Squelettes algorithmiques asynchrones : application aux langages orientés domaine

Antoine Tran Tan

NNT : 2015SACLS025

THESE DE DOCTORAT
DE L'UNIVERSITE PARIS-SACLAY,
préparée à l'Université Paris-Sud

ÉCOLE DOCTORALE N°580
Sciences et technologies de l'information et de la communication

Spécialité de doctorat : Informatique

Par

M. Antoine TRAN TAN

Squelettes algorithmiques asynchrones : application aux langages orientés domaine

Thèse présentée et soutenue à Gif-sur-Yvette, le 8 octobre 2015 :

Composition du Jury :

M. Frédéric Loulergue, Professeur, Université d'Orléans, Rapporteur
M. Stéphane Vialle, Professeur, CentraleSupélec, Campus de Metz, Rapporteur
M. Sylvain Conchon, Professeur, Université Paris-Sud, Président du Jury
M. François Irigoin, Directeur de Recherche, Mines ParisTech, CRI, Examinateur
M. Joël Falcou, Maître de Conférences, Université Paris-Sud, Co-encadrant
M. Daniel Etiemble, Professeur émérite, Université Paris-Sud, Directeur de thèse

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Squelettes algorithmiques asynchrones : application aux langages orientés domaine

Mots clés : Programmation asynchrone, DSELs, Programmation générative, Programmation générique, C++

Résumé

Les architectures parallèles sont aujourd'hui présentes dans pratiquement tous les ordinateurs, que ce soit les superordinateurs, les PCs de bureau ou les smartphones. Or l'utilisation efficace de ce type de système implique généralement un effort supplémentaire de la part des développeurs et des scientifiques que ce soit dans la prise en compte des nouveaux aspects architecturaux ou bien dans la mise en pratique de la programmation parallèle. Sur système à mémoire partagée, la programmation par threads s'impose encore comme un standard pour l'exploitation du parallélisme entre les coeurs CPU. Elle force cependant l'utilisateur à faire face aux nombreuses problématiques tels que la gestion des threads et la concurrence.

Pour palier ce problème, d'autres modèles sont explorés : on cite en particulier le modèle *Futures* qui intègre le concept plus général de *programmation parallèle asynchrone*. Dans ce modèle, le travail de création et de gestion des threads est délégué au runtime et des *graphes de dépendances* sont définis par l'utilisateur pour garantir la consistence du calcul. Dans cette thèse, nous présentons l'intégration de ce modèle dans NT^2, une bibliothèque développée par l'équipe *Parsys* du LRI qui propose une interface simple et intuitive pour le calcul numérique. NT^2 s'appuie notamment sur des techniques de méta-programmation par templates C++ pour analyser, restructurer et optimiser des portions de codes arbitraires en amont du compilateur C++. Plus précisément, NT^2 emploie des *Expression Templates* pour forcer les expressions à renvoyer un type encodant leur *arbre de syntaxe abstraite* (ou *AST* comme *Abstract Syntax Tree*) au lieu d'être évaluées tout de suite. Une fois analysés, les ASTs sont dirigés vers des familles de nids de boucles puis transformés en appels de *squelette algorithmique*.

Notre première contribution dans cette thèse est d'implanter des versions asynchrones de ces squelettes algorithmiques pour garantir la performance du code généré. Pour ce faire, nous décomposons chaque squelette sous forme de tâches en suivant le modèle *worker/spawner* : les *workers* sont des noyaux de calcul monotâche possiblement vectorisables, et les *spawners* sont des fonctions chargées de paralléliser les calculs en créant plusieurs workers en charge d'un intervalle multi-dimmensionnel de données. La gestion correcte des dépendances entre tâches est réalisée par résolution des aléas de données.

Notre deuxième contribution dans cette thèse est d'établir des modèles de coût permettant d'estimer la performance des versions séquentielles et parallèles des

squelettes NT^2 dans une architecture visée. Ces modèles de coût sont calculés *en ligne* afin de sélectionner *à la volée* les meilleurs versions de squelette. Les prévisions de performance se basent sur trois caractéristiques : les métriques de ressources mesurables par microbenchmarks, les caractéristiques du code de l'application déterminables pendant la compilation et les caractéristiques du jeu de données disponibles pendant l'exécution.

Nous évaluons ces contributions sur quatre applications : une (LU) est de type *compute bound* et les trois autres (Black & Scholes, Lattice Boltzmann et GMRES) sont plutôt *memory bound*. LU est très favorable à l'utilisation de l'asynchronisme alors que la structure et le coté itératif de GMRES limitent l'impact de l'asynchronisme. Les deux autres correspondent à une situation intermédiaire où l'asynchronisme peut cependant être utilisé. Les résultats montrent un gain significatif dans les programmes favorables à l'asynchronisme sans qu'il y ait de surcoût significatif pour les programmes se prêtant moins à l'utilisation de l'asynchronisme.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Asynchronous algorithmic skeletons : application to domain specific languages

Keywords : Asynchronous programming, EDSLs, Generative programming, Generic programming, C++

Abstract

Parallel architectures are used in almost every computers from supercomputers to desktop computers or smartphones. Yet, the effective use of this type of system still requires extra effort from developers and scientists that must consider the new architectural features to develop parallel programs. In shared memory systems, thread-based programming is still considered a standard to exploit parallelism between the CPU cores. This forces the user to deal with many issues such as thread management or concurrency.

To overcome this problem, other models are explored. One of the most prominent model is the *Future* model which integrates the general concept of *asynchronous parallel programming*. In this model, the task of creating and managing threads is delegated to the runtime and *dependency graphs* are defined by the user to ensure the consistency of the computation. In this thesis, we present the integration of this model in the NT^2 library developed by *Parsys* team at LRI. It encapsulates a domain specific language and provides a simple and intuitive interface for numerical computations. NT^2 uses *C++ template metaprogramming* to analyze, restructure and optimize portions of arbitrary codes at compile-time. More specifically, NT^2 employs *Expression Templates* to cause expressions to build a representation of their abstract syntax trees (ASTs) instead of being evaluated eagerly. Once analyzed, the ASTs are matched to a given family of loop nests and then transformed into *algorithmic skeleton* calls.

Our first contribution is to implement asynchronous versions of algorithmic skeletons to guarantee the performance of the generated code. To do this, we decompose each skeleton in the form of lightweight tasks following the *worker/spawner* model : the *workers* are single task kernels that are possibly vectorized, and *spawners* are functions dedicated to parallelize computations by creating several workers in charge of a multi-dimensional data range. The proper management of dependencies between tasks is performed by resolving data hazards.

Our second contribution is to establish cost models to estimate the performance of sequential and parallel versions of NT^2 skeletons for a target architecture. These cost models are calculated *online* to select the best skeleton version. Performance predictions are based on three specifications: the resources metrics measured with microbenchmarks, the application code characteristics determined at compile-time and the dataset features that are available at execution time.

We evaluate these contributions with four applications of the scientific computing domain: one (LU) is *compute bound* and the three other ones (Black & Scholes, Lattice Boltzmann and GMRES) are *memory bound*. LU is suitable for asynchronous computing while GMRES structure and iterative computation limit the impact of asynchronous computing. The two other ones correspond to an intermediate position where asynchronous computing can still be used. The results show a significant performance gain for programs that are suitable for asynchronous computing without significant overheads for programs less suitable to the use of the asynchronous approach.

Remerciements

Je tiens tout d'abord à remercier Pr. Daniel Etiemble et Dr. Joël Falcou pour la confiance qu'il m'ont témoigné tout au long de mes trois années de thèse et pour leur implication dans la rédaction de ce manuscrit. J'ai vraiment apprécié leur expertise et leur disponibilité.

J'aimerais ensuite remercier Dr. Hartmut Kaiser et toute l'équipe Ste||ar de l'université de Louisiane pour m'avoir accueilli dans leur laboratoire et pour leur aide précieuse dans l'utilisation d'HPX.

Je remercie aussi Messieurs les professeurs Frédéric Loulergue et Stéphane Vialle pour m'avoir fait l'honneur de relire ce manuscrit et pour avoir contribué par leurs remarques et suggestions à la qualité de ce mémoire.

Je remercie ensuite Messieurs les professeurs François Irigoin et Sylvain Conchon pour m'avoir fait l'honneur de participer au Jury de soutenance et pour leurs remarques constructives.

Je tiens enfin à adresser toute ma gratitude aux personnes avec qui j'ai eu des discussions fructueuses et qui ont contribué plus ou moins directement à l'avancement de mes travaux : Rémi, Lénaïc, Mathias, Pierre, Mikolaj, Riadh, Adrien, Yushan, Amal, Long, Ian, JT, Marie, Alan, Anchen, Sylvain, Christine, Loïc, Florence, Lionel, Laurent, Andrea, Farouk.

Tables des matières

Liste des Figures

Liste des Tableaux

Introduction

Sommaire

Parallélisme et concurrence

Les dernières avancées en matière d'architecture des processeurs ont permis aux supercalculateurs modernes d'effectuer plus d'un million de milliards d'opérations flottantes par seconde ; la transition vers l'ère de l'exaflop est d'ailleurs annoncée à l'horizon 2020. Le facteur clé permettant aujourd'hui d'atteindre de telles performances sont les *architectures parallèles*.

La mise en œuvre des architectures parallèles n'est pas une révolution mais a pris de l'ampleur ces dernières années depuis que les fréquences d'horloge ont cessé de croître. Afin de toujours améliorer les performances sans augmenter la consommation énergétique, les fondeurs de processeurs se sont progressivement tournés vers les technologies multi-cœurs. Il devient alors indispensable pour un programmeur de maîtriser la *programmation parallèle* pour profiter pleinement de la performance des nouvelles architectures. Le fossé entre la programmation classique et la programmation parallèle continue toutefois de se creuser. Nous discutons dans la suite des deux principaux modèles de programmation parallèle.

Dans les systèmes à mémoire partagée, la tendance est de suivre le *modèle thread*. Cette tendance s'explique d'abord par le fait que les threads ont émergé bien avant l'adoption des processeurs multi-cœurs, et qu'ils étaient déjà présents en grand nombre dans les applications de bureau. Cette tendance est portée également par la popularité des bibliothèques telles que Pthreads ou OpenMP car elles fournissent chacune une API facilitant la mise en œuvre des threads système natifs. Ces bibliothèques présentent toutefois un inconvénient majeur. Comme elles ne définissent pas de règles strictes pour les programmes présentant des *data races*, *i.e.* situations dans lesquelles des variables partagées sont lues et écrites de manière concurrente

par plusieurs threads à la fois, le seul moyen d'éviter d'éventuelles *situations de compétition* est de faire appel aux primitives de synchronisation (*verrous*, *sémaphores* et *barrières*), mais leur utilisation doit se faire avec prudence car ces mécanismes sont connus pour entraîner un surcoût substantiel en terme de temps d'exécution.

Dans les systèmes à mémoire distribuée, la tendance est de suivre le *modèle par passage de messages*. Ce modèle, qui se prête bien aux gros calculs scientifiques, a émergé de la communauté HPC pour utiliser au mieux les ressources massivement parallèles. En utilisant PVM ou MPI, les programmeurs peuvent d'une part écrire explicitement dans un même programme le code des différents processus émetteurs et récepteurs de messages, et d'autre part organiser leurs calculs sous forme de blocs synchrones – on parle de *Bulk Synchronous Programming*. Dans ce schéma, chaque processus exécute les mêmes instructions en parallèle, et effectuent par intermittence des opérations collectives pour garantir la consistance du calcul. L'inconvénient de ce schéma est que lorsqu'un système regroupe différents types de processeurs, l'utilisateur devra se confronter aux problèmes de *répartition de charge* pour garantir la performance.

Pour résumer, le principal problème de la programmation parallèle est qu'elle repose encore principalement aujourd'hui sur des modèles de programmation bas niveau et fait face aux nombreuses problématiques qui en découlent. D'une part, la mauvaise utilisation des différents mécanismes de synchronisation peut être source de dégradations de performance. D'autre part, il devient inéluctable que l'utilisateur ait à interagir avec différents modèles de programmation selon que le système à utiliser est de type partagé, distribué, hétérogène (ex : système combinant CPUs et GPUs), ou les trois à la fois. Nous discutons dans la suite des principaux types de parallélisme ainsi que des approches permettant de faire évoluer plus efficacement la programmation parallèle avec un objectif ambitieux : tirer parti des cœurs disponibles avec une programmation proche de la programmation séquentielle.

Défis de la programmation parallèle

Tout au long du processus de programmation, de la formulation de l'algorithme jusqu'à l'exécution du programme résultant, deux types de parallélisme sont constamment considérés: le *parallélisme de données* et le *parallélisme de tâches*.

Le parallélisme de données se définit comme la situation dans laquelle une instruction, une (ou plusieurs) routine(s) sont appliquées à chaque élément d'une structure de données. Le premier cas correspond à l'utilisation des instructions SIMD qui se sont développées depuis le milieu des années 90. Pour les jeux d'instructions IA-32 et Intel 64, ce sont les différentes extensions SSE et AVX. Elles permettent d'exploiter au mieux le parallélisme de données existant dans un code au niveau de chaque processeur (ou cœur) individuel avant même d'aborder les problèmes de

parallélisation entre différents cœurs ou clusters de multi-cœurs. Le second cas correspond au modèle SPMD où différents processeurs (cœurs) exécutent le même code sur des sous-ensembles de données différents. L'exécution n'étant pas synchrone au niveau de chaque instruction, des barrières de synchronisation sont nécessaires, que ce soit en mémoire partagée ou en mémoire distribuée, pour tenir compte des différences de temps d'exécution liées aux structures conditionnelles. Dans le cas des architectures distribuées, il faut préalablement répartir la structure de données entre les mémoires locales des différents processeurs. Ensuite, pour toutes les architectures, chaque thread (ou processus) effectue en parallèle le même calcul sur leur propre segment de données, en faisant éventuellement appel à des opérations de réduction. Le parallélisme de données est la source de parallélisme qui est généralement exploitée en premier car il arrive souvent que l'accélération désirée soit atteinte en remplaçant simplement les nids de boucle séquentiels très gourmands en calcul par des nids de boucle parallèles.

Le parallélisme de tâches se définit comme la situation dans laquelle l'ensemble (ou une partie) du code séquentiel est divisé en plusieurs unités de travail qui peuvent s'exécuter de manière concurrente et potentiellement en parallèle. Par conception, le parallélisme de tâches est plus compliqué à mettre en œuvre que le parallélisme de données et cela pour deux raisons. D'abord, le programmeur doit évaluer la *granularité des tâches*, *i.e.* c'est-à-dire la quantité de travail à attribuer à chaque tâche, en gardant à l'esprit que selon le système utilisé, des tâches à gros grain peuvent entraîner un *déséquilibre de charge* alors que des tâches à grain fin peuvent entraîner un surcoût en temps d'exécution. Ensuite, exploiter le parallélisme de tâches implique d'employer un *graphe de dépendances* et de mettre en œuvre des techniques permettant d'ordonnancer les différents noeuds du graphe. Comme l'ordonnancement d'un algorithme sur une architecture multi-processeurs est un problème *NP-complet* [58, 27, 64], la tendance est plus orientée vers l'utilisation d'un ordonnanceur qui va émettre, soit de manière statique, soit de manière dynamique, des décisions basées sur des heuristiques pour ordonnancer l'ensemble des tâches vers les processeurs.

Comme nous l'avons vu précédemment, la programmation parallèle implique de confronter plusieurs aspects de la programmation pour exploiter au mieux le parallélisme potentiel des applications sur une architecture donnée. Bien qu'il soit incontestable que les techniques bas niveau (modèle par passage de messages et modèle thread) soient là pour durer, leur mauvaise utilisation peut nuire à la fois sur l'expressivité des codes et sur leur performance. C'est pourquoi de nombreux groupes de recherche travaillent activement sur des approches de plus haut niveau pour réduire la complexité d'écriture des programmes parallèles tout en maintenant un haut niveau de performance. Parmi ces approches, on cite celles qui facilitent l'exploitation du parallélisme de données sur systèmes à mémoire partagée [43, 50], celles qui facilitent l'exploitation des deux types de parallélisme sur systèmes à mémoire distribuée [60, 25, 24], puis les **Langages Orientés Domaine** (ou *DSL*s

comme *Domain Specific Languages*).

Pour les applications séquentielles, l'efficacité des Langages Orientés Domaine est bien établie ; la sémantique haut niveau du domaine d'application est préservée et la portabilité est assurée pour tout type de systèmes informatiques. Malgré leurs résultats en terme de productivité et d'économies de temps, les *DSL*s peuvent entraîner des coûts importants incluant les coûts d'apprentissage du langage et les coûts associés à sa mise en œuvre. Pour limiter ces contraintes, les *Langages* ***Enfouis*** *Orientés Domaine* (ou *DSEL*s comme *Domain Specific Embedded Languages* [48, 68]) ont été proposées. Les DSELs sont des langages imbriqués dans un langage hôte ; ils sont compilés ou interprétés dans le même écosystème que le langage hôte. C'est cette approche qu'a développé Joel Falcou dans les bibliothèques NT^2, et Boost SIMD, qui a fait l'objet de publications [41, 39]. Les outils développés autour de NT^2 ont permis d'obtenir des résultats spectaculaires. C'est donc le point de départ de notre travail.

NT^2 : Une bibliothèque pour le calcul scientifique

NT^2 – Numerical Template Toolbox – est une bibliothèque C++ conçue pour aider les non-spécialistes issus de différents domaines à développer des applications haute performance [40]. Afin de fournir un outil performant qui soit utilisable par le plus grand nombre, NT^2 implémente l'essentiel des fonctionnalités du langage MATLAB via la mise en œuvre d'un *DSEL* écrit en C++ tout en garantissant un haut niveau de performance. La structure de données centrale utilisée dans NT^2 est la classe template `table` qui est l'équivalent d'une matrice dans MATLAB. Les spécificités comme l'indexation commençant par 1, la gestion des dimensions et les opérations de redimensionnement sont préservées. Voici un exemple de code NT^2 qui calcule l'écart quadratique entre deux vecteurs.

```
// Matlab: A1 = 1:1000;
table<double> A1 = _(1.,1000.)

// Matlab: A2 = A1 + randn( size(A1) );
table<double> A2 = A1 + randn(size(A1));

// Matlab: rms = sqrt( sum((A1(:) - A2(:)).^2) / numel(A1) );
double rms = sqrt( sum(sqr(A1(_) - A2(_))) / numel(A1) );
```

Listing 1: Calcul d'un écart quadratique dans NT^2

Dans cette exemple de code, remarquons la sémantique de premier ordre des objets *tables* via l'utilisation d'opérations globales, l'utilisation de `_` à la place de l'opérateur colonne (:) de MATLAB et la mise à disposition de fonctions telles que `numel` et `size` qui assurent le même calcul que leur équivalent MATLAB. Plus de 80% des fonctions de base de MATLAB sont disponibles. L'interfaçage avec le langage C++ est garanti pour assurer la compatibilité avec les algorithmes de la norme C++ et les bibliothèques majeures issues de la suite logicielle Boost C++ [34]. NT^2 se repose sur trois types d'optimisations : le parallélisme d'instructions par l'usage implicite des extensions SIMD via BOOST.SIMD [39], le parallélisme au niveau thread via OpenMP ou Intel TBB et les *Expression Templates* que nous présenterons dans le chapitre 2.

Néanmoins, les performances de NT^2 restent encore entravées par les mécanismes de synchronisation liés au modèle de programmation par threads. Or, des techniques relâchant les contraintes de synchronisation ont été étudiées depuis plusieurs années et sont généralement connues sous la terminologie de *programmation parallèle asynchrone*. Elles correspondent notamment au modèle *Futures et Promesses*, qui ont été intégrées dans la nouvelle norme C++ puis étendues dans l'outil HPX conçu par l'équipe *Ste||ar* du Centre de Calcul et Technologie de l'Université d'État de Louisiane. HPX est un runtime multi-plateformes encapsulé dans une bibliothèque C++. Les threads système sont remplacés par des tâches légères que nous présenterons dans le chapitre 1.

Objectifs

L'objectif de cette thèse est l'intégration des techniques de programmation asynchrone dans NT^2, et l'évaluation des performances obtenues sur des benchmarks significatifs. Les objectifs de notre travail sont :

Mettre en œuvre des tâches légères pour le calcul scientifique

La recherche actuelle en terme de modèles de programmation parallèle met en exergue l'utilisation de tâches à très faible grain et de runtimes fonctionnant dans des plate-formes de plus en plus larges et de plus en plus hétérogènes. Le premier objectif de ce travail est d'explorer ces modèles de programmation afin de mettre en œuvre une conversion automatique de codes orientés domaine en codes de production haute performance.

Intégrer des modèles de coût pour la génération automatique de code

Il existe de nombreux modèles de programmation parallèle dans la littérature qui visent à contraindre les programmes à n'être construit qu'à partir de noyaux à temps d'exécution déterministe. Les squelettes algorithmiques, l'un de ces modèles, ont

l'avantage de fournir à la fois une abstraction pour le parallélisme et une efficacité qu'il est possible de prédire. Le deuxième objectif de ce travail est donc d'établir des modèles de coût pour un sous-ensemble de squelettes algorithmiques afin d'orchestrer au mieux la génération automatique de code.

Notre contribution

À travers ce travail, nous avons proposé une solution pour augmenter un système initialement conçu pour faciliter le portage haute performance d'applications scientifiques existantes, en utilisant des squelettes algorithmiques asynchrones pour garantir et contrôler la performance des codes résultants. Ce manuscrit est organisé de la manière suivante :

- **Chapitre 1, Programmation multi-threads en C++.** Dans ce chapitre, nous présentons à travers les évaluations du langage C++ les concepts clés de la programmation par threads tout en soulignant les problématiques liées à la concurrence sur système à mémoire partagée.

- **Chapitre 2, Expression Templates et squelettes orientés domaine.** Dans ce chapitre, nous présentons les principaux outils utilisés dans NT^2 pour garantir la performance. Nous verrons ensuite les limites rencontrés et présentons dans ce contexte l'outil HPX, un runtime C++ qui met en œuvre des tâches légères et qui offre une API asynchrone totalement conforme avec la nouvelle norme C++.

- **Chapitre 3, Génération automatique de code concurrent pour un langage enfoui orienté domaine.** Pour **améliorer** le déploiement d'un code NT^2 sur systèmes à mémoire partagée, nous avons reconstruit son système de construction de squelettes algorithmiques multi-niveaux afin que le code utilisateur soit traduite sous forme d'une composition de tâches asynchrones.

- **Chapitre 4, Modèles de coût pour la génération de code.** Dans ce chapitre, nous présentons l'extension du système de génération automatique de squelettes algorithmiques afin d'y inclure des modèles de coût statiques qui tirent parti, d'une part de métriques de ressources pertinentes et d'autre part des estimations de coût extraites directement des expressions du DSEL. Ces modèles de coût sont ensuite raffinés pendant l'exécution afin de décider *à la volée* des différentes versions de squelette à utiliser.

- **Chapitre 5, Évaluation de performances.** Dans ce chapitre, nous évaluons nos outils en utilisant une collection d'applications scientifiques du monde réel. Ces benchmarks se divisent en deux catégories : les applications *limitées par la bande passante mémoire* et les applications *limitées par la performance crête*.

- **Conclusion et perspectives.** Dans ce chapitre final, nous synthétisons l'ensemble des résultats obtenus dans ce travail. En conclusion, nous discutons des possibles orientations pour de futurs travaux de recherche.

Chapitre 1

Programmation multi-threads en C++

Sommaire

Malgré le gain de popularité de la programmation multi-threads et cela dès le début des années 90, le comité ANSI/ISO a choisi de ne pas en tenir compte lors de la refonte de la norme C++ en 1998. Ainsi jusqu'à il y a quelques années, les programmeurs C++ désirant manipuler les threads avaient recours à des extensions spécifiques aux plate-formes [52, 62], le plus souvent implémentées comme surcouche de Pthreads ou de Windows threads. Conscient de l'importance de ce modèle de programmation, accentuée par la mise sur le marché des processeurs multi-cœurs, le comité de normalisation du langage C++ a décidé en 2011 d'inclure de nouveaux composants permettant à la fois de faciliter la programmation multi-threads et à la fois d'écrire du code parallèle portable qui puisse fonctionner sur différentes plate-formes. Nous passons en revue dans cette partie les mécanismes clés de la programmation par threads et les fonctionnalités correspondantes présentes dans la norme C++ 2011.

1.1 Gestion des threads

Tout programme décrit en C++ lance au moins un thread : celui qui exécute la fonction `main()`. Un programme peut ensuite lancer d'autres threads en leur spécifiant d'autres points d'entrée. Ces threads vont alors s'exécuter en concurrence avec le thread initial. Voici un exemple de code montrant comment lancer un thread avec la nouvelle norme C++:

```
void f(std::string const & s)
{
  std::cout << "Hello " << s << std::endl;
}

std::string mot = "World!";
std::thread t (&f,mot);
/* ... */
t.join();
```

❶ ❷ ❸

Listing 1.1: Lancer un thread et attendre celui-ci

Dans cet exemple, il s'agit de lancer une séquence d'instructions ❶ dans un nouveau thread. Pour cela on instancie un objet thread ❷ en lui spécifiant son point d'entrée et les paramètres à utiliser. Pour attendre que cette fonction ait fini de s'exécuter, on appelle la méthode `join()` ❸ sur l'objet thread.

Une des fonctionnalités intéressantes de la norme C++ 2011 est de pouvoir définir des fonctions locales *anonymes* capables de capturer des variables dans leur *portée* : ce sont les fonctions *lambda*. La syntaxe est la suivante:

```
[ liste de capture ]( paramètres de la fonction ) -> type de retour
{ corps de la fonction }
```

Avec une fonction lambda, la création du thread précédent peut s'écrire comme ci-dessous:

```
std::thread t ( [mot]()  ❶
                { std::cout << "Hello" << mot << std::endl; }
              );
```

Listing 1.2: Lancer un thread avec passage d'une fonction lambda

Notons ici que le type de retour n'est pas indiquée ❶. La lambda ne retournant aucune valeur, le type de retour est implicitement défini comme `void`.

Une fois le thread lancé, l'utilisateur doit s'assurer, ou bien d'appeler la méthode `join()` pour attendre que le thread se termine, ou bien, d'appeler la méthode `detach()` pour permettre au thread de s'exécuter en arrière-plan, et cela avant que l'instance de `std::thread` ne soit détruite. Dans le cas contraire, c'est le destructeur de la classe `std::thread` qui se charge d'interrompre le programme entier.

La méthode `join()` est la manière la plus simple d'attendre qu'un thread ait fini son travail, mais ce mécanisme reste un peu faible en terme d'efficacité et a surtout le défaut de n'être invocable qu'une fois, car en sortie d'appel à `join()`, le thread système sous-jacent aura disparu.[1]

[1] Nous verrons en section 1.3 qu'il existe d'autres mécanismes dans la norme C++ permettant d'améliorer la granularité des synchronisations.

1.2 Gestion des accès aux données

Pour revenir sur notre propos, la raison clé qui incite une bonne majorité des programmeurs à utiliser le modèle de programmation multi-thread est la simplicité du mode de communication entre threads vu qu'ils partagent le même espace mémoire. Il faut toutefois faire attention aux accès concurrents aux données partagées et donc dans la plupart des cas, avoir recours aux *mutexes* (ou *exclusions mutuelles*). Voici un exemple de code illustrant l'utilisation des mutexes avec la nouvelle norme C++:

```
std::vector<int> v;
std::mutex m;                    <-- 1

void ajouter_valeur(int valeur)
{
    m.lock();                    <-- 2
    v.push_back(valeur);
    m.unlock();                  <-- 3
}

std::thread t1 (& ajouter_valeur, 1);
std::thread t2 (& ajouter_valeur, 2);
```

Listing 1.3: Protéger l'accès concurrent à une donnée

Ici le mutex est créé au moment où la variable `m` est instanciée ❶. Une section critique débute alors en appelant la méthode `lock()` ❷ de l'objet `std::mutex` et se termine en appelant la méthode `unlock()` ❸. Pour plus de sûreté et une gestion efficace des éventuelles exceptions, la nouvelle norme C++ recommande d'utiliser des objets *verrous* par l'intermédiaire des classes template `std::lock_guard` et `std::unique_lock`[2]. Conformément à la notion d'*Acquisition de Ressources par l'Initialisation* (ou RAII en anglais), c'est au moment de la création de l'objet verrou que le mutex est verrouillé et au moment de sa destruction qu'il est déverrouillé. Ces classes C++ s'avèrent pratiques vu qu'il arrive souvent que la portée des corps de fonctions coïncident avec les sections critiques désirées. De plus, si une exception devait survenir dans la portée de ces objets, le mécanisme de *stack unwinding* avant retour de la fonction appellera automatiquement les destructeurs de ces objets. Les ressources acquises jusqu'ici seront donc proprement restituées quoi qu'il arrive. Le code précédent se réécrit donc comme ci-dessous:

```
void ajouter_valeur(int valeur)
{
  std::lock_guard<std::mutex> l(m);
  v.push_back(valeur);
}
```

Listing 1.4: Protéger un accès concurrent par verrou

[2]La classe `std::unique_lock` fonctionne comme la classe `std::lock_guard` mais possède des méthodes supplémentaires `lock()`, `try_lock()` et `unlock()` permettant à l'utilisateur de gérer manuellement le verrouillage/déverrouillage des mutexes.

Ayant présenté les fondamentaux de la programmation multi-threads, nous discutons dans la suite de la problématique de synchronisation des threads dans le cas très particulier où il s'agit de faire en sorte qu'un thread **attende** qu'un autre thread ait fini une opération (et non pas qu'il se termine) pour pouvoir continuer.

1.3 Synchronisation sur événement ponctuel

L'idée la plus simple pour réaliser un *point de synchronisation*, *i.e.* faire en sorte qu'un thread attende qu'un événement se produise, est d'utiliser une méthode par *scrutation de flags*. En d'autres termes, le thread en attente scrute en boucle le changement d'état d'une variable partagée, le changement d'état pouvant être effectué par un autre thread. Le problème avec ce type de solution est qu'il s'agit d'une attente active impliquant forcément de monopoliser un thread actif et donc de gaspiller une ressource CPU pour ne rien faire. Pour réduire le nombre d'attentes actives, l'idée est de mettre en sommeil les threads en attente et de laisser le système les réveiller.

1.3.1 Synchronisation par variables conditionnelles

L'approche la plus conventionnelle permettant d'effectuer une synchronisation sur événement ponctuel est d'utiliser des *variables conditionnelles*. Voici un exemple de code basé sur le schéma *producteur/consommateur* montrant comment réaliser ce type de point de synchronisation avec la norme C++ 2011:

```
std::mutex m;
std::condition_variable condition;          <-- 1
int reponse;

// Code du thread producteur
void producteur()
{
  std::unique_lock<std::mutex> l1(m);       <-- 2
  reponse = 42;                             <-- 3
  condition.notify_one();                   <-- 4
}

// Code du thread consommateur
void consommateur()
{
  std::unique_lock<std::mutex> l2(m);       <-- 5
  condition.wait(l2);                       <-- 6
  std::cout << "Reponse a la grande question de l'univers: "
  << reponse << std::endl;
}
```

Listing 1.5: Synchroniser deux threads avec une variable conditionnelle

Ici, la variable conditionnelle est créée au moment où la variable `condition` est instanciée ❶ ; par convention, l'utilisateur doit toujours se servir d'un mutex pour chaque variable conditionnelle qu'il utilise. Comme indiqué dans le code exemple, le thread producteur doit successivement:

- Verrouiller le mutex
- Produire la valeur ❸
- Notifier l'événement ❹
- Déverrouiller le mutex

Le thread consommateur doit successivement:

- Verrouiller le mutex
- Attendre l'événement ❻. Cette étape déverrouille le mutex, puis met en veille le thread en attente. Le réveil par le système se fait après notification de l'événement, ce qui va reverrouiller le mutex.
- Traiter l'événement
- Déverrouiller le mutex

Notons dans l'exemple, que les étapes de verrouillage et déverrouillage du mutex sont implémentés dans les deux cas de manière **simple** et **sécurisé** via l'instanciation d'un objet verrou `std::unique_lock`. ❷❺

1.3.2 Synchronisation par Futures

La méthode de synchronisation par variable conditionnelle a l'avantage de retranscrire des concepts déjà présents dans les API Pthreads et Windows threads, mais semble cependant assez contraignante dans sa mise en œuvre (utilisation explicite d'un mutex, attente du consommateur **avant** signalement du producteur, gestion des *faux réveils*, *i.e.* situation dans laquelle un thread est réveillé sans signalement d'une variable conditionnelle, etc.). Nous voyons dans cette partie un mécanisme de synchronisation moins conventionnel qui met à profit une sémantique plus proche du schéma producteur/consommateur laissant, de fait, le soin au compilateur de gérer les aspects sous-jacents liés à la concurrence.

Dans la littérature, on parle plus de modèle *Futures et Promesses* que de modèle *Futures*. Conceptuellement parlant, les *Futures* sont des proxys vers des résultats qui seront disponibles plus tard. A contrario, les *Promesses* donnent le droit à leur détenteurs d'écrire ces résultats. Pour revenir au schéma producteur/consommateur, on définit le producteur comme étant le thread qui détient la Promesse, et le consommateur comme le thread qui détient la Future.

Voici un exemple de code montrant comment réaliser un point de synchronisation par Future avec la nouvelle norme C++:

```
std::promise<int> p;                                    ←❶

// Code du thread producteur
void producteur()
{
  p.set_value(42);                                      ←❷
}

// Code du thread consommateur
void consommateur()
{
  std::future<int> f = p.get_future();                  ←❸

  /* ... */

  std::cout << "Reponse a la grande question de l'univers: "
  << f.get() << std::endl;                              ←❹
}
```

Listing 1.6: Synchroniser deux threads avec une Future

Ici, nous voyons que la Future ❸ et la Promesse ❶ ont chacun un paramètre template de type `int` : c'est le type du résultat commun. Ce résultat est implémenté sous forme d'*état partagé* qui est créé dynamiquement au moment où `p` est instanciée ❶. En appelant la méthode `get_future()` ❸, le consommateur obtient une Future référençant le même état partagé que `p`.

Quand le consommateur appelle la méthode `get()` ❹, plusieurs scénarios peuvent intervenir[3] ; ici deux sont possibles:

- Si l'état partagé est **non prêt**, c'est-à-dire que la méthode `set_value()` ❷ n'a pas encore été appelée, le thread consommateur est mis en sommeil jusqu'à que l'état partagé soit prêt.
- Si l'état partagé est **prêt**, c'est-à-dire que la méthode `set_value()` a déja été appelée, le résultat est retourné immédiatement.

Il y a deux avantages par rapport à l'approche par variables conditionnelles: on ne voit apparaître aucune instance explicite de verrous, et l'ordre dans lequel s'effectuent le signalement de l'événement et la capture de l'événement est indifférenciée. Un détail reste quand même à souligner : la gestion de l'état partagé dans la mémoire. La nouvelle norme C++ permet maintenant d'utiliser des *pointeurs intelligents*, *i.e.* des pointeurs capables de gérer automatiquement la destruction des objets pointés. Dans le cas de l'état partagé, celui-ci est alloué dynamiquement par l'intermédiaire d'un objet `std::shared_ptr`, un type de pointeur intelligent mettant en œuvre un compteur atomique de possesseurs de l'objet pointé. La Promesse est donc le premier *possesseur* de l'état partagé. Quand la Future est créée (via la méthode `get_future()`), celle-ci incrémente ce compteur. La règle est donc simple : c'est lorsque le dernier possesseur de l'état partagé est détruit que l'état partagé

[3] Nous parlerons d'un de ces scénarios plus tard en Section 1.4

est détruit.

```
std::future<int> f;
std::shared_future<int> fcopie = f.share();     <-- 1

std::thread t1( [fcopie]()   <-- 2
                { printf( "Thread 1 repond: %d\n", fcopie.get() ); }
              );

std::thread t2( [fcopie]()   <-- 3
                { printf( "Thread 2 repond: %d\n", fcopie.get() ); }
              );
```

Listing 1.7: Instancier des Futures référençant le même résultat

Ajoutons qu'une instance de `std::future` n'est pas copiable[4]. Ainsi pour faire en sorte que deux threads consommateurs se synchronisent sur un même événement, la *copie* doit s'effectuer en déléguant d'abord la possession de l'état partagé de l'objet `std::future` à un objet `std::shared_future` ❶, puis en faisant autant de copies de l'objet `std::shared_future` que de threads consommateurs ❷ ❸. Notons qu'après délégation, l'appel de la méthode `get()` de l'objet `std::future` est considéré comme un *comportement indéfini*.

Chaque nouvelle copie de l'objet `std::shared_future` incrémente le nombre de possesseurs de l'état partagé. Ici, les threads consommateurs ayant leur propre instance de Future peuvent donc concurremment appeler la méthode `get()` sans que cela ne génère de *race conditions*.

1.4 Des threads aux tâches asynchrones

Dans la partie précédente, nous avons exposé les différents mécanismes présents dans la nouvelle norme C++ permettant de gérer les threads et avons conclu sur la synchronisation de threads par Futures et Promesses, outil puissant alliant la simplicité dans la sémantique et une gestion efficace et sécurisé des différents threads acteurs. Nous allons voir dans cette partie qu'il est plutôt rare qu'un utilisateur ait à manipuler des *Promesses*, étant donné que ce sont des objets instanciés très souvent en *arrière-plan* dans le cas du langage C++. Un point important à souligner que nous abordons maintenant, est que le schéma producteur/consommateur quand on se place uniquement du côté du consommateur revient à aborder le concept d'*asynchronisme*.

Dans notre cas nous définissons l'asynchronisme comme une manière alternative de faire appel à une fonction. Au lieu que la fonction soit exécutée directement, celle-ci est soumise sous forme de requête au runtime C++. La réponse à cette requête, c'est-à-dire le résultat de cette fonction, sera donc disponible plus tard et transmis via une Future. En attendant, le consommateur peut exécuter d'autres fonctions et

[4]La copie d'un objet non copiable échoue à la compilation

à tout moment récupérer le résultat de la requête. Voici un exemple de code montrant comment lancer un appel asynchrone de fonction avec la nouvelle norme C++:

```
std::string mot = "World!";
std::future<void> t
  = std::async([mot]()
               { std::cout << "Hello" << mot << std::endl; }
              );
/* ... */
t.get();
```

Listing 1.8: Appeler une fonction lambda de manière asynchrone

La fonction `std::async` exécute successivement les étapes suivantes:

- construit un couple Future/Promesse et l'état partagé associé
- délègue (potentiellement dans un nouveau thread) l'ensemble comprenant la Promesse, le pointeur de fonction et les paramètres de la fonction
- retourne la Future

Soulignons le fait qu'un appel de fonction via `std::async` n'implique pas forcément la création d'un nouveau thread. En effet, la fonction `std::async` peut éventuellement mettre l'état partagé dans un état spécial qui n'est ni **prêt**, ni **non prêt**. Cet état signale au premier consommateur qui aura appelé la méthode `get()` qu'une fonction *attachée* à la Future est en attente d'exécution : c'est donc ce consommateur qui va exécuter la fonction. Il y a deux raisons qui expliquent l'intégration de ce mode de fonctionnement. La première est de profiter de la sémantique des Futures pour forcer l'*évaluation paresseuse* de fonctions [5]. La seconde raison est qu'en cas de *sur-souscription*, *i.e.* situation dans laquelle il y a plus de threads prêts à être exécutés que de cœurs logiques, ou bien d'épuisement de threads, *i.e.* situation dans laquelle le nombre limite de threads est atteint, le runtime C++ puisse dynamiquement basculer d'un mode d'exécution asynchrone à un mode d'exécution séquentiel.

Pour résumer, les utilisateurs ont le choix ou de suivre le *modèle thread*, c'est-à-dire que la décomposition de leur programme sous forme de tâches concurrentes aboutit à une assignation **manuelle** de celles-ci aux threads (voir section 1.1), ou de suivre le *modèle asynchrone basé sur les tâches*, c'est-à-dire que la décomposition de leur programme sous forme de tâches concurrentes aboutit à une assignation **automatique** de celles-ci aux threads. L'avantage clé du modèle asynchrone est donc que l'utilisateur n'a plus à se soucier des problèmes de répartition de charge, de sur-souscription, et d'épuisement de threads.

[5] Pour cela l'utilisateur ajoute `std::launch::deferred` en premier paramètre de la fonction `std::async`

Néanmoins des problèmes persistent. Premièrement, la fonction `std::async` telle qu'elle est définie dans la norme C++ 2011 reste encore trop proche du modèle thread. En effet, chaque nouvelle tâche qui résulte d'un appel à `std::async` implique la création d'un nouveau thread système. Cela met de côté la possibilité d'exploiter efficacement le parallélisme de tâches à grain fin. Deuxièmement, le modèle asynchrone - encore une fois celui défini par la norme C++ - souffre d'un manque de composabilité. Voici un exemple illustrant ce propos:

```
std::shared_future< std::string > premier                    <-- ❶
  = std::async([]() -> std::string
               { return "Hello"; }
              );

std::future< std::string > second                            <-- ❷
  = std::async([premier]() -> std::string
               {
                 std::string r = premier.get();              } ❸
                 return r + " World!";
               }
              );

std::string resultat = second.get();
```

Listing 1.9: Séquencement de deux tâches avec la norme C++ 2011

Le problème ici est que les tâches associées aux variables Futures `premier` ❶ et `second` ❷ vont potentiellement s'exécuter en même temps, ce qui va inutilement mettre en sommeil le thread exécutant la deuxième fonction ❸. Pour remédier à cela, il faudrait d'une part qu'il y ait un moyen de spécifier le fait que certaines tâches peuvent *dépendre* d'autres tâches et d'autre part qu'il y ait un ordonnanceur de tâches qui puisse décider où et quand exécuter ces tâches lorsque leurs dépendances sont satisfaites.

On conclut cette partie par le fait que la norme C++ a bien répondu aux attentes des utilisateurs en ce qui concerne la gestion manuelle des threads mais reste encore incomplète du point de vue du modèle asynchrone basé sur les tâches. De nombreuses propositions ont été soumises au comité de la norme C++ et se rejoignent à peu près toutes sur l'idée qu'il faille remplacer le mode d'exécution «un thread - une tâche » par le mode d'exécution «*pool de threads* - file d'attente de tâches », mode d'exécution plus conventionnel qui est connu pour améliorer la concurrence et mieux exploiter le parallélisme de tâches à grain fin. Nous parlerons dans la suite plus longuement de ce mode d'exécution.

1.5 Concurrence par tâches légères

La décomposition d'applications en tâches à granularité fine a longtemps été cantonnée aux problèmes irréguliers[6] mais commence aujourd'hui à se généraliser au fur et à mesure de l'augmentation de la complexité des nouvelles architectures. En effet, pour permettre à n'importe quel système d'assurer une répartition de charge optimale et un ordonnancement dynamique efficace, le grain des tâches invocables dans un programme doit être d'une taille suffisamment faible. Nous rappelons dans cette partie le principe de fonctionnement du mode d'exécution *pool de threads - file d'attente de tâches* ainsi que la notion de tâches légères.

1.5.1 Gestion des files d'attente de tâches

Le terme «tâche » est souvent utilisé lorsque l'on veut désigner une simple séquence d'instructions. Dans cette partie, on définit une *tâche* comme un objet créé dynamiquement et encapsulant à la fois cette séquence d'instructions, une liste de paramètres et, éventuellement, un compteur de tâches *prédecesseurs* et une liste de tâches *successeurs* que l'on désigne plus souvent sous le terme de **continuations**. Quand une tâche est créé, celle-ci est généralement placée dans une file d'attente et ordonnancée dès qu'une ressource de calcul se libère. On parle alors de *tâche légère* quand les ressources de calcul ne sont plus des processeurs physiques mais bien des threads système préalablement placés dans un pool de threads. Cela implique d'implanter un ordonnanceur logiciel de tâches qui se rajoute à celui du système hôte.

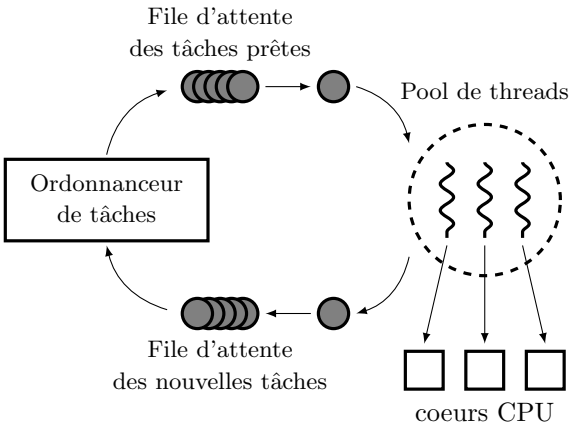


Figure 1.1: Schéma du mode d'exécution *pool de threads - file d'attente de tâches*

La Figure 1.1 illustre le mode d'exécution «pool de threads - file d'attente de tâches ». Lorsqu'une tâche s'exécute, celle-ci ayant été récupérée par un thread, il

[6]Problèmes intrinsèquement parallèles mais difficilement expressibles sous forme de nids de boucles

arrive en cours d'exécution que de nouvelles tâches se créent. Les nouvelles tâches sont donc concurremment placées dans une autre file spécialement dédiée. Le travail de l'ordonnanceur de tâches est donc de vérifier à la volée que les dépendances de chacune des nouvelles tâches sont satisfaites et de placer les tâches vérifiant cette condition dans la file d'attente principale. Il est aussi possible de décentraliser la file d'attente principale en attribuant une file d'attente à chaque thread. L'intérêt de cette approche est de permettre à l'ordonnanceur de tâches de distribuer équitablement les tâches – on parle alors de *load balancing* [75] – et la possibilité aux threads (en attente de tâches) de *dérober* les tâches dans les files d'attentes voisines – on parle alors de *work stealing* [18].

La raison principale expliquant l'engouement tardif pour ce mode d'exécution est que celui-ci a longtemps eu comme préjudice d'être peu efficace, car la contention globale dans un programme pouvait très vite dégénérer vu que les threads devaient continuellement verrouiller/déverrouiller des mutexes pour récupérer/déposer des tâches dans les files. Or, l'apparition des algorithmes *sans verrous* [56] (ou *lock-free*) et surtout du modèle de file *sans-verrou* de Michael et Scott [7] ont changé la donne. Il existe aujourd'hui pléthore de solutions logicielles basées sur les tâches légères. En comparant ces différentes solutions, on relève deux facteurs dinstinctifs :

- En premier, leur niveau d'*intrusivité*. Au niveau le plus faible, il y a les solutions basées sur des **bibliothèques** comme Intel TBB [62] ou HPX [51]. Au niveau le plus fort, il y a les solutions basées sur des **extensions de langage** comme OpenMP ou Intel Cilk Plus [1].
- En second, leur niveau de *flexibilité*. Nous verrons dans la suite que la flexibilité d'une API dépendra étroitement du *modèle de tâche légère* que ces solutions logicielles mettent en œuvre.

1.5.2 Modèles de tâches légères et graphes de dépendances

Comme nous l'avons vu précédemment, il existe un très grand nombre d'outils de programmation permettant de faciliter la mise en œuvre de tâches légères. Nous discutons dans cette partie des modèles de tâches légères qui prédominent aujourd'hui, puis nous nous intéresserons aux API permettant de les utiliser dans des applications.

1.5.2.1 Tâches non préemptibles, approche par Codelets

Le mode d'exécution par Codelets consiste à manipuler des tâches dont la séquence d'instructions peut contenir n'importe quelle instruction **sauf** celles susceptibles de préempter l'exécution en cours ou bien de générer une attente longue comme

[7] Dans ce modèle [57], l'accès conventionnel par verrou est remplacé par des opérations atomiques *Compare And Swap*. Cela diminue très fortement le coût des accès concurrents.

les opérations I/O. Une propriété fondamentale de ce mode d'exécution est qu'il assure l'*atomicité* des tâches, c'est-à-dire que lorsqu'une tâche occupe une ressource de calcul, la ressource n'est restituée qu'après terminaison de cette tâche. Nous illustrons en Figure 1.2 le cycle de vie d'un codelet.

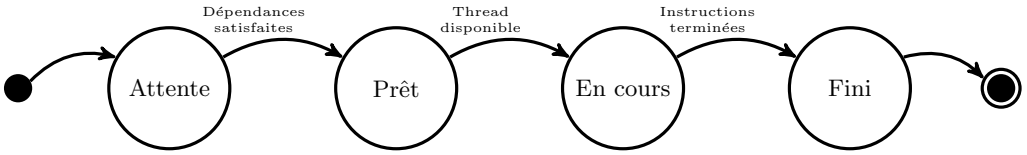


Figure 1.2: Cycle de vie d'un Codelet

L'intérêt de ne pas préempter les tâches est de limiter le nombre de changements de contexte car ils impliquent forcément un coût (sauvegarde de l'état du CPU, du contenu des registres, etc ...) et éviter au maximum que ces opérations ne *pollue* les caches CPU. En effet, pour des problèmes limités par la bande passante mémoire, il arrive souvent qu'une tâche ait besoin de travailler sur des données contigües en mémoire qui ont été chargées optimalement dans le cache grâce au mécanisme de *prefetch*. Si un changement de contexte devait survenir au milieu de l'exécution d'une tâche, le mécanisme de *remplacement des lignes de cache* ferait disparaître certaines données utiles à la tâche préemptée ce qui peut l'obliger, au retour de préemption, de recharger ces données depuis la mémoire.

L'approche par Codelets est celle qui est la plus utilisée, en particulier pour les solutions logicielles génériques dans lesquelles les tâches peuvent aussi bien être ordonnancées dans des threads CPU que dans des threads GPU. Les solutions qui l'implémentent sont OpenMP [61], OmpSs [10], Intel TBB [62], Intel Cilk Plus [1], StarPU [9], et SWARM [54].

1.5.2.2 Tâches préemptibles, approche par threads légers

Le mode d'exécution par threads légers est très semblable au mode d'exécution par Codelets, à ceci près que les tâches que l'on qualifie ici de *threads légers* sont préemptibles. Sur système conventionnel, la préemption survient généralement quand il y a sursouscription (cf. Section 1.4). Des plages de temps sont alors attribuées aux threads et lorsqu'une de ces plages est expirée, l'ordonnanceur effectue un changement de contexte. Fondamentalement, les threads légers fonctionnent comme des threads systèmes classiques. La différence est que généralement l'ordonnanceur n'a pas le pouvoir de préempter les threads légers. À la place, ce sont les threads eux-mêmes qui se préemptent : on parle alors de fonctionnement *multi-tâches coopératif*. Nous illustrons en Figure 1.3 le cycle de vie d'un thread léger.

L'intérêt de ce mode d'exécution est double : il s'agit d'une part de préempter *utilement* les tâches (ex : synchronisation sur événements ponctuels, appel à une

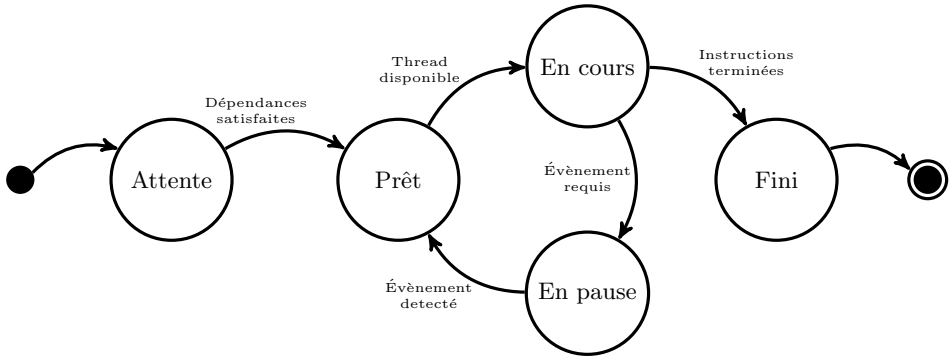


Figure 1.3: Cycle de vie d'un thread léger

fonction I/O, etc.) de sorte que d'autres tâches prêtes puissent être exécutées, et d'autre part de conserver le bénéfice du mode d'exécution par Codelets, c'est-à-dire de minimiser le nombre de changements de contexte. L'inconvénient, cependant, est que ce mode d'exécution implique de réimplanter des versions à granularité fine d'une grande partie des mécanismes classiques liés aux threads – les verrous et mutexes en premier – de sorte que la préemption des threads légers ne génère pas de *deadlocks*.

En effet, rendre des tâches préemptibles, c'est aussi leur donner la possibilité, au retour de préemption, de migrer d'un thread système à un autre or si une tâche attachée à un thread devait : prendre l'acquisition d'un verrou système, se préempter, et ensuite revenir de préemption en étant attachée à un autre thread, il serait impossible pour cette tâche de relâcher le verrou car le système aura identifié le premier thread comme étant son actuel possesseur. L'utilisation d'un verrou à granularité fine est donc nécessaire pour éviter ce problème. Bien que l'approche par threads légers soit plus difficile à mettre en œuvre, de nombreux projets implémentent cette approche. Parmi ces projets, on cite Microsoft PPL [52], QThreads [73] et HPX [51].

1.5.2.3 Graphes de dépendances sous OpenMP

OpenMP est l'API la plus utilisée pour concevoir des programmes qui mettent en œuvre des tâches légères. Le comité de standardisation d'OpenMP a néanmoins tardé pour permettre aux utilisateurs de les instancier explicitement. En effet, avant la version 3.0, celles-ci étaient invisibles vis à vis du programmeur et instanciées uniquement lors des décompositions de boucles. Ce manque a permis aux bibliothèques tels qu'Intel TBB [62] ou Intel Cilk Plus [1] de se présenter comme de réelles alternatives à OpenMP. Il reste cependant un facteur limitant : leur manque de composabilité. En effet, en utilisant l'une de ces deux solutions, il n'y a pas de moyens naturels qui permettent à l'utilisateur d'expliciter les dépendances entre tâches car celles-ci se limitent au seul mode d'invocation *fork-join*.

Or la prise en charge des dépendances revêt une importance capitale dans l'exploitation du parallélisme de tâches. Elle permet d'une part de préserver la sémantique du programme en indiquant à l'ordonnanceur quelles tâches peuvent s'exécuter en même temps, et permet d'autre part de mettre en œuvre des heuristiques pour un ordonnancement optimal des différentes tâches. C'est pourquoi, des runtimes comme OmpSs [10] ou SWARM [54] utilisent les graphes de dépendances (ou *Directed Acyclic Graphs*) comme élément charnière de leur API. La version 4.0 d'OpenMP s'est d'ailleurs largement inspirée du runtime OmpSs pour ajouter des clauses relatives aux dépendances de tâches. Voici la version du code producteur/consommateur avec OpenMP 4.0 :

```
#pragma omp parallel                                          ❶
#pragma omp single                                            ❷
{
   std::string message;

   #pragma omp task shared(message) depend(out: message)      ❸
   {
     message += "Hello";
   }

   #pragma omp task shared(message) depend(in: message)       ❹
   {
     message += " World!";
   }

   #pragma omp taskwait                                       ❺
}
```

Listing 1.10: Séquencement de deux tâches avec OpenMP 4.0

Les primitives `pragma omp parallel` ❶ et `pragma omp single` ❷ indiquent qu'un nouveau pool de threads est créé et que la suite ne sera exécutée que par un seul thread. Dans cet exemple, les blocs d'instructions précédés par la primitive `pragma omp task` ❸ ❹ constituent les tâches légères. Les clauses `depend(out:)` ❸ et `depend(in:)` ❹ permettent de spécifier qu'il y a dépendance de flot entre les deux tâches : le fait que la variable `message` apparaisse dans la clause `depend` de chacune des tâches force le séquencement des tâches dans l'ordre où elles ont été instanciées. Notons que les tâches OpenMP sont capables de récupérer des données dans leur portée : le mot-clé `shared` ❸ ❹ permet par exemple de lister les variables qu'une tâche va capturer par *référence*. La primitive `pragma omp taskwait` ❺ permet de placer une barrière de synchronisation : le thread courant ne passe pas à la suite tant que les tâches invoqués jusqu'ici ne se sont pas toutes terminées.

Bien qu'OpenMP ait réussi à combler son vide en rendant les tâches légères beaucoup plus accessibles qu'elles ne l'étaient autrefois, l'API reste basée sur des directives `pragma` parfois difficile à mettre en œuvre dans un contexte générique et souffre d'un problème de composabilité. Comme nous l'avons énoncé précédemment,

d'autres solutions existent et certaines d'entre elles s'adaptent plus naturellement au langage C++. Nous allons voir plus tard dans ce manuscrit (cf. chapitre 2) qu'il est possible en restant conforme au modèle Futures mis en place par la norme C++11 de relier le mécanisme des tâches légères à une sémantique fondamentalement asynchrone.

Chapitre 2

Expression Templates et Squelettes Algorithmiques orientés domaine

Sommaire

Dans [41], un langage orienté domaine mis sous la forme d'une bibliothèque C++ méta-programmée a été élaboré pour faciliter le portage haute performance de codes scientifiques vers les architectures multi-cœurs. Ce premier prototype appelé NT^2 a ensuite été étendu [38] pour capturer les spécificités des différentes architectures durant le processus de génération de code. Nous exposons brièvement dans ce chapitre les principales techniques utilisées pour garantir la performance. À la lumière des expériences qui ont suivi, nous exposons les limites rencontrées puis présentons l'outil HPX, un runtime mettant en œuvre des tâches légères qui se présente comme une réponse à ces limites.

2.1 Application des Expression Templates pour la gestion statique d'ASTs

Les Expression Templates [72, 71] sont une technique permettant l'implantation d'une forme d'évaluation paresseuse en C++ [65]. Cet idiome permet de construire, à la compilation, un type représentant l'AST d'une expression arbitraire. Le principe général consiste à abuser de la surcharge de fonctions et d'opérateurs afin de leur faire renvoyer un type paramétrique récursif qui encode une vue aplatie de l'AST qu'il génère. Une fois construit, ce type peut être manipulé afin de générer du code. Le champ d'application de cette technique est relativement large, allant

au delà de la simple optimisation de copies et des temporaires mais peu de projets l'utilisent de manière poussée car la conception et la maintenance de ce type de code est très complexe. Pour simplifier ce processus, Eric Niebler a proposé Boost Proto [59] - une bibliothèque C++ permettant de définir des Langages Orientées Domaines. Boost Proto permet aux développeurs de spécifier des grammaires et des actions sémantiques pour des DSELs et propose une génération semi-automatique de toutes les structures templates nécessaires à la capture de l'AST. Comparativement à des DSELs basés sur des Expression Templates écrits à la main, la construction d'un nouveau DSL avec Boost Proto est réalisée à un niveau d'abstraction supérieur en se basant sur la définition et l'utilisation de *transformations Proto*, *i.e.* fonctions opérant sur des expressions du DSEL par *reconnaissance de motifs* (ou *pattern matching*). NT^2 utilise Boost Proto comme générateur d'expressions templates et remplace la méthode conventionnelle de parcours statique des ASTs par l'exécution d'un algorithme de parcours mixte ayant lieu à la fois à la compilation et pendant l'exécution grâce à des structures standardisées pré-définies. La Figure 2.1 montre un exemple d'Expression Template et donne une illustration de l'AST qu'elle représente.

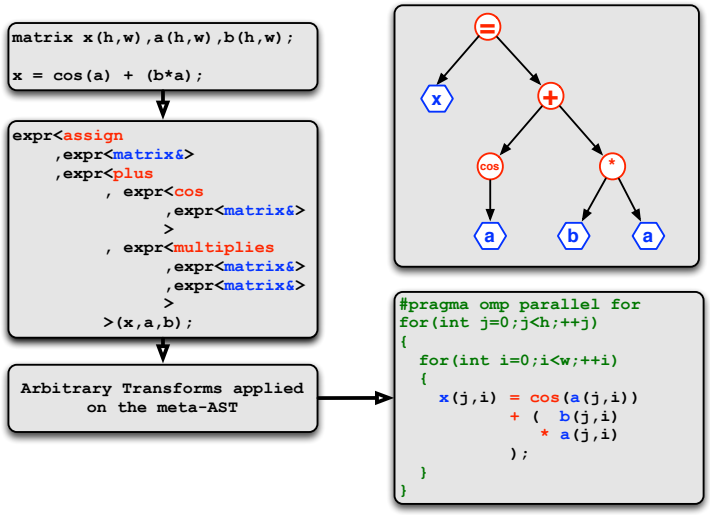


Figure 2.1: Principe général des *Expression Templates* – *La surcharge des opérateurs C++ permet de construire un type récursif encodant l'AST de l'expression original.*

2.2 Des ASTs aux squelettes algorithmiques

Les *squelettes algorithmiques* (ou *squelettes parallèles* [30]) sont des motifs de conception que l'on retrouve souvent en programmation parallèle. Ils se présentent généralement sous forme de fonctions d'ordre supérieur, *i.e.* des fonctions paramétrées par des fonctions et qui peuvent potentiellement retourner des fonctions. Cette composabilité réduit la complexité d'écriture des programmes parallèles car leur combi-

naison est viable par conception. Le deuxième avantage clé des squelettes est leur capacité d'enfouir dans leur structure tous les mécanismes liés à la synchronisation et à l'ordonnancement des tâches qu'ils impliquent. Ainsi lorsqu'une sémantique de squelettes est définie, les programmeurs n'ont plus à se préoccuper de ces différents mécanismes. Cela a deux influences: premièrement, il est possible d'utiliser les squelettes comme unités d'abstraction pour mettre en œuvre des programmes qui s'adaptent aux aspects architecturaux ; deuxièmement, leurs comportements face aux problématiques de calculs et de communications sont connus à l'avance et peuvent potentiellement être optimisés [4, 37] (pour plus de détails, voir chapitre 4).

Bien qu'un bon nombre de squelettes ait été proposés dans la littérature [53, 28], NT^2 se focalise sur trois squelettes orientés données :

- **transform** qui effectue une opération arbitraire à chaque (ou certains) élément(s) d'une table en entrée et qui range le résultat dans une table en sortie.
- **fold** qui effectue une réduction partielle des éléments d'une table en entrée le long d'une dimension donnée et qui range le résultat dans une table en sortie.
- **scan** qui effectue un balayage préfixe des éléments d'une table en entrée le long d'une dimension donnée et qui range le résultat dans une table en sortie.

Ces squelettes sont associés à des familles de nids de boucles plus ou moins imbriquables:

- Les **nids de boucles point par point** qui représentent les nids de boucles implantables via un appel au squelette **transform** et qui peuvent s'imbriquer dans d'autres **nids de boucles point par point**.
- Les **nids de boucles de réduction** qui représentent les nids de boucles implantables via un appel au squelette **fold**. Deux réductions consécutives ne peuvent pas s'imbriquer l'une dans l'autre car elles peuvent opérer sur deux dimensions différentes, mais peuvent chacune imbriquer des nids de boucles point par point.
- Les **nids de boucles de balayage préfixe** qui représentent les nids de boucles implantables via un appel au squelette **scan**. Deux balayages préfixes consécutifs, tout comme les réductions, ne peuvent pas s'imbriquer l'un dans l'autre, mais peuvent chacun imbriquer des nids de boucles point par point.

Ces familles de nids de boucles sont utilisées pour discriminer les fonctions fournies par NT^2 de sorte que le type représentant la partie opérative de l'AST puisse être évaluée et orientée vers une famille de nids de boucles. Les ASTs d'expressions comprenant au moins un nœud terminal de type `table` ou `_` sont évalués à la compilation. Il faut donc veiller lors de la construction d'un AST à séparer les parties qui

peuvent potentiellement se traduire en nids de boucles non-imbriquables. Pour cela, la recherche de l'appartenance d'un AST à une famille de nids de boucles démarre à partir de son nœud racine.

Par exemple, le nœud racine de l'expression `A=B/sum(C+D)` est l'opérateur `(=)`. L'AST entre donc dans la catégorie *nid de boucle point par point*. Par récursion, la même analyse est appliquée sur les sous-arbres du nœud `(=)` c'est-à-dire `A` et `B/sum(C+D)`, puis ainsi de suite. Durant ce parcours, l'AST `sum(C+D)` sera classé comme *nid de boucle de réduction* : une scission D'ASTs devra donc être réalisée car comme énoncé précédemment on ne peut pas imbriquer un nid de boucle de réduction dans un nid de boucle point par point.

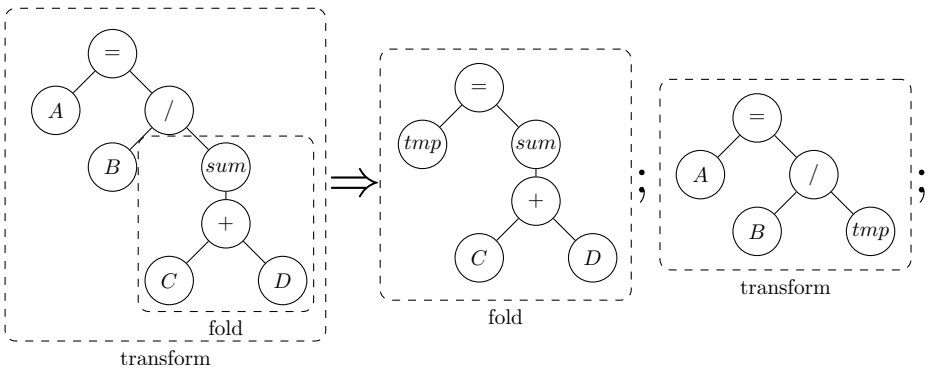


Figure 2.2: Processus d'extraction de squelettes parallèles – *L'imbrication de différents types de squelettes dans une expression est automatiquement déroulée pendant la compilation sous forme de suite d'expressions à squelette unique.*

Pour illustrer ce mécanisme de scission, nous montrons en Figure 2.2 comment l'expression `A=B/sum(C+D)` est construite et séparée en deux sous-ASTs chacun composé d'un seul type de squelettes. Ici les ASTs scindés sont chaînés de manière logique via l'ajout d'une variable temporaire insérée comme résultat du premier AST et comme opérande du second AST. La durée de vie de cette variable temporaire est garantie par la mise en œuvre d'un pointeur intelligent C++ pour faire en sorte que la donnée calculée à l'interface entre les deux sous-AST puisse vivre assez longtemps lors du passage d'un AST à l'autre.

Notons que, comme l'AST `C+D` se traduit par un nid de boucles point par point, celui-ci reste imbriqué dans le nœud `sum`. NT^2 utilise ensuite l'*imbriquabilité* des squelettes algorithmiques pour appeler la version SIMD ou séquentielle de chaque squelette impliqué dans une suite d'expressions afin d'exploiter les ressources machine de manière récursive et hiérarchique. À la fin de la compilation, chaque expression NT^2 est convertie en une propre série de nids de boucles imbriquant chacun une combinaison de codes vectoriels, OpenMP et séquentiels.

Comme exposé ci-dessus, une fois que les expressions NT^2 ont été émises soit directement par l'utilisateur, soit par l'intermédiaire des découpages d'ASTs, celles-ci sont exécutées en suivant le modèle simple fork-join disponibles dans OpenMP ou TBB. Au fur et à mesure que le nombre d'expressions augmente, le coût total combinant celui des synchronisations, des variables temporaires et des défauts de cache issus d'une mauvaise gestion de la localité peut limiter la performance. Pour résoudre ces problèmes, il est primordial d'exploiter le parallélisme de tâches. Nous avons vu en chapitre 1 que la meilleure façon d'exploiter ce type de parallélisme est d'utiliser le concept avancé des tâches légères. Nous présentons dans la suite un outil qui met en œuvre ce concept tout en offrant une API asynchrone conforme avec la nouvelle norme C++.

2.3 Une API asynchrone pour l'invocation des tâches légères

Nous avons vu en chapitre 1 deux manières de concevoir les tâches légères et présenté un exemple d'API basé sur les graphes de dépendances. Ce modèle d'API se combine naturellement avec le modèle Codelets mais comporte des lacunes pour synthétiser le modèle thread léger (manque d'un mécanisme explicite pour préempter les tâches). Nous étudions dans cette partie, un système qui répond à cette attente, tout en restant conforme aux fonctionnalités du langage C++.

2.3.1 HPX - Un runtime pour des applications parallèles et distribuées

HPX [51] est un runtime polyvalent mis sous la forme d'une bibliothèque C++ qui a pour vocation de faciliter le développement d'applications parallèles déployables sur tous types de plate-formes, qu'elles soient à mémoire partagée ou distribuée. HPX combine à la fois l'utilisation d'un espace d'adressage global, un mode d'exécution local par threads légers, une communication distribuée basée sur les *messages actifs*, et une sémantique pour les calculs distants quasi-identique à celle pour les calculs locaux.

Un point intéressant à souligner dans HPX est que son API s'aligne fortement sur la bibliothèque multi-thread définie par la norme C++11 (Voir chapitre 1). La différence majeure qui nous pousse à migrer vers le runtime HPX est que celui-ci éradique toutes les occurrences aux threads systèmes pour les remplacer par des threads légers ; c'est-à-dire que lorsqu'une opération implique de créer ou de préempter un thread, ce n'est pas un thread système que l'on manipule mais bien une tâche légère. Ici, le *modèle asynchrone basé sur les tâches* prend tout son sens, car d'une part les threads système deviennent invisibles à l'utilisateur – ce qui était déjà le cas grâce à la fonction `std::async` – et d'autre part parce qu'il nous permet

d'exploiter véritablement le parallélisme de tâches à grain fin.

Revenons maintenant au problème de composabilité évoqué en Section 1.4 pour la classe `std::future`. Nous avons vu qu'en utilisant les seules fonctionnalités définies par la norme C++11, l'exécution d'une séquence de tâches pouvait générer des attentes inutiles [1]. C'est pourquoi, de nombreuses propositions ont été soumises au comité de la norme C++ et ont abouti à l'introduction de mécanismes *sans attente* (ou *wait free*) dans la prochaine refonte de la norme C++ qui aura lieu en 2017. Ces mécanismes permettront entre autres à certaines tâches de n'être ordonnancées qu'au moment où une (ou plusieurs) tâches auront fini de s'exécuter : cela revient à répondre à la problématique de *gestion des dépendances* soulevée en Section 1.5.2.3. En regroupant une majeure partie de ces idées, HPX a étendu le modèle Futures en rajoutant un certain nombre de fonctionnalités supplémentaires, notamment celles permettant de composer *séquentiellement* et *parallèlement* les objets Futures.

2.3.2 Composition séquentielle des Futures

La *composition séquentielle* de Futures est le mécanisme de base dans HPX permettant d'exécuter dans l'ordre une séquence de tâches en parallèle avec d'autres tâches. Nous présentons dans le Listing 2.1 une version du code producteur/consommateur en utilisant la *composition séquentielle* de Futures.

```
using Fstring =  hpx::future< std::string >;          <-- 1

Fstring premier = hpx::async([]() -> std::string
                             { return "Hello"; }
                            );

Fstring second  = premier.then([](Fstring p) -> std::string
                               { return p.get() + " World!"; }
                              );                                  } 2

std::string resultat = second.get();
```

Listing 2.1: Séquencement de deux tâches avec HPX

Une des fonctionnalités intéressantes de la norme C++11 est de définir des *alias* de types, par l'intermédiaire du mot-clé `using`. Dans cet exemple, `Fstring` ❶ est un alias pour le type `hpx::future< std::string >`. La méthode `then()` ❷ de l'instance de Future `premier` permet **d'attacher dynamiquement une fonction** à celle-ci. Cette fonction sera convertie en tâche successeur ou *continuation* et ordonnancée dès que le résultat de `premier` sera disponible. L'appel de cette méthode renvoie en outre une nouvelle instance de Future correspondant au résultat de la continuation. Pour utiliser la méthode `then()`, une contrainte doit être respectée : la fonction donnée en paramètre de `then()` doit être une fonction devant accepter la

[1] Ce problème est comparable à celui soulevé pour les tâches légères dans Intel TBB et Intel Cilk Plus.

Future courante comme seul paramètre. En effet, dans un contexte de programmation asynchrone, mettre en œuvre des continuations revient à définir des *fonctions de rappel* (ou *callbacks*). Or par définition, toute fonction de rappel est capable de *traiter l'événement* qui est à l'origine de son lancement. Notons, dans cet exemple, que l'appel de la méthode `get()` dans la continuation ne sera pas bloquant ; à l'instant où cette ligne sera exécutée, `p` pointera sur le même résultat que `premier`, qui sera déjà disponible.

2.3.3 Composition parallèle des Futures

La *composition parallèle* de Futures est le mécanisme de base dans HPX permettant à une tâche de n'être ordonnancée qu'au moment où **toutes** ses dépendances sont satisfaites. Nous présentons dans le Listing 2.2 un exemple de code mettant en œuvre un graphe de dépendances avec des dépendances **statiques** qui utilise la composition parallèle de Futures.

```
using Fstring =  hpx::future< std::string >;
using Ftuple  =  hpx::future< std::tuple< Fstring,Fstring > > ;

Fstring premier = hpx::async([]() -> std::string
                             { return "Hello"; }
                            );

Fstring second  = hpx::async([]() -> std::string
                             { return " World!"; }
                            );


Fstring troisieme = hpx::when_all(premier,second).then(
                     [](Ftuple fjoin) -> std::string
                     {
                        std::tuple< Fstring,Fstring > join
                        = fjoin.get();

                        return std::get<0>(join).get()
                             + std::get<1>(join).get();
                     }
                    );

std::string resultat = troisieme.get();
```

Listing 2.2: Graphe de dépendances avec HPX - dépendances statiques

Dans cet exemple, la fonction `when_all()` ❶ est le cœur du mécanisme de composition parallèle. Cette fonction renvoie une nouvelle instance de Future qui ne sera prête que lorsque les instances de Futures passées en paramètres seront prêtes. Notons que la fonction `when_all` ne fait rien sauf renvoyer une Future pointant sur un tuple composé de ses paramètres. Dans le contexte des graphes de dépendances, nous pouvons voir qu'il est possible d'instancier un *nœud de jointure*, *i.e.* nœud représentant une tâche dépendante de plusieurs tâches, en utilisant la combinaison

`when_all().then()` ❶ comme montré dans cet exemple.

Pour des nœuds de jointure dont le nombre de dépendances n'est connu qu'à l'exécution, une surcharge de la fonction `when_all` a été prévue pour accepter un vecteur de Futures comme seul paramètre. Nous présentons dans le Listing 2.3 le code d'un graphe de dépendances avec des dépendances **dynamiques**.

```
using Fstring =  hpx::future< std::string >;
using Fvector =  hpx::future< std::vector< Fstring > > ;

std::vector< Fstring > deps;                                    <-- ❶

deps.push_back( hpx::async([]() -> std::string
                           { return "Hello"; }
                          )
              );                                                ❷

deps.push_back( hpx::async([]() -> std::string
                           { return " World!"; }
                          );
              );                                                ❸


Fstring troisieme = hpx::when_all(deps).then(
                     [](Fvector fjoin) -> std::string
                     {
                        std::vector< Fstring > join
                        = fjoin.get();

                        std::string resultat;
                        for(int i = 0; i<join.size(); i++)
                        { resultat += join[i].get(); }
                        return resultat;
                     }
                    );                                          ❹

std::string resultat = troisieme.get();
```

Listing 2.3: Graphe de dependances avec HPX - dependances dynamiques

Dans cet exemple, notons que la Future ❹ renvoyée par `when_all()` ne pointe plus sur un tuple de paramètres, mais simplement sur le vecteur de Futures ❶ qui lui a été fourni en paramètre. La différence importante par rapport à l'exemple précédent est qu'au lieu de créer les tâches prédécesseurs via des instanciations individuelles, celles-ci sont empilées dynamiquement ❷❸ dans un vecteur de Futures. Cela permet par exemple de faciliter le regroupement de dépendances notamment pour des problèmes décrits par des graphes de dépendances homogènes.

On conclut ce chapitre par le fait qu'HPX répond de manière tout à fait remarquable à tous les critères que l'on attend d'un système basé sur les tâches légères, tout en s'appuyant sur un API fondamentalement asynchrone et acceptée par la

communauté du langage C++. Pour ces raisons, nous utilisons cet outil comme base solide pour l'ensemble du travail décrit dans ce manuscrit.

Chapitre 3

Génération automatique de code concurrent pour un langage enfoui orienté domaine

Comme nous l'avons souligné dans le chapitre 2, des problèmes subsistent dans NT^2. Premièrement, NT^2 souffre d'un manque d'extensibilité qui implique de coder manuellement chaque nouvelle composante architecturale. Deuxièmement, il n'exploite que le parallélisme de données. L'exécution d'un algorithme contenant un nombre arbitraire d'opérations matricielles indépendantes pourrait potentiellement être améliorée si celles-ci pouvaient s'exécuter en même temps. **Notre proposition** dans ce chapitre consiste à remanier NT^2 pour prendre en compte le parallélisme de tâches dans des environnements à mémoire partagée et cela de manière générique grâce à des **squelettes parallèles asynchrones** multi-niveaux. Pour cela, nous mettons à profit le système automatique de découpage d'ASTs pour en déduire un graphe de dépendances inter-expressions construit dynamiquement grâce au mécanisme des Futures.

3.1 Intégration des Futures dans NT^2

L'intégration des Futures dans NT^2 est faite de la manière suivante:

- **Mettre en œuvre une implémentation générique des Futures**. Bien que NT^2 utilise HPX comme backend de choix pour le parallélisme de tâches, la plupart des systèmes ont tendance à plutôt choisir des runtimes comme OpenMP ou TBB. Ainsi, nous mettons en œuvre une classe template `Future` qui fera office de *wrapper* générique associant le choix courant du runtime à sa propre implémentation de tâches légères et aux fonctions associés.

- **Adaptation des squelettes pour le découpage en tâches légères**. Les squelettes NT^2 ont été modifiés de sorte que leur fonctionnement reposent à la fois sur les Futures et sur les appels asynchrones de fonctions. Pour ce faire, les squelettes NT^2 sont maintenant implantés sous forme de tâches légères via le modèle *worker/spawner*. Le `worker` est un objet fonction C++ qui prend en paramètre une paire de paramètres {début,fin}. Cette paire définit un intervalle (potentiellement sur plusieurs dimensions) sur lequel itérer et permet au *worker* d'effectuer des calculs *sans threads* préalablement optimisés.

Le `spawner` est une fonction *template* qui représente le squelette algorithmique : il invoque plusieurs *workers* en les encapsulant dans des tâches légères, et cela suivant un schéma spécifique correspondant au type de squelette déduit d'une expression NT^2.

- **Ajout d'un gestionnaire de dépendances dans NT^2.** La dernière phase de cette intégration repose sur le processus de chaînage des tâches asynchrones lancées par les squelettes extraits des ASTs. Cela est réalisé en utilisant les mécanismes de composition séquentielle et parallèle des Futures pour définir les différentes dépendances requises pour l'évaluation des expressions. Des pipelines de tâches sont alors créés à l'interface des AST temporaires pour préserver la sémantique du code NT^2 et résoudre les *aléas de données* (cf Section 3.2).

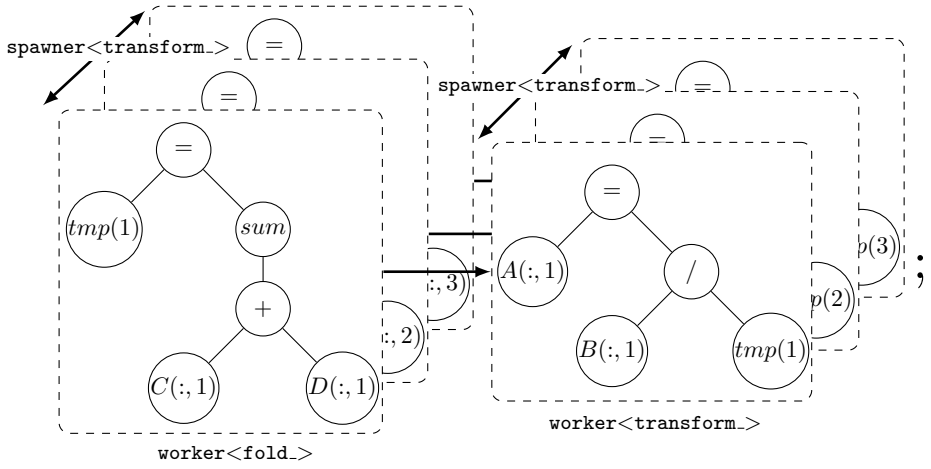


Figure 3.1: Découpage d'un AST sous forme de tâches légères. *Le déroulage précédent sous forme de suite d'expressions est augmentée par l'insertion d'un pipeline asynchrone entre les AST générés.*

La Figure 3.1 montre la traduction finale de l'expression `A=B/sum(C+D)` sous forme de composition de tâches. L'expression est parallélisée en utilisant à la fois le modèle *worker/spawner* pour profiter du parallélisme de données, et des pipelines placés à l'interface des appels de squelettes.

Le parallélisme d'instructions est maintenu en utilisant des *workers* qui tirent parti au maximum des instructions vectorielles afin de garantir le meilleur niveau de performance pour la couche *data-parallèle*. L'optimisation inter-expressions est **l'avantage principal** de notre système de génération de code concurrent. Il nous permet en effet de préserver *potentiellement* la localité des données et d'assurer ainsi un accès optimal à la mémoire. Ajoutons que de telles optimisations sont souvent difficiles à mettre en œuvre avec les Expressions Templates classiques car celles-ci ne peuvent accéder que statiquement à la structure des expressions NT^2.

3.2 Résolution des aléas de données

L'ordonnancement de l'ensemble des tâches invoquées dans un programme est guidé par deux paramètres importants : l'**ordre** dans lequel les expressions sont appelées, et la **liste des instances de tables** utilisées en lecture ou en écriture dans chacune des expressions. Ces paramètres sont ceux que nous utilisons pour résoudre dynamiquement les aléas de données définis par Bernstein [16]. Pour résoudre ces aléas, nous partons de l'hypothèse qu'une Future joue le rôle de **marqueur temporel** pour un bloc de données. Ainsi, chaque table NT^2 encapsule, en plus de ses données brutes, un tableau de Futures utilisées pour **marquer** des sous-morceaux de données brutes. Nous choisissons par défaut un tuilage avec le même grain pour chaque dimension de table. On présente dans le Listing 3.1 un fragment de code du *spawner* transform asynchrone sur 2 dimensions. Le but de cette fonction est de lancer de manière asynchrone une série de tâches asynchrones encapsulant chacune un **calcul sur un sous-morceau**. La variable `w` ❶ est un objet fonction encapsulant l'expression à évaluer (`in`), une référence à la table de sortie (`out`), et une surcharge de l'opérateur `()` dans lequel le calcul *sans threads* est effectué.

Dans l'ordre le spawner `transform` effectue les opérations suivantes :

- récupère le contexte `s` de la table de sortie ❷
- agrège dans une liste appartenant à la table de sortie l'adresse du contexte de chacune des tables d'entrée ❸ (*gestion des dépendances Read After Write*)

Puis pour chaque tuile de données, le spawner `transform`

- évalue la **zone** de la tuile de données à mettre à jour ❹
- agrège dans un vecteur temporaire de Futures les marqueurs des tuiles qui couvrent cette zone, et cela pour chacune des tables contenues dans la liste de `s` ❺
- lance une tâche *successeur* et met à jour le marqueur associé à la tuile considérée ❻

Enfin, le spawner `transform`

- réinitialise la liste de contextes de la table de sortie et y ajoute son contexte ❼ (*gestion des dépendances Write After Write*)
- ajoute l'adresse du contexte de la table de sortie dans la liste de contextes de chacune des tables d'entrée ❽ (*gestion des dépendances Write After Read*)

```
Worker & w = w_;                                                    ❶
auto s = details::get_specifics(w.out);                             ❷

int dummy;
details::get_cards()(w.in, dummy, s.calling_cards);                 ❸

for(int nn=0, n=0; nn<s.nblocks_col; ++nn, n+=s.grain)
{
   for(int mm=0, m=0; mm<s.nblocks_row; ++mm, m+=s.grain)
   {
      int chunk_m = (mm<s.nblocks_row-1)
                  ? s.grain
                  : s.last_chunk_row;

      int chunk_n = (nn<s.nblocks_col-1)                            ❹
                  ? s.grain
                  : s.last_chunk_col;

      std::tuple<int,int> offset = {m,n};
      std::tuple<int,int> chunk  = {chunk_m,chunk_n};

      std::vector< nt2::future<int> > deps;

      for(auto it = s.calling_cards.begin();
          it != s.calling_cards.end();
          ++it)
       {
          details::insert_dependencies( deps, **it,  offset, chunk);  ❺
       }

       if( deps.size() == 0 )
       {
         s.tile(mm,nn) = nt2::async<Arch>(Worker(w), offset, chunk);
       }

       else
       {
          s.tile(mm,nn)
          = nt2::when_all<Arch>(deps)                               ❻
            .then( [w,begin,chunk](auto)
                   {
                     w(offset,chunk);
                   }
          );
       }
   }
}

s.calling_cards.clear();
s.insert(&s);                                                       ❼

details::set_cards()(w.in, dummy, &s);                              ❽
```

Listing 3.1: Extrait du spawner `transform` asynchrone dans sa version 2D

Un point à souligner est que les étapes d'agrégation ❸ et de propagation ❽ des adresses de contextes implique d'analyser dynamiquement l'AST de l'expression à évaluer sans pour autant l'évaluer. Pour cela, un algorithme récursif de parcours d'AST a été réalisé et implanté sous forme d'une *grammaire Proto*. Une grammaire

Proto est un objet fonction qui accepte trois arguments : l'expression à analyser, un état initial pour la *variable d'état*[1] et une donnée auxiliaire qui est passée par référence. Dans notre cas de figure, seul le premier et le troisième argument nous intéressent. On présente ci-dessous la grammaire générique qui détecte la présence de nœuds terminaux de type *table* dans un AST. (Voir Listing 3.2)

```
template<class F>                                          ❶
struct aggregate_nodes
:boost::proto::or_<                                        ❷

 boost::proto::when<
    boost::proto::and_<
      boost::proto::terminal< boost::proto::_ >
     ,boost::proto::not_<
       boost::proto::if_<
          meta::is_container_or_ref< boost::proto::_value>()   ❸
       >
      >
    >
   ,boost::proto::_state
  >

 ,boost::proto::when<
    boost::proto::and_<
      boost::proto::terminal< boost::proto::_ >
     ,boost::proto::if_<
        meta::is_container_or_ref< boost::proto::_value>()     ❹
     >
    >
  ,F(boost::proto::_value,boost::proto::_data)
  >

 ,boost::proto::otherwise<
    boost::proto::fold<
      boost::proto::_
     ,boost::proto::_state                                     ❺
     ,aggregate_nodes<F>
    >
  >
>
{};
```

Listing 3.2: Détection de nœuds terminaux de type `table` d'un AST en utilisant Boost Proto

On remarque dans ce code, que l'algorithme de parcours est défini entièrement sous forme de *spécifications* de paramètres templates ; en effet, la grammaire Proto n'a besoin que du type de l'expression pour effectuer son introspection[2]. Sous certaines conditions grammaticales, une action sémantique est effectuée. La combinaison `boost::proto::or_` ... `boost::proto::when` ❷ est le cœur de l'algorithme de parcours. Elle permet de réaliser l'équivalent d'un switch case sur le nœud d'AST courant. En commençant par le nœud racine de l'AST, la grammaire effectue les actions suivantes :

[1] variable statique attribuée à la grammaire

[2] le type des expressions NT^2 est une représentation de leur AST

- **Si** le nœud courant est un nœud terminal et n'est pas une table, *retourner* la valeur courante de la variable d'état ❸
- **Sinon si**, le nœud courant est un nœud terminal et une table, *appeler* F en lui passant le nœud courant, et la donnée auxiliaire en paramètres ❹
- **Sinon**, appliquer la grammaire sur chacun des nœuds fils (dans l'ordre du plus à gauche au plus à droite) ❺

Une fois la grammaire définie, il faut lui imposer un comportement pendant l'exécution. Cela revient à définir le contenu du paramètre template `F` ❶ qui joue le rôle d'action sémantique. L'astuce de passer par un paramètre template permet d'utiliser le même code de grammaire pour l'étape d'agrégation des adresses et l'étape de propagation. Voici la définition des objets fonctions qui ont permis d'agréger et de propager les adresses de contexte dans le Listing 3.1 :

```
struct F_get_cards : boost::proto::callable
{
  typedef int result_type;

  template <class Container,class ProtoData>
  inline int operator()(Container & in, ProtoData & p) const
  {
    p.insert( & details::get_specifics(in) );
    return 0;
  }
};

struct F_set_cards : boost::proto::callable
{
  typedef int result_type;

  template <class Container,class ProtoData>
  inline int operator()(Container & in, ProtoData & p) const
  {
    details::get_specifics(in).calling_cards.insert(p);
    return 0;
  }
};

namespace details
{
  using get_cards = aggregate_nodes<F_get_cards>;
  using set_cards = aggregate_nodes<F_set_cards>;
}
```

Listing 3.3: Objets fonctions pour l'agrégation et la propagation d'adresses de contexte

3.3 Travaux associés

De nombreuses solutions ont été proposées pour résoudre les problèmes décrits dans ce chapitre. Nous présentons ci-dessous une liste de bibliothèques et de langages qui ont implanté ces solutions.

- **eSkel** [29] [15] (Edinburgh Skeleton Library) est une bibliothèque C basée sur l'outil de programmation MPI qui fournit à la fois un modèle de programmation et une API orientée squelettes parallèles. Comme dans NT^2, eSkel a étudié l'idiome du squelette *pipeline* afin de définir les interactions entre les tâches impliquées dans une séquence d'appels de squelettes. Le problème dans eSkel est son manque d'expressivité car son API implique que l'utilisateur soit relativement familier avec MPI.

- **Muesli** [28] (Muenster Skeleton Library) est une bibliothèque de templates C++ qui d'une part exploite les outils MPI et OpenMP et d'autre part met en œuvre des squelettes qui couvrent les deux types de parallélisme (parallélisme de données et parallélisme de tâches). La couche data-parallèle repose sur des structures de données distribuées (tableaux, matrices et matrices creuses) qui sont divisées en sous-morceaux, chaque sous-morceau étant attribué à un processus MPI. Actuellement, la différence avec NT^2 est que Muesli ne profite pas des fonctionnalités de gestion des tâches disponibles dans OpenMP vu que l'outil n'est utilisé que pour gérer la couche data-parallèle.

- **Muskel** [3] est un framework Java orienté squelette qui cible prioritairement les grappes et les grilles de postes de travail. La dernière version de Muskel fournit un environnement étendu pour les architectures multi-cœurs y compris une méthode pour gérer les *Macro Dataflows* (équivalent des graphes de tâches dans NT^2) en surcouche d'un runtime dédié. L'approche de Muskel qui consiste à transformer automatiquement les squelettes en graphes de tâches est semblable à celle employée dans NT^2.

- **STAPL** [7] (Standard Template Adaptive Parallel Library) est une bibliothèque C ++ basée sur des composants semblables à ceux proposées par la bibliothèque de la norme C++. STAPL offre des équivalents parallèles des conteneurs C++ (pContainers) et des algorithmes (pAlgorithms) en les faisant s'interagir via le concept d'*intervalles* (Prange). STAPL fournit des supports pour les architectures à mémoire partagée et à mémoire distribuée et comprend à la fois un runtime complet et des règles strictes pour garantir son extensibilité. STAPL utilise un système similaire aux Futures de HPX mais défini pour leur propre runtime.

- **SaC** [43] (Single Assignment C) est un langage de tableaux inspiré du langage C qui prend en charge les tableaux multidimensionnels en tant que classe d'objets de premier ordre. SaC intègre une parallélisation implicite basée sur le modèle thread en utilisant un compilateur spécifique. NT^2 présente des caractéristiques similaires, mais repose d'une part sur une parallélisation orientée tâches légères et d'autre part sur un DSEL qui ne dépend que du compilateur C++.

- **Chapel** [24] est un langage de programmation à part entière développé par Cray pour simplifier la programmation parallèle. Chapel fournit des abstrac-

tions pour le parallélisme de données sous la forme d'objets *arrays* qui sont l'équivalent des *tables* dans NT^2, et fournit des abstractions pour la parallélisme de tâches sous la forme de *variables de synchronisation* qui sont l'équivalent des Futures dans NT^2.

- **ELI** [26] est un langage de tableaux inspiré du langage APL qui utilise un compilateur ELI vers C pour générer du code C parallèle qui exploite l'outil OpenMP. La différence avec NT^2 est que le langage ELI a besoin d'un environnement extérieur pour le processus de génération de code alors que NT^2 ne se base que sur l'utilisation de bibliothèques et les caractéristiques du langage C++.

- **FastFlow** [5] est un framework C++ généraliste qui fournit un sous-ensemble de squelettes parallèles construits de manière hiérarchique en surcouche de leur propre système de tâches. La principale différence avec NT^2 est que l'ensemble des squelettes FastFlow sont centrés sur l'extensibilité, alors que NT^2 restreint son ensemble de squelettes dans le contexte de conception d'un DSEL pour le calcul scientifique.

- **PaRSEC** [19] (ou DAGuE) est un framework générique développé par l'ICL à l'Université du Tennessee qui peut être utilisé pour extraire des motifs *dataflow* à partir d'un code séquentiel C, générant une représentation sous forme de graphes de tâches pendant la compilation. PaRSEC utilise un runtime dédié pour instancier le graphe de tâches sous forme de tâches de calcul. La différence avec NT^2 est que, dans PaRSEC, l'utilisateur est responsable d'une grande partie de la chaîne de conception notamment de l'étape d'inspection des graphes de tâches générés.

Chapitre 4

Modèles de coût pour la génération de code

Sommaire

De nombreuses recherches dans la littérature proposent d'élever le niveau d'abstraction dans l'écriture de codes parallèles. Le thème des squelettes algorithmiques (voir chapitre 2) en est un exemple. Ce modèle a comme principal avantage de permettre aux programmeurs de composer leur code en assemblant des briques de base intrinsèquement parallèles sans se préoccuper des problèmes de communication et de synchronisation qui en découlent. Par conception, ces fonctions d'ordre supérieur facilitent la programmation parallèle. Cependant des problèmes persistent.

Les architectures devenant de plus en plus complexes, transformer de telles structures en tâches asynchrones au lieu de les appeler sous forme de fonctions (éventuellement *inlinées*) n'est pas nécessairement bénéfique. D'un côté, le fait d'ajouter davantage de tâches amplifie le *surcoût lié aux tâches*, *i.e.* le temps consommé pour mettre une tâche en file d'attente, l'assigner à un thread et mettre en sommeil ce thread une fois la tâche terminée. Ce surcoût, généralement faible, peut devenir problématique si son ordre de grandeur atteint celui du calcul utile. D'un autre côté, il reste primordial de maximiser le parallélisme de tâches à grain fin pour à la fois exploiter pleinement les ressources de calcul disponibles via l'asynchronisme et assurer une bonne répartition de charge.

4.1 Modèles analytiques de performance pour les multi-cœurs

Les recherches récentes sur les modèles de performance ont tendance à préciser continuellement la loi d'Amdahl pour l'adapter aux architectures multi-cœurs. Après avoir passé en revue ces modèles de performance, nous examinerons d'autres modèles de performance prenant en compte de manière plus précise l'impact des coûts de communication dans l'exécution des codes parallèles.

4.1.1 Analyses de performance basées sur les contraintes

La première formulation de l'accélération d'un programme parallèle fut émise par Amdahl [6]. En considérant n'importe quel programme à *charge de travail fixe*, en notant f le taux de calcul parallèle, $1-f$ le taux de calcul séquentiel et P le nombre de processeurs, l'accélération du parallélisme s'écrit :

$$\Gamma_A(f,P) = \frac{1}{(1-f)+\frac{f}{P}} \tag{4.1}$$

Cette formulation pessimiste suppose que l'accélération maximale atteignable est égale à $1/(1-f)$. Gustafson, pour sa part, a une vision plus optimiste [46] pour définir cette grandeur. Il suppose qu'une machine parallèle contenant P processeurs est toujours capable d'effectuer P unités de travail en *une durée fixe* correspondant à celui que met un processeur pour effectuer une unité de travail. L'accélération devient donc :

$$\Gamma_G(f,P) = (1-f)+P.f \tag{4.2}$$

XH Sun et Ni [67] ont admis un modèle similaire mais qui définit la charge de travail comme étant une fonction polynomiale $g(x)$ avec x la taille des données. En supposant que la taille des données augmente linéairement avec le nombre de processeurs (soit k le facteur d'échelle associé), l'accélération devient :

$$\Gamma_{SN}(f,P) = \frac{(1-f)+g(k.P)f}{(1-f)+\frac{g(k.P)f}{P}} \tag{4.3}$$

Ce modèle est bien adapté à l'évaluation de performance pour des programmes limités par la bande passante mémoire du fait qu'il tient compte scrupuleusement de la complexité arithmétique des calculs en concordance avec la taille des données. En général, on fait souvent référence à ces lois au moment de conjecturer sur le passage à l'échelle d'un programme à partir de ses résultats expérimentaux. On utilise alors la loi d'Amdahl pour le passage à l'échelle au sens strict, ou *strong scaling* et la loi de Gustafson pour le passage à l'échelle au sens large, ou *weak scaling*. Mais ces lois restent éloignées de la réalité étant donné que la seule métrique de ressources prise en compte est le nombre de processeurs. Hill et Marty ont émis la première extension de la loi d'Amdahl qui prenne en compte l'impact des architectures multi-cœurs [47]. En appelant P le budget global en *BCE* (ou *base core elements*), r le

nombre de BCEs par cœur de processeur et $perf(r)$ la performance par cœur de processeur, l'accélération pour une puce symétrique devient :

$$\Gamma_{HMS}(f, P, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f.r}{perf(r)P}} \tag{4.4}$$

Pour une puce asymétrique composée de $P - r$ cœurs *standards* et un seul cœur capable de délivrer une performance $perf(r)$, l'accélération devient :

$$\Gamma_{HMA}(f, P, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+P-r}} \tag{4.5}$$

Ces deux formulations confirment l'hypothèse d'Amdahl supposant que la part séquentielle des programmes limite l'accélération du parallélisme mais défendent aussi le fait qu'il est possible d'atteindre une meilleure accélération en mélangeant dans la même puce des cœurs de processeur complexes, plus aptes à exécuter du code séquentiel, avec des cœurs de processeurs simples, plus aptes à exécuter du code parallèle. Une troisième formulation a été émise pour des machines à fonctionnement dynamique, *i.e.* des machines pouvant mettre en sommeil un ou plusieurs cœurs de processeurs ou mettre en œuvre la *spéculation au niveau thread*, mais nous n'entrerons pas plus en détail sur ce sujet. Bien qu'Hill et Marty reconnaissent les faiblesses de leur modèle car il ne tient compte d'aucun surcoût lié au matériel, leur approche a ouvert de nouvelles pistes de réflexion et a permis d'intensifier la recherche ces dernières années autour des modèles de performance. [45, 66] Comme exemple, on cite le modèle introduit par XH Sun et Chen [66] qui prend en compte la contention dans les mémoires caches et l'acheminement des données à partir (ou vers) la mémoire principale comme étant des facteurs clés limitant la performance. En supposant que les accès aux données sont la principale contrainte affectant la scalabilité, les auteurs ont mis en corrélation la part séquentielle des codes avec le temps dépensé à accéder ou communiquer des données et la part parallèle des codes avec le temps dépensé à traiter ces données.

Gunther et al., pour leur part, sont allés plus loin en introduisant un modèle plus abstrait appelé *loi universelle de scalabilité* [45] qui s'inspire d'un modèle précis issu de la théorie des files d'attentes [44]. Ce modèle appelé *modèle du réparateur* introduit une chaîne abstraite d'assemblage composé de P machines opérantes qui vont chacune tomber en panne au bout d'un temps fixe connu à l'avance. Lorsqu'une machine tombe en panne, un réparateur vient la réparer en occupant un temps de service fixe. Par contre si plusieurs machines tombent simultanément en panne, celles-ci sont placées en file d'attente et ensuite réparées sur la base du *premier arrivé premier servi*. Pour faire l'analogie avec l'analyse de performances des codes parallèles, on considère les machines comme étant des processeurs (ou des threads), leur durée de vie comme étant leur temps respectif de calcul utile, et le nombre moyen de machines en marche par unité de temps comme étant l'accélération du parallélisme. Avec σ le coefficient de *contention*, et κ le coefficient de *cohérence*, l'accélération s'écrit :

$$\Gamma_{USL}(P) = \frac{P}{1 + \sigma(P-1) + \kappa P(P-1)} \tag{4.6}$$

La Loi universelle de scalabilité a l'avantage de quantifier la scalabilité de nombreux problèmes car l'accélération parallèle s'exprime sous forme d'une fonction rationnelle avec un polynôme caractéristique en puissance de P au dénominateur. Les termes associés à σ et κ dans ce polynôme ont en outre une certaine consistance physique du fait qu'ils représentent chacun un comportement concret que sont respectivement la baisse de performance lié au partage des ressources et la baisse de performance lié à une scalabilité rétrograde. La question à se poser cependant est de déterminer les différentes métriques de performance propre à l'architecture matérielle qui permettent d'établir ces coefficients. Pour faciliter leur identification, l'idée serait de construire un *modèle haut-niveau de machine* qui soit à la fois :

- Détaillé, pour accepter les spécificités des différentes machines existantes
- Abstraite, pour maintenir la portabilité des modèles de coût
- Réaliste, pour se prêter à une analyse de performance la plus consistante possible

La section suivante passera en revue ces modèles en question tout en restant focalisée sur la problématique des architectures multi-cœurs.

4.1.2 Coûts orientés temps d'exécution

Modéliser l'architecture implique de prendre en compte un certain nombre de ressources clés et de sélectionner leurs métriques correspondantes afin de refléter au mieux leur impact dans la performance parallèle. En pratique, deux paramètres influents sont considérés : la latence mémoire et la communication inter-processeur. Les programmes sont donc organisés en deux parties : une partie calcul qui se compose exclusivement du travail utile et une partie communication qui se compose du travail supplémentaire, effet secondaire résultant du parallélisme. Il existe dans la littérature un certain nombre de modèles de programmation parallèle qui acceptent ce postulat et qui définissent en conséquence des modèles de coût basés sur la connaissance de l'architecture et plus précisément de ses métriques de ressources. Les modèles les plus notables sont PRAM [42], LogP [31] et BSP [69]. Dans les pages suivantes, nous choisissons délibérément de limiter notre discussion à BSP. Il reste néanmoins possible de généraliser nos observations pour les autres modèles haut-niveau de machine cités.

Le modèle **Bulk Synchronous Parallel** (BSP) [69] a été défini par Valiant et al. comme une passerelle entre le matériel et le logiciel dont le but est de simplifier le développement d'algorithmes parallèles. Le modèle BSP se caractérise par un modèle haut-niveau de machine comprenant un certain nombre de processeurs

parallèles reliés via un réseau d'interconnexion. Cette machine est définie par trois métriques de ressource distinctes : P le nombre de processeurs, g l'inverse du débit de communication (en unités de temps par mot) et L le coût de synchronisation.

Du point de vue de la programmation parallèle, BSP contraint les développeurs à écrire leurs programmes sous forme d'une séquence de *super-étapes* comprenant chacune trois sous-étapes :

- *Calcul local*, effectué en concurrence par chaque processus
- *Communication*, effectué pair à pair pour échanger des données
- *Barrière de synchronisation*, qui contraint les processus à s'attendre mutuellement jusqu'à que tout le monde ait fini sa communication

Du point de vue des modèles de coût, BSP reconnaît les limites dues aux synchronisations des processus et assume que le fait de partager des données implique forcément des latences dues aux communications. À partir de cette considération, BSP associe un coût à chaque super-étape et définit le coût des programmes comme étant la somme totale des coûts des super-étapes. Avec ω_i le coût du calcul local dans le processus i et h_i le nombre de messages envoyés et reçus par le processus i, le coût d'une super-étape devient :

$$\Theta_{super-étape}(P, g, L) = max_{i=1}^{P}(\omega_i) + max_{i=1}^{P}(h_i.g) + L \tag{4.7}$$

Malgré sa précision et son acceptation par la communauté en programmation parallèle, le modèle BSP original reste limité pour deux raisons importantes. Premièrement, BSP est un modèle haut-niveau qui a été créé avant tout *en surcouche* des systèmes à mémoire distribuée et de la programmation par passage de messages. Toutefois, le passage vers les machines multi-cœurs implique d'examiner en détail de nouveaux problèmes qui découlent à la fois du niveau système et logiciel avec le multi-thread et le multi-tâche, et du niveau matériel avec la hiérarchie des caches mémoire. Deuxièmement, BSP contraint le programmeur à n'utiliser qu'un seul modèle de communication/synchronisation ; modèle qui peut parfois être difficile à conjuguer avec le parallélisme de tâches exprimé au plus haut niveau d'abstraction logicielle.

Pour résoudre ces problèmes, une première idée serait de réviser le modèle haut-niveau de machine pour mieux prendre en compte les spécificités des architectures multi-cœurs. Par exemple, Valiant a étendu le modèle BSP en introduisant le modèle Multi-BSP [70] qui définit à chaque niveau de mémoire (ou de cache), une taille de mémoire (ou de cache) m_i et un triplet distinct de métriques de ressources BSP P_i , g_i, L_i. Une autre idée serait de dépendre d'une passerelle conceptuelle secondaire qui fasse le lien entre le logiciel et la couche basée sur le modèle haut-niveau de machine. Les squelettes algorithmiques, par exemple, sont un concept particulièrement adapté pour cette approche. Nous passons en revue dans la section suivante les différents

outils et frameworks existants ayant implanté des modèles de coût spécifiques pour ce type de structures.

4.1.3 Des frameworks de squelettes orientés modèles de coût

L'idée de définir des modèles de coût pour les squelettes algorithmiques n'est pas récente : une première suggestion a été émise par Darlington et al. [32] pour le squelette *Divide and Conquer* quelques années après que Cole ait introduit le concept de squelettes [30]. Nous présentons ici une liste de frameworks orientés squelettes ayant définis de tels modèles.

Pisa Parallel Programming Language (P^3L) [11] est un langage de programmation parallèle qui construit automatiquement des squelettes à partir d'une sémantique proche du *dataflow* pour générer, via un compilateur spécifique, du code C parallèle déployable sur grilles de calcul. P^3L implémente des modèles de coût fondés sur BSP mais prenant en compte des coûts additionnels survenant du mode d'exécution fork-join.

Automatic Model Generation Tool (AMoGeT) [14] est un framework générant automatiquement des modèles de coût paramétriques pour squelettes à partir de leur formulation en langage PEPA. AMoGeT transforme chacune des expressions PEPA en mini-codes de simulation CMTC [1] qui vont servir à calculer numériquement les modèles de coût.

Orleans Skeleton Library (OSL) [49] est une bibliothèque C++ implémentant à la fois des squelettes méta-programmés et des conteneurs distribués pour un fonctionnement dans l'environnement MPI. La précision d'analyse des performances d'OSL est à peu près comparable à celle de BSP du fait qu'OSL élabore ses modèles de coût exclusivement à partir de modèles de coût BSP.

HWSkel [8] est une bibliothèque de squelettes orientée cluster de processeurs multi-cœurs et construite en surcouche au modèle de programmation hybride OpenMP + MPI. Au lieu de prédire des temps d'exécution, HWSkel définit un modèle de coût unique qui répartit efficacement le travail entre les nœuds d'un cluster. Ce modèle prend en compte pour chaque nœud : le nombre de cœurs CPU, la vitesse d'un cœur CPU et la taille du cache L2.

4.2 Modèles de coût pour la paramétrisation de squelettes

Comme expliqué précédemment, les modèles de coût sont des outils d'analyse que l'on utilise généralement pour conjecturer des performances obtenues expérimentalement. La précision d'un modèle de coût se juge alors par l'erreur numérique des valeurs numériques par rapport aux valeurs réelles. Dans cette partie, nous décrivons une approche duale où il s'agit d'implémenter un système de décision simple qui, en fonction d'estimations par modèles de coût, va sélectionner pendant l'exécution

[1] Chaîne de Markov à Temps Continu

les meilleurs versions de squelette impliquées dans chaque expression NT^2. Pour minimiser le surcoût de ce système, une part importante des modèles de coût est calculée à la compilation par méta-programmation C++. Nous jugeons cette solution viable car le calcul se fait essentiellement à partir d'opérandes disponibles avant exécution.

4.2.1 Métriques de ressource portables pour des squelettes réalistes

Pour faire la passerelle entre le logiciel et le matériel, le système de squelettes employé dans NT^2 a besoin d'un modèle haut-niveau de machine qui dissocie sans ambiguïté le coup du calcul utile et le coût des communications. Pour cette raison, nous partons du modèle BSP. Afin d'évaluer au mieux le comportement des squelettes utilisés dans NT^2 sur les systèmes à mémoire partagée, nous admettons les hypothèses suivantes :

- Les instances de squelettes peuvent être vues comme des super-étapes BSP
- Il y a autant de processus BSP que de threads systèmes qui s'exécutent, et autant de threads système qu'il y a de cœurs physiques
- Le nombre de messages envoyés et reçus par les processus BSP dans une instance de squelette est nulle
- Le coût de synchronisation, qui varie avec le type de squelette, combine le coût de la barrière de synchronisation et le surcoût lié aux tâches

Avec N_i le nombre de tâches attribués au processus i, ω le coût attribué aux opérations flottantes **par tâche** et γ le coût global attribué au chargement/rangement de données, le modèle de coût pour un squelette devient

$$\Theta_{\substack{Squelette \\ Multi-Tâches}}(P, N_1, \ldots, N_P) = (max_{i=1}^{P} N_i.\omega) + \gamma + L_{Squelette}(\sum_{i=1}^{P} N_i) \qquad (4.8)$$

Un point important à souligner est que pour chaque instance de squelette dans un programme NT^2, il est possible d'évaluer le nombre exact d'opérations flottantes et de chargements/rangements effectué dans une itération de boucle. Connaissant la taille des données, la partie calcul du coût devient parfaitement définissable. À ce stade, le modèle classique de BSP ne nous suffit plus car nous avons besoin de prendre en compte des facteurs clés des architectures multi-cœurs tels que la bande passante mémoire. Pour résoudre ce problème, nous choisissons d'étendre le modèle BSP en ajoutant deux métriques de ressource : la **bande passante mémoire maximale** b_{max} (à ne pas confondre avec la bande passante réseau qui vient avec le terme g) et la **performance crête** $Peak_{max}$. Quel que soit le squelette utilisé, les fragments de code obtenus contiennent au plus deux boucles imbriquées : une qui

itère sur des sous-morceaux de *table(s)* et une autre qui itère sur les éléments d'un sous-morceau. En appelant k_{outer} la taille de la première boucle et k_{inner} la taille de la seconde, le coût attribué aux opérations flottantes **par tâche** dans la partie calcul est :

$$\omega = \frac{k_{inner}.k_{outer}}{\sum_{i=1}^{P} N_i} \left(\frac{Flops_{par\ itération}}{Peak_{max}} \right) \tag{4.9}$$

Et le coût correspondant aux chargements/rangements dans la partie calcul est :

$$\gamma = k_{outer}.\left(k_{inner} \frac{Loads_{par\ itération}}{b_{max}} + \frac{Stores_{par\ itération}}{b_{max}} \right) \tag{4.10}$$

Pour notre prédiction de performance, nous devons comparer ce coût avec son équivalent séquentiel. Nous définissons ce temps comme le temps cumulé des fonctions empaquetés dans les tâches ci-dessus lorsqu'elles sont exécutées séquentiellement par un seul processeur. Sous cette condition, le coût de synchronisation devient nul et le coût du squelette séquentiel est (en théorie) égal à $(\sum_{i=1}^{P} N_i.\omega)+\gamma$. En pratique, lorsqu'un seul processeur est utilisé, les métriques de ressources changent selon la logique de *scalabilité matérielle*. Pour prendre en compte ce facteur, nous notons respectivement b_1 et $Peak_1$ la bande passante mémoire maximale et la performance crête d'un seul cœur CPU. Le modèle de coût associé aux squelettes séquentiels devient donc :

$$\begin{aligned} \Theta_{\substack{Squelette \\ Séquentiel}} = k_{outer}.\Bigg(k_{inner}.\Bigg(\frac{Flops_{par\ itération}}{Peak_{\ 1}} + \frac{Loads_{par\ itération}}{b_1} \Bigg) \\ + \frac{Stores_{par\ itération}}{b_1} \Bigg) \end{aligned} \tag{4.11}$$

À ce stade, la partie la plus difficile est d'affecter une valeur numérique à chacune des métriques citées ci-dessus. Le paramètre BSP P est obtenu en appelant une fonction utilitaire (ex : `omp_get_num_threads()` pour OpenMP) qui renvoie le nombre de threads lancés par le runtime. Pour la performance crête et la bande passante mémoire, leurs valeurs numériques sont trouvables de manière plus ou moins directe dans des notices techniques. En pratique, il est utile de passer par les formules suivantes :

$$Peak_{max} = Nombre\ de\ coeurs \times Flops\ par\ cycle \times Fréquence_{\ CPU} \tag{4.12}$$

$$\begin{aligned} b_{max} = Nombre\ de\ sockets \times Octets\ par\ canal \\ \times Nombre\ de\ canaux \times Fréquence_{Mémoire} \end{aligned} \tag{4.13}$$

Ces équations nous permettent de trouver par la même occasion les métriques des squelettes séquentiels : $Peak_1$ en faisant le rapport de $Peak_{max}$ sur le nombre de cœurs, et b_1 en faisant le rapport de b_{max} sur le produit { nombre de sockets $\times$ nombre de canaux }. Cependant, pour des raisons de portabilité et de précision suffisante, nous choisissons de déterminer ces valeurs par mise en œuvre de microbenchmarks. Nous utilisons donc les benchmarks Linpack [36] et STREAM [55] avec, d'une part, tous les cœurs CPU pour évaluer $Peak_{max}$ et b_{max} et avec, d'autre part, un seul cœur CPU (si possible, le plus performant) pour évaluer $Peak_1$ et b_1.

La dernière métrique de ressources à déterminer est le paramètre BSP L. Pour trouver cette valeur, certains aspects de la couche multi-tâches doivent être considérés. Selon le type de squelette (transform, fold, scan), et le mode d'exécution (master-worker ou fork-join), un type de barrière est choisi : les plus courants sont les barrières centralisées (*centralized barrier*), les barrières à tableaux (*dissemination barrier*) et les barrières arborescentes (*static tree barrier*). Avec N, le nombre de tâches issues de la décomposition de boucles, L peut autant être de complexité $O(N)$ que de complexité $O(log_2(N))$. Il est donc nécessaire d'assigner un coût de synchronisation qui soit propre à chaque type de squelette.

En raison du manque de certitude quant au type de barrière utilisé, nous choisissons de reprendre une approche par microbenchmarks. Nous nous inspirons donc des microbenchmarks EPCC OpenMP [21, 22] pour mesurer les temps de synchronisation. Pour ces tests, nous fixons le travail séquentiel assigné à chaque tâche, et invoquons autant de threads qu'il y a de cœurs physiques. La seule quantité qui varie est le nombre de tâches. Un détail à prendre en compte est la quantité de travail à assigner à chaque tâche. Celui-ci doit être suffisamment petit pour ne pas noyer la métrique L dans le bruit de mesure et suffisamment grand pour assurer une répartition de charge sur l'ensemble des processus BSP. En soustrayant la quantité $N.\omega/P$ aux temps mesurés, on peut tracer l'évolution du paramètre L en fonction de N. Un point à souligner est que les tests effectués pour chacun de squelettes (Voir Figures 4.1 et 4.2) dans différents environnements multi-tâches ont montré que, dans tous les cas, L varie linéairement avec N. Cela laisse entendre que le paramètre L a une complexité plus proche de $O(N)$ que de $O(log_2(N))$. Les deux raisons possibles sont les suivantes :

- la barrière utilisée est une barrière *centralisée* ou un dérivé de celle-ci.
- le surcoût lié aux tâches varie linéairement avec N et prend le dessus sur le coût de la barrière de synchronisation

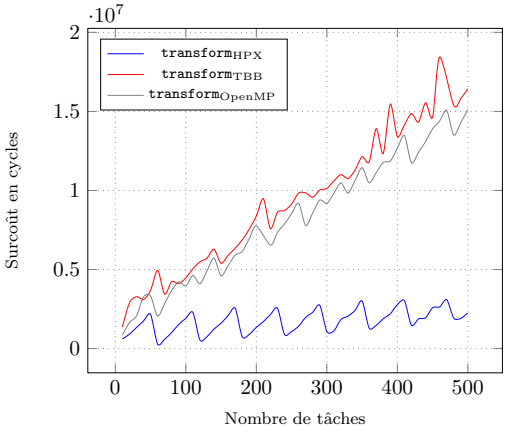


Figure 4.1: Coût de la barrière du squelette `transform` en fonction du nombre de tâches - Essai sur machine 2 × Intel Westmere (6 cœurs)

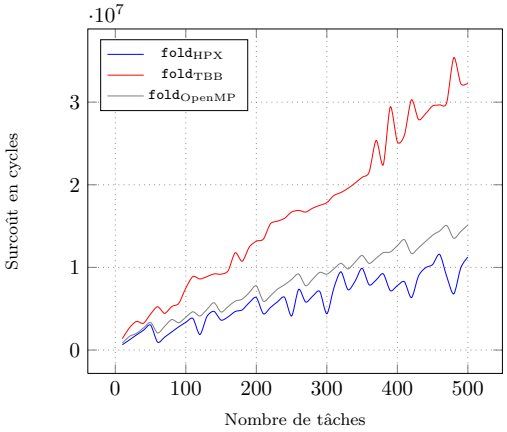


Figure 4.2: Coût de la barrière du squelette `fold` en fonction du nombre de tâches - Essai sur machine 2 × Intel Westmere (6 cœurs)

De ces observations, nous proposons donc une formule simpliste pour identifier L :

$$L_{Squelette}(N) = l_{Squelette}.N \tag{4.14}$$

Ici $l_{Squelette}$ est un facteur d'échelle caractéristique du squelette. Nous déterminons cette valeur par *régression linéaire* sur les échantillons obtenus.

Ayant défini notre modèle haut-niveau de machine, les métriques de ressources et présenté nos modèles de coût, nous expliquons dans la suite comment combiner la méta-programmation C++ et calculs entiers pour réaliser la sélection *à la volée* des meilleurs versions de squelette directement à partir des expressions NT^2.

4.2.2 Des arbres de syntaxe abstraits aux prédictions fondés sur la connaissance de l'application

Dans le chapitre 3, nous avons décrit une méthode basée sur les transformations Proto et les évaluations d'expressions pour assembler sous condition(s) grammaticale(s) des fragments de code parallèle. Cette section décrit comment il est possible de réutiliser ces techniques pour agréger des coûts numériques à partir de l'AST. En supposant que les métriques de ressources sont déjà disponibles, une première étape est d'extraire le nombre d'opérations flottantes et le nombre d'éléments à virgule flottante chargés/rangés afin d'évaluer le coût de la partie calcul (voir section 4.2.1). De là, nous admettons les trois relations suivantes :

- Nombre de nœuds terminaux entrées de type `table` = Nombre de chargements à virgule flottante par itération de la boucle interne
- Nombre de nœuds terminaux sorties de type `table` = Nombre de rangements à virgule flottante par itération de la boucle externe
- Coût cumulé des nœuds non terminaux = Nombre d'opérations flottantes par itération de la boucle interne

On peut voir en Figure 4.3 la traduction de l'expression $A = B + C * D$ en liste de coûts numériques. De cet exemple, on peut supposer que tous les coûts impliquées dans une itération de boucle sont déterminables à la compilation. L'hypothèse pour le nombre d'opérations flottantes est vraie tant que les opérations qui composent l'AST sont élémentaires. L'hypothèse pour les chargements/rangements l'est un peu moins. En effet, si une même *table* apparaît plusieurs fois dans un AST, on comptera plus de chargements/rangements qu'il n'y en a en réalité. Nous réglons ce problème en passant un conteneur associatif de type `set` comme donnée auxiliaire de la transformation Proto utilisée pour agréger les coûts de chargements/rangements. En utilisant cette méthode, nous nous assurons que chaque instance de *table* n'est comptée qu'une fois.

La deuxième étape consiste à déterminer le nombre de tâches pour lequel le coût du squelette parallèle doit être déterminé. La tendance en HPC est de laisser l'utilisateur choisir à l'exécution la taille de grain des tâches, ce qui revient à changer le nombre de tâches lorsque le problème est de taille fixe. Pour notre prédiction, nous choisissons consciencieusement de calculer le coût de squelette pour une seule taille de grain : la plus petite qui soit acceptable. La première raison est de placer le système de prédiction autour d'un point de fonctionnement qui élimine le déséquilibre de charge. La deuxième raison est de fournir une limite supérieure au temps d'exécution. Un bon compromis pour la taille de grain minimale est la taille de la ligne de cache L2. Le principal argument est que pour atteindre un minimum de scalabilité, la seule contrainte à respecter est d'éviter le faux-partage (ou *false sharing*), *i.e.* situation dans laquelle une même ligne de cache est modifiée contigüement par plusieurs threads en même temps. Dans cette situation, le surcoût lié au

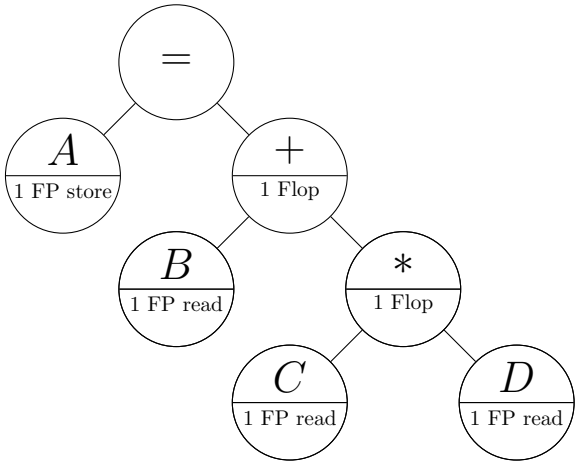


Figure 4.3: Extraction de coûts à partir d'un AST Proto

mécanisme de cache augmente considérablement. La **ligne de cache L2** est ainsi la toute dernière métrique de ressources que l'on ajoute à notre modèle haut-niveau de machine. À ce stade, on calcule le nombre de tâches N en utilisant la formule suivante :

$$N = \sum_{i=1}^{P} N_i = \frac{k_{inner}.k_{outer}}{taille\ de\ la\ ligne\ de\ cache\ L2} \tag{4.15}$$

En supposant que le système à mémoire partagée est symmétrique, *i.e.* chaque cœur CPU a la même caractéristique et se voit donc attribué le même nombre de tâches, le coût du squelette parallèle est :

$$\begin{aligned}\Theta_{\substack{Squelette \\ Multi-Tâches}} = k_{outer}.\Bigg(k_{inner}.\Bigg(&\frac{Flops_{par\ itération}}{Peak_{max}} + \frac{Loads_{par\ itération}}{b_{max}} \\ &+ \frac{l_{Squelette}}{taille\ de\ la\ ligne\ de\ cache\ L2}\Bigg) \\ &+ \frac{Stores_{par\ itération}}{b_{max}}\Bigg)\end{aligned} \tag{4.16}$$

Le coût de la partie calcul est donc déterminé pendant l'exécution en utilisant à la fois l'équation 4.11 pour la version séquentielle et l'équation 4.16 pour la version parallèle.

La troisième et dernière étape pour notre système de prédiction est l'étape de décision. Nous implémentons cette étape en effectuant la comparaison suivante :

$$\Theta_{\substack{Squelette \\ Multi-Tâches}} < \Theta_{\substack{Squelette \\ Séquentiel}}$$

Si le résultat est **vrai**, on sélectionne la version parallèle du squelette. Si le résultat est **faux**, on sélectionne la version séquentielle du squelette. Ajoutons que l'appel de la version séquentielle d'un squelette est précédée par une barrière de synchronisation appliquée sur tous les *marqueurs* (cf. chapitre 3) des tables identifiés comme nœuds terminaux de l'AST.

4.2.3 Avantages et inconvénients d'une approche par modèles de coût statiques

La méthode usuelle quand on décide de paralléliser un programme est d'inspecter le code séquentiel pour trouver les points chauds, puis de profiler le programme pour identifier les composantes les plus coûteuses, et enfin de sélectionner les fractions de code qui valident les deux premières étapes. Une fois la stratégie déterminée, une version parallèle du code est écrite. Le reste consiste en un processus de réglages pendant lequel la taille de grain des tâches et le nombre de processeurs actifs sont ajustés en effectuant une série de tests d'exécution dans le but d'atteindre les performances voulues. Dans la section 4.2.2, nous avons intégré des modèles de coût statiques pour décider automatiquement si des fragments de code expressibles sous forme d'instances de squelette devaient oui ou non être parallélisés. Cette approche a l'avantage d'une part de réduire le temps de conception d'un code parallèle et d'autre part de retirer l'étape d'inspection du code séquentiel.

Néanmoins, les modèles de coût dans NT^2 ne couvrent pas tout l'espace de recherche car ils ne prédisent le temps d'exécution que pour un seul couple {Nombre de threads, Taille de grain}. Une idée serait de réaliser des prédictions pour davantage de points dans l'espace de recherche, mais cette solution reste quelque peu limitée par le fait que nos modèles de coût sont calculés *en ligne* pour chaque instance de squelette dans un programme. Pour résoudre ce problème, une possibilité serait de laisser l'utilisateur avoir le contrôle sur la taille de grain des tâches, à la fois pour ne pas se restreindre à des prédictions sous-optimales et à la fois pour ouvrir les programmes résultants à une étape de réglage. Cette étape peut ainsi se faire soit manuellement – on parle alors de *hand-tuning*, soit automatiquement via des méthodes d'apprentissage automatique – on parle alors d'*auto-tuning*.

Comme premier exemple, on peut utiliser l'auto-tuning pour recalibrer des modèles de coût et s'en servir à l'exécution via un ordonnanceur orienté modèles de coût. Cette solution est celle proposée par Augonnet et al. [9] dans la bibliothèque StarPU. Comme deuxième exemple, on peut utiliser l'auto-tuning pour trouver la configuration optimale pour différents *back-ends* et ensuite générer un *plan d'exécution* à la fois pour guider la sélection de version pendant l'exécution, et le réglage *par back-end* de la taille de grain des tâches. Cette solution est celle proposée par Dastgeer et al. [33] dans la bibliothèque SkePU.

Dans NT^2, nous avons décidé de ne pas aller plus loin avec l'auto-tuning ; le sujet sera abordé ultérieurement pour une possible extension de NT^2 vers les systèmes multi-cœurs avec GPU. Néanmoins, notre contribution dans ce chapitre a été de montrer qu'il est possible d'une part de profiter de l'évaluation paresseuse des expressions et d'autre part de la connaissance des caractéristiques machine pour exécuter la meilleure version (séquentielle ou parallèle) d'un fragment de code expressible sous forme de squelettes.

Chapitre 5

Évaluation de performances

Sommaire

Avant de présenter nos benchmarks, rappelons que les algorithmes implantés dans ce chapitre sont issus du domaine du calcul scientifique et qu'ils comportent par conséquent de nombreux calculs manipulant des nombres en virgule flottante. Or il existe aujourd'hui un fossé significatif entre la vitesse des opérations flottantes et la vitesse de chargement/rangement des données nécessaires. Pour prendre en compte ces limites, Williams et al. [74] ont conçu le modèle *roofline* dans le but de donner une visualisation concise des critères de performance et de ses limites intrinsèques. Pour cela, le modèle se base sur l'analyse des goulets d'étranglements (performance crête, bande passante mémoire, ...) pour définir une borne supérieure de performance pour un algorithme en fonction de son *intensité opérationnelle*, *i.e.* le rapport entre le nombre total de flops et la quantité totale de données à traiter.

Dans ce contexte, nous classons les benchmarks en deux catégories qui sont les applications *limitées par la bande passante mémoire* que l'on définit comme des applications ayant une faible intensité opérationnelle et les applications *limitées par la performance crête* des machines que l'on définit comme les applications ayant une forte intensité opérationnelle. L'évaluation de la première classe d'applications (Black & Scholes, LBM, GMRES) est effectuée sur une machine de petite échelle – Mini-Titan – mais ayant de meilleures caractéristiques mémoire. L'évaluation de la seconde (LU tuilée) est effectuée sur une machine de plus grande échelle – Lyra – mais ayant de moins bonnes caractéristiques mémoire. Nous présentons dans le tableau 5.1 leurs principales caractéristiques.

Dans ce chapitre, nous avons ordonné les benchmarks du cas d'exploitation de l'asynchronisme le plus favorable au cas le moins favorable. Ainsi, nous présentons :

- **LU** car l'apport de l'asynchronisme y est évident. Notre implémentation doit donc être le plus proche possible de l'état de l'art.
- **Black & Scholes** car le code se compose d'une succession de squelettes purement data-parallèles. L'apport de l'asynchronisme se limite donc à augmenter la localité en traversant les nids de boucles. On s'attend ici à ce que les performances asynchrones soient similaires aux performances synchrones, démontrant ainsi le faible surcoût de notre système dans les cas limites.
- **Lattice Boltzmann** car il expose une structure proche des applications à stencils, problèmes notoirement connus pour être fortement limités par la bande passante mémoire. Ce benchmark vise à démontrer que notre système apporte néanmoins un gain sur ces algorithmes.
- **GMRES** car il se situe à l'autre extrémité du spectre car sa structure algorithmique et son caractère itératif limite l'impact de l'asynchronisme.

Nom de la machine	**Mini-Titan**	**Lyra**
Nom des processeurs	Intel Westmere	AMD Istanbul
Nombre de cœurs	2 × 6 cœurs	8 × 6 cœurs
Taille du cache de dernier niveau (L3)	12 Mo	5 Mo
Nombre de nœuds NUMA	2	8
Jeu d'instructions SIMD	SSE4.2	SSE4a
Type de mémoire	DDR3	DDR2
Nombre de canaux du contrôleur mémoire	3	2
Performance crête simple précision	210 GFlop/s	403 GFlop/s
Bande passante mémoire	24 Goctets/s	24 Goctets/s

Tableau 5.1: Caractéristiques des plate-formes de tests

5.1 Décomposition LU - version tuilée

La décomposition LU [35] est une méthode d'algèbre linéaire visant à mettre une matrice A d'ordre N sous la forme $A = P * L * U$ où L est une matrice triangulaire unitaire inférieure, U une matrice triangulaire supérieure et P une matrice de

permutation. La version de l'algorithme que l'on retrouve le plus souvent dans les bibliothèques d'algèbre linéaire est celle dite à *pivotement partiel*. Dans cette version, la sélection du pivot pendant les étapes d'élimination de Gauss n'est effectuée qu'à partir des éléments de la colonne sur laquelle l'élimination est appliquée.

Présentation de l'algorithme

Il arrive plus souvent que la factorisation LU soit faite *sur place*, c'est-à-dire que les données de A sont remplacées par celles qui constituent les matrices L et U. En outre, la parallélisation de cet algorithme se fonde sur une approche par blocs. Nous illustrons en Figure 5.1 le principe de cet algorithme.

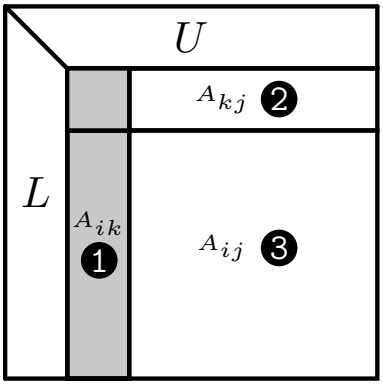


Figure 5.1: Schéma de principe de la décomposition LU par blocs

À l'itération k, l'algorithme effectue successivement les étapes suivantes:

- une factorisation LU **élémentaire** est effectuée sur un *panneau* ❶ de largeur B et délivre un vecteur de pivotement.
- le vecteur de pivotement est appliqué sur le reste de la matrice sous forme de permutations de lignes
- Une résolution triangulaire est effectuée sur les B premières lignes ❷ de la sous-matrice située à droite du panneau
- la multiplication de matrice $A_{i,j} \leftarrow A_{i,j} - A_{i,k} \times A_{k,j}$ est appliquée pour *mettre à jour* les $N - B$ lignes restantes ❸ de cette sous-matrice.

L'inconvénient de cette version est la présence d'une barrière de synchronisation entre chaque itération de l'algorithme. Pour contourner ce problème, une *version tuilée* [23] de la décomposition LU a été proposée pour supprimer ces barrières. Cette approche qui est le cœur des algorithmes implantés dans la bibliothèque PLASMA [2] (The Parallel Linear Algebra for Multicore Architectures) consiste à décomposer

l'algorithme LU en unités de travail avec dépendances qui opèrent chacune sur des portions carrés d'une matrice. Cela conduit naturellement à un graphe de dépendances qui, une fois couplé à un runtime basé sur des tâches asynchrones, peut apporter de meilleures performances. Nous illustrons en Figure 5.2 le principe de cet algorithme.

Dans la figure 5.2, nous remarquons deux niveaux de pipeline ; le pipeline intra-itération dans laquelle des tuiles de données peuvent être mises à jour potentiellement en parallèle et le pipeline inter-itération permettant de ne pas attendre que l'intégralité de la matrice soit mise à jour pour démarrer les mises à jour aux itérations suivantes. Soulignons le fait que pour rendre possible cette décomposition en tâches asynchrones, la stratégie de pivotement de l'algorithme LU a été sensiblement modifiée. L'étape de factorisation du panneau, est remplacée par une séquence de factorisations par **paire de tuiles**. Ce changement peut conduire à une baisse de la stabilité de l'algorithme du fait que le pivot n'est sélectionné que sur une fraction de colonne durant l'étape d'élimination de Gauss. En pratique, l'impact de cette baisse de stabilité est considéré comme mineur.

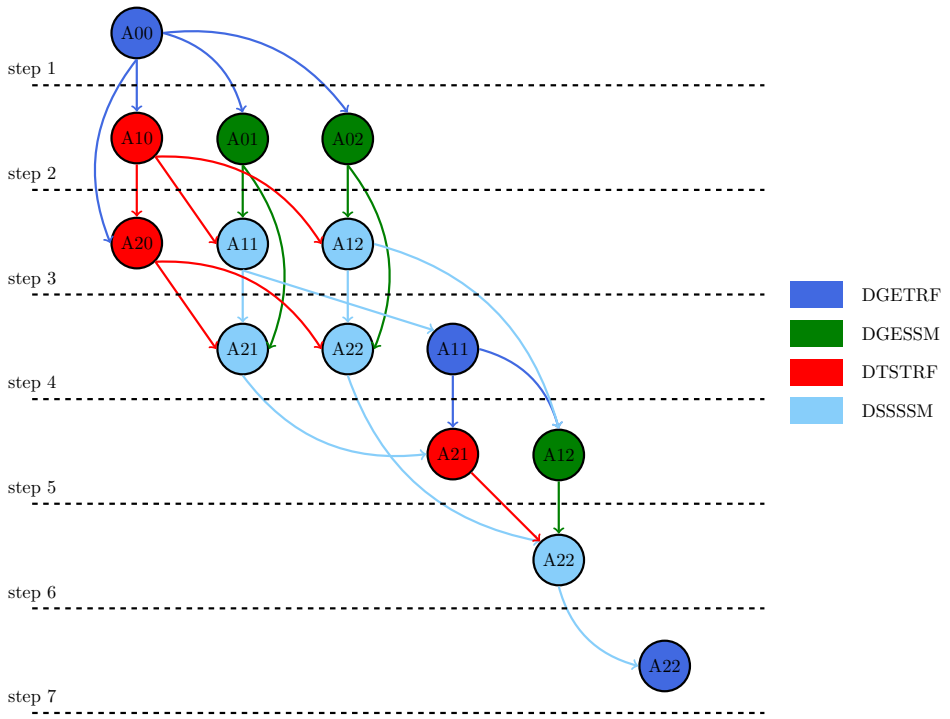


Figure 5.2: Schéma de principe de la décomposition LU tuilée

Résultats

Pour notre évaluation de performances, la version tuilée de la factorisation LU est implantée avec des tables NT^2 d'éléments à virgule flottante double précision via le système de gestion de dépendances par Futures que l'on a décrit dans le Chapitre 3. Nous comparons cette implémentation d'une part avec celle fournie par la bibliothèque PLASMA et d'autre part avec la version par blocs fournie dans la bibliothèque Intel MKL.

Les Figures 5.3, 5.4, 5.5 montrent que la version tuilée de l'algorithme prend progressivement le dessus sur la version classique par blocs. Étant donné que la machine Lyra est composée de 8 nœuds NUMA comprenant chacun 6 cœurs de processeurs, la performance de la version MKL chute autour de 20 cœurs. Cette chute s'explique par le fait que **plus** de 3 nœuds NUMA sont utilisés pour effectuer les transactions mémoire. Par conception, la version tuilée de l'algorithme LU intègre de l'asynchronisme pour *couvrir* ces transactions mémoire par le calcul utile, d'où le gain de performance.

Notons d'ailleurs qu'en augmentant le nombre de cœurs, la performance de PLASMA reste meilleure que celle de NT^2. En terme de temps d'exécution brute, lorsqu'on se retrouve en dessous d'un certain seuil de temps et bien que l'algorithme LU ait une intensité opérationnelle suffisante, les temps d'accès mémoire commencent à reprendre le dessus sur le calcul. Contrairement à PLASMA qui se base sur un *réarrangement* des données[1] en plus du mode d'exécution asynchrone, NT^2 conserve l'arrangement *Column Major* définie par Lapack afin de ne pas interférer sur les calculs appelant nos squelettes. Ainsi, notre version comptera plus de défauts de cache (un par colonne accédée dans une tuile) que celle de PLASMA.

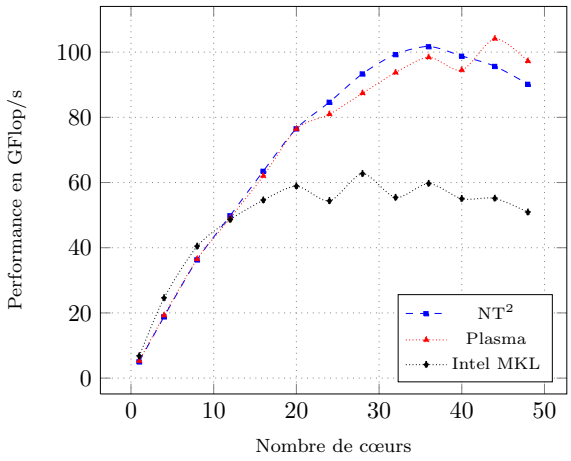


Figure 5.3: Performance de LU pour une matrice d'ordre 4000

[1] les éléments dans chaque tuile de données sont placées contigüement en mémoire.

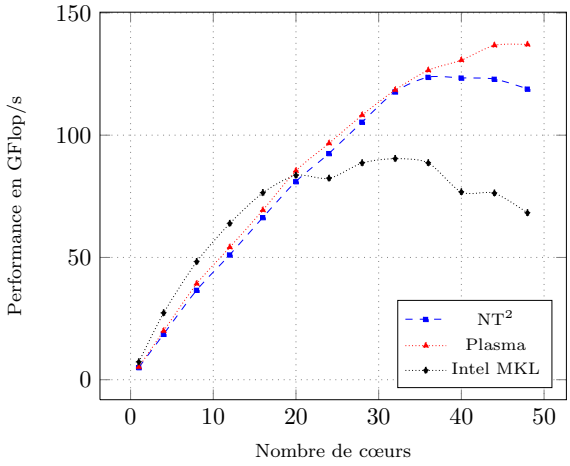


Figure 5.4: Performance de LU pour une matrice d'ordre 8000

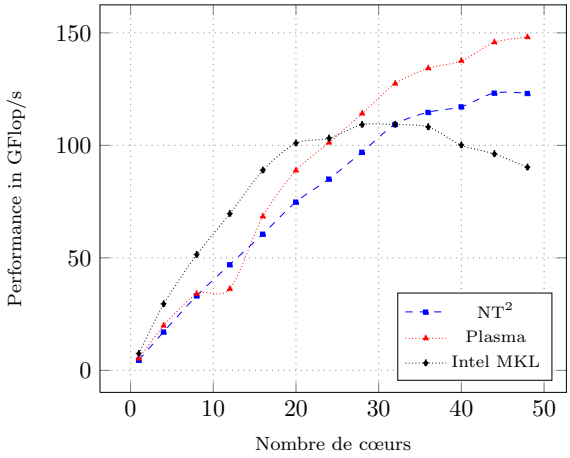


Figure 5.5: Performance de LU pour une matrice d'ordre 12000

Points à retenir

- La factorisation LU à une intensité opérationnelle suffisante qui permet d'exploiter l'asynchronisme sans se préoccuper des problèmes de localité des données
- HPX peut fournir des performances équivalentes à des runtimes de tâches asynchrones reconnues. On prouve par ce benchmark qu'HPX est effectivement une base solide pour le travail d'exploration de l'asynchronisme.

5.2 Black & Scholes

L'algorithme Black & Scholes [17] représente un modèle mathématique capable de donner une estimation théorique du prix d'une option européenne. Le code NT^2 est défini dans le Listing 5.1. L'algorithme Black & Scholes implique à la fois une haute latence et l'utilisation d'un grand nombre de registres. La version SIMD des opérations `log`, `exp` et `normcdf` utilise des polynômes et des étapes de raffinement de précision qui consomment une grande quantité de registres. En outre, cette implémentation invoque un certain nombre d'expressions dans lesquelles la localité des données joue un rôle majeur dans la performance.

```
table<float> blackscholes ( table<float> const& S, table<float> const& X
                          , table<float> const& T, table<float> const& r
                          , table<float> const& v
                          )
{
  table<float> d  = sqrt(T);
  table<float> d1 = log(S/X)+(fma(sqr(v),0.5f,r)*T)/(v*d);
  table<float> d2 = fma(-v,d,d1));

  return S*normcdf(d1) - X*exp(-r*T)*normcdf(d2);
}
```

Listing 5.1: Implémentation NT^2 du code Black & Scholes

L'algorithme Black & Scholes se présentant sous la forme d'une séquence d'appels au squelette *transform*, une taille minimale de table est requise pour obtenir une certaine performance. Ce benchmark est donc évaluée pour des tables de nombres en virgule flottante simple précision de taille variant de 64 millions à 1024 millions éléments. La performance *par élément* variant lentement avec la taille du problème (de l'ordre d'un cycle par élément), nous utilisons les *cycles par élément* comme unités de mesure et ne conservons que la valeur médiane des résultats obtenus pour une implémentation donnée.

La Figure 5.6 montre que l'implémentation NT^2 de l'algorithme est meilleure que la version Boost SIMD avec un gain de performance de **4**. Étant donné que l'on utilise tous les cœurs de processeurs (12 cœurs sur Mini-Titan), l'accélération idéale devrait être égale à 12 lorsque les latences dues aux synchronisations et aux communications sont ignorées. Cela montre qu'il y un réel manque de scalabilité dû à la présence des barrières implicites entre chaque expression.

Nous intégrons ensuite l'optimisation consistant à ajouter des pipelines inter-expressions et comparons cette nouvelle version avec celle qui conserve les barrières. La Figure 5.7 montre que la version avec barrières reste meilleure que la version avec pipelines mais que sa scalabilité a une progression **faible** lorsque la taille de grain des tâches est augmentée. La version avec pipelines n'est pas optimale pour de très petites tailles de grains mais progresse relativement bien lorsque celles-ci changent. Étant donné qu'HPX utilise la règle du *premier arrivé premier servi* comme politique d'ordonnancement par défaut, la localité des données n'est pas

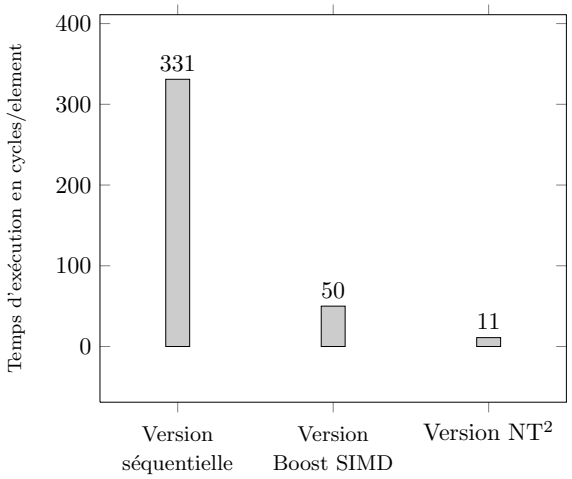


Figure 5.6: Comparaison des temps d'exécution du code Black & Scholes

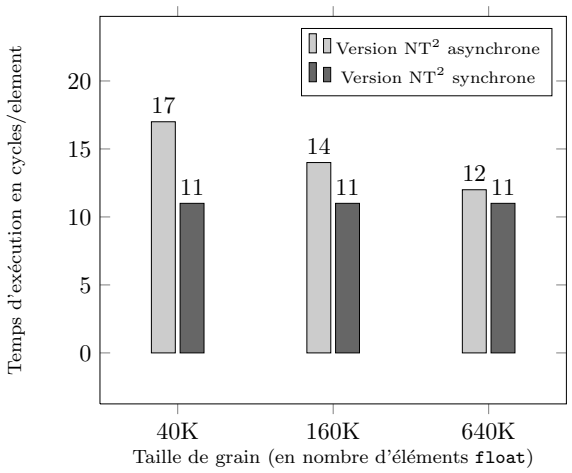


Figure 5.7: Influence de la taille de grain pour le code Black & Scholes NT^2

préservée lorsque des squelettes *transform* successifs sont reliés par pipeline.

Ainsi, l'invocation des tâches successeurs ne dépendant que d'une seule tâche est optimisée. Pour cela, nous intégrons les conditions suivantes:

- La Future est *non prête* - un callback est attaché à la Future, de sorte que le thread en charge de résoudre la Future exécute cette continuation.
- La Future est *prête* - Le thread en train d'instancier la continuation l'exécute immédiatement.

Bien que l'intégration de cette condition fonctionne en théorie, les problèmes soulignés précédemment demeurent. En effet, l'exécution d'une tâche peut avoir fini

avant que la continuation ne soit instanciée. Dans cette situation, le thread ayant exécuté cette tâche aura déjà récupéré une nouvelle tâche dans sa file d'attente. La continuation va donc être exécutée par un mauvais thread, ce qui va entraîner une perte de localité mémoire. Pour conserver cette localité, la complexité du calcul encapsulée dans la tâche (ex: produit Matrice-Vecteur) doit être suffisamment grande pour assurer un ordonnancement optimal.

Points à retenir

- L'algorithme Black & Scholes est une séquence de nids de boucles pouvant potentiellement être fusionnés pour améliorer la performance.
- La localité des données ne peut pas être préservée par les pipelines inter-expressions car la résolution des objets Futures se fait plus souvent avant instanciation de leur continuation
- En choisissant bien la taille de grain, il n'y a pas de surcoût par rapport à une approche purement data-parallèle

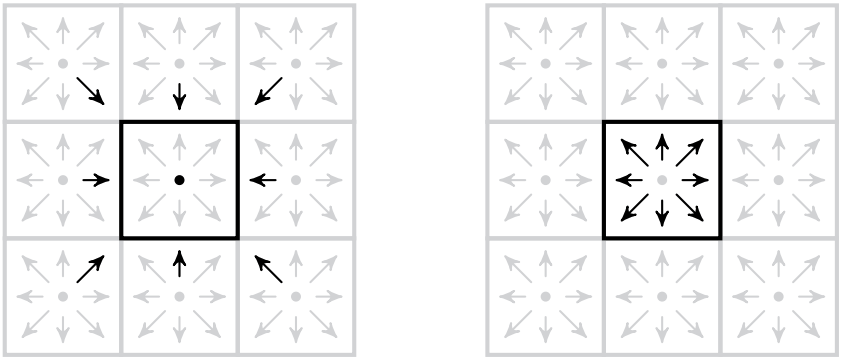
5.3 Lattice Boltzmann - version D2Q9

La méthode de Lattice Boltzmann [20] est une méthode de calcul utilisée en dynamique des fluides pour simuler des problèmes d'écoulements de fluides complexes. La méthode met en œuvre un maillage de dimension n dans lequel chaque particule se voit attribuée des fonctions de distribution calculées en temps discret. À chaque pas de temps, et pour chaque point du maillage, on effectue un calcul qui prend en compte la vitesse (sur plusieurs directions) de ce point, celle de ses voisins en vis à vis, et celle de ses voisins diagonaux. On appelle *problème DnQm*, un problème appliqué à un maillage de n dimensions et dans lequel chaque point se voit attribué m composantes de vitesse. Pour notre évaluation, nous choisissions la version D2Q9.

Présentation de l'algorithme

À un pas de temps donné, le calcul pour chaque point se décompose en deux phases : la *phase de propagation* et la *phase de collision*. La phase de propagation consiste à mémoriser la valeur d'**une** composante de vitesse de chacun des 8 points voisins, plus la valeur d'une des composantes du point courant. La phase de collision consiste à confronter ces valeurs avec les 8 valeurs restantes du point courant et ainsi mettre à jour les valeurs de vitesse correspondant à ce point. Plusieurs approches existent pour implémenter cet algorithme. La version la plus naïve est celle à *2 maillages - 2 phases* qui range les valeurs de vitesse post-propagation dans un autre maillage intermédiaire, puis qui écrase les valeurs du maillage d'origine avec les valeurs post-collision. Une deuxième version est celle à *2 maillages - 1 phase* dans laquelle les phases de propagation et de collision sont fusionnées, et dans laquelle, à chaque

pas de temps, l'adresse du maillage d'origine et celle contenant les valeurs post-collision sont permutées. Cette version permet de réduire sensiblement le temps total d'accès aux données car les écritures ne sont effectuées que sur un maillage à la fois (Voir Figure 5.8). Pour notre évaluation de performances, nous implémentons cette version.



(a) Phase de Propagation (b) Phase de Collision

Légendes
: Élément à lire
: Position du point à mettre à jour

Figure 5.8: Schéma de principe de l'algorithme Lattice Boltzman D2Q9

À ce stade, il s'agit de savoir comment ranger en mémoire les données caractéristiques du maillage. La méthode naïve est de placer contigüement en mémoire les 9 composantes de chaque point : cela revient à manipuler des tableaux de structures (ou *AOS* comme *Array of Structures*). Les principaux atouts de cette disposition de données est, d'une part d'avoir une représentation qui colle parfaitement au modèle *ensemble de points*, et d'autre part, de rapprocher au mieux les données nécessaires pour la phase de collision. La méthode moins naïve pour ranger les données caractéristiques du maillage, est de regrouper les valeurs par numéro de composante : cela revient à remplacer le tableau de structures, par une structure de tableaux (ou *SOA* comme *Structure of Arrays*). Les avantages de cette disposition, est cette fois de tirer parti des instructions vectorielles SIMD, les briques élémentaires n'étant plus des structures mais bien des nombres flottants, tout en profitant des instructions de rangement *non-temporelles*[2], pour éviter de polluer les chargements effectués pendant la phase de propagation. Nous étudierons en détail dans la suite l'impact du choix de l'une de ces deux méthodes.

[2]Les instructions *non-temporelles*, disponible par exemple dans les extensions SSE, permettent de ranger des données directement dans la mémoire en contournant les mémoires cache

Taille du maillage	version séquentielle	version NT^2 avec boucles (SOA)	version NT^2 avec boucles (AOS)	version NT^2 avec l'opérateur '_' sans threads (SOA)	version NT^2 avec l'opérateur '_' avec threads (SOA)
(512,256)	5.27	6.07	6.11	7.68 (× 1.46)	38.38 (× 7.28)
(1024,512)	7.04	6.24	7.01	7.32 (× 1.10)	11.62 (× 1.65)
(2048,1024)	6.77	5.99	7.13	7.47 (× 1.10)	12.89 (× 1.9)
(4096,2048)	3.99	4.12	7.16 (× **1.79**)	6.03 (× **1.51**)	14.24 (× **3.57**)
(8192,4096)	3.77	3.86	7.14 (× **1.89**)	5.93 (× **1.57**)	16.23 (× **4.31**)

Tableau 5.2: Performance en MLUP/s pour différentes versions de LBM D2Q9 en simple précision

Résultats

Pour notre évaluation de performances, nous comparons nos résultats par rapport à ceux correspondant à un code séquentiel écrit en C++. Pour cela, nous commençons par écrire une version NT^2 qui conserve les boucles parcourant les points et remplaçons les accès par pointeurs par des accès élémentaires dans le style MATLAB. Une des caractéristiques de NT^2 est d'imposer une certaine convention dans l'ordre de rangement des données des *tables*. Pour une *table* f de dimension N et dont la taille *effective*, *i.e.* prenant en compte d'éventuelles *strides*, est $n_1 \times n_2 \times ... \times n_N$, l'élément $f(i_1, i_2, ..., i_n)$ se situera à un offset mémoire de $i_1 + i_2 \times n_1 + ... + i_N \times (n_1 \times n2 \times ... \times n_{N-1})$ par rapport à l'élément $f(0, 0, ..., 0)$. Ainsi pour mettre en œuvre le rangement mémoire d'un maillage, on utilisera une *table* à 3 dimensions. Notons qu'avec NT^2, le passage SOA/AOS est trivial : cela revient à faire un changement de numéro de dimension pour le numéro de la composante de vitesse (3 pour SOA et 1 pour AOS). Après avoir réalisé ces tests, nous nous focalisons sur la disposition SOA et remplaçons les boucles par des calculs par sous-morceaux. Le squelette `transform` sera donc fortement sollicité car les opérations seront majoritairement des opérations point par point. Notons dans ce calcul que le tuilage asynchrone par défaut est utilisé pour la dimension 1 et 2 des tables, mais qu'il faut **explicitement** fixer le grain de la dimension 3 à 1. Pour ces tests, nous observons les deux cas SIMD et SIMD avec threads.

Les résultats des différentes versions du code LBM pour des maillages rectangulaires de taille variant de 512 × 256 à 8192 × 4096 sont synthétisés dans le tableau 5.2. Les gains de performance indiqués sont ceux obtenus en comparant les chiffres par rapport à ceux du code séquentiel.

En premier lieu, les résultats montrent que le remplacement des accès par pointeurs par des accès élémentaires sous NT^2 ne dégrade pas la performance. En second lieu, les résultats montrent d'une part que l'usage d'instructions SSE n'améliore la performance qu'en faible proportion (× 1.5 au lieu de 4), et d'autre part que le simple passage du format SOA au format AOS améliore davantage les performances (× 1.8). Cette différence s'explique par le fait que l'étape de collision à une part plus importante dans le calcul que l'étape de propagation car le format AOS priv-

ilégie justement cette étape. En dernier lieu, les résultats montrent que la meilleure version reste le format SOA lorsque l'on engage plusieurs threads. Cependant, les performances restent éloignées de la performance crête ($\times$ 2.3 par rapport à la version sans threads). Pour expliquer cette différence, nous étudions de plus près le passage à l'échelle de cette version du code.

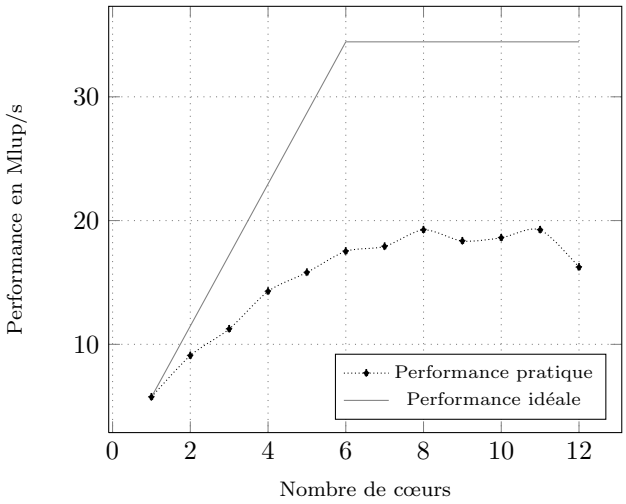


Figure 5.9: Performance MLUP/s de la version NT^2 de LBM D2Q9 en fonction du nombre de cœurs CPU

D'après la Figure 5.9, l'efficacité de Lattice Boltzmann suit difficilement celle souhaitée lorsque l'on augmente le nombre de processeurs. À la place, on voit clairement un palier limite de 20 MLUP/s, ce qui revient à un gain de performance de 4. Vu que la machine dans laquelle les tests ont été effectués contient 2 nœuds NUMA connectés à des processeurs dotés d'un contrôleur mémoire à 3 canaux, le gain par la multiplication de threads tendra sensiblement vers 6. Ce qui prouve encore une fois que l'efficacité de l'application est effectivement limitée par la bande passante mémoire.

Points à retenir

- La scalabilité obtenue est typique d'une application limitée par la bande passante mémoire.

- Le format SOA dans le code LBM n'est avantageux qu'à partir du moment où l'on combine instructions SIMD et multi-thread. L'apport du multi-thread est d'exploiter tous les canaux des contrôleurs mémoire (3 au lieu d'1 dans Mini-Titan)

5.4 GMRES

La méthode GMRES [63] ou *Generalized Minimal Residual Method* fait partie des méthodes itératives du sous-espace de Krylov dont le but est de résoudre le système d'algèbre linéaire $Ax = b$, A étant une matrice carrée d'ordre N, x le vecteur solution et b le vecteur de second membre de l'équation. Cette classe de méthodes résout ce système en cherchant à chaque pas d'itération k la solution $\mathrm{x}^{(k)}$ définie comme une combinaison des vecteurs base formant le sous-espace de Krylov $\kappa^k(A; r^{(0)})$. Dans cette expression, $r^{(0)}$ est le résidu associé à $x^{(0)}$, la valeur initiale de la solution x.

GMRES utilise une approche dite *de minimisation du résidu* impliquant, à chaque itération que le terme $\|b - Ax^{(k)}\|$ soit minimal dans le sous-espace $\kappa^k(A; r^{(0)})$. Le point clé dans GMRES est de s'appuyer sur une base orthonormale du sous-espace de Krylov pour d'une part, augmenter l'exactitude du résultat en minimisant le risque d'obtenir des vecteurs de base dégénérés, et d'autre part pour transformer le problème des moindres carrés $\min_x \|b - Ax^{(k)}\|$ en un problème beaucoup plus petit basé sur une matrice de Hessenberg supérieure. Les vecteurs de base du sous-espace de Krylov sont donc obtenus via une *procédure d'Arnoldi*, et le problème des moindres carrés est résolu en suivant les étapes suivantes :

- Réduction de la matrice de Hessenberg en un système triangulaire
- Résolution du système triangulaire
- Combinaison des vecteurs base en utilisant les coefficients obtenues

Présentation de l'algorithme

À chaque pas d'itération k, la *procédure d'Arnoldi* illustrée en Figure 5.10 est effectuée. Notons que le nombre de vecteurs base augmente au fur et à mesure que k augmente, ce qui a pour conséquence de modifier à la fois la quantité de données à traiter et la quantité de calculs à effectuer. Le problème des moindres carrés n'est en général invoqué qu'à la fin, c'est-à-dire au moment où la norme du résidu valide le critère d'arrêt. La performance de GMRES repose donc essentiellement sur la manière dont la procédure d'Arnoldi est implantée. Au bout de m itérations, si le critère d'arrêt n'a pas encore été atteint, le vecteur $x^{(m)}$ est calculé, puis l'algorithme est relancé en prenant $x^{(m)}$ comme solution initiale.

À ce stade, il s'agit de savoir comment organiser les différentes données en utilisant NT^2. Pour les vecteurs base, nous choisissons de les ranger dans une *table* de dimension 2 et de taille $N \times (m + 1)$, chaque colonne de cette *table* représentant un vecteur base. Du point de vue du tuilage asynchrone, nous utilisons le grain par défaut pour la dimension 1 de cette table, et fixons **explicitement** le grain de la dimension 2 à 1.

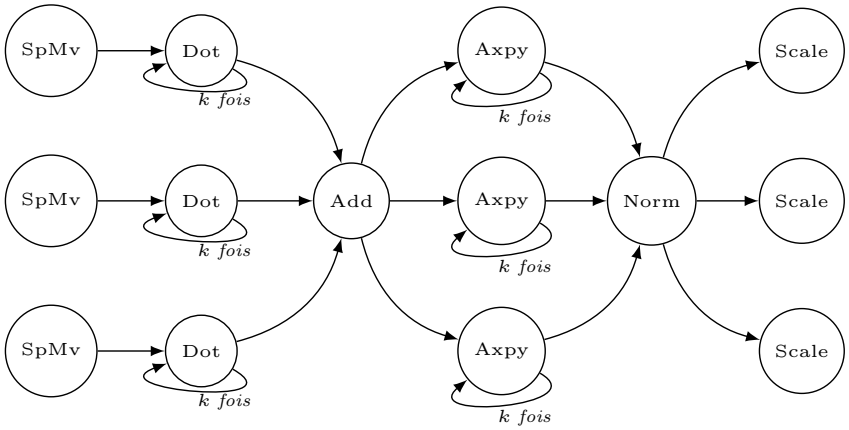


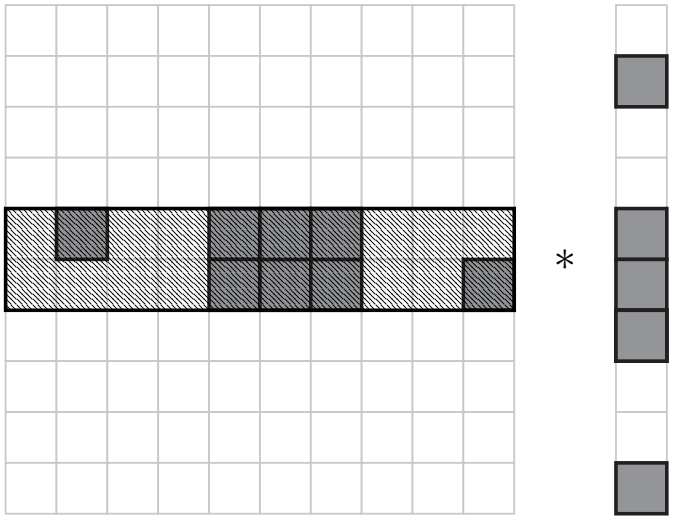
Figure 5.10: Schéma de principe de la procédure parallèle d'Arnoldi dans l'algorithme GMRES

Pour la matrice creuse A, on utilise le format CSR ou *Compressed Row Storage* connu pour être le format le plus général parmi les structures creuses existantes. Trois tables à une dimension sont utilisées : la table `val` contenant les valeurs non nulles $a_{i,j}$, la table `col_ind` contenant les indices j de ces éléments, et la table `row_ptr` contenant les *offsets* dans la table `val` des éléments $a_{i,j}$ avec l'indice j le plus petit pour chaque ligne i. Le produit matrice creuse - vecteur, ou la fonction *SpMV* (*Sparse Matrix Vector*) est implanté sous forme d'une fonction distincte et parallélisée explicitement en interne. Cette fonction met à jour les Futures associées au vecteur base issu du calcul. Le reste est écrit dans le style MATLAB et s'inspire presque exclusivement du code GMRES fourni par la *Netlib* [13]. En revenant sur la Figure 5.10, nous voyons que la composition `fold` $\circ$ `transform` sera sollicitée 2 $\times$ k fois à la k-ème itération.

Soulignons le fait que l'étape SpMV est la partie la plus longue dans l'algorithme GMRES. En effet, bien que cette opération se résume à une somme de produits, le calcul effectue des chargements de données qui peuvent être spatialement espacées dans la mémoire. Si l'on parallélise le calcul par *blocs de lignes* de la matrice creuse et si les indices j des éléments $a_{i,j}$ du bloc sont trop dispersés, des éléments non contigus du vecteur opérande devront être chargés; ce qui va limiter la performance car le nombre de chargements est équivalent au nombre d'opérations flottantes. Nous illustrons ce propos dans la Figure 5.11.

Résultats

Pour notre évaluation de performances, nous comparons l'implémentation NT^2 de l'algorithme GMRES par rapport à celle fournie par la bibliothèque Petsc [12]. Reconnue comme une référence dans le milieu du HPC, Petsc effectue ses calculs parallèles en utilisant MPI. Nous effectuons nos tests de performance en mettant



Légendes :
: Élément à lire
: Bloc de lignes, les éléments grisés dans la zone hachurée sont les valeurs non-nulles de la matrice placés dans l'ordre {de gauche à droite, puis de haut en bas} de manière contigüe en mémoire

Figure 5.11: Schéma de principe de l'étape SPMV par blocs

en œuvre deux problèmes tirés de la ressource web *Matrix Market Collection* et en utilisant dans les deux cas la même solution initiale et un vecteur de second membre calculé au préalable. Le paramètre de redémarrage m est fixée à 30 afin de travailler sur une quantité raisonnable de données et le nombre maximal d'itérations fixé à 1000, juste assez pour les problèmes traités.

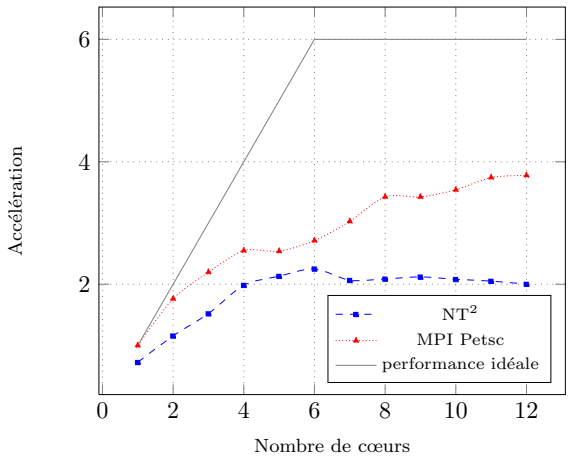


Figure 5.12: Performance de GMRES - Problème memplus (Matrice d'ordre 17758)

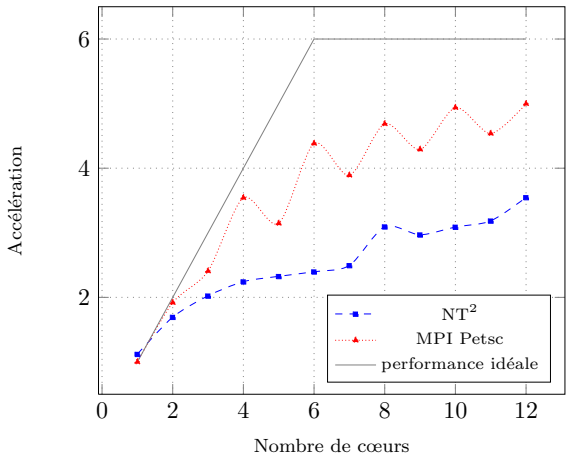


Figure 5.13: Performance de GMRES - Problème s3dkt3m2 (Matrice d'ordre 90449)

Globalement, Les Figures 5.12, 5.13 montrent que le code Petsc reste meilleur que le code NT^2. Contrairement à Petsc qui travaille sur des matrices distribuées, NT^2 travaille sur des matrices dont les données brutes sont placées logiquement de manière contiguë en mémoire mais entrelacées physiquement parmi les nœuds NUMA. Dans Petsc, chaque processus MPI est *pinné* par défaut sur un cœur CPU, et leur mémoire *pinnée* dans le domaine NUMA dans lequel ils se trouvent ; cette distribution de données réduit donc les communications inter-nœud NUMA.

Points à retenir

- Le benchmark GMRES souligne le besoin de travailler avec des tables ayant des données brutes distribuées et de pouvoir optionnellement donner aux tâches une affinité CPU.
- Nous considérons l'étape SpMV de GMRES comme le *pire cas* pour notre système de gestion des tâches car le calcul utile a une intensité opérationnelle extrêmement faible, ce qui oblige à placer minutieusement les données et les threads devant les traiter pour gagner en performance.

5.5 Synthèse des résultats obtenus

Dans ce chapitre, nous avons testé l'intérêt d'exploiter l'asynchronisme pour des algorithmes références issus du domaine du calcul scientifique.

Notre première question a été de savoir s'il existait des cas d'étude dans lesquels la suppression de barrières de synchronisation aurait un impact positif dans un environnement à mémoire partagée. La réponse est oui. Nous confirmons cette réponse

grâce au benchmark LU.

Notre deuxième question a été de savoir s'il était possible d'augmenter la localité en tirant parti des spécifications des dépendances. La réponse est oui *en théorie* mais non *en pratique*. Le benchmark Black & Scholes est en effet un cas d'étude dans lequel la marge de temps pour instancier les tâches successeurs n'est pas assez suffisante pour espérer préserver cette localité.

Notre ultime question a été de savoir si la gestion du parallélisme de tâches amenait un surcoût par rapport à une approche purement data-parallèle même pour les cas limites. La réponse est non. Les benchmarks Lattice Boltzmann et GMRES ont montré qu'il est possible d'apporter de l'efficacité même si l'asynchronisme y joue un rôle mineur.

Conclusions et Perspectives

Conclusions

Dans ce manuscrit, nous avons présenté une extension de NT^2, une bibliothèque encapsulant un langage dédié destiné à aider les non-spécialistes à développer des applications haute performance. Les principaux intérêts de cette extension est de maximiser l'asynchronisme dans les calculs et de pallier les limites du modèle d'optimisation statique des Expressions Templates. Pour cela nous avons fait évoluer le système de gestion du parallélisme vers un système basé sur la programmation asynchrone.

Notre contribution principale a été de mettre en œuvre le découpage automatique d'expressions exprimées dans un langage dédié sous forme de tâches légères asynchrones. Pour assurer à la fois la portabilité et l'efficacité de notre approche, nous avons développé des squelettes parallèles asynchrones multi-niveaux qui permettent d'exploiter non seulement le parallélisme de données mais aussi le parallélisme de tâches. Mettre en avant les dépendances de données au sein d'un algorithme permet d'exploiter au mieux son parallélisme latent. C'est pourquoi, nous avons décidé d'employer des graphes de dépendances. Comme nous l'avons montré au cours de ces travaux, le problème des aléas de données rend l'exploitation du parallélisme de tâches très complexe. Ce problème a pu être résolu grâce à HPX et au modèle Futures qui présente des avantages conséquents en terme d'expressivité de l'asynchronisme et d'opportunités de composition. En outre, l'utilisation du runtime HPX a permis d'afficher des performances proches de celles des runtimes de l'état de l'art comme TBB ou OpenMP.

Nous avons aussi contribué au modèle d'Expressions Templates en rajoutant des modèles de coûts inspirés du modèle BSP calculés en partie pendant la compilation et en partie pendant l'exécution. Ces coûts sont exploités dans le but de guider l'évaluation des expressions. La granularité des tâches étant un facteur important dans l'exploitation du parallélisme de tâches, nous avons conçu un système basé sur l'introspection des ASTs et l'utilisation de micro-benchmarks pour prédire une borne inférieure de performance et activer de manière opportune la parallélisation d'un code formulé sous forme de squelettes.

Durant la phase d'évaluation de performances, nous avons montré l'intérêt d'exploiter l'asynchronisme en testant la version tuilée asynchrone de l'algorithme LU et vérifié la viabilité du modèle Futures. Nous avons ensuite évalué notre approche jusqu'à ses limites en testant des algorithmes connus pour avoir des performances bornées par la bande passante mémoire. Nous avons observé qu'il n'y avait pas de réelles pertes

face à une approche purement data-parallèle, mais qu'il était nécessaire de tendre vers une gestion à la fois de données distribuées et de l'affinité CPU des tâches pour exploiter au mieux la bande passante mémoire d'une machine.

Perspectives

À la lumière de nos résultats, nous proposons quelques pistes pour de futures recherches. Comme nous l'avons vu, un grand nombre de problèmes issus du domaine du calcul scientifique manipulent intensivement les données et voient leur performance bornée par la bande passante mémoire et non la performance crête des architectures. Pour pallier ce problème, on peut suggérer deux solutions.

- Soit étendre le système de parallélisation automatique de NT^2 vers une programmation hybride de type MPI + tâches légères afin de travailler sur des données distribuées et profiter de l'asynchronisme *localisée* dans chaque nœud NUMA.
- Soit exploiter les GPUs car ceux-ci sont capables de fournir une bande passante 10 fois supérieure à celle des CPU – à condition – que les données restent sur le GPU et que les applications (ex : Black & Scholes) se prêtent bien aux accès dits *coalescents* à leur mémoire globale.

Pour des problèmes dont les briques de calcul ont une intensité opérationnelle comparable à celle d'un produit de matrices (ex : Décomposition LU), l'asynchronisme pourrait être exploré à plus grande échelle via des graphes de tâches distribués. Une des forces majeures du runtime HPX que nous n'avons pas exploitée jusqu'ici est qu'il a étendu le modèle Futures vers les systèmes distribués en surcouche d'un espace d'adressage global. Il existe aujourd'hui de nombreux problèmes distribués issus du domaine du calcul scientifique (ex : méthodes de résolution multi-grilles) qui souffrent du coût de synchronisation des barrières MPI et qui nécessitent un plus haut niveau d'abstraction. En effet sous MPI, l'exploitation de la programmation parallèle asynchrone ne peut se faire que via les mécanismes *one-sided*, des primitives connus pour être très difficiles à mettre en œuvre. D'une part, le modèle Futures pourrait faciliter l'implantation des graphes de tâches distribués, comme ce fut le cas pour les graphes de tâches sur système à mémoire partagée. D'autre part, la prise en charge par NT^2 des matrices multi-dimensionnelles pourrait hautement améliorer l'expressivité de ce type de code.

Les résultats dans ce manuscrit ont montré l'efficacité du modèle asynchrone à travers un langage dédié au calcul scientifique. Nos perspectives présentés dans ce dernier chapitre tendent vers une généralisation du modèle Futures pour des systèmes distribués localement hétérogènes.

Bibliographie

[1] Intel Cilk Plus, 2013. http://software.intel.com/en-us/intel-cilk-plus. (Cited on pages 19, 20 and 21.)

[2] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julie Langou, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Asim YarKhan. Plasma users' guide. *Electrical Engineering and Computer Science Department, University of Tennessee, Tech. Rep., http://icl. cs. utk. edu/projectsfiles/plasma/pdf/users guide. pdf*, 2009. (Cited on page 59.)

[3] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4), 2001. (Cited on page 41.)

[4] Marco Aldinucci, Marco Danelutto, and Jan Dünnweber. Optimization techniques for implementing parallel skeletons in grid environments. In Sergei Gorlatch, editor, *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 35–47, Stirling, Scotland, UK, July 2004. Universitat Munster, Germany. (Cited on page 27.)

[5] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, October 2014. (Cited on page 42.)

[6] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. (Cited on page 44.)

[7] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In HenryG. Dietz, editor, *Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208. Springer Berlin Heidelberg, 2003. (Cited on page 41.)

[8] Khari Armih, Greg Michaelson, and Phil Trinder. Cache size in a cost model for heterogeneous skeletons. In *Proceedings of the Fifth International Workshop on High-level Parallel Programming and Applications*, HLPP '11, pages 3–10, New York, NY, USA, 2011. ACM. (Cited on page 48.)

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous

multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. (Cited on pages 20 and 55.)

[10] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009. (Cited on pages 20 and 22.)

[11] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P3l: A structured high-level parallel language, and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995. (Cited on page 48.)

[12] Satish Balay, William D. Gropp, Lois C. McInnes, and Barry F. Smith. Efficienct management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, page 163–202. Birkhauser Press, 1997. (Cited on page 70.)

[13] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994. (Cited on page 70.)

[14] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Evaluating the performance of skeleton-based high level parallel programs. In Marian Bubak, GeertDick van Albada, PeterM.A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, volume 3038 of *Lecture Notes in Computer Science*, pages 289–296. Springer Berlin Heidelberg, 2004. (Cited on page 48.)

[15] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pages 761–770. Springer-Verlag, Berlin, Heidelberg, 2005. (Cited on page 41.)

[16] A.J. Bernstein. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757–763, Oct 1966. (Cited on page 37.)

[17] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973. (Cited on page 63.)

[18] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999. (Cited on page 19.)

[19] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, and Jack Dongarra. From serial loops to parallel execution on distributed systems. In Christos Kaklamanis, Theodore Papatheodorou, and PaulG. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 246–257. Springer Berlin Heidelberg, 2012. (Cited on page 42.)

[20] M'hamed Bouzidi, Dominique d'Humières, Pierre Lallemand, and Li-Shi Luo. Lattice boltzmann equation on a two-dimensional rectangular grid. *Journal of Computational Physics*, 172(2):704 – 717, 2001. (Cited on page 65.)

[21] J Mark Bull. Measuring synchronisation and scheduling overheads in openmp. In *Proceedings of First European Workshop on OpenMP*, volume 8, page 49, 1999. (Cited on page 51.)

[22] J.Mark Bull, Fiona Reid, and Nicola McDonnell. A microbenchmark suite for openmp tasks. In BarbaraM. Chapman, Federico Massaioli, MatthiasS. Müller, and Marco Rorro, editors, *OpenMP in a Heterogeneous World*, volume 7312 of *Lecture Notes in Computer Science*, pages 271–274. Springer Berlin Heidelberg, 2012. (Cited on page 51.)

[23] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009. (Cited on page 59.)

[24] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. (Cited on pages 3 and 41.)

[25] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005. (Cited on page 3.)

[26] Wai-Mee Ching and Da Zheng. Automatic parallelization of array-oriented programs for a multi-core machine. *International Journal of Parallel Programming*, 40(5):514–531, 2012. (Cited on page 42.)

[27] Philippe Chretienne, JK Lenstra, Z Liu, et al. Scheduling theory and its applications. *Journal of the Operational Research Society*, 48(7):764–765, 1997. (Cited on page 3.)

[28] Philipp Ciechanowicz and Herbert Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 108–113. IEEE, 2010. (Cited on pages 27 and 41.)

[29] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004. (Cited on page 41.)

[30] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989. (Cited on pages 26 and 48.)

[31] 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*, volume 28. ACM, 1993. (Cited on page 46.)

[32] John Darlington, Anthony J Field, Peter G Harrison, Paul HJ Kelly, David WN Sharp, Qian Wu, and Ronald L While. Parallel programming using skeleton functions. In *PARLE'93 Parallel Architectures and Languages Europe*, pages 146–160. Springer, 1993. (Cited on page 48.)

[33] Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. Auto-tuning skepu: A multi-backend skeleton programming framework for multi-gpu systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM. (Cited on page 55.)

[34] Beman Dawes, David Abrahams, and Rene Rivera. Boost C++ libraries. `http://www.boost.org`. (Cited on page 5.)

[35] J. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. (Cited on page 58.)

[36] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. (Cited on page 51.)

[37] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Domain-specific optimization strategy for skeleton programs. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 705–714. Springer Berlin Heidelberg, 2007. (Cited on page 27.)

[38] Pierre Esterie, Joel Falcou, Mathias Gaunard, Jean-Thierry Lapresté, and Lionel Lacassagne. The numerical template toolbox: A modern c++ design for scientific computing. *Journal of Parallel and Distributed Computing*, 74(12):3240–3253, 2014. (Cited on page 25.)

[39] Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte Rozoy. Boost. simd: generic programming for portable simdization. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 431–432. ACM, 2012. (Cited on pages 4 and 5.)

[40] Joel Falcou, Mathais Gaunard, and Jean-Thierry Lapresté. The numerical template toolbox, 2013. http://www.github.com/MetaScale/nt2. (Cited on page 4.)

[41] Joel Falcou, Jocelyn Sérot, Lucien Pech, and Jean-Thierry Lapresté. Meta-programming applied to automatic smp parallelization of linear algebra code. In *Euro-Par 2008–Parallel Processing*, pages 729–738. Springer Berlin Heidelberg, 2008. (Cited on pages 4 and 25.)

[42] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978. (Cited on page 46.)

[43] Clemens Grelck and Sven-Bodo Scholz. Sac—a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006. (Cited on pages 3 and 41.)

[44] Donald Gross. *Fundamentals of queueing theory*. John Wiley & Sons, 2008. (Cited on page 45.)

[45] Neil J Gunther. A general theory of computational scalability based on rational functions. *arXiv preprint arXiv:0808.1431*, 2008. (Cited on page 45.)

[46] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. (Cited on page 44.)

[47] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008. (Cited on page 44.)

[48] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996. (Cited on page 4.)

[49] Noman Javed and Frédéric Loulergue. Osl: Optimized bulk synchronous parallel skeletons on distributed arrays. In Yong Dou, Ralf Gruber, and JosefM. Joller, editors, *Advanced Parallel Processing Technologies*, volume 5737 of *Lecture Notes in Computer Science*, pages 436–451. Springer Berlin Heidelberg, 2009. (Cited on page 48.)

[50] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M T Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 383–414, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (Cited on page 3.)

[51] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009. (Cited on pages 19, 21 and 29.)

[52] Kenny Kerr. Windows with C++ - Visual C++ 2010 and the Parallel Patterns Library. *MSDN magazine*, page 107, 2009. (Cited on pages 9 and 21.)

[53] Herbert Kuchen. *A skeleton library*. Springer, 2002. (Cited on page 27.)

[54] Christopher Lauderdale and Rishi Khan. Towards a codelet-based runtime for exascale computing: Position paper. In *Proceedings of the 2Nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM. (Cited on pages 20 and 22.)

[55] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/. (Cited on page 51.)

[56] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991. (Cited on page 19.)

[57] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM. (Cited on page 19.)

[58] R Garey Michael and S Johnson David. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979. (Cited on page 3.)

[59] Eric Niebler. Proto : A compiler construction toolkit for DSELs. In *Proceedings of ACM SIGPLAN Symposium on Library-Centric Software Design*, 2007. (Cited on page 26.)

[60] Robert W Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998. (Cited on page 3.)

[61] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013. (Cited on page 20.)

[62] James Reinders. *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2010. (Cited on pages 9, 19, 20 and 21.)

[63] Youcef Saad and Martin H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986. (Cited on page 69.)

[64] Behrooz A Shirazi, Krishna M Kavi, and Ali R Hurson. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Society Press, 1995. (Cited on page 3.)

[65] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91 – 99, 2001. (Cited on page 25.)

[66] Xian-He Sun and Yong Chen. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010. (Cited on page 45.)

[67] Xian-He Sun and Lionel M Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, 1993. (Cited on page 44.)

[68] Laurence Tratt. Model transformations and tool integration. *Software & Systems Modeling*, 4(2):112–122, 2005. (Cited on page 4.)

[69] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. (Cited on page 46.)

[70] Leslie G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, January 2011. (Cited on page 47.)

[71] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. (Cited on page 25.)

[72] Todd Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995. (Cited on page 25.)

[73] K.B. Wheeler, R.C. Murphy, and D. Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008. (Cited on page 21.)

[74] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. (Cited on page 57.)

[75] Chengzhong Xu and Francis C. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. (Cited on page 19.)