 \Downarrow^{\mathtt{b}}_{p} < v_0, ..., v_n > /C_2/ < C_{2.i} > \\ \forall i \in p \quad \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}\} \vdash f_i\ i \Downarrow^{\mathtt{l}}_{p.i} v'_i/C_i\end{array}}{\mathcal{M} \vdash \textbf{apply } e_1\ e_2 \Downarrow^{\mathtt{b}}_{p} < v'_0, ..., v'_n > /C_p \oplus C_1 \oplus C_2 \oplus 1/ < C_{p.i} \oplus C_{1.i} \oplus C_{2.i} \oplus C_i >}$$

$$\text{PROJ}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e \Downarrow^{\mathtt{b}}_{p} < v_0, ..., v_n > /C_1/ < C_{1.i} > \\ \forall i \in p \quad \mathcal{M} \oplus_{p.i} \{f \mapsto \overline{(e')[\mathcal{E}']}\} \vdash f\ i \Downarrow^{\mathtt{b}}_{p.i} v_i/0/ < 0 > \qquad \mathcal{C}omm_{||\mathcal{M}||_p}(v_i)\end{array}}{\mathcal{M} \vdash \textbf{proj } e \Downarrow^{\mathtt{b}}_{p} \overline{(e')[\mathcal{E}']}/C_p \oplus 1 \oplus \max_{i=0}^{n}(C_{1.i}) \oplus \sum_{i=0}^{n}(\mathcal{S}_{\mathcal{M}}(v_i)) \otimes \mathbf{g} \oplus \mathbf{l} \oplus \max_{i=0}^{n}(C_{p.i})/ < 0 >}$$

$$\text{PUT}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e \Downarrow^{\mathtt{b}}_{p} < \overline{(e_0)[\mathcal{E}_0]}, ..., \overline{(e_n)[\mathcal{E}_n]} > /C_1/ < C_{1.i} > \\ \forall i, j \in p \\ \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}\} \vdash f_i\ j \Downarrow^{\mathtt{l}}_{p.i} v_{ij}/C_i \\ \mathcal{M} \oplus_{p.j} \{f'_j \mapsto \overline{(e'_j)[\mathcal{E}'_j]}\} \vdash f'_j\ i \Downarrow^{\mathtt{l}}_{p.j} v_{ij}/C_j\end{array}}{\mathcal{M} \vdash \textbf{put } e \Downarrow^{\mathtt{b}}_{p} < \overline{(e'_0)[\mathcal{E}'_0]}, ..., \overline{(e'_n)[\mathcal{E}'_n]} > f/C_{put}/ < 0 >}$$

Figure 5.10: BSP evaluation rules with costs.

$$\text{MULTI__NODE}\quad \frac{\begin{array}{l} isNode(p) \\ \mathcal{M} \vdash e_1 \Downarrow_p^{\mathtt{l}} \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']}/C_1 \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathtt{l}} v/C_2 \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \oplus_p \{f \mapsto \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']}\} \vdash e_1' \Downarrow_p^{\mathtt{b}} v'/C/ < C_i > \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathtt{l}} v'/C_p \oplus 1 \oplus C_1 \oplus C_2 \oplus C \oplus \max_{i=0}^{n}(C_i)}$$

$$\text{MULTI__LEAF}\quad \frac{\begin{array}{l} isLeaf(p) \\ \mathcal{M} \vdash e_1 \Downarrow_p^{\mathtt{l}} \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']}/C_1 \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathtt{l}} v/C_2 \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \vdash e_2' \Downarrow_p^{\mathtt{s}} v'/C \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_p^{\mathtt{l}} v'/C_p \oplus 1 \oplus C_1 \oplus C_2 \oplus C}$$

Figure 5.11: Local evaluation rules with costs.

$$\text{MULTI__DEF}\quad \frac{\begin{array}{l} \mathcal{M}' = select(||\mathcal{M}||_{p_multi}, \mathcal{F}(\textbf{multi}\ \ f\ \ x \to e_1 \dagger e_2)) \\ v \equiv \overline{(\textbf{multi}\ \ f\ \ x \to e_1 \dagger e_2)[\mathcal{M}']} \qquad \mathcal{C}omm_{||\mathcal{M}||_{root}}(v) \end{array}}{\mathcal{M} \vdash (\textbf{multi}\ \ f\ \ x \to e_1 \dagger e_2) \Downarrow_{p_multi}^{\mathtt{m}} v/C_{p_multi} \oplus 1}$$

$$\text{MULTI__CALL}\quad \frac{\begin{array}{l} \mathcal{M} \vdash e_1 \Downarrow_{p_multi}^{\mathtt{m}} \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']}/C_1 \qquad \mathcal{M} \vdash e_2 \Downarrow_{p_multi}^{\mathtt{m}} v/C_2 \\ \mathcal{M} \oplus_{root} \{x \mapsto v\} \oplus_{root} \{f \mapsto \overline{(\textbf{multi}\ \ f\ \ x \to e_1' \dagger e_2')[\mathcal{M}']}\} \vdash e_1' \Downarrow_{root}^{\mathtt{b}} v'/C/ < C_i > \end{array}}{\mathcal{M} \vdash (e_1\ \ e_2) \Downarrow_{p_multi}^{\mathtt{m}} v'/C_{p_multi} \oplus 1 \oplus C_1 \oplus C_2 \oplus C \oplus \sum_{i=0}^{n}(C_i) \oplus broadcast(v')}$$

Figure 5.12: MULTI-BSP evaluation rules with costs.

evaluation. For example, the FUN_APP requires to push values to the top of the memory stack to proceed the evaluation. Thus, we estimate such a cost to one.

The Figure 5.10 gives an overview of the BSP rules. Those rules are the core of the MULTI-ML language and they concentrate most of the costs relative to parallelism. As we are dealing with BSP rules, the cost is now expressed with two quantifiers: (1) the cost of the current node and (2) the cost of the sub-nodes, generated by asynchronous computations.

The cost of the REPLICATE rule is highly related to computation time of the sub-nodes. Each ith sub-node of the current node p performs the computation for a cost C_i. Thus, the cost engendered by such an asynchronous computation is added to the current sub-nodes cost $C_{p.i}$. A cost of 1 is added and correspond to the signal of evaluation which is sent to the sub-nodes. This operation is widely discuss in the implementation chapter (Chapter 5) and is necessary to handle multiple implementations.

The DOWN rule cost is simple. The value v is communicated to the sub-nodes with a cost relative to its size, namely $\mathcal{S}_S(v)$. As the down primitive must communicate a value from a node to its sub-nodes, we must execute a synchronisation barrier. In addition to the cost due to the bandwidth and the latency, we propagate the asynchronous computation costs of the sub-nodes by adding the $\max_{i=0}^{n}(C_{p.i})$. Thereafter, the sub-node's cost is set to zero. On the top of that, we just add lookup cost of one concerning the variable evaluation.

The MKPAR rule is similar to the REPLICATE rule in the sense that it creates a parallel vector. Nevertheless, its cost is quite different. It is composed by the sum of the i computation costs (C_0 to C_n) plus the sum of the v_i transfers costs. Indeed, as the v_i values are computed sequentially on the node before, we need to take into account both the i computations costs plus their communication cost. The asynchronous costs are propagated, as a synchronisation barrier is done. It is conceivable to imagine a finer cost estimation by considering the overlapping of both communications and computations.

As the evaluation of the APPLY rule is done asynchronously, we simply add the evaluation costs of e_1 and e_2. Then, asynchronous cost are added on each sub-components of the node p.

The cost of PROJ is related to the evaluation cost of e. As all the values of the parallel vector $< v_0, ..., v_n >$ must be communicated to the memory of the node p, the cost consists in the sum of communication costs of those values. In addition to that, as PROJ is synchronous, we add the $\max_{i=0}^{n}(C_{p.i})$, and set the sub-node's costs to zero.

The PUT cost is quite dense because of the number of communications which are done during the evaluation of the primitive. Thus, the expression of C_{put} is the following:

$$\begin{aligned} C_{put} = \; & C_p \oplus 1 \oplus C_1 \oplus \max_{i=0}^{n}(C_{1.i}) \oplus \max_{i=0}^{n}(C_i) \oplus \max_{i=0}^{n}(C_j) \\ & \oplus \max_{i=0}^{n}(max(\sum_{j=0}^{n}(\mathcal{S}_{\mathcal{M}}(v_{ij})), \sum_{j=0}^{n}(\mathcal{S}_{\mathcal{M}}(v_{ji})))) \otimes \mathbf{g} \oplus \mathbf{l} \end{aligned}$$

First, we need to add the evaluation cost of e. Then, C_i and C_j describes, respectively, the cost of building the value *to send* and *handling* the received value. As such computations are done in parallel, we took the maximum of C_i and C_j costs. We also need to take into account

the communication costs of all those values. To do so, for each i, we select the maximum value between the cost of *sending* a value from i to j and the cost of *sending* a value from j to i. Finally, we take the maximum of all the ij exchanges.

The Figure 5.11 gives the local rules. Here, we compute the cost of the multi-function's recursive call cost.

Here, only the MULTI_NODE and MULTI_LEAF rules can be evaluated. The costs of the multi-function recursive call taking place on both the node and the leaf is simple. We just add the evaluation cost of e_1 and e_2, plus the multi-function call cost, resulting in the recursive call. The MULTI_NODE rule adds the C_i costs which result from the potential asynchronous computations done on the node. Thus, we collect all the costs engendered by multi-function recursion. As expected, this mechanism is not necessary on the MULTI_LEAF rule, as there is no parallel computation at this level.

Finally, Figure 5.12 describes the rules that are evaluated at the MULTI-BSP level.

The cost of MULTI_DEF is directly related to the size of the multi-function closure. As before, for conciseness reason, we estimate this cost at one.

Concerning the MULTI_CALL rule, we simply add the evaluation cost of e_1 and e_2, plus the multi-function call cost. Like the MULTI_NODE rule, we collect all the costs engendered by multi-function recursion at the *root* node. It is important to note that, as the MULTI-BSP memory may be distributed, we must take into account the cost of broadcasting the value v' to all the component of the MULTI-BSP architecture. In practice, the $broadcast(v')$ cost can be done in a logarithmic way.

We now have the description of a semantics with costs. However, we do not propose an abstract machine relying on this cost. As it seems interesting and as it does not appear to introduce difficulties, it is reserved for future work.

5.3 The compilation toolchain

The compilation toolchain structure is, most of the time, structured by three blocks. A simple toolchain could be structured as follow:

(1) The *frontend* checks the syntax and the semantics of the language, according to its specifications. It could also perform type checking and code analysis to collect informations such as types, errors and warnings. The output of this phase results in an abstract syntax tree (AST).

(2) The *middle-end* takes the AST produced by the front end and performs transformations or optimisations.

(3) The *backend* takes the output of the *middle-end* to produce a machine code, suitable for a particular architecture. It is then possible to execute this code on a machine.

In the context of the MULTI-ML language, the *frontend* performs the parsing and lexing phase, to make sure that the MULTI-ML syntax is respected. Then, the type checker is called to ensure the execution safety of the program. The *middle-end* consists in a code transformation

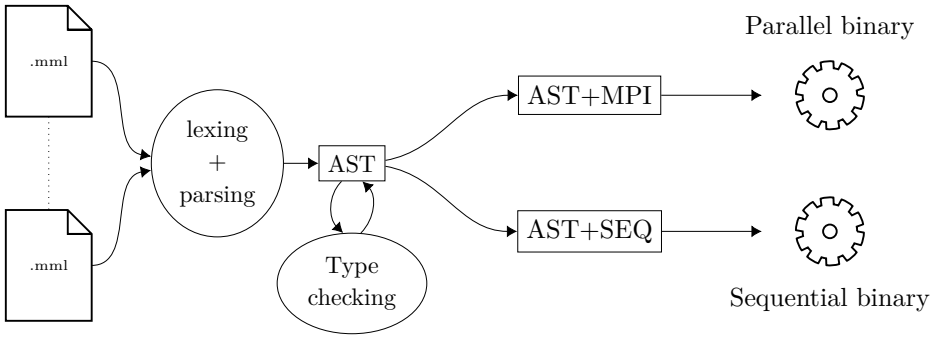


Figure 5.13: The MULTI-ML compilation toolchain.

which takes the abstract syntax tree as input. Then, depending on the requested operation, it transforms the AST by adding parallel or sequential instructions. Finally, the *backend* compiles the given AST to produce a parallel (or sequential) executable. Figure 5.13 summarises the compilation steps.

To build the MULTI-ML toolchain we choose to start from the OCAML 4.02.1 sources. Then, we modified the OCAML parser and lexer, by reducing its expressivity and adding the MULTI-ML constructions (see Section 5.3.1). Then we choose not to modify the typing system of OCAML. Indeed, it is a twisted and rather complicated piece of code which is difficult to handle. We choose to implement our own type checker, as explained in Section 5.3.2. Finally, the code transformation is described Section 5.3.3 and the way that MULTI-ML programs are executed in parallel or sequential is explained in Section 5.3.4.

One may wonder why we did not use the OCAML tools which aims to extend the language. For example, CAMLP5 is a preprocessor and pretty-printer for OCAML programs. Its preprocessor allows to extend the syntax of OCAML and redefine the whole syntax of the language. The pretty printer allows to convert one syntax to another and check the results of syntax extensions. Such a tool seems to be convenient. However, it is not that easy to use. For example, BSML is build over CAMLP4 and its maintenance was difficult. Indeed, CAMLP4 introduced a backward-incompatible update which was very laborious to adapt in order to get a working version of BSML. To avoid such inconvenience, we choose not to use those tools. Of course, as MULTI-ML is based on a particular version of OCAML, all the new syntax or type features introduced by new versions would not be available in MULTI-ML. As our language does not allow the cutting edge OCAML features, it does not matter. Moreover, if compilation optimisations involving performance gains are added to a future version of the OCAML compiler, we will benefit from them, as the OCAML compiler is directly used, in the *backend*.

One unfortunate thing of this approach is the impossibility of bootstrapping MULTI-ML, that is, to be able to recompile MULTI-ML sources with the newly created compiler.

5.3.1 Lexing and parsing

The first phase of the syntactical analysis is lexing. Lexing is a simple process which tokenizes a stream of text: it *breaks a program in words or phrases*. Basically, the lexer identifies all the tokens defined in the language (such as **`let`**, **`fun`**, **`if`**, **`mkpar`**, *etc.*) and generates a new stream of tokens: the parse tree. The definition of the MULTI-ML lexer can be found in the file `lexer.mll`. Thanks to `ocamllex`, which produces a lexical analyser from a set of regular expressions, we obtain a standalone lexer. Then, we must parse our program, using the lexical analyser generated by `ocamllex`. To do so, we use the `ocamlyacc` command, which produces a parser from a context-free grammar specification with attached semantic actions, in the style of `yacc`. From the definition

of the grammar, which can be found in file `parser.mly`, we can produce a parser, written in OCAML.

As previously explained, we took the original OCAML lexer and parser and we modify them to be able to recognise the μMULTI-ML language.

Without going into details, we must also define all the structures needed to handle the MULTI-ML particular constructions. For example, we enrich the parse tree structure (which is declared in file `parsetree.mli`) by adding expressions such as trees, multi-functions and all the MULTI-ML primitives. Thus, the AST produced as an output of the parsing phase is now available for type checking.

5.3.2 Type system implementation

As explained in Chapter 4, the type inference consists in two phases: constraint generation and constraint solving.

From a freshly produced AST, we are able to generate a set of constraints. This set is composed by the constraints on types and their localities, directly obtained by the constraint generation rules, derived from the inference rules.

The constraint set produced is then given, as input, to the constraint solver. It aims to decompose the constraint and, thanks to a constraint solving algorithm, a type unification algorithm and a locality unification algorithm, solve the constraints.

As the AST produced in the frontend respects the syntax, there is no reason for the failure of the constraint generation phase. However, the type errors will be raised during the constraint solving phase.

It is known that generating a *"good"* error message from a type error is an hard task. As explained in [28], there exist many ways to improve the message related to an error. Here, we do not aim to improve the OCAML type error messages. However, as typing MULTI-BSP programs introduces a large amount of potential errors related to parallelism, we have tried to give specific error messages. Thus, to ease the understanding of typing errors and help the programmer to resolve them, we have introduced a new set of *errors*. In addition to the regular OCAML typing errors (such as *Unbound Values*, *Wrong types*, *etc.*) we have defined:

- `Wrong_Locality`: raised when a locality is not acceptable. It is the case if the *accessibility* or *definability* predicated are not respected;
- `Loc_Unification`: raised when the unification between two localities is not acceptable (incoherent localities);
- `Vector_Imbrication`: raised when a nested parallel vector is detected. It is, basically, a specialisation of the `Wrong_Locality` error;
- `Non_Communicable`: raised when a noncommunicable value tries to be communicated.

Those type error messages are not always very accurate, due to the fact that the locality unification may occur at an unexpected time, and thus rely on a locality constraint introduced by an expression far away from the location of the displayed error. However, this set of errors gives significant information helping to correct a piece of MULTI-ML code.

5.3.3 Code transformation

The code transformation is an important step occurring during the middle-end phase. It consists in the transformation of the (intermediate) AST produced by the frontend. As explained in Chapter 4, we need to transform the program written using the MULTI-ML syntactic sugar

into a *core language*. Thus, the code is expressed with a small set of primitives which are implemented depending on their purpose (parallel or sequential code generation). For the parallel code transformation, we obtain an MULTI-ML program, written in OCAML, with a communication module using the MPI routines. This code can be compiled by the standard OCAML compiler and the C compiler for MPI: `mpicc`.

5.3.4 Code execution

In this section we briefly explain how the current MULTI-ML implementation works and how it could be improved.

5.3.4.1 Parallel execution

As MULTI-ML aims to be executed on any hierarchical architectures, we propose a system of generic functor which takes a configuration file as input. This file contains all the information of an architecture, shaped like a tree. We can distinguish the depth, the components and their siblings and also the capabilities of each level, based on the latency, bandwidth and memory size.

To be compatible with a wide set of parallel systems, it is also possible to use different communication libraries. To do so, the communication module is based on a generic functor which takes a communication module as a parameter. For now, the MPI version is the only communication module available. Nevertheless, as the communication primitives are based on a minimal set of instructions, it is easy to write several communication modules based on, for example, the PUB or TCD/IP, as it is done in BSML.

After the compilation of a MULTI-ML code using the `multimlopt.mpi` compiler, it is possible to execute the program on the hierarchical architecture. To do so, we need to execute one instance of the produced executable on each computing unit of the machine. As the binary produced is a SPMD program, the same binary is executed everywhere. Only their initialisation values distinguish them. The MPI tools and this configuration files handle this initial step, seamlessly.

To simplify the MULTI-ML code generation we use closures to communicate values through components. Thus, when a multi-function is called recursively within the scope of a parallel vector, we communicate, to its sub-nodes, a closure containing the multi-function code plus its arguments. As the generated code is SPMD and the type system ensures that the multi-function is declared at the MULTI-BSP level, there is no need to do such a thing: the multi-function code is already known by components, only the arguments need to be communicated. As explained previousely, a solution to avoid the multi-function communication can consist in referencing all the multi-functions with *static identifiers*. Then, only the *identifiers* will be communicated. The same idea can also be used to reference all the MULTI-BSP value. Furthermore, it is also possible to avoid the communication of the code within the scope of parallel vectors. To do so, we need to generate, at runtime, *dynamic identifiers* to reference parallel vectors between a node and its sub-nodes. In combination with the static identifiers, we avoid useless code communications. Such a technique would drastically decrease the size of the communicated values, and would avoid to communicate values that are already known by the receiver. As this technique requires to develop a *global addressing* (to reference values with common identifiers) with *dynamic addressing* between two levels, its implementation is not trivial. To ease the implementation of MULTI-ML, we choose not to develop this ingenious solution.

In the current MULTI-ML implementation, we do not use the shared memory as it could be. Indeed, all the communications are done using the serialisation of OCAML: `Marshal`. This operation takes an expression and produces a closed term which is possible to communicate to another process. This operation imposes to copy a piece of data and communicate it. When a communication is done between two components accessing the same memory, such an operation

is useless. Without side effects, we could simply give, to the receiver, a *pointer* to the data. The usage of `mmap`, a POSIX API which allows to share a memory through processes using a memory-mapped file I/O, could drastically increase the performance of the components accessing a shared memory.

Another concern is the garbage collection. The OCAML garbage collection is designed to be executed with sequential programs. In this context of MULTI-BSP programming, we must deal with parallel data structures which cannot be collected by the standard OCAML garbage collector. Ideally, we could have a dedicated parallel garbage collector. However, writing such an algorithm is not a trivial task [43]. For example, we should be able to clean all the parallel vector. To do so, a node must notify all its sub-nodes that a parallel vector will not be used any more. It requires a fine collection mechanism and a lightweight system of notification between nodes. Currently, the MULTI-ML implementation does not deal with the parallel garbage collection.

As the current implementation of MULTI-ML relies on a set of *daemons* which are running for each components of the MULTI-BSP architecture, the number of process instances is high. This method simplifies the way of implementing MULTI-ML but could be improved. For example, we can imagine that all the leaves will run only one process (instead of $n > 1$). This process will schedule a set of components composed by the leaf itself, plus some of the top nodes. Thus, the leaf process would dynamically switch its execution contexts to evaluate several stages of the architecture. Such an implementation may allow communication optimisations and better performances. However, such a method may introduce problems such as process interleaving and potential deadlocks.

5.3.4.2 The interactive loop

The interactive loop, commonly referenced as *toplevel*, is a very useful OCAML feature. In this mode, the system repeatedly reads OCAML phrases from the input, then typechecks, compiles and evaluates them, then prints the inferred type and result value, if any. Such a feature is very interesting to test and debug auxiliary functions of a more complex program. It is possible to check that the type inferred by OCAML is the one we expect, and then, we can test the behaviour of a piece of code.

As parallel programming may introduce more complex behaviour, the toplevel seems to be almost essential. In BSML, an interactive loop is available to test BSP programmes. It seems mandatory to propose the same feature with MULTI-ML.

Consequently, we propose a toplevel to interactively evaluate MULTI-BSP algorithms. As MULTI-ML can deal with complex hierarchical architectures, we propose to run the toplevel with a given architecture. Thus, it is possible to test the behaviour of a MULTI-ML program on a virtual HPC machine, or on any imaginary architectures. For example, we can trace the data load balancing of our algorithm on various architectures, in the blink of an eye. To declare a new architecture, we just need to define a tree using the OCAML STYLE, as shown in Figure 5.14. Here, for each node and leaf, we give a triplet to denote the latency (in FLOPS), bandwidth (in FLOPS) and memory capacities (in bytes): $(\mathbf{l}, \mathbf{g}, \mathbf{m})$. In this example, we describe an architecture of depth 3 with:

- Level 0; one core as a basic computing unit plus L1/L2 caches: $(\mathbf{p}_0 = 1, \mathbf{g}_0 = 1, \mathbf{L}_0 = 0, \mathbf{m}_0 = 0)$
- Level 1; one chip has 8 cores plus L3 caches: $(\mathbf{p}_1 = 2, \mathbf{g}_1 = 10, \mathbf{L}_1 = 40, \mathbf{m}_1 = 4MB)$
- Level 2; one node has 4 chips plus RAM: $(\mathbf{p}_2 = 2, \mathbf{g}_2 = 100, \mathbf{L}_2 = 25, \mathbf{m}_2 = 16GB)$

```
Node((100,25,17_179_869_184),[
  Node((10,40,4_000),
    [
      Leaf(1,0,0);
      Leaf(1,0,0)
    ])
  ;Node((10,40,4_194_304),
    [
      Leaf(1,0,0);
      Leaf(1,0,0)
    ])
  ])
```

Figure 5.14: A MULTI-ML toplevel configuration file.

```
#let t = mktree (to_string gid);;
- : val t : string tree
# t;;
o "0"
|
--o "0.0"
| |--> "0.0.0"
| |--> "0.0.1"
--o "0.1"
| |--> "0.1.0"
| |--> "0.1.1"
```

Figure 5.15: Interactive tree visualisation.

The tree structure can be visualised through the interactive loop. As the toplevel point of view is directed to the MULTI-BSP level, all the executions *start* within locality `m`. In Figure 5.15, we combine the use of the `mktree` primitive (which builds a tree from an expression) and the `gid` (which returns the global identifier of a component) to build a tree of *gids*. The output value `t` is thus displayed, in the toplevel, with the shape of a tree where the nodes are represented by a circle (`o`) and the leaves by an arrow head (`>`). The depth and hierarchy of the tree is symbolised using separation characters and indentation.

Other standard BSML, and thus OCAML, features work as expected.

5.4 Experimentation and benchmarks

In this section, we propose several examples of MULTI-BSP algorithms. As expected, they are written using the MULTI-ML language and are executed on real architectures.

5.4.1 The execution platforms

To test the performances of the algorithms, we use two clusters. Those machines are thus composed by both shared and distributed memories. The machines used are named MIREV2 and

$p_0 = 0$	$g_3 = 3$	$L_3 = 0$	$m_3 = 0$	$p_0 = 0$	$g_3 = 5$	$L_3 = 0$	$m_3 = 0$
$p_1 = 16$	$g_2 = 6$	$L_2 = 1800$	$m_2 = 20Mb$	$p_1 = 4$	$g_2 = 6$	$L_2 = 800$	$m_2 = 6Mb$
$p_2 = 2$	$g_1 = 89$	$L_1 = 1100$	$m_1 = 64Gb$	$p_2 = 2$	$g_1 = 14$	$L_1 = 472$	$m_1 = 16Gb$
$p_3 = 4$	$g_0 = \infty$	$L_0 = 149000$	$m_0 = 0$	$p_3 = 8$	$g_0 = \infty$	$L_0 = 195400$	$m_0 = 0$

Table 5.1: Multi-BSP parameters of Mirev3 (left) and Mirev2 (right).

MIREV3 and belong to the LIFO: "**L**aboratoire d'**I**nformatique **F**ondamental d'**O**rléans, Orléans, France". Their specifications are the following:

- MIREV2: 8 nodes with 2 quad-core (AMD 2376) at 2.3Ghz with 16Gb of memory per node and a 1Gb/s network.
- MIREV3: 4 nodes with 2 hyper-threaded octo-core CPUs with 16 threads(Intel XEON E5−2650) at 2.6Ghz with 64Gb of memory per node and a 10Gb/s network.

As previously, the version 4.02.1 of OCAML, GCC 4.9 and MPICH 3.1 were used. To estimate the MULTI-BSP parameters of both architectures, we use the *probe* method described in [9]. It measures the time to perform some random collective all-to-all operations of growing size and, then, a least squares methods is applied to compute the values g and L. As the *probe* method was designed for BSP architectures, we recursively call it on each stage of the MULTI-BSP architecture. Thus, we have an acceptable approximation of the MULTI-BSP parameters, even if the configuration of the machines may produce some *noise*. The resulting values can be found in Table 5.1. We can observe that the latency L_1 and L_2 (regarding respectively the RAM and cache memories) are in the same order of magnitude. On the contrary, the L_0 latency, corresponding to network accesses, are way slower. The same behaviour can be observed with the bandwidth g. As the *probe* method is quite sensitive to *noise* (due to the machine configuration), the MULTI-BSP parameters may be inaccurate. However, they illustrate well the hierarchy of memory with their different capabilities and can be used to predict the execution time of an algorithm. In Section 1.4.1.3, we give the evolution of the MULTI-BSP parameters on each architecture, depending on the thread distribution (Figure 1.5 and Figure 1.6)

5.4.2 Eratosthenes's sieve

Algorithm The sieve of Eratosthenes is a simple algorithm used to find all prime numbers up to any given limit. It works by iteratively marking as composite (*i.e.* not prime) the multiples of each prime, starting with the multiples of 2. The multiples of a given prime number are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime.

Here, we present a simple and rather naive implementation of the sieve of Eratosthenes. To do so, it is possible to use a scan with a particular operator. The sieve of Eratosthenes is thus done by the propagation of the prime numbers, thanks to the scan. The MULTI-ML code of a scan is given in Figure 5.16.

The scan algorithm (also known as prefix sums algorithm) implementation is standard functional and high order programming. The scan algorithm takes a binary associative operator $\oplus$ and a set of n elements $[a_0, a_1, ..., a_{n-1}]$ and returns the sums of the prefixes: $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$. The sequential version of the scan algorithm is trivial and its implementation can be easily found in published works. Moreover, it is available in most of the standard library of high order languages, such as OCAML. Here, we describe the behaviour of the MULTI-ML

```
let multi_ml_scan op e l =
  let multi tree m_scan flag lst t =
    where node =
      if flag then
        let splt = mkpar (fun i -> split nprocs i lst ) in
        let r1 = << m_scan true $splt$ t >> in
        let deep = scan_direct op e << at_l $r1$ >> in
        let r2 = << if $pid$ != 0 then
                      mscan false [$deep$] (at_l $r1$)
                    else
                      at_l $r1$ >>
        let v = last << at_l $r2$ >> in
        finally v r2
      else
        let r3 = << m_scan false #lst# t >> in
        let v = last << at_l $r3$ >> in
        finally v r3
    where leaf =
      if flag then
        seq_scan op e lst
      else
        fst (map_and_last op lst (at_s t))
  in m_scan true l (mktree [])

  (* scan_direct:('a->'a->'a)->'a->'a par->'a par *)
  (* last:'a par->'a => gives the last element of a vector *)
  (* seq_scan:('a->'a->'a)->'a'->'a list->'a*'a list
  => computes the scan and also returns the last element *)
  (* map_and_last:'a list->('a->'a)->'a*'a list
  => do a map and also returns the last element *)
```

Figure 5.16: A MULTI-ML scan.

scan. First, on line 1, we define the `multi_ml_scan` value which aims to compute the scan on a MULTI-BSP architecture. It takes, as arguments, an associative operator *op*, a neutral element e and a list l. We then define (on line 2) the multi-tree, function which is going to be used to compute the scan. The function takes three arguments: 1) a `flag` which is used to control the recursion, 2) the list (`lst`) on which the operator must be applied, 3) and a tree `t` which is used to build the final tree. As every multi-tree-function, the body of `m_scan` is divided in two parts: the node and the leaf code.

First, when the evaluation takes place on a node, we check the value of the `flag` (line 4). It allows to distinguish two phases: 1) the first recursion level which distributes the list toward the leaves, 2) the second and following recursion level which only propagate the prefix-sum *to the right*. To distribute the list, we use the `mkpar` primitive and a split function, on line 5. The `split: int -> int -> int list -> int list` function takes a number of slices, an element i and a list, and returns the i^{th} slice of the list. We recall that `nprocs` returns the number of children of the current node. When the distribution is complete, we can call recursively the `m_scan` function to continue the distribution of the list toward the leaves (line 6). When the recursion is terminated, the parallel vector `r1` contains `nprocs` trees, where the scan algorithm was successfully applied. Now, we need to retrieve the roots of the sub-branches (which contain the values of the prefix-sums of the sub-levels) and propagate them. The retrieving of the value to propagate downward is done by using a BSP scan (`scan_direct`) onto the root of the sub-branches (line 7). Then, on line $8-11$, we recall the `m_scan` onto the sub-nodes, with a new list composed by the freshly computed prefixes. We use a conditional statement to avoid the recursive call of the multi-tree-function onto the left-most node, which is useless as its prefix-sum is already computed. To end the node code, we take the right-most element of the `r2` parallel vector, which is the prefix that must be propagated at the next step, on the upper node. Finally, on line 13, we return the value `v` and the vector of branches containing the computed prefix-sum of the sub-levels. Thus, we build the branch of the current level.

The leaf code is much easier. When the `flag` is `true` (which correspond to the first time we reach the leaves: the end of the list distribution), we compute the local scan, in sequential. Otherwise (line 22), we must take into account the new element given in parameter, which correspond to the propagation of a prefix, from a upper node. In both cases, we return, as a value of the leaf, the computed value.

One can note that the usage of the flag is *only* used to distinguish two phases: the list distribution and the prefix propagation. In the presented algorithm, we choose to mix those two phases to reduce the size of the code. However, it is absolutely possible to write two multi-tree-functions, computing each phases independently, and call them, one after the other.

The sieve of Eratosthenes is easy to implement using a simple operator and the scan algorithm (Figure 5.17). The function `op_erato` aims to eliminate all the values of a given list, with the `elim` function. The `erato` function computes the Eratosthenes's sieve on a given list of values (`vals`) using the MULTI-BSP version of scan (`multi_ml_scan`), the Eratosthenes's operator (`op_erato`) and a list of *initial prime numbers*. Thanks to the scan algorithm, the `op_erato` is applied on all the distributed values *"from the left to the right"* with a growing set of values to eliminate.

Cost Analysis The MULTI-BSP cost of the scan algorithm is the following:

$$\sum_{i\in[0\ldots\mathbf{d}[} V_{sp}(i) + \mathcal{O}(|l_{\mathbf{d}}|) + \sum_{i\in[0\ldots\mathbf{d}[} C_i.$$

```
(* elim: int list -> int -> int list *)
let elim l m =
  let rec aux l accu =
    match l with
      [] -> List.rev accu
      | hd::tl -> if (hd mod m=0)
        then aux tl accu
        else aux tl (hd::accu)
  in aux l []

(*op_erato: int list -> int list -> int list*)
let op_erato new_val vals =
  let rec aux cri l =
  begin
    match cri with
    | [] -> l
    | (h::t) ->
      let n_elim = elim l h in
      aux t n_elim
  end
in aux new_val vals

(*erato: (int list -> int list -> int list) -> int list -> int list -> int list*)
let erato op vals =
  multi_ml_scan op_erato [2;3;5] vals
```

Figure 5.17: The sieve of Eratosthenes.

	100_000		500_000		1_000_000		3_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	0.7	1.8	22.4	105.0	125.3	430.7	×	×
16	0.5	0.8	13.3	50.3	68.1	331.5	1200.0	×
32	0.3	0.5	2.6	18.9	11.3	122.2	173.2	×
48	0.5	0.4	1.75	14.5	5.5	88.4	69.3	×
64	0.3	0.3	1.3	8.7	4.1	56.1	51.1	749.9
96	0.3	0.38	1.6	6.3	3.9	30.8	38.1	576.1
128	0.5	0.45	2.1	5.2	4.7	24.3	30.6	443.7

Table 5.2: Execution time (in seconds) of Eratosthenes using MULTI-ML and BSML on Mirev3.

	100_000		500_000		1_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	1.5	1.7	64.5	106.1	402.9	1538.1
16	0.45	0.93	16.0	49.3	91.4	631.7
32	0.14	0.45	4.1	18.7	21.1	219.7
48	0.13	0.40	2.6	11.0	10.8	123.5
64	0.11	0.34	1.89	7.5	8.2	80.5

Table 5.3: Execution time (in seconds) of Eratosthenes using MULTI-ML and BSML on Mirev2.

Where $\sum_{i\in[0...\mathbf{d}[} V_{sp}(i)$ is the total cost of splitting the list toward the leaves (at depth $\mathbf{d}-1$); $\mathcal{O}(|l_{\mathbf{d}}|)$ is the time needed to compute the local sums in the leaves; and $\sum_{i\in[0...\mathbf{d}[} C_i$ corresponds to the cost of diffusing partial sum back to leaves and to add these values to the values held by leaves. This diffusion is done once per node. $V_{sp}(i)$ is the work done at level i to split the locally held chunk l_i and scatter it among children nodes. Splitting l_i in $\mathbf{p}_i$ chunks costs $\mathcal{O}(|l_i|)$ where $|l_i|$, the size of l_i, is $n * \prod_{j\in[0...i[} \frac{1}{\mathbf{p_j}}$ where n is the size of the initial list held by the root node. Scattering it among children nodes costs $\mathbf{p_i} * \mathbf{g_{i-1}} + \frac{n_i}{\mathbf{p_i}} + \mathbf{L_i}$. The sequential list scan cost at leaves is $\mathcal{O}(|l_d|) = \mathcal{O}(n * \prod_{i\in[0...\mathbf{d}[} \frac{1}{\mathbf{p_i}})$. The cost C_i at level i is the cost of a BSP scan, of the diffusion of the computed values toward the leaves, in addition to the sequential cost of a map on list held by leaves. Let s be the size of the partial sum, the cost of BSP scan at level i is $s * \mathbf{p_i} * \mathbf{g_{i-1}} + \mathbf{L_i}$, the diffusion cost is $\sum_{j\in]i...\mathbf{d}]} \mathbf{g_j} * s + l_j$ and the final map cost is $\mathcal{O}(s_{\mathbf{d}})$. The size s may be difficult to evaluate: for a sum of integers it will simply be the size of an integer, but for Eratosthenes sieve, the size of exchanged lists varies depending on which data are held by the node.

Benchmarks We measured the time to compute the Eratosthenes sieve without taking into account the time to generate the initial list of values. The experiments have been done on MIREV2 and MIREV3 using BSML (MPI version) and MULTI-ML over lists of increasing size on an increasing number of processors. The processes have been assigned to machines in order to scatter as much as possible the computation among the machines, *i.e.* a 16 process run will use one core on each processor of the 8 machines of MIREV3. Tables 5.2 and 5.3 show the results of our experimentation, where × stands for unfinished computations.

We notice that, on both MIREV2 and MIREV3, the efficiency of MULTI-ML on small lists is poor. It can be explained by the cost overhead introduced by the initialisation of the MULTI-ML *"system"*, which is heavier that the BSML one. On MIREV3, as the number of processes grows, the execution time variance is smaller, but is still in favour of BSML.

Nevertheless, as the size of the data grows, MULTI-ML exceeds BSML. For example, on

MIREV3 with 1.000.000 elements and 64 processors, MULTI-ML is 13 times faster than BSML. The same behaviour can be observed on MIREV2, using the same parameters, with a speedup of 10. This difference is due to the fact that BSML communicates through the network at every super step; while MULTI-ML is focusing on communications through local memories and finally communicates through the network. Thus, the BSML algorithm is going to saturate the network with a large amount of communications. This congestion drastically decrease the performances. On the contrary, the MULTI-ML algorithm will communicate through local memories, and only a few communications will take place on the network. The network is not saturated and the performances are maximised.

6

Conclusion and perspectives

Parallel computations are used almost everywhere. They offer the possibility to compute data faster than sequentially by taking advantage of parallel architectures ranging from small embedded chips to cutting edge hyper-clusters. However, designing and programming parallel algorithms is challenging and a rather not straightforward task. Several complex parameters must be taken into account in order to obtain performance and scalability and portability. Furthermore, parallel algorithms are prone to errors (such as deadlocks, data-races, *etc.*) and we must ensure safe executions. To do so, structured programming models and bridging models have been proposed in published materials. We choose to explore the MULTI-BSP model, which is an extension of the BSP model. BSP has proven itself a highly reliable bridging model that can tackle all the parallel programming problematics. Nevertheless, BSP was designed for flat architectures which was the standard of yesterday's parallel machines. As today's machines tend to be more and more hierarchical, we think that MULTI-BSP is the logical answer to this evolution. Its multi-stage capabilities allow us to take advantage of the properties of today's architectures with a simple bridging model to unify them all.

In this thesis we introduce MULTI-ML as a new programming language "à la ML", which is based on the MULTI-BSP model. MULTI-ML is an extension of BSML, a language conceived to program BSP algorithms "à la ML". Its aim is to draw strength from the MULTI-BSP model and enable us to program hierarchical architectures with ease. To do so, we offer a way to manage distant memories through the use of parallel vectors. We also define *trees* as a new parallel data structure that handle multi-stage data, where the data distribution is managed by the programmer. Furthermore, we introduce the concept of multi-function, which has the ability to recursively go through the hierarchical architecture.

In the following, we summarise the contributions regarding MULTI-ML semantics, a type system and its implementation. Finally, we discuss the perspectives around the evolution of MULTI-ML.

(1) The typing of expression e generates a valid set of constraints;

(2) The typing system and the semantics ensure a safe evaluation;

(3) The abstract compilation scheme produces a generic code;

(4) The cost model gives a performances prediction;

(5) The implementation allows the execution of a MULTI-ML program.

Figure 6.1: Summary of this thesis.

6.1 Contributions

Figure 6.1 aims to summarise all the contributions of this thesis. The reading of this figure is similar to the manuscript. From the left hand side, we describe the founding elements of MULTI-ML: the BSP model and BSML, its ML implementation (unfilled arrows). Both have inspired the MULTI-ML language. The dashed arrows symbolise hierarchical extensions. Then, we unfold (using plain arrows) the main steps of the conception of MULTI-ML: from the type system, the semantics and the abstract compilation scheme, to the implementation and testing phase. We also describe all the properties resulting from our approach.

In (1) we can see that we develop an algorithm of type inference for MULTI-ML. The typing system (2) ensures the safe execution of MULTI-ML programs thanks to a formal semantics. We can observe that the *locality constraints* dues to the MULTI-BSP model influences both the typing system and the semantics. Thus, with (3) , we are able to compile a MULTI-ML program. The cost model of MULTI-BSP (4) allows to make performance predictions. In association with (5) , we propose an implementation relying on communication modules such as MPI.

If a type system gives a valid type for e and if the evaluation of e produces a value then the compilation of e is valid. Then, we propose the following result:

Final result 6.1 (*Correctness of a* MULTI-ML *program*)

Let e be an expression standing for a MULTI-ML *program, Γ a type environment and C a set of constraints generated from e, then: $C \vdash e : \tau_\pi/\varepsilon[c]$ implies $\Gamma \vdash e : \tau_\pi/\varepsilon[c]$ implies $e \Rightarrow_{safe}$. Finally, if $\mathcal{WF}_(e)$ we have $\langle\langle e, ..., e\rangle\rangle \Rightarrow_{safe}$.*

6.1.1 The Multi-ML language

MULTI-ML has been designed to program hierarchical architectures with multiple levels of memory. As it is inspired by the MULTI-BSP model, its programming model is based on a tree structure with nested components. The MULTI-ML language slightly differs from the original idea of MULTI-BSP in the sense that nodes have computation capabilities. While MULTI-BSP trees are composed by nodes with storage capacities and leaves with computation capabilities, in MULTI-ML it is possible to execute codes on nodes. The node codes must be as lightweight as possible and focus on data scattering and gathering. This ability is mainly used to manage parallel computation during the multi-stage execution. Such a parallel computation is written using the BSML primitives plus some additional MULTI-BSP primitives. Moreover, we propose a new type of recursion by introducing the *multi-functions*. Thanks to the combination of parallel vectors and multi-functions, it is easy to recurse through the different stagess of a hierarchical architecture. To strengthen its hierarchical abilities, we also propose a new parallel data structure that handles hierarchically distributed values: the *tree* structure.

Such hierarchical architectures are composed by layers of memory that tend to decrease in size as we draw closer to the leaves. As all those memories are physically different in terms of both size and bandwidth, we must take it into account. To do so, MULTI-ML proposes a system of locality that gives information on the memory where the data belongs. It is useful to manage data in order to get the best performance and avoid expensive communication. The locality information is also used as a safeguard to ensure well founded communications.

We could almost say that MULTI-ML is an extension of BSML with hierarchical capabilities. Nevertheless, some concepts that are the essence of BSML differ from MULTI-ML. As the execution model of MULTI-ML is dedicated to hierarchical machines, memories are separated. Thus, a value defined in a node must be explicitly communicated to a sub-node in order to be used on that sub-node. This behaviour differs from the *global* execution that is used in BSML. Indeed, as the

global code of a BSML program is replicated, every global value is available in every memory, through parallel vectors. This is not the case with MULTI-ML.

The MULTI-ML language aims to propose a way to program hierarchical architectures more efficiently than the BSP model. Even if its implementation does not rely on a low-level and high performance language commonly used in HPC, its strength resides in its structured programming model. MULTI-BSP is disconnected from the implementation details and places the emphasis on a way to exploit hierarchical architectures by designing efficient algorithms instead of writing bare metal optimisations.

Compared to BSP, MULTI-BSP adds complexity in its model in order to enhance its expressivity. Its ability to handle multi-stage memories and allow sub-synchronisations is attractive but, as MULTI-BSP is a young model, it has to prove itself. One can ask if MULTI-BSP is too complex to be adopted in place of BSP. As expected, it seems harder to design MULTI-BSP algorithms instead of BSP ones. Thus, programming such MULTI-BSP algorithms (using MULTI-ML) seems more complicated than programming BSP algorithms (using BSML). This overhead may discourage algorithmicists and programmers. Even if it seems harder to program with MULTI-ML, the language can be seen as a tool for experts who want to develop skeletons for *turnkey solutions*.

The promise of unifying various hierarchical architectures under a bridging model that ensures efficiency, safe execution, performance prediction and scalability must be thoroughly researched.

6.1.2 Semantics and typing

To get a formal description of the MULTI-ML language, we choose to write an operational semantics. It allows us to describe the behaviour of the evaluation of an expression. As the semantics is disconnected from the current implementation, it allows us to reason on well founded terms. In addition to that, and thanks to the MULTI-BSP cost model, we propose an operational semantics with costs. This annotated semantics is useful to determine the cost of all constructions of the language. It also gives hints on the way the implementation must be done in order to respect the MULTI-ML concepts. Furthermore, it could be used to predict performance of a given algorithm.

As well as in ML, type safety ensures that a program is well-formed and thus, *"cannot go wrong"*. To guarantee the evaluation safety of MULTI-ML programs, we have defined a specific type system. This type system guarantee parallel program safety of MULTI-ML algorithm by rejecting unsound constructions. It mainly deals with replicated coherency, parallel structures nesting and memory level compatibility. To do so, we choose to enrich the ML type system with type annotations, effects and constraints. Thanks to the conjunction of a purely constraint based type system and a constraint solver, we are able to propose an algorithm that can reject unsound expressions. For example, we can ensure replicated coherency and avoid programs with erroneous communications or unauthorised parallel structure nesting. Currently, the type system is quite limited. All the OCAML features are not yet available and the multi-functions have high-order limitations. Thus, it is not yet possible to compose multi-functions or make partial applications. In Section 6.2.6, we propose solutions to overcome those limitations.

6.1.3 Implementation

We propose a formalisation of an abstract compilation scheme for a MULTI-ML program. Thus, we provide a detailed formal description of a parallel runtime system for MULTI-ML. To do so, we have defined a set of *elementary blocks* that describes the essential features which are disconnected from the low-level implementation details. Those *elementary blocks* are used to propose a compilation scheme for each MULTI-ML primitive. On top of that, we define the execution

scheme of a MULTI-ML program execution. This formalised abstract compilation scheme also ensures a greater confidence for the design of the MULTI-ML compiler.

MULTI-ML is based on ML and is implemented over OCAML. Thus, we choose to modify the OCAML sources from the parsing and lexing phase to the abstract syntax tree generation. The MULTI-ML type inference system is based on the OCAML type system, however it is not directly based on its implementation. As we choose to start the MULTI-ML implementation by using a limited core language: μ-MULTI-ML, the type system works on a few OCAML constructions. New features can be added, as new constraints, to the type system, gradually.

To complete the compilation toolchain, we take a sound abstract syntax tree in order to generate a valid OCAML parallel code. Currently, the parallel implementation relies on MPI. The usage of MPI allows good performances on a wide variety of architectures. However, as the implementation relies on generic modules, it is possible to use any communication library. For example, one can imagine an implementation dedicated to shared-memory models. A sequential version of MULTI-ML is also available to simulate the execution on various hierarchical architectures, in sequential. A toplevel based on the sequential implementation is available to provide an interactive loop for test and debug.

6.1.4 Benchmarks and results

Through practical experimentation, we have shown the benefits of the MULTI-BSP model. This approach takes advantage of the hierarchical architectures by allowing the sub-synchronisations of each machine of the cluster. Conversely, the BSP model interprets the whole machine as a flat set of processors communicating through a common network. When algorithms require communications between all the processors at the same time, BSP algorithms induce a network congestion due to the network bottleneck. On the contrary, the MULTI-BSP model avoids the network congestion by allowing communications on sub-groups of components. Thus, the communications are first done locally, between a few components; and finally, the last communications are done through the network.

In addition to avoid network congestion, the MULTI-BSP model favours communications between components sharing a low-level and high performance memory. It is thus possible to take advantage of the hierarchical memories of an architecture.

Our benchmark shows that, in practice, the performances of bottleneck sensitives algorithms are better using MULTI-ML than BSML.

6.2 Perspectives

The MULTI-ML language provides many perspectives of future work. In the following, we present the most promising research directions to our mind.

6.2.1 Certified parallel programming

As BSML is a purely functional extension of ML, it is possible to use a theorem prover (COQ) to get an axiomatization of the BSML primitives. Thus, a certified parallel program can be extracted, directly from the program proof written in COQ [61, 68]. Such a property is quite rare in the world of parallel programming and is worthwhile. As MULTI-ML is an extension of BSML, it seems legitimate to wish the same property, for MULTI-BSP algorithms. To do so, we need a syntactic extension to be able to handle the multi-function concept in COQ. This *syntactic sugar* is necessary to express the MULTI-ML primitives in *pure* COQ. To do so, we may use shallow embedding, as it is done in BSML, for a fragment of MULTI-ML. The recursive calls due to the

recursion through the hierarchical architecture may introduce tricky proof of termination. The formalisation of MULTI-ML in COQ seems to be a difficult and rather non trivial task. Such a tool would lead to systematic development of certified MULTI-BSP algorithms, and thus, to certified parallel programs.

Alternatively, it is also interesting to imagine a way of programming MULTI-BSP algorithms with a skeleton. We could use *hylomorphisms* [115] to capture our recursion schemes in order to generalise them. Intuitively, for a simple divide-and-conquer algorithm, `hylo T f g` is a recursive algorithm where `T` is the recursive call tree, `f` is the *scatter* part and `g` the *gather* part. The required "muscles" of such a skeleton could be: 1) `down` for data distribution; 2) `leaf` for leaves computations; 3) `up` for results scattering; 4) and `control` to control recursion. Even if the expressivity of this approach is similar to the MULTI-ML language, the simulation is not trivial.

6.2.2 GPU integration

As **G**raphical **P**rocessing **U**nits (or GPU's) are widespread on hierarchical architectures, it seems interesting to handle them in the MULTI-BSP language. Indeed, **G**eneral **P**urpose **G**raphics **P**rocessing **U**nits (or GPGPU) is a non negligible way of computing in the world of HPC. The support of GPU's in MULTI-ML is thus a promising feature which could be achieved using SPOC [16]: a set of tools for GPGPU programming in OCAML.

Basically, a GPU is a computing unit composed by many computational units accessing a very high bandwidth memory. A GPU can be seen as a shared memory architecture. Nevertheless, as GPUs work in a very particular way (due to their graphical capabilities), they need a particular programming scheme. It is not possible to program them *as usual*. A *kernel* is a code written in a specific language that can be run on a GPU. We can imagine an extension of the MULTI-ML multi-function which consists in adding a kernel code, next to the node and leaf codes. The kernel code would be executed on GPU, when it is available.

However, adding a GPU into the MULTI-BSP tree is not a that trivial extension. In the following figures, we represent a GPU as a triangle, where the thick lines stand for the special *link* needed to integrate this new component. The GPU integration can be seen in many ways, for example:

1) A simple unit of computation (Figure 6.2a), that is to say a leaf. In this case, the GPU is accessible at the same stage of the leaves;

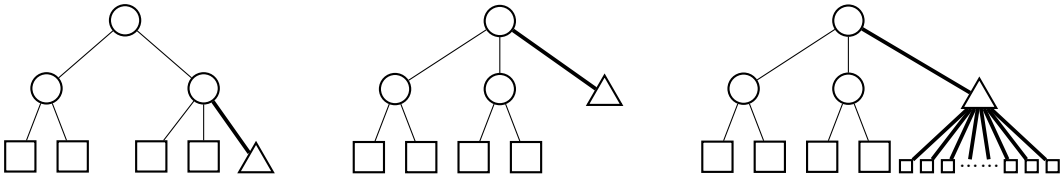
2) A particular node available at an arbitrary stage of the architecture that runs a kernel code (Figure 6.2b);

3) A node with many leaves corresponding to the GPU threads or streaming multiprocessors (Figure 6.2c). It is thus possible to finely program the GPU.

Those proposals give two principal points of views. The first and second one interpret the GPU as a *black box* where the GPU execution is done seamlessly, without fine control. On the contrary, the third approach aims to allow a precise way of programming GPUs by decomposing it as a tree. Those solutions introduce an unbalanced tree, something which is not considered by the MULTI-BSP model and must be studied in details.

As it seems non trivial to modify the MULTI-BSP model in its essence, we may favour a balanced approach. Thus, we can image the following methods:

1) At a given stage, a GPU is accessed by several nodes of the architecture, at the same time (Figure 6.3a);



(a) Single GPU leaf. (b) Single GPU node. (c) Detailed GPU node.

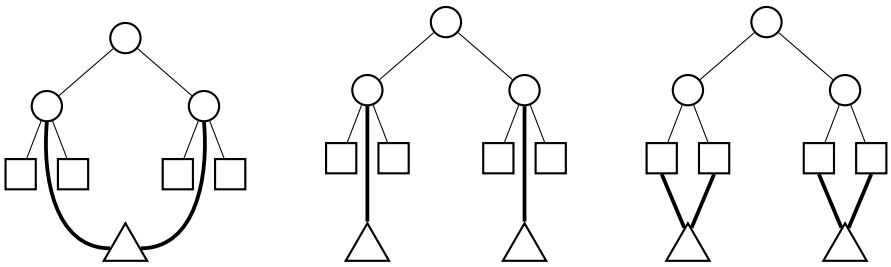
Figure 6.2: Unbalanced GPU integration.

2) All the nodes of a given stage can access their *own* GPU (Figure 6.3b). We can imagine an implementation where the GPU is *virtually* split;

3) All the leaves have access to their *own* GPU (Figure 6.3c);

Those approaches are balanced and seem in accordance to the MULTI-BSP model. Moreover, one can imagine that accessing a GPU must be done from a node with RAM memory in order to achieve better performance. On the contrary, a GPU accessed from a node (or leaf) with a small amount of memory may be inefficient.

All those considerations may lead to an extended version of MULTI-ML that handles GPUs, thanks to SPOC. Thereby, we will propose a language able to program hybrid and hierarchical architectures, which is the standard of today's cutting edge machines.



(a) Shared GPU. (b) Shared GPU node. (c) Shared leaf GPU.

Figure 6.3: Balanced GPU integration.

6.2.3 Examples

MULTI-BSP is a young model which is not yet widespread. There are few algorithms based on this model in published materials. Moreover, there is no complete implementation that are available for testing. It seems important to write MULTI-BSP algorithms in order to demonstrate the benefits offered by the model. As a proof of concept, writing MULTI-BSP algorithms using MULTI-ML would justify the existence of such a language. Then, it will be possible to benchmark a given algorithm on various architectures to experience the scalability of MULTI-BSP algorithms. Comparing the results with the cost predictions of the MULTI-BSP model would also point out the benefits of such a structured model.

6.2.4 Costs analysis

The cost analysis of an algorithm is one of the key features of the MULTI-BSP model. Reasoning on algorithmic costs allows to develop efficient algorithms running on MULTI-BSP architectures. Moreover, writing a MULTI-BSP algorithm with an optimal cost allows to obtain an *immortal*

algorithm that can be run optimally on every hierarchical architecture. As the cost model is embedded in the semantics of the language, it seems interesting to propose a way to extract *automatically* the cost of a given program. One can imagine a compilation tool that gives a runtime estimation of a program, based on the computation and network capabilities of an architecture. Furthermore, one can imagine to combine the resource bound analysis proposed in [89] with MULTI-ML in order to estimate the amount of resources exploited by a program at runtime. In terms of resources, it is also possible to consider a way to estimate the power consumption of a system, based on its specifications. Thus, it would be possible to obtain the efficiency of an algorithm, based on its execution time and energy consumption.

6.2.5 Multi-ML implementation

Currently, MULTI-ML relies on a relatively un-optimised implementation. As we use closures to communicate data, the communication between components is not optimised. The usage of closures has simplified the development of MULTI-ML and will also ease its maintenance. However, one can imagine a unified memory addressing that would avoid closure communications. Thus, the communications would only rely on pure data communication with shared function identifier. Moreover, the implementation of an efficient parallel garbage collector would improve the memory management of MULTI-ML programs.

6.2.6 Multi-ML extensions

Language A certain amount of work is also needed to manage parallel input/output (I/O), as it was done in [62] with BSML. Indeed, I/O at the MULTI-BSP level (when all the computing units are concerned) is not trivial to handle. We must be able to synchronise all the computing units on a single channel. Such consideration introduces many questions on determinism, coherency and performances. It seems interesting to propose a MULTI-ML library to ease I/Os. However, it also raises many questions regarding the design and implementation of such features.

Exceptions is an important feature that must be embedded in the MULTI-ML language. The MULTI-ML implementation does not propagate exceptions in a proper way, yet. It seems natural to propose, as in BSML, a way to handle exceptions with a pattern matching of the form: `trypar … withpar …`. This pattern matching involves the definition of a small language of regular expressions to be able to match parallel exceptions easily, as it is the case in BSML. It seems quite trivial to do the same thing in MULTI-ML.

It also seems very interesting to propose a primitive of *superposition*, at it is the case in BSML. The *superposition* is a purely functional construction that fuses the communications and synchronisations of the evaluation of two expressions given in argument. Such an executions would preserve the MULTI-BSP model and would allow to overlap the communications and synchronisations of two multi-stage executions, resulting in better performance. However, the implementation of *superposition* was a very difficult task in BSML and could be even more difficult in MULTI-ML.

Type system As we said previously, MULTI-ML does not implement all the OCAML features. In the first time, we may add the variants (or sum types). This construction seems trivial but it introduces many locality constraints. Therefore, a way to match patterns in parallel vectors, define records and modules may be considered. However, modules introduces difficulties regarding exception handling. Indeed, as exceptions can be defined in modules, their definition must be known by the whole architecture to be considered.

Regarding the MULTI-ML type system itself, we can imagine a way to exceed the limitations of multi-functions composition. As a multi-function defined outside the scope of another one is

not directly usable by the latter (because of its locality), we may imagine a way to relax this constraint. For example, we can introduce a new keyword to be able to *re-bind*, locally, a multi-function inside the definition context of another one. To do so, we just need to add `where f = f` to the definition of a multi-function, where `f` is a predefined multi-function. The multi-function `f` is thus *re-bound* during the initialisation of another multi-function. This solution is simple and does not involve important changes. We could also update the grammar of constraints in order to be able to have multiple latent effects on arrows types. Adding the disjunctions to the grammar would allow us to have the latent effect $\xrightarrow{\epsilon}$ where $\epsilon = \mathtt{m} \vee \mathtt{l}$. Thus, we could use multi-functions in a high-order way.

We may also use the type system's informations to offer code optimisations. For example, by proposing a smarter way to serialise values, based on their types and localities.

Bibliography

[1] Marcelo Alaniz, Sergio Nesmachnow, Brice Goglin, Santiago Iturriaga, Veronica Gil Gosta, and Marcela Printista. "MBSPDiscover: An Automatic Benchmark for MultiBSP Performance Analysis". In: *High Performance Computing*. Communications in Computer and Information Science 485. Springer Berlin Heidelberg, Oct. 20, 2014, pp. 158–172 (page 29).

[2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. "LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation". In: *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '95. New York, NY, USA: ACM, 1995, pp. 95–105 (page 35).

[3] Martin Helmut Alt. "Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance". Münster University, 2007 (page 18).

[4] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and Kai Tan. "Generating Parallel Programs from the Wavefront Design Pattern". In: *Proceedings 16th International Parallel and Distributed Processing Symposium*. Proceedings 16th International Parallel and Distributed Processing Symposium. Apr. 2002, 8 pp– (page 43).

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58 (page 21).

[6] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. "MPI on a Million Processors". In: *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 20–30 (pages 1, 3).

[7] Jost Berthold, Mischa Dieterle, Oleg Lobachev, and Rita Loogen. "Parallel FFT with Eden Skeletons". In: *Parallel Computing Technologies*. Ed. by Victor Malyshkin. Lecture Notes in Computer Science 5698. Springer Berlin Heidelberg, Aug. 31, 2009, pp. 73–83 (page 64).

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004 (page 20).

[9] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, 2004 (pages 11, 13–15, 22, 23, 156).

[10] Guy E. Blelloch. *NESL: A Nested Data-Parallel Language*. Pittsburgh, PA, USA: Carnegie Mellon University, 1992 (pages 62, 80).

[11] Guy E. Blelloch, Jeremy T Fineman, Phillip B. Gibbons, and Julian Shun. “Internally Deterministic Parallel Algorithms Can Be Fast”. In: Conference: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2012 (pages 10, 15).

[12] Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. “Parallel Programming Must Be Deterministic by Default”. In: *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*. HotPar’09. Berkeley, CA, USA: USENIX Association, 2009, pp. 4–4 (page 10).

[13] Olaf Bonorden. “Versatility of Bulk Synchronous Parallel Computing : From the Heterogeneous Cluster to the System on Chip”. Jan. 1, 2008 (page 22).

[14] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. “The Paderborn University BSP (PUB) Library”. In: *Parallel Computing* 29.2 (Feb. 2003), pp. 187–207 (page 54).

[15] George Horaţiu Botorog and Herbert Kuchen. “Efficient High-Level Parallel Programming”. In: *Theoretical Computer Science* 196.1 (Apr. 6, 1998), pp. 71–107 (page 3).

[16] Mathias Bourgoin, Emmanuel Chailloux, and Jean Luc Lamotte. “SPOC: GPGPU Programming through Stream Processing with OCaml”. In: *Parallel Processing Letters* 22.2 (May 2012). 12 pages, p. 1240007 (pages 7, 168).

[17] Wadoud Bousdira, Frédéric Gava, Louis Gesbert, Frédéric Loulergue, and Guillaume Petiot. “Functional Parallel Programming with Revised Bulk Synchronous Parallel ML”. In: *2010 First International Conference on Networking and Computing (ICNC)*. 2010 First International Conference on Networking and Computing (ICNC). Nov. 2010, pp. 191–196 (page 20).

[18] F. Cappello, P. Fraigniaud, B. Mans, and A. L. Rosenberg. “HiHCoHP-Toward a Realistic Communication Model for Hierarchical Hyperclusters of Heterogeneous Processors”. In: *Parallel and Distributed Processing Symposium., Proceedings 15th International*. Parallel and Distributed Processing Symposium., Proceedings 15th International. Apr. 2001, 6 pp.– (page 37).

[19] F. Cappello, A. Guermouche, and M. Snir. “On Communication Determinism in Parallel HPC Applications”. In: *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. 2010 Proceedings of 19th International Conference on Computer Communications and Networks. Aug. 2010, pp. 1–8 (page 10).

[20] Franck Cappello, Pierre Fraigniaud, Bernard Mans, and Arnold L. Rosenberg. “An Algorithmic Model for Heterogeneous Hyper-Clusters: Rationale and Experience”. In: *International Journal of Foundations of Computer Science* 16 (02 Apr. 1, 2005), pp. 195–215 (page 37).

[21] Luca Cardelli and Andrew D. Gordon. “Mobile Ambients”. In: *Theoretical Computer Science* 240.1 (June 6, 2000), pp. 177–213 (page 6).

[22] Nicholas J Carriero, David Gelernter, Timothy G Mattson, and Andrew H Sherman. “The Linda Alternative to Message-Passing Systems”. In: *Parallel Computing*. Message Passing Interfaces 20.4 (Apr. 1, 1994), pp. 633–655 (page 63).

[23] Hojung Cha and Dongho Lee. “H-BSP: A Hierarchical BSP Computation Model”. In: *The Journal of Supercomputing* 18.2 (Feb. 2001), pp. 179–200 (page 37).

[24] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and W. Pfannenstiel. “Nepal - Nested Data Parallelism in Haskell”. In: *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*. Euro-Par ’01. London, UK, UK: Springer-Verlag, 2001, pp. 524–534 (page 63).

[25] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. “Data Parallel Haskell: A Status Report”. In: *Declarative Aspects of Multicore Programming* (2007) (page 63).

[26] Albert Chan, Frank Dehne, and Ryan Taylor. “CGMGRAPH/CGMLIB: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines”. In: *Int. J. High Perform. Comput. Appl.* 19.1 (Feb. 2005), pp. 81–97 (page 22).

[27] Barbara Chapman. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2008 (page 5).

[28] Arthur Charguéraud. “Improving Type Error Messages in OCaml”. In: *Electronic Proceedings in Theoretical Computer Science* 198 (Dec. 5, 2015), pp. 80–97. arXiv: 1512.01897 (page 152).

[29] Philipp Ciechanowicz and Herbert Kuchen. “Enhancing Muesli’s Data Parallel Skeletons for Multi-Core Computer Architectures”. In: *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*. HPCC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 108–113 (page 45).

[30] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. *The Münster Skeleton Library Muesli — A Comprehensive Overview*. University of Münster, Tech. Rep., 2009 (page 45).

[31] Murray Cole. “Algorithmic Skeletons: Structured Management of Parallel Computation”. University of Glasgow, 1989 (pages 3, 40).

[32] Murray Cole. “Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming”. In: *Parallel Comput.* 30.3 (Mar. 2004), pp. 389–406 (pages 15, 18, 43).

[33] V. G. Costa, A. M. Printista, and M. Marin. “A Parallel Search Engine with BSP”. In: *Third Latin American Web Congress (LA-WEB’2005)*. Third Latin American Web Congress (LA-WEB’2005). Oct. 2005, 10 pp.– (page 22).

[34] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. “LogP: Towards a Realistic Model of Parallel Computation”. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’93. New York, NY, USA: ACM, 1993, pp. 1–12 (page 34).

[35] Rodrigo Da Rosa Righi, Laércio Lima Pilla, Nicolas Maillard, Alexandre Carissimi, and Philippe Olivier Alexandre Navaux. “Observing the Impact of Multiple Metrics and Runtime Adaptations on Bsp Process Rescheduling”. In: *Parallel Processing Letters* 20 (02 June 1, 2010), pp. 123–144 (page 22).

[36] Luis Damas and Robin Milner. “Principal Type-Schemes for Functional Programs”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’82. New York, NY, USA: ACM, 1982, pp. 207–212 (page 122).

[37] Marco Danelutto and Massimo Torquati. “Structured Parallel Programming with “Core” FastFlow”. In: *Central European Functional Programming School*. Lecture Notes in Computer Science 8606. Springer International Publishing, 2015, pp. 29–75 (page 44).

[38] Olivier Danvy and Jan Midtgaard. “Abstracting Abstract Machines : Technical Perspective”. In: *Association for Computing Machinery, Inc.* Communication of the ACM 54.91 (2011) (page 131).

[39] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. "Parallel Programming Using Skeleton Functions". In: *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*. PARLE '93. London, UK, UK: Springer-Verlag, 1993, pp. 146–160 (page 3).

[40] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113 (pages 21, 41, 44).

[41] Narsingh Deo and Paulius Micikevicius. "Coarse-Grained Parallelization of Distance-Bound Smoothing for the Molecular Conformation Problem". In: *Distributed Computing*. Ed. by Sajal K. Das and Swapan Bhattacharya. Lecture Notes in Computer Science 2571. Springer Berlin Heidelberg, Dec. 28, 2002, pp. 55–66 (page 22).

[42] Edsger W. Dijkstra. "Letters to the Editor: Go to Statement Considered Harmful". In: *Commun. ACM* 11.3 (Mar. 1968), pp. 147–148 (page 2).

[43] Damien Doligez and Georges Gonthier. "Portable, Unobtrusive Garbage Collection for Multiprocessor Systems". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. New York, NY, USA: ACM, 1994, pp. 70–83 (page 154).

[44] N. Drosinos and N. Koziris. "Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*. 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. Apr. 2004, pp. 15– (page 7).

[45] Ralph Duncan. "A Survey of Parallel Computer Architectures". In: *Computer* 23.2 (Feb. 1990), pp. 5–16 (page 4).

[46] Jörn Eisenbiegler, Welf Löwe, and Wolf Zimmermann. "BSP, LogP, and Oblivious Programs". In: *Euro-Par'98 Parallel Processing*. Ed. by David Pritchard and Jeff Reeve. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Sept. 1, 1998, pp. 865–874 (page 22).

[47] Johan Enmyren and Christoph W. Kessler. "SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems". In: *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*. HLPP '10. New York, NY, USA: ACM, 2010, pp. 5–14 (page 48).

[48] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. "Quaff: Efficient C++ Design for Parallel Skeletons". In: *Parallel Computing*. Algorithmic Skeletons 32 (7–8 Sept. 2006), pp. 604–615 (page 48).

[49] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. "A Report on the Sisal Language Project". In: *J. Parallel Distrib. Comput.* 10.4 (Dec. 1990), pp. 349–366 (page 63).

[50] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. "Implicitly Threaded Parallelism in Manticore". In: *J. Funct. Program.* 20 (5-6 Nov. 2010), pp. 537–576 (page 63).

[51] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. "Manticore: A Heterogeneous Parallel Language". In: *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*. DAMP '07. New York, NY, USA: ACM, 2007, pp. 37–44 (page 63).

[52] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Trans. Comput.* 21.9 (Sept. 1972), pp. 948–960 (page 4).

[53] Jean Fortin. "BSP-Why: A Tool for Deductive Verification of BSP Programs". 2013 (page 55).

[54] Jean Fortin and Frédéric Gava. “BSP-Why: A Tool for Deductive Verification of BSP Algorithms with Subgroup Synchronisation”. In: *International Journal of Parallel Programming* (Mar. 31, 2015), pp. 1–24 (page 21).

[55] Jean Fortin and Frédéric Gava. “Towards Mechanised Semantics of HPC: The BSP with Subgroup Synchronisation Case”. In: *Algorithms and Architectures for Parallel Processing*. International Conference on Algorithms and Architectures for Parallel Processing. Springer, Cham, Nov. 18, 2015, pp. 222–237 (page 85).

[56] Steven Fortune and James Wyllie. “Parallelism in Random Access Machines”. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC ’78. New York, NY, USA: ACM, 1978, pp. 114–118 (page 33).

[57] Ian Foster, Robert Olson, and Steven Tuecke. “Productive Parallel Programming: The PCN Approach”. In: *Scientific Programming* 1.1 (1992), pp. 51–66 (page 63).

[58] Matthew I. Frank, Anant Agarwal, and Mary K. Vernon. “LoPC: Modeling Contention in Parallel Algorithms”. In: *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’97. New York, NY, USA: ACM, 1997, pp. 276–287 (page 35).

[59] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Computer Society Pr, 2003 (page 61).

[60] Jacques Garrigue. “Relaxing the Value Restriction”. In: *Functional and Logic Programming*. Ed. by Yukiyoshi Kameyama and Peter J. Stuckey. Lecture Notes in Computer Science 2998. Springer Berlin Heidelberg, Apr. 7, 2004, pp. 196–213 (page 120).

[61] Frédéric Gava. “Formal Proofs of Functional Bsp Programs”. In: *Parallel Processing Letters* 13 (03 Sept. 1, 2003), pp. 365–376 (pages 15, 20, 167).

[62] Frédéric Gava. “Parallel I/O in Bulk-Synchronous Parallel ML”. In: *Computational Science - ICCS 2004*. International Conference on Computational Science. Springer, Berlin, Heidelberg, June 6, 2004, pp. 331–338 (page 170).

[63] Frédéric Gava and Frédéric Loulergue. “A Functional Language for Departmental Metacomputing”. In: *Parallel Processing Letters* 15 (03 Sept. 1, 2005), pp. 289–304 (page 80).

[64] David Gelernter and Nicholas Carriero. “Coordination Languages and Their Significance”. In: *Commun. ACM* 35.2 (Feb. 1992), pp. 97–107 (page 63).

[65] Alexandros V. Gerbessiotis. “Extending the BSP Model for Multi-Core and out-of-Core Computing: MBSP”. In: *Parallel Computing* 41 (Jan. 2015), pp. 90–102 (pages 23, 40).

[66] Louis Gesbert. “Développement Systématique et Sûreté d’exécution En Programmation Parallèle Structurée”. 2009 (page 118).

[67] Louis Gesbert, Frédéric Gava, Frédéric Loulergue, and Frédéric Dabrowski. “Bulk Synchronous Parallel ML with Exceptions”. In: *Future Generation Computer Systems* 26.3 (Mar. 2010), pp. 486–490 (pages 15, 121).

[68] Louis Gesbert, Zhenjiang Hu, Frédéric Loulergue, Kiminori Matsuzaki, and Julien Tesson. “Systematic Development of Correct Bulk Synchronous Parallel Programs”. In: *2010 International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). Dec. 2010, pp. 334–340 (pages 15, 20, 167).

[69] P. B. Gibbons. “A More Practical PRAM Model”. In: *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’89. New York, NY, USA: ACM, 1989, pp. 158–168 (page 34).

[70] Sergei Gorlatch. “Send-Receive Considered Harmful: Myths and Realities of Message Passing”. In: *ACM Trans. Program. Lang. Syst.* 26.1 (Jan. 2004), pp. 47–56 (pages 2, 21).

[71] Clemens Grelck and Sven-Bodo Scholz. “Classes and Objects as Basis for I/O in SAC”. In: *In Proceedings of the 7th International Workshop on Implementation of Functional Languages (IFL'95),* Båstad, Sweden, 1995, pp. 30–44 (page 63).

[72] Gaétan Hains. “Subset Synchronization in BSP Computing”. In: *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications* (July 1998), pp. 242–246 (page 35).

[73] Håkan Mattsson and Christoph Kessler. “Towards a Bulk-Synchronous Distributed Shared Memory Programming Environment for Grids”. In: *Applied Parallel Computing. State of the Art in Scientific Computing*. International Workshop on Applied Parallel Computing. Springer, Berlin, Heidelberg, June 20, 2004, pp. 519–526 (page 61).

[74] K. Hamidouche, F. M. Mendonca, J. Falcou, and D. Etiemble. “Parallel Biological Sequence Comparison on Heterogeneous High Performance Computing Platforms with BSP++”. In: *2011 23rd International Symposium on Computer Architecture and High Performance Computing*. 2011 23rd International Symposium on Computer Architecture and High Performance Computing. Oct. 2011, pp. 136–143 (page 57).

[75] Khaled Hamidouche, Alexandre Borghi, Pierre Esterie, Joel Falcou, and Sylvain Peyronnet. “Three High Performance Architectures in the Parallel APMC Boat”. In: *Proceedings of the 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology*. PDMC-HIBI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 20–27 (pages 7, 57).

[76] Khaled Hamidouche, Joel Falcou, and Daniel Etiemble. “A Framework for an Automatic Hybrid MPI+OpenMP Code Generation”. In: *Proceedings of the 19th High Performance Computing Symposia*. HPC '11. San Diego, CA, USA: Society for Computer Simulation International, 2011, pp. 48–55 (page 57).

[77] Khaled Hamidouche, Fernando Machado Mendonca, Joel Falcou, Alba Cristina Magalhaes Alves de Melo, and Daniel Etiemble. “Parallel Smith-Waterman Comparison on Multicore and Manycore Computing Platforms with BSP++”. In: *International Journal of Parallel Programming* 41.1 (Feb. 1, 2013), pp. 111–136 (page 57).

[78] Kevin Hammond. “Glasgow Parallel Haskell (GpH)”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, 2011, pp. 768–779 (page 63).

[79] Kevin Hammond. “Parallel Functional Programming: An Introduction”. In: *International Symposium on Parallel Symbolic Computation*. 1994, p. 46 (page 62).

[80] Kevin Hammond. “The Dynamic Properties of Hume: A Functionally-Based Concurrent Language with Bounded Time and Space Behaviour”. In: *Implementation of Functional Languages*. Ed. by Markus Mohnen and Pieter Koopman. Lecture Notes in Computer Science 2011. Springer Berlin Heidelberg, Sept. 4, 2000, pp. 122–139 (page 64).

[81] Kevin Hammond and Greg Michelson, eds. *Research Directions in Parallel Functional Programming*. London, UK, UK: Springer-Verlag, 2000 (pages 3, 62).

[82] Bruce Hendrickson. “Computational Science: Emerging Opportunities and Challenges”. In: *Journal of Physics: Conference Series* 180.1 (2009), p. 012013 (pages 1, 22).

[83] D. S. Henty. “Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling”. In: *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. SC '00. Washington, DC, USA: IEEE Computer Society, 2000 (page 7).

[84] Todd Heywood and Sanjay Ranka. “A Practical Hierarchical Model of Parallel Computation I. The Model”. In: *Journal of Parallel and Distributed Computing* 16.3 (Nov. 1, 1992), pp. 212–232 (page 34).

[85] J. M. D. Hill and D. B. Skillicorn. “Practical Barrier Synchronisation”. In: *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, 1998. PDP ’98.* Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, 1998. PDP ’98. Jan. 1998, pp. 438–444 (page 14).

[86] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. “BSPlib: The BSP Programming Library”. In: *Parallel Computing* 24.14 (Dec. 1998), pp. 1947–1980 (pages 16, 52).

[87] Konrad Hinsen, Hans Petter Langtangen, Ola Skavhaug, and \AAsmund Ødeg\a ard. “Using BSP and Python to Simplify Parallel Programming”. In: *Future Gener. Comput. Syst.* 22.1 (Jan. 2006), pp. 123–157 (page 58).

[88] C.A.R. Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. “The Verified Software Initiative: A Manifesto”. In: *ACM Comput. Surv.* 41.4 (Oct. 2009), 22:1–22:8 (page 3).

[89] Jan Hoffman, Das Ankush, and Weng Shu-Chun. “Towards Automatic Resource Bound Analysis for OCaml”. In: *In Proceedings of the 44th Symposium on Principles of Programming Languages* (2017) (page 170).

[90] Qiming Hou, Kun Zhou, and Baining Guo. “BSGP: Bulk-Synchronous GPU Programming”. In: *ACM SIGGRAPH 2008 Papers*. SIGGRAPH ’08. New York, NY, USA: ACM, 2008, 19:1–19:12 (pages 14, 22, 58).

[91] Gérard Huet. “Résolution d’équations Dans Des Langages d’ordre 1,2,..., ”. Université Paris VII, Sept. 1976 (page 123).

[92] Vladimir Janjic, Christopher Mark Brown, Max Neunhoeffer, Kevin Hammond, Stephen Alexander Linton, and H.-W. Loidl. “Space Exploration Using Parallel Orbits: A Study in Parallel Symbolic Computing”. In: *Accelerating Computational Science and Engineering (CSE)* 25 (2014), pp. 225–232 (page 43).

[93] Yan Jiang, Weiqin Tong, and Wentao Zhao. “Resource Load Balancing Based on Multi-Agent in ServiceBSP Model”. In: *Proceedings of the 7th International Conference on Computational Science, Part III: ICCS 2007*. ICCS ’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 42–49 (page 22).

[94] Daniel Johansson, Mattias Eriksson, and Christoph W. Kessler. “Bulk-Synchronous Parallel Computing on the CELL Processor”. In: *PARS*. 2007 (page 61).

[95] Ben H. H. Juurlink and Harry A. G. Wijshoff. “The E-BSP Model: Incorporating General Locality and Unbalanced Communication into the BSP Model”. In: *Euro-Par’96 Parallel Processing*. Lecture Notes in Computer Science 1124. Springer Berlin Heidelberg, Aug. 26, 1996, pp. 339–347 (page 36).

[96] Y. Karasawa and H. Iwasaki. “A Parallel Skeleton Library for Multi-Core Clusters”. In: *2009 International Conference on Parallel Processing*. 2009 International Conference on Parallel Processing. Sept. 2009, pp. 84–91 (page 41).

[97] Paul H.J. Kelly. “Functional Programming for Loosely-Coupled Multiprocessors”. In: *Functional Programming for Loosely-Coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989, p. 44 (page 63).

[98] Christoph W. Kessler. “Managing Distributed Shared Arrays in a Bulk-Synchronous Parallel Programming Environment”. In: *Concurrency and Computation: Practice and Experience* 16 (2-3 Feb. 1, 2004), pp. 133–153 (page 60).

[99] Christoph W. Kessler. “NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model”. In: *The Journal of Supercomputing* 17.3 (Nov. 1, 2000), pp. 245–262 (pages 60, 61).

[100] Thilo Kielmann and Sergei Gorlatch. “Bandwidth-Latency Models (BSP, LogP)”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, 2011, pp. 107–112 (page 13).

[101] Peter Krusche and Alexander Tiskin. “New Algorithms for Efficient Parallel String Comparison”. In: *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’10. New York, NY, USA: ACM, 2010, pp. 209–216 (page 22).

[102] Alexey Kukanov. “The Foundations for Scalable Multi-Core Software in Intel Threading Building Blocks”. In: *Intel Technology Journal* 11.4 (2007), pp. 309–322 (page 5).

[103] Edward A. Lee. “The Problem with Threads”. In: *Computer* 39.5 (May 2006), pp. 33–42 (pages 2, 7, 21).

[104] Xavier Leroy and Hervé Grall. “Coinductive Big-Step Operational Semantics”. In: *Information and Computation*. Special issue on Structural Operational Semantics (SOS) 207.2 (Feb. 2009), pp. 284–304 (pages 84, 93).

[105] Chong Li and Gaétan Hains. “SGL: Towards a Bridging Model for Heterogeneous Hierarchical Platforms”. In: *International Journal of Parallel Programming* 7.2 (Apr. 2012), pp. 139–151 (page 38).

[106] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. “On the Versatility of Parallel Sorting by Regular Sampling”. In: *Parallel Computing* 19.10 (Oct. 1, 1993), pp. 1079–1103 (page 19).

[107] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á J. Rebón, and P. W. Trinder. “Comparing Parallel Functional Languages: Programming and Performance”. In: *Higher Order Symbolic Computing* 16.3 (Sept. 2003), pp. 203–251 (page 63).

[108] Rita Loogen, Yolanda Ortega-mallén, and Ricardo Peña-marí. “Parallel Functional Programming in Eden”. In: *J. Funct. Program.* 15.3 (May 2005), pp. 431–475 (page 63).

[109] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. “Models of Parallel Computation: A Survey and Synthesis”. In: *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*. Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences. Vol. 2. Jan. 1995, 61–70 vol.2 (page 13).

[110] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. “Pregel: A System for Large-Scale Graph Processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. New York, NY, USA: ACM, 2010, pp. 135–146 (pages 21, 59, 60).

[111] Louis Mandel and Luc Maranget. “Programming in JoCaml (Tool Demonstration)”. In: *17th European Symposium on Programming ({ESOP 2008})*. 2008, pp. 108–111 (page 64).

[112] Charles Martel and Arvind Raghunathan. “Asynchronous PRAMs with Memory Latency”. In: *J. Parallel Distrib. Comput.* 23.1 (Oct. 1994), pp. 10–26 (page 34).

[113] Jeremy M. R. Martin and Alexander Tiskin. “BSP Modelling of Two Tiered Architectures”. In: *Proceedings of {W}o{TUG}-22: Architectures, Languages and Techniques for Concurrent Systems* (Mar. 1999), pp. 47–56 (page 38).

[114] Jeremy M. R. Martin and Alexander Tiskin. “Dynamic BSP: Towards a Flexible Approach to Parallel Computing over the Grid”. In: *Communicating Process Architectures 2004* (Sept. 2004), pp. 219–226 (page 39).

[115] Erik Meijer, Maarten Fokkinga, and Ross Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: Springer-Verlag, 1991, pp. 124–144 (page 168).

[116] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375 (pages 79, 113).

[117] Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. New York, NY, USA: Cambridge University Press, 1999 (page 6).

[118] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. “Scalable Parallel Programming with CUDA”. In: *Queue* 6.2 (Mar. 2008), pp. 40–53 (page 5).

[119] Martin Odersky, Martin Sulzmann, and Martin Wehr. “Type Inference with Constrained Types”. In: *Theor. Pract. Object Syst.* 5.1 (Jan. 1999), pp. 35–55 (pages 101, 122).

[120] C. Ouwehand, P. Hijma, and W. J. Fokkink. “GPU Programming in Functional Languages A Comparison of Haskell GPU Embedded Domain Specific Languages”. In: (2013) (page 7).

[121] Matthew Felice Pace. “BSP vs MapReduce”. In: *Procedia Computer Science* 9 (2012), pp. 246–255. arXiv: 1203.2081 (page 44).

[122] Frédéric Peschanski. “Parallel Computing with the Pi-Calculus”. In: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. DAMP ’11. New York, NY, USA: ACM, 2011, pp. 45–54 (page 6).

[123] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2002 (page 122).

[124] François Pottier and Didier Rémy. “The Essence of ML Type Inference”. In: *Advanced Topics in Types and Programming Languages*. MIT Press, May 20, 2003, Chapter 10 (pages 101, 122, 129).

[125] R. d R. Righi, L. L. Pilla, A. Carissimi, P. Navaux, and H. U. Heiss. “MigBSP: A Novel Migration Model for Bulk-Synchronous Parallel Processes Rescheduling”. In: *2009 11th IEEE International Conference on High Performance Computing and Communications*. 2009 11th IEEE International Conference on High Performance Computing and Communications. June 2009, pp. 585–590 (pages 22, 56).

[126] Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, and Frantisek Mikulu. “WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core”. In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Ed. by Björn Lisper. Vol. 15. OpenAccess Series in Informatics (OASIcs). The printed version of the WCET’10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 90–100 (page 10).

[127] Andreas Rossberg. “Typed Open Programming : A Higher-Order, Typed Approach to Dynamic Modularity and Distribution”. Universitat des Saarlandes, 2007 (page 64).

[128] Fatima K. Abu Salem and Laurence T. Yang. “Parallel Methods for Absolute Irreducibility Testing”. In: *The Journal of Supercomputing* 46.3 (Dec. 1, 2008), pp. 181–212 (page 22).

[129] A. Savadi and H. Deldari. “A Bridging Model for Branch-and-Bound Algorithms on Multi-Core Architectures”. In: *2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming*. 2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming. Dec. 2012, pp. 235–241 (page 40).

[130] Abdorreza Savadi and Hossein Deldari. “Measurement of the Latency Parameters of the Multi-BSP Model: A Multicore Benchmarking Approach”. In: *The Journal of Supercomputing* 67.2 (Feb. 1, 2014), pp. 565–584 (page 29).

[131] Abdorreza Savadi, Morteza Moradi, and Hossein Deldari. “Multi-DaC Programming Model: A Variant of Multi-BSP Model for Divide-and-Conquer Algorithms”. In: *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*. DAMP ’12. New York, NY, USA: ACM, 2012, pp. 41–46 (page 40).

[132] Norman Scaife, Greg Michaelson, and Susumu Horiguchi. “Empirical Parallel Performance Prediction from Semantics-Based Profiling”. In: *Computational Science – ICCS 2005*. Ed. by Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra. Lecture Notes in Computer Science 3515. Springer Berlin Heidelberg, May 22, 2005, pp. 781–789 (page 64).

[133] Hermes Senger, Veronica Gil-Costa, Luciana Arantes, Cesar A. C. Marcondes, Juan Mauricio Marin Caihuan, Liria M. Sato, Da Silva, and Fabrício A. B. “BSP Cost and Scalability Analysis for MapReduce Operations”. In: *Concurrency Computation* (2015) (page 44).

[134] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. “Ott: Effective Tool Support for the Working Semanticist”. In: *Journal of Functional Programming* (2010), pp. 71–122 (page 81).

[135] Hanmao Shi and Jonathan Schaeffer. “Parallel Sorting by Regular Sampling”. In: *J. Parallel Distrib. Comput.* 14.4 (Apr. 1992), pp. 361–372 (pages 19, 58).

[136] Vincent Simonet. “Type Inference with Structural Subtyping: A Faithful Formalization of an Efficient Constraint Solver”. In: *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS’03)*. Vol. volume 2895 of Lecture Notes in Computer Science. Beijing, China: Springer-Verlag, Nov. 2003, pp. 283–302 (page 97).

[137] Vincent Simonet and Inria Rocquencourt. “Flow Caml in a Nutshell”. In: *Proceedings of the First APPSEM-II Workshop*. 2003, pp. 152–165 (page 122).

[138] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. “MultiMLton: A Multicore-Aware Runtime for Standard ML”. In: *Journal of Functional Programming* 24 (06 2014), pp. 613–674 (page 64).

[139] D. B. Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. “Optimising Data-Parallel Programs Using the BSP Cost Model”. In: *Euro-Par’98 Parallel Processing*. European Conference on Parallel Processing. Springer, Berlin, Heidelberg, Sept. 1, 1998, pp. 698–703 (page 3).

[140] D. B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. “Questions and Answers about BSP”. In: *Scientific Programming* 6.3 (1997), pp. 249–274 (pages 13, 21).

[141] Gert Smolka. “The Oz Programming Model”. In: *Mathematical Methods in Program Development*. Ed. by Manfred Broy and Birgit Schieder. NATO ASI Series 158. Springer Berlin Heidelberg, 1997, pp. 409–432 (page 64).

[142] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference*. 2nd. (Revised). Cambridge, MA, USA: MIT Press, 1998 (pages 5, 14, 15, 49).

[143] Joar Sohl. *A Scalable Run-Time System for NestStep on Cluster Supercomputers*. Master Thesis. Linköpings universitet, 2006 (page 61).

[144] Antony Joseph Sunu. *Evaluating Threading Building Blocks Pipelines*. 2007 (page 5).

[145] Jean-Pierre Talpin and Pierre Jouvelot. "The Type and Effect Discipline". In: *Information and Compution* 111.2 (June 1994), pp. 245–296 (page 67).

[146] Alexander Tiskin. "The Design and Analysis of Bulk-Synchronous Parallel Algorithms". Nov. 1998 (pages 19, 20).

[147] Pilar de la Torre and Clyde P. Kruskal. "Submachine Locality in the Bulk Synchronous Setting". In: *Euro-Par'96 Parallel Processing*. Lecture Notes in Computer Science 1124. Springer Berlin Heidelberg, Aug. 26, 1996, pp. 352–358 (page 36).

[148] Theo Ungerer et al. "Parallelizing Industrial Hard Real-Time Applications for the parMERASA Multicore". In: *ACM Trans. Embed. Comput. Syst.* 15.3 (May 2016), 53:1–53:27 (page 10).

[149] Leslie G. Valiant. "A Bridging Model for Multi-Core Computing". In: *J. Comput. Syst. Sci.* 77.1 (Jan. 2011), pp. 154–166 (pages 23, 24, 26–29).

[150] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111 (pages 11, 13).

[151] Vasil P. Vasilev. "Bspgrid: Variable Resources Parallel Computation and Multiprogrammed Parallelism". In: *Parallel Processing Letters* 13 (03 Sept. 1, 2003), pp. 329–340 (page 39).

[152] Jue Wang, ChangJun Hu, JiLin Zhang, and JianJiang Li. "OpenMP Compiler for Distributed Memory Architectures". In: *Science China Information Sciences* 53.5 (May 1, 2010), pp. 932–944 (page 5).

[153] Tom White. *Hadoop: The Definitive Guide*. 1st. O'Reilly Media, Inc., 2009 (page 44).

[154] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". In: *Inf. Comput.* 115.1 (Nov. 1994), pp. 38–94 (page 115).

[155] Andrew K. Wright. "Typing References by Effect Inference". In: *ESOP '92*. Ed. by Bernd Krieg-Brückner. Lecture Notes in Computer Science 582. Springer Berlin Heidelberg, Feb. 26, 1992, pp. 473–491 (page 120).

[156] A.n. Yzelman and Rob H. Bisseling. "An Object-Oriented Bulk Synchronous Parallel Library for Multicore Programming". In: *Concurrency and Computation: Practice and Experience* 24.5 (Apr. 10, 2012), pp. 533–553 (page 14).

[157] Lukasz Ziarek, Siddharth Tiwary, and Suresh Jagannathan. "Isolating Determinism in Multi-Threaded Programs". In: *Runtime Verification*. International Conference on Runtime Verification. Springer, Berlin, Heidelberg, Sept. 27, 2011, pp. 63–77 (page 64).

Index

A
Accessibility 102
Algorithmic skeletons 40
Annotated type 97

B
BSML language 15
BSP model 13

C
Communicable predicate 86
Component 25

D
Definability 103
Distributed memory 5

F
Free type variable 99

H
Hybrid architectures 6

L
Latent Effect 97
Level 65, 79
Locality 98
Lookup predicate 86

M
MPI 49
MULTI-BSP model 24
Multi-function 68

P
Parallel architectures 3
Parallel vector 15
Propagation 104

S
Select predicate 90
Serialisation 104
Shared memory 4
Stage 25
Superstep 14

T
Tree structure 70

W
Weakening 106

Table of notations

Greek Symbols

α, β Type variable . . . page 97

Γ Environment . . . page 99

κ Kind . . . page 123

Λ Evaluation environment . . . page 107

λ, π Locality variable . . . page 97

$\mathcal{F}$ Free type variable . . . page 99

$\mathcal{V}_\kappa$ Set of type variables . . . page 125

σ Type scheme . . . page 100

τ Type . . . page 97

τ_π Annotated type . . . page 97

ε Latent effect . . . page 97

φ, ψ Substitution . . . page 99

Roman Symbols

α, β Range of type variables . . . page 125

`b` BSP level . . . page 98

`c` Communicable level . . . page 98

`l` Local level . . . page 98

`m` Multi level . . . page 98

`s` Sequential level . . . page 98

$\overline{\alpha}, \overline{\beta}$ Finite range of type variables . . . page 125

$\vec{\alpha}$ Vector of distinct type variables . . . page 125

$e, e_1, ..., e_n$ Expression . . . page 107

List of URL

1 MPI: http://www.mpi-forum.org/

2 TBB: http://threadingbuildingblocks.org

3 CUDA: http://developer.nvidia.com/category/zone/cuda-zone

4 top 500: https://www.top500.org/ (november 2016)

5 COQ: https://coq.inria.fr/

6 BSML: http://traclifo.univ-orleans.fr/BSML/

7 OCAML: http://caml.org

8 BSPLIB: http://www.bsp-worldwide.org/implmnts/oxtool/

9 LIFO: http://www.univ-orleans.fr/lifo/

10 Skeleton wiki: https://en.wikipedia.org/wiki/Algorithmic_skeleton#Frameworks_comparison

11 SKLML: hsklml.inria.fr/

12 A SKLML example: http://sklml.inria.fr/doc/htm/UserManual.htm

13 A FASTFLOW examples: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:tutorial

14 A MUESLI examples: http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformaticanetworking/spm/lessonmuesli1.pdf

15 A SKEPU examples: http://www.ida.liu.se/labs/pelab/skepu/

16 A MulticoreBSP example: http://albert-jan.yzelman.net/education/multibsp-uu14.pdf

17 GIRAPH: http://giraph.apache.org/

18 A NESL example: http://www.cs.cmu.edu/~scandal/nesl/algorithms.html

19 Learn OCAML: https://ocaml.org/learn/

20 OCAML doc: http://caml.inria.fr/pub/docs/manual-ocaml/

21 OTT: http://www.cl.cam.ac.uk/~pes20/ott/

Appendix

A.1 Semantics

A.1.1 Terminating semantics is deterministic

Lemma A.2 (*Evaluation is deterministic*)

Let e be a program, let $\mathcal{M}$ be an environment, $\mathcal{L}$ an evaluation locality, p a position and v_1 and v_2 be values. If $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_1$ and $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_2$ then $v_1 = v_2$.

Proof: By induction on the derivation of $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_1$ and case analysis on $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_2$. For each rule concluding the derivation tree, we have:

- Case VALUES (VALUES). For all locality of evaluation and for all position, the only applicable evaluation rule is VALUES, and thus, the two evaluation judgement are identical. Hence, the result.
- Case OP_EVAL (**op**). As previously.
- Case VAR (x). As previously.
- Case CLOSURE (the expression e is of the form $\textbf{fun}\ x \to e\ )$. As previously.
- Case REC_CLOSURE ($\textbf{rec}\ f\ x \to e\ )$. As previously.
- Case FUN_APP. The expression e is of the form $(e_1\ e_2)$. By definition of the FUN_APP rule, e_1 is evaluated into $\overline{(\textbf{fun}\ x \to e'\ )[\mathcal{E}]}$. By induction hypothesis, $\forall v_1', \mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} v_1'$. Thus, we have that $v_1' = \overline{(\textbf{fun}\ x \to e'\ )[\mathcal{E}]}$, as it is the only valid case. The derivation rule is the following:

$$\frac{\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{fun}\ x \to e'\ )[\mathcal{E}']} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v' \qquad \mathcal{M} \oplus_p \mathcal{E}' \oplus_p \{x \mapsto v'\} \vdash e' \Downarrow_p^{\mathcal{L}} v_1'}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_1}$$

  Using the FUN_APP rule, the derivation of $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_2$ is the following:

$$\frac{\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{fun}\ x \to e''\ )[\mathcal{E}'']} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v'' \qquad \mathcal{M} \oplus_p \mathcal{E}'' \oplus_p \{x \mapsto v''\} \vdash e'' \Downarrow_p^{\mathcal{L}} v_2'}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_2}$$

  By induction hypothesis, $v_1' \equiv v_2'$. Then, as a conclusion of the FUN_APP rule, $v_1 \equiv v_2$.
- Case REC_FUN_APP. As previously, with $e_1 \equiv \overline{(\textbf{rec}\ f\ x \to e'\ )[\mathcal{E}]}$.
- Case OP_APP. As previously, with $e_1 \equiv \textbf{op}$.
- Case LET_IN. With $e \equiv (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)$, as there is only one applicable rule, by induction hypothesis, all the premises are the valid. Hence the result.

- Case REPLICATE. As previously.
- Case DOWN. As previously, using the VAR rule.
- Case MKPAR. As previously, by induction $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} e'$.
- Case APPLY. As previously.
- Case PROJ. As previously.
- Case PUT. As previously.
- Case MULTI_NODE. Similar to the FUN_APP rule where the expression e is of the form $(e_1\ e_2)$ and, by induction hypothesis, e_1 is of the form $\overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']}$, as it is the only applicable rule at locality $\mathtt{l}$. As the predicates $isNode(p)$ and $isLeaf(p)$ are mutually exclusive, $isNode(p)$ is valid within the application context.
- Case MULTI_LEAF. Similar to the MULTI_NODE rule but where the predicate $isLeaf(p)$ is valid.
- Case MULTI_DEF. Similar to the CLOSURE rule where the expression e is of the form $(\textbf{multi}\ f\ x \to e_1' \dagger e_2')$ at locality $\mathtt{m}$.
- Case MULTI_CALL. Similar to the MULTI_NODE rule, where the locality of evaluation is $\mathtt{m}$. ■

A.1.2 Diverging semantics properties

Lemma A.3 (*Evaluation and divergence exclusivity*)

Let e be a program, let $\mathcal{M}$ be an environment, $\mathcal{L}$ an evaluation locality, p a position and v a value. If $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$ and $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty$ then there is a contradiction.

Proof: By induction on the derivation of $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$ and case analysis on $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty$, using Lemma 4.3. We have:

- Case VALUES (VALUES). For all locality of evaluation and and for all position, there is no coinductive rule for constants. Hence the contradiction.
- Case OP_EVAL (**op**). As previously.
- Case VAR (x). As previously.
- Case CLOSURE, the expression e is of the form $(\textbf{fun}\ x \to e\ )$. As previously.
- Case REC_CLOSURE $(\textbf{rec}\ f\ x \to e\ )$. As previously.
- Case FUN_APP. The expression e is of the form $(e_1\ e_2)$. By definition of the FUN_APP rule, $\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{fun}\ x \to e'\ )[\mathcal{E}]}$, $\mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v_2$ and $\mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x \mapsto v_2\} \vdash e' \Downarrow_p^{\mathcal{L}} v'$. By induction hypothesis on the premises, we have:
  - $\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \infty$ (rule APP-L). Using the previous inductive hypothesis, we get a contradiction.
  - $\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{fun}\ x \to e'\ )[\mathcal{E}]}$, as it is the only valid case, and $\mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} \infty$ (rule FUN_APP-R). Using the previous inductive hypothesis, we get a contradiction.
  - $\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{fun}\ x \to e'\ )[\mathcal{E}]}$, $\mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v'$ (rule FUN_APP-E), we have:

$$\frac{\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{fun}\ x \to e'\ )[\mathcal{E}]} \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v' \qquad \mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x' \mapsto v'\} \vdash e' \Downarrow_p^{\mathcal{L}} \infty}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow_p^{\mathcal{L}} \infty}$$

  By induction, we obtain:

  - $\mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x' \mapsto v'\} \vdash e' \Downarrow_p^{\mathcal{L}} v$
  - if $\mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x' \mapsto v'\} \vdash e' \Downarrow_p^{\mathcal{L}} \infty$, then we have a contradiction.
- Case REC_FUN_APP. As previously, with $e_1 \equiv \overline{(\textbf{rec } f \ x \to e' \ )[\mathcal{E}]}$.
- Case OP_APP. As previously, with $e_1 \equiv \textbf{op}$.
- Case LET_IN. When $e \equiv \textbf{let } x = e_1 \textbf{ in } e_2$, the case is direct as there is a single applicable rule.
- Case REPLICATE. By induction hypothesis, if at least one of the i ($i \in p$) evaluations leads to a diverging evaluation, then we have a contradiction.
- Case DOWN. There is no rule for DOWN, similarly to VALUES. Hence the contradiction.
- Case MKPAR. As previously, as the premises are mutually exclusives.
- Case APPLY. As previously.
- Case PROJ. As previously.
- Case PUT. As previously.
- Case MULTI_NODE. Similarly to the FUN_APP rule, if $\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{multi } f \ x \to e_1' \dagger e_2')[\mathcal{M}']}$, $\mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v'$ and $\mathcal{M}' \oplus_p \{x \mapsto v'\} \oplus_p \{f \mapsto \overline{(\textbf{multi } f \ x \to e_1' \dagger e_2')[\mathcal{M}']}\} \vdash e_1' \Downarrow_p^{\mathfrak{b}} v$, by induction hypothesis, it excludes diverging rules.
- Case MULTI_LEAF. As previously.
- Case MULTI_CALL. As previously.

■

A.2 Typing

A.2.1 Type soundness

Lemma A.4 (*Validity of localities within the evaluation context*)

If $\Lambda, \Gamma \vdash e : \tau_\pi/\varepsilon \ [c]$ *then* $[c]$ *holds.*

Proof: By induction on e.

- Case VAR. The result is explicit in the derivation rule: $\pi' \lhd \Lambda$.
- Case CST. As previously.
- Case OP. As previously.
- Case PAIR. By induction hypothesis on e_1 and e_2, hence the result.
- Case IF THEN ELSE. By induction hypothesis on e_1, e_2 and e_3, hence the result.
- Case LET IN. By induction hypothesis on e_1 and e_2, hence the result.
- Case LET REC. As previously.
- Case DEFFUN. The expression e is of the form $\textbf{fun } x \to e'$ and is typed using the rule:

$$\text{DEFFUN} \quad \frac{\begin{array}{l}\Lambda', \Gamma; x : \tau_{\pi_1}^1/\varepsilon_1 \vdash e' : \tau_{\pi_2}^2/\varepsilon_2 \ [c_1] \\ c_2 = [\varepsilon_2 \blacktriangleleft \Lambda, \pi_1 \lhd \varepsilon_2, c_1]\end{array}}{\Lambda, \Gamma \vdash \textbf{fun } x \ \to \ e' : (\tau_{\pi_1}^1 \xrightarrow{\varepsilon_2} \tau_{\pi_2}^2)_\Lambda/\varepsilon_2 \ [c_2]}$$

By induction hypothesis on e', π_2 is valid in Λ'. Thus, $e' : (\tau_{\pi_1}^1 \xrightarrow{\varepsilon_2} \tau_{\pi_2}^2)_\Lambda/\varepsilon_2$ with $\varepsilon_2 \blacktriangleleft \Lambda$ ensuring the definability and $\pi_1 \lhd \varepsilon_2$ ensuring the accessibility of x.

- Case APP. The expression e is of the form $f\ e'$. By induction hypothesis, we have $f : (\tau^1_{\pi_1} \xrightarrow{\varepsilon_1} \tau^2_{\pi_2})_{\Lambda'}/\varepsilon_2$ and $e' : \tau^1_{\pi_3}/\varepsilon_3$, which are both valid in Λ. Thus, we have $\pi_3 \lhd \pi_2$ to ensure the validity of the argument, $\varepsilon_1 \lhd \Lambda$ ensures the validity of evaluation of f within Λ and $\pi_2 \lhd \Lambda$ forces the the produced value to be valid locally. The constraint $\varepsilon_1 \blacktriangleleft \Lambda$ is used to ensure the validity of references.
- Case VECTOR. The expression e is of the form $<< e' >>$. By induction hypothesis on e', π is valid in $\mathtt{c}$. Then, by definition, $\varepsilon \lhd \mathtt{c}$ and $\pi \lhd \mathtt{c}$ ensure the validity of the term.
- Case COPY. By definition of the **Seria** predicate, $\tau_\pi = \mathbf{Seria}_{\mathtt{c}}(\tau_{\mathtt{b}})$ thus $\pi \lhd \mathtt{c}$.
- Case OPEN. The rule is direct.
- Case OP'. By definition, $\Gamma(\mathbf{op}')$ leads to a valid type. Similarly to the APP rule, $\pi_3 \lhd \mathtt{b}$ ensures the validity of the argument and $\varepsilon \lhd \mathtt{b}$ ensures the validity of evaluation of $\mathbf{op}'$ within Λ. Hence the result.
- Case REPLICATE. By induction hypothesis on f and by ensuring that the evaluation and the value produced is valid within $\mathtt{b} : \varepsilon_1 \lhd \mathtt{c}$ and $\pi_2 \lhd \mathtt{c}$.
- Case DOWN. Similarly to the COPY rule.
- Case APPLY. By induction hypothesis on e_1 and e_2. Then, similarly to the APP rule, by ensuring that $\pi_3 \lhd \pi_1$, $\varepsilon \lhd \mathtt{c}$ and $\pi_2 \lhd \mathtt{c}$.
- Case MULTI_FUN. The expression e is of the form $(\tau^1_{\mathtt{m}} \xrightarrow{\mathtt{m}} \tau^3_{\mathtt{m}})_{\mathtt{m}}/\mathtt{m}$. By induction hypothesis, $e_1 : \tau^3_{\pi_3}/\varepsilon_1$ and $e_2 : \tau^3_{\pi_4}/\varepsilon_2$. By definition of the **Seria** predicate, π_3 is valid in $\mathtt{m}$ and π_4 is valid $\mathtt{c}$.

■

Lemma A.18 (*Typing is stable under substitution*)

Under the following hypothesis which defines the substitution:

- $\Lambda, \Gamma; x : \forall \overrightarrow{\alpha}\ \overrightarrow{\lambda}[c_1].\tau^1_{\pi_1} \vdash e : \tau^2_{\pi_2}/\varepsilon\ \ [c_2; \exists \overrightarrow{\alpha}\ \overrightarrow{\lambda}.c_1]$
- $\Lambda, \Gamma \vdash v : \tau^1_{\pi_1}/\varepsilon\ \ [c_1; c_2]$
- $\overrightarrow{\alpha}\ \overrightarrow{\lambda} \cap \mathcal{F}(c_2) \cap \mathcal{F}(\Gamma) = \emptyset$

We have: $\Lambda, \Gamma \vdash \{v/x\}e : \tau^2_{\pi_2}/\varepsilon\ \ [\exists \overrightarrow{\alpha}\ \overrightarrow{\lambda}.c_1; c_2]$ *where* $\{v/x\}e$ *stands for the substitution of* x *by* v *in expression* e.

Proof: We define $\Gamma_x = \Gamma; x : \forall \overrightarrow{\alpha}\ \overrightarrow{\lambda}[c_1].\tau^1_{\pi_1}$ and $c = c_2; \exists \overrightarrow{\alpha}\ \overrightarrow{\lambda}.c_1$ for conciseness reasons. The first hypothesis is $\Lambda, \Gamma_x \vdash e : \tau^2_{\pi_2}/\varepsilon\ \ [c]$. The proof is made by induction on the derivation of this typing judgement.

- Case VAR: we define $e = x'$, thus we have:

$$\text{VAR}\quad \frac{\begin{array}{l}\Gamma_x(x') = \forall \overrightarrow{\alpha'}\ \overrightarrow{\lambda'}[c_0].\tau_{\pi'}/\varepsilon\ \ [c_0] \\ c = [\pi' \lhd \Lambda, c_0]\end{array}}{\Lambda, \Gamma_x \vdash x' : \tau^2_{\pi_2}/\varepsilon\ \ [c]}$$

  - If $x' \neq x$, then $\{v/x\}e = e$ and $\Gamma_x(x') = \Gamma(x)$. Hence the result.
  - If $x' = x$, then $\{v/x\}e = v$ and, by hypothesis: $\Lambda, \Gamma \vdash \{v/x\}e : \tau^1_{\pi_1}/\varepsilon\ \ [c_1; c_2]$. Then, we have $\Gamma_x = \forall \overrightarrow{\alpha}\ \overrightarrow{\lambda}[c_1].\tau^1_{\pi_1} = \forall \overrightarrow{\alpha'}\ \overrightarrow{\lambda'}[c_1].\tau^2_{\pi_2}$. Thus, $\tau^1_{\pi_1}$ and $\tau^2_{\pi_2}$ are equals modulo renaming (alpha-conversion).
- Case OP. Similar to VAR.

- Case LET-IN: we define $e = \textbf{let}\ \ x' = e_1\ \ \textbf{in}\ \ e_2$. The derivation rule is thus the following:

$$\text{LET IN}\quad \frac{\begin{array}{l}\Lambda, \Gamma_x \vdash e_1 : \tau^1_{\pi_1}/\varepsilon_1\ \ [c'_1]\\ \Lambda, \Gamma_x; x' : \forall\overrightarrow{\alpha'}\overrightarrow{\lambda'}[c'_1].\textbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1) \vdash e_2 : \tau^2_{\pi_2}/\varepsilon_2\ \ [c_2]\\ c_3 \equiv [\Psi = \textbf{Propgt}(\varepsilon_1, \varepsilon_2), c_1, c_2]\end{array}}{\Lambda, \Gamma_x \vdash \textbf{let}\ \ x' = e_1\ \ \textbf{in}\ \ e_2 : \tau^2_{\pi_2}/\Psi\ \ [c_3]}$$

  - If $x = x'$, then $\Lambda, \Gamma_x; x' : \forall\overrightarrow{\alpha'}\overrightarrow{\lambda'}[c'_1].\textbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1) = \Lambda, \Gamma; x' : \forall\overrightarrow{\alpha'}\overrightarrow{\lambda'}[c'_1].\textbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1)$. Thus, $\Lambda, \Gamma; x' : \forall\overrightarrow{\alpha'}\overrightarrow{\lambda'}[c'_1].\textbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1) \vdash e_2 : \tau^2_{\pi_2}/\varepsilon_2\ \ [c_2]$. By using the LET-IN rule, we get $\Lambda, \Gamma \vdash \textbf{let}\ \ x' = \{v/x\}e_1\ \ \textbf{in}\ \ e_2 : \tau^2_{\pi_2}/\Psi\ \ [c_3]$ and, by definition of the substitution, $\Lambda, \Gamma \vdash \{v/x\}\textbf{let}\ \ x' = e_1\ \ \textbf{in}\ \ e_2 : \tau^2_{\pi_2}/\Psi\ \ [c_3]$.
  - If $x \neq x'$, then $\Gamma_x; x' : \forall\overrightarrow{\alpha'}\overrightarrow{\lambda'}[c'_1].\textbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1) = \Gamma; x' : \forall\overrightarrow{\alpha'}\overrightarrow{\lambda'}[c'_1].\textbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1); x : \forall\overrightarrow{\alpha'}\overrightarrow{\lambda'}[c'_1].\textbf{Weak}(\tau^3_{\pi_3}, \varepsilon_3)$. By using the LET-IN rule, we get $\Lambda, \Gamma \vdash \textbf{let}\ \ x' = \{v/x\}e_1\ \ \textbf{in}\ \ \{v/x\}e_2 : \tau^2_{\pi_2}/\Psi\ \ [c_3]$ and thus $\Lambda, \Gamma \vdash \{v/x\}(\textbf{let}\ \ x' = e_1\ \ \textbf{in}\ \ e_2) : \tau^2_{\pi_2}/\Psi\ \ [c_3]$.
- Case LET REC. Similar to the LET IN case.
- Case DEFFUN: we define $e = \textbf{fun}\ \ x' \to e'$, thus, we have $\tau^2_{\pi_2} = (\tau'^1_{\pi'_1} \xrightarrow{\varepsilon'} \tau'^2_{\pi'_2})_\Lambda$. The derivation rule is:

$$\text{DEFFUN}\quad \frac{\begin{array}{l}\Lambda', \Gamma_x; x' : \tau'^1_{\pi'_1}/\varepsilon_1 \vdash e' : \tau'^2_{\pi'_2}/\varepsilon'\ \ [c_1]\\ c_2 = [\varepsilon' \blacktriangleleft \Lambda, \pi_1 \lhd \varepsilon', c_1]\end{array}}{\Lambda, \Gamma_x \vdash \textbf{fun}\ \ x'\ \ \to\ \ e' : (\tau'^1_{\pi'_1} \xrightarrow{\varepsilon'} \tau'^2_{\pi'_2})_\Lambda/\varepsilon'\ \ [c_2]}$$

  - If $x \neq x'$, similarly to the LET-IN rule, we have: $\Gamma_x; x' : \tau'^1_{\pi'_1}/\varepsilon_1 = \Gamma_x; x' : \tau'^1_{\pi'_1}/\varepsilon_1; x : \forall\overrightarrow{\alpha'}\overrightarrow{\lambda'}[c'_1].\tau^1_{\pi_1}$. By induction of the first premise, $\Lambda, \Gamma; x' : \tau'^1_{\pi'_1}/\varepsilon_1 = \Gamma_x; x' : \tau'^1_{\pi'_1}/\varepsilon_1 \vdash \{v/x\}e' : \tau'^2_{\pi'_2}/\varepsilon'\ \ [c_2]$. Hence the result.
  - If $x = x'$, then $\Gamma_x; x' : \tau'^1_{\pi'_1} = \Gamma; x' : \tau'^1_{\pi'_1}$ and
- Case COPY. Similar to the VAR case, where $\Gamma_x(x') = \forall\overrightarrow{\alpha'}[c_0].\tau_\mathtt{b}/\varepsilon\ \ [c_0]$
- Case OPEN. Similar to the VAR case, where $\Gamma_x(x') = \forall\overrightarrow{\alpha'}[c_0].\tau\ \mathtt{par}_\mathtt{b}/\varepsilon\ \ [c_0]$
- Case OP'. Where $e \equiv \textbf{op}'\ \ e'$, similarly to the VAR case and by induction hypothesis on e'.
- Case DOWN. Similarly to the VAR case with the locality of τ constrained to $\mathtt{b}$.
- Case MULTI_FUN. Similar to the DEFFUN case.
- Other cases: the remaining rules are resolved directly as they do not deal with the environment Γ. ■

Proposition A.19 (*Semantic generalisation*)

Let v be a value and τ_π be a type such that $\models v : \tau_\pi$. Then, for all substitution φ, we have $\models v : \varphi(\tau_\pi)$.

Proof: By structural induction over v.

- Case where $v \equiv \textbf{cst}$ and $\tau = \mathtt{Base}$. With Hypothesis H0 4.17, we have $\varphi(\tau) = \tau$. Hence the result.
- Case where $v \equiv \textbf{op}$ and $\sigma = TC(\textbf{op})$. With Hypothesis H0 4.17 and α of $TC(\textbf{op})$ out of reach of φ. Hence the result.

- Case where $v \equiv \overline{(\textbf{fun}\ \ x \to e\ )[\mathcal{M}]}$ and $\tau_\pi = (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_{\Lambda'}$. Let Γ be a typing environment such that $\models \mathcal{M} : \Gamma$ and $\Gamma \vdash (\textbf{fun}\ \ x \to e\ ) : (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_{\Lambda'}$. By Lemma 4.18, we have $\varphi(\Gamma) \vdash (\textbf{fun}\ \ x \to e\ ) : \varphi((\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_{\Lambda'})$. It remains to prove that $\models \mathcal{M} : \varphi(\Gamma)$. For any $y \in Dom(\mathcal{M})$, we write $\Gamma(y)$ as $\forall \alpha_1, ..., \alpha_n \lambda_1, ..., \lambda_n[c].\tau_y$ with the α_i and λ_i chosen out of reach of φ. We then have $\varphi(\Gamma(y)) = \forall \alpha_1, ..., \alpha_n \lambda_1, ..., \lambda_n[c].\varphi(\tau_y)$. We need to prove that $\mathcal{M}(y) : \tau'_{\pi'}$ for all instances of $\tau'_{\pi'}$ of $\varphi(\Gamma(y))$. Let $\tau'_{\pi'}$ be such an instance. We therefore have $\tau'_{\pi'} = \psi(\varphi(\tau_y))$ for some substitution ψ. We also have $\models \mathcal{M}(y) : \tau_y$, by definition of $\models$ over type schemes. We can therefore apply the inductive hypothesis to the value $\mathcal{M}(y)$, to the type τ_y and to the substitution $\psi \circ \varphi$. It follows that $\models \mathcal{M}(y) : \tau'_{\pi'}$ (this holds for all instances $\tau'_{\pi'}$ of $\varphi(\Gamma(y))$). Hence $\models \mathcal{M}(y) : \varphi(\Gamma(y))$ (this holds for all $y \in Dom(e)$). Hence $\models \mathcal{M} : \varphi(\Gamma)$ and the expected result.
- Case where $v \equiv \overline{(\textbf{rec}\ \ f\ \ x \to e\ )[\mathcal{M}]}$. As previously.
- Case where $v \equiv \overline{(\textbf{multi}\ \ f\ \ x \to e_1 \dagger e_2)[\mathcal{M}]}$. As previously.
- Case where $v \equiv (v_1, v_2)$ and $\tau_\pi = \tau^1_{\pi_1} \times \tau^2_{\pi_2}$. We apply the induction hypothesis to both v_1 and v_2. It follows $\models v_1 : \varphi(\tau^1_{\pi_1})$ and $\models v_2 : \varphi(\tau^2_{\pi_2})$. Hence the result.
- Case where $v \equiv\!<< v >>$ and $\tau_\pi = \tau^1_{\pi_1}\ \mathtt{par_b}$. By induction hypothesis on v and by definition of $\models$ we have $\models v : \varphi(\tau^1_{\pi_1}\ \mathtt{par_b})$. Hence the result. ■

Lemma A.20 (*Evaluation progress*)

Let e be an expression, τ_π be a type, Γ be a typing environment, $\mathcal{M}$ be an evaluation environment such that $\Lambda, \Gamma \vdash e : \forall \overrightarrow{\alpha}\ \overrightarrow{\lambda}[c].\tau_\pi$ and $\models \mathcal{M} : \Gamma$. If there exists a value v such that $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p v$ then $\models v : \tau_\pi$

Proof: By induction on $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p v$ and by case analysis of e (and thus on the concluding rule used in the typing tree derivation).

- Case CST. e can only be reduced into itself. We must ensure that $\models \textbf{cst} : TC(\textbf{cst})$ for all constants **cst**. By definition, it is true for the constants and type assignment TC. Hence the result.
- Case OP. As previously.
- Case VAR. The only applicable typing rule is VAR. The typing system guarantees that $x \in Dom(\Gamma)$ and thus, by hypothesis on $\models \mathcal{M} : \Gamma$, $x \in Dom(\mathcal{M})$. Thus, $\mathcal{M} \vdash x \Downarrow^{\mathcal{L}}_p v$ is the only evaluation possible. By hypothesis $\models \mathcal{M} : \Gamma$, we have $\models \mathcal{M}(x) : \Gamma(x)$ and thus $\models \mathcal{M}(x) : \tau_\pi$ by definition of $\models$ over type schemes.
- Case LET_IN. The LET IN typing rule is the following:

$$\text{LET IN}\quad \frac{\begin{array}{l}\Lambda, \Gamma \vdash e_1 : \tau^1_{\pi_1}/\varepsilon_1\ \ [c_1] \\ \Lambda, \Gamma; x : \textbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1) \vdash e_2 : \tau^2_{\pi_2}/\varepsilon_2\ \ [c_2] \\ c_3 \equiv [\Psi = \textbf{Propgt}(\varepsilon_1, \varepsilon_2), c_1, c_2]\end{array}}{\Lambda, \Gamma \vdash \textbf{let}\ \ x = e_1\ \ \textbf{in}\ \ e_2 : \tau^2_{\pi_2}/\Psi\ \ [c_3]}$$

  We thus have that $\Lambda, \Gamma \vdash e_2 : \tau^2_{\pi_2}$
  Following the LET_IN semantics rule, the last evaluation step is:

$$\text{LET_IN}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e_1 \Downarrow^{\mathcal{L}}_p v_1 \\ \mathcal{M} \oplus_p \{x \mapsto v_1\} \vdash e_2 \Downarrow^{\mathcal{L}}_p v_2\end{array}}{\mathcal{M} \vdash \textbf{let}\ \ x = e_1\ \ \textbf{in}\ \ e_2 \Downarrow^{\mathcal{L}}_p v_2}$$

By induction hypothesis on e_1, we have $\models v_1 : \tau^1_{\pi_1}$. Using Lemma 4.19, $\models v_1 : \mathbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1)$. We define $\mathcal{M}_1 = \mathcal{M} \oplus \{x \mapsto v_1\}$ and $\Gamma_1 = \Gamma \oplus \{x \mapsto \mathbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1)\}$. Therefore, we have $\models \mathcal{M}_1 : \Gamma_1$. We apply the induction hypothesis on e_2 with $\mathcal{M}_1$ and Γ_1. Thus, we have $\models v_2 : \tau^2_{\pi_2}$.

- Case LET_REC. Similar to the LET_IN case.
- Case DEFFUN. The only applicable typing rule is DEFFUN. Thus $\mathcal{M} \vdash (\mathbf{fun}\ x \to e\ ) \Downarrow^{\mathcal{L}}_p \overline{(\mathbf{fun}\ x \to e\ [\mathcal{M}])}$. By definition of $\models$, we have $\models \overline{(\mathbf{fun}\ x \to e\ [\mathcal{M}])} : (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_\Lambda$.
- Case MULTI_FUN. The only applicable typing rule is MULTI_FUN as the evaluation locality is constrained to $\mathtt{m}$. Thus $\mathcal{M} \vdash (\mathbf{multi}\ f\ x \to e_1 \dagger e_2) \Downarrow^{\mathtt{m}}_{multi} \overline{(\mathbf{multi}\ f\ x' \to e'_1 \dagger e'_2)[\mathcal{M}']}$. By definition of $\models$, we have $\models \overline{(\mathbf{multi}\ f\ x' \to e'_1 \dagger e'_2)[\mathcal{M}']} : (\tau^1_{\mathtt{m}} \xrightarrow{\mathtt{m}} \tau^2_{\mathtt{m}})_{\mathtt{m}}$.
- Case APP. Here, we have two possible evaluation: (1) the application of a function closure and (2) the application of a multi-function.

  (1) For the first case (function closure) we have:

$$\text{FUN_APP}\quad \frac{\mathcal{M} \vdash e_1 \Downarrow^{\mathcal{L}}_p \overline{(\mathbf{fun}\ x' \to e'\ )[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow^{\mathcal{L}}_p v \qquad \mathcal{M} \oplus_p \mathcal{M}' \oplus_p \{x' \mapsto v\} \vdash e' \Downarrow^{\mathcal{L}}_p v'}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow^{\mathcal{L}}_p v'}$$

  By induction hypothesis, we get:
  - $\models \overline{(\mathbf{fun}\ x \to e\ )[\mathcal{M}]} : (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_{\Lambda'}$
  - $\models v : \tau^3_{\pi_3}$ (with $\pi_3 \lhd \pi_1$)

  By definition of $\models$, there exists a typing environment Γ' such that $\models \mathcal{M}' : \Gamma'$ and $\Lambda, \Gamma' \vdash (\mathbf{fun}\ x' \to e'\ ) : (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_{\Lambda'}$. As there is only on typing rule that can be derived, its premises must hold: $\Lambda, \Gamma' \oplus \{x \mapsto \tau^1_{\pi_3}\} \vdash e' : \tau^2_{\pi_2}$. Using the following environments:
  - $\mathcal{M}_1 = \mathcal{M}' \oplus \{x \mapsto v\}$
  - $\Gamma_1 = \Gamma' \oplus \{x \mapsto \tau^1_{\pi_3}\}$

  We have $\models \mathcal{M}_1 : \Gamma_1$ and $\Lambda, \Gamma_1 \vdash e' : \tau^2_{\pi_2}$ (by definition of $\models \mathcal{M}' : \Gamma'$). By applying the induction hypothesis to the evaluation of e', we get : $\models v' : \tau^2_{\pi_2}$.

  (2) For the second case (multi-function closure) we have:

$$\text{FUN_APP}\quad \frac{\mathcal{M} \vdash e_1 \Downarrow^{\mathtt{m}}_{multi} \overline{(\mathbf{multi}\ f\ x' \to e'_1 \dagger e'_2)[\mathcal{M}']} \qquad \mathcal{M} \vdash e_2 \Downarrow^{\mathtt{m}}_{multi} v \qquad \mathcal{M} \oplus_{root} \mathcal{M}' \oplus_{root} \{x' \mapsto v\} \vdash e' \Downarrow^{\mathtt{b}}_{root} v'}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow^{\mathtt{m}}_{multi} v'}$$

  By induction hypothesis, we get:
  - $\models \overline{(\mathbf{multi}\ f\ x' \to e'_1 \dagger e'_2)[\mathcal{M}']} : (\tau^1_{\mathtt{m}} \xrightarrow{\mathtt{m}} \tau^2_{\mathtt{m}})_{\mathtt{m}}$
  - $\models v : \tau^1_{\mathtt{m}}$

  By definition of $\models$, there exists a typing environment Γ' such that $\models \mathcal{M}' : \Gamma'$ and $\Lambda, \Gamma' \vdash (\mathbf{multi}\ f\ x \to e_1 \dagger e_2) : (\tau^1_{\mathtt{m}} \xrightarrow{\mathtt{m}} \tau^2_{\mathtt{m}})_{\mathtt{m}}$. As there is only on typing rule that can be derived, its premises must hold: $\Lambda, \Gamma' \oplus \{x \mapsto \tau^1_{\mathtt{m}}\} \vdash e'_1 : \tau^2_{\mathtt{m}}$. Using the following environments:
  - $\mathcal{M}_1 = \mathcal{M}' \oplus \{x \mapsto v\}$
  - $\Gamma_1 = \Gamma' \oplus \{x \mapsto \tau^1_{\mathtt{m}}\}$

  We have $\models \mathcal{M}_1 : \Gamma_1$ and $\Lambda, \Gamma_1 \vdash e'_1 : \tau^2_{\mathtt{m}}$ (by definition of $\models \mathcal{M}' : \Gamma'$). By applying the induction hypothesis to the evaluation of e'_1, we get : $\models v' : \tau^2_{\mathtt{m}}$.

- Case VECTOR. By induction hypothesis on e, which is derived from the semantics rules of **replicate**, **down** and **apply** because of the transformation, and by definition of $\models$, we have: $\models << e >> : \tau\ \mathtt{par_b}$.
- Case COPY. Similar to the VAR case.
- Case OPEN. Similar to the VAR case.
- Case OP'. Similar to the APP case as the rule consists in the application of a MULTI-ML primitive on an expression.
- Case REPLICATE. The last evaluation step of the rule is:

$$\text{REPLICATE}\quad \frac{\forall i \in p \quad \mathcal{M} \oplus_{p_i} \{f \mapsto (\textbf{fun}\ _ \to e\ )\} \vdash f\ ()\ \Downarrow^{\mathtt{l}}_{p_i} v_i}{\mathcal{M} \vdash \textbf{replicate}\ (\textbf{fun}\ _ \to e\ )\ \Downarrow^{\mathtt{b}}_{p} < v_0, ..., v_n >}$$

  By induction hypothesis, we have $\models v_i : \tau_\pi$. By definition of $\models$, there exists a typing environment Γ' such that $\models \mathcal{M}' : \Gamma'$ and $\mathtt{b}, \Gamma' \vdash << e >> : \tau_\pi\ \mathtt{par_b}$. Thus, we have $\models << e >> : \tau_\pi\ \mathtt{par_b}$.
- Case DOWN. Similar to the COPY case.
- Case APPLY. Similar to the APP case where the only possible evaluation of e_1 is a parallel vector of expressions of the form $\overline{(\textbf{fun}\ x \to e\ )[\mathcal{M}]}$.
- The remaining rules (PAIR and IF THEN ELSE) are direct. ■

Lemma A.21 (*Co-evaluation progress*)

Let e be an expression, τ_π be a type, Γ be a typing environment, $\mathcal{M}$ be an evaluation environment such that $\Lambda, \Gamma \vdash e : \tau_\pi$ and $\models \mathcal{M} : \Gamma$. Let v be a value; if it is impossible to obtain $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p v$; Then $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p \infty$.

Proof: (Using classical logic) The proof is done by coinduction and case analysis over e:

- Cases $e \equiv \textbf{cst}$ (CST), $e \equiv \textbf{op}$ (OP), $e \equiv (\textbf{fun}\ x \to e\ )$ (DEFFUN) and $e \equiv (\textbf{multi}\ f\ x \to e_1 \dagger e_2)$ (MULTI_FUN) lead to a contradiction, since they can be reduced to themselves or into a value from the environment $\mathcal{M}$.
- Case where $e \equiv x$ (VAR). We have a contradiction if $\Lambda, \Gamma \vdash x : \tau_\pi$ and $\not\models \mathcal{M} : \Gamma$.
- Case where $e \equiv (e_1\ e_2)$ (APP). Using the excluded middle, either e_1 evaluates to a value f_1, or not. In the latter case, $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p \infty$ follows from one of the rule of the application and from $\mathcal{M} \vdash e_1 \Downarrow^{\mathcal{L}}_p \infty$, which was obtained by the coinduction hypothesis. Thus, f_1 is typed $(\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_{\Lambda'}$ by Lemma 4.20.

  Using, again, the excluded middle, either e_2 evaluates to a value v, or not. In the latter case $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p \infty$ follows from one of the rule on of the rule of the application and by the coinduction hypothesis. In the former case, we have $\Lambda, \Gamma \vdash e_2 : \tau^1_{\pi_1}$. The case when f_1 is an operator clearly leads to a contradiction since, by Hypothesis H1 4.17, all operators apply to a well-typed value leads to a value. Regarding f_1, we have two cases:
  - Assuming that $f_1 \equiv \overline{(\textbf{fun}\ x' \to e'\ )[\mathcal{M}']}$, by Lemma 4.20, $\models f_1 : (\tau^1_{\pi_1} \xrightarrow{\varepsilon} \tau^2_{\pi_2})_{\Lambda'}$. Hence, there exists an environment Γ' such that $\models \mathcal{M}' : \Gamma'$. We define the two following environments:
    * $\mathcal{M}_1 = \mathcal{M}' \oplus \{x' \mapsto v\}$
    * $\Gamma_1 = \Gamma' \oplus \{x' \mapsto \tau^1_{\pi_1}\}$

We have $\models \mathcal{M}_1 : \Gamma_1$ and $\Lambda, \Gamma_1 \vdash e' : \tau^2_{\pi_2}$. With the excluded middle, we have that $\Gamma_1 \vdash e' \Downarrow^{\mathcal{L}}_p \infty$, otherwise e would evaluate to some value. Thus, the result $\mathcal{M} \vdash e \Downarrow^{\mathcal{L}}_p \infty$ follows from one of the rule of the application and coinduction hypothesis with environments Γ_1 and $\mathcal{M}_1$.

– Assuming that $f_1 \equiv \overline{(\textbf{multi } f \ x \to e_1 \dagger e_2)[\mathcal{M}]}$, by Lemma 4.20, $\models f_1 : (\tau^1_{\mathtt{m}} \xrightarrow{\mathtt{m}} \tau^2_{\mathtt{m}})_{\mathtt{m}}$. Using the excluded middle the procedure is similar to the function closure case.

- Case where $e \equiv \textbf{let } x = e_1 \textbf{ in } e_2$ (LET_IN). Similarly to the APP case.
- Case where $e \equiv \textbf{let rec } f \ x = e_1 \textbf{ in } e_2$ (LET_IN). As previously.
- Case where $e \equiv << e' >>$ (VECTOR). Using the excluded middle, $\forall i \in p$, e' can be evaluated into a value v_i ($\mathcal{M} \vdash e' \Downarrow^{\mathtt{c}}_{p_i} v_i$), or not. In the latter case, $\exists i \in p$ such as $\mathcal{M} \vdash e' \Downarrow^{\mathcal{L}}_p \infty$. Hence the result.
- Case $e \equiv \textbf{copy}$ COPY. Similar to the VAR case.
- Case $e \equiv \textbf{open}$OPEN. Similar to the VAR case.
- Case $e \equiv \textbf{op}'$ OP'. Similar to the app case as the rule consists in the application of a MULTI-ML primitive on an expression.
- Case $e \equiv \textbf{replicate}$ REPLICATE. Using the excluded middle, $\textbf{replicate } (\textbf{fun } _ \to e')$ can be evaluated into a value or not. In the latter case, by coinduction hypothesis $f \mapsto (\textbf{fun } _ \to e' \ )\} : (\mathtt{unit}_{\pi_1} \xrightarrow{\varepsilon_1} \tau^2_{\pi_2})_{\Lambda'}$ and $\mathcal{M} \vdash e' \Downarrow^{\mathtt{c}}_{p_i} \infty$. Thus, $\exists i \in p$ such that $\mathcal{M} \oplus_{p_i} \{f \mapsto (\textbf{fun } _ \to e \ )\} \vdash f \ () \Downarrow^{\mathtt{l}}_{p_i} \infty$. Hence the result.
- Case $e \equiv \textbf{down}$ DOWN. Similar to the COPY case.
- Case $e \equiv \textbf{apply}$ APPLY. Similar to the APP case where the only possible evaluation of e_1 is a parallel vector of functions $\overline{(\textbf{fun } x \to e \ )[\mathcal{M}]}$.
- The remaining rules (PAIR and IF THEN ELSE) are direct. ■