Définition et évaluation d'INUKTITUT : un interface pour l'environnement de programmation parallèle asynchrone Athapascan

Nhien An Le Khac

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__/__/__/__/__/__/__/__/__/__/

THÈSE

pour obtenir le grade

DOCTEUR DE L'INPG

Spécialité : " Informatique : Systèmes et Logiciels"

préparée au laboratoire Informatique et Distribution
dans le cardre de ***l'Ecole Doctorale "Mathématique, Sciences et Technologies de l'Information, Informatique"***

présentée et soutenue publiquement

par

Le Khac Nhien An

Le 09 Mars 2005

Titre :

Définition et évaluation d'INUKTITUT : un interface pour l'environnement de programmation parallèle asynchrone Athapascan

---

Directeur de thèse : Brigitte Plateau
Co-encadrant : Thierry Gautier
Co-directeur : Dong Thi Bich Thuy

JURY

M. VAN DAT CUNG	, Président
M. RAYMOND NAMYST	, Rapporteur
M. CONG DUC PHAM	, Rapporteur
Mme. BRIGITTE PLATEAU	, Directeur de thèse
M. THIERRY GAUTIER	, Co-encadrant
Mme. DONG THI BICH THUY	, Co-directeur

À ma famille,
Ba, Mẹ và anh An
À ma femme,
Thanh Thoa

Remerciements

Mes premiers remerciements s'adressent à mon tuteur, M. Thierry Gautier pour avoir dirigé ma thèse en m'offrant un encadrement de qualité, pour ses précieux conseils, des suggestions pertinentes et des critiques constructives.

J'adresse également tous mes remerciements au Professeur Brigitte Plateau pour avoir accepté de diriger cette thèse au laboratoire ID-IMAG, pour son encouragement et sa grande gentillesse.

Je tiens à remercier infiniment le Professeur Dong Thi-Bich-Thuy sans qui je n'aurai pu entamer cette thèse. Je la remercie aussi pour son soutien dans les moments difficiles, sa compréhension et son encouragement.

Je remercie le Professeur Van-Dat Cung qui m'a fait l'honneur de présider le jury de ma soutenance. Je voudrais aussi remercier le Professeur Raymont Namyst, M. Cong-Duc Pham pour l'honneur qu'ils m'ont fait en consacrant une part de leur temps à rapporter sur mes travaux.

Je n'oublie pas de remercier mes amis Euloge Edi, Vincent Duong et AnTe Nguyen pour la correction française de mon manuscrit.

Je voudrais remercier les permanents, les doctorants et les ingénieurs au laboratoire ID pour les discussions intéressantes sur le travail ainsi que sur la vie quotidienne.

Je voudrais remercier également mes amis vietnamiens pour leur soutien, leur encouragement et pour les moments agréables.

Enfin, les mots ne sauraient exprimer ma profonde gratitude et mes remerciements infinis à mes parents et mon frère qui m'ont soutenu et encouragé durant ces années de thèse ; particulièrement à ma femme pour son amour, sa présence, son soutien et sa compréhension.

Table des matières

Table des figures

Liste des tableaux

1

Introduction

L'informatique joue un rôle important dans la recherche scientifique et technique d'aujourd'hui. Le calcul scientifique a de plus en plus besoin de puissance de calcul que ce soit pour traiter des problèmes de plus en plus complexes ou bien pour les traiter plus rapidement. Citons par exemple les applications de simulation moléculaire, la simulation océanique, la simulation numérique multi-physique, la génétique, *etc*.

L'une des manières d'étendre la capacité des calculateurs pour augmenter leur puissance de traitement est d'utiliser conjointement plus ordinateurs pour résoudre un même problème, c'est-à-dire en utilisant les méthodes et les outils étudiés par le *« calcul parallèle à haute performance »* [DS98] (*High Performance Computing*). Ce domaine de recherche considère la résolution de problèmes sur des architectures construites par assemblage de plusieurs ordinateurs reliés par un réseau de communication et qui sont appelées *« grappes »* (*cluster*).

Ces grappes sont constituées par l'interconnexion des stations de travail (monoprocesseurs ou multiprocesseurs) via un réseau local plus ou moins performant. De nombreux protocoles et interfaces de programmation ont été développés pour exploiter ces grappes, que ce soit pour la partie « calcul » et l'exploitation efficace des processeurs grâce à la multiprogrammation légère [IEEE94, MN95, Open], ou pour la partie « communication » et le développement de protocoles efficaces [SOH96, Ste94, ABMN02, Aum02, Myr, FTK96] permettant le transfert d'octets avec une très faible

latence. Ces architectures offrent des plates-formes de calcul à hautes performances avec du matériel à faible coût. Aussi, de nos jours, ces grappes rencontrent un large succès dans le domaine du calcul scientifique. Pourtant la programmation d'une application ou le portage d'un environnement de programmation parallèle sur ces grappes est un travail difficile du fait de la complexité et de la variété des caractéristiques de ces architectures et des bibliothèques disponibles.

De nombreux travaux ont débouché sur la définition de bibliothèques pour la programmation portable sur les grappes. L'objectif de ces travaux est de garantir la portabilité des codes par la définition d'un ensemble de fonctions d'interface exporté à l'application et dont l'implantation, sur différentes architectures, permet de rendre les codes portables. La difficulté rencontrée concerne le choix de la définition des fonctions d'interfaces afin de garantir de bonnes performances sur plusieurs architectures de grappes. L'exemple le plus connu concerne la bibliothèque *MPI* [SOH96] qui permet la portabilité du code d'applications parallèles régulières sur un large ensemble d'architectures de grappes. De nombreuses implantations de MPI existent, LAM-MPI [LAM65] et MPI-CH [mpich] sont les deux implantations les plus connues car elles sont disponibles gratuitement et possèdent de bonnes performances sur différentes grappes et réseaux de communication. D'autres bibliothèques, telles que *Socket*[Ste94], *Madeleine*[ABMN02, Aum02], *GM/Myrinet*[Myr], *Corba*[Vin98], *Nexus*[FTK96], existent sur grappes et possèdent des implantations plus performantes ou des fonctionnalités spécifiques.

Le travail de ce doctorat s'inscrit dans le cadre du développement du logiciel Athapascan et de la définition d'une couche de portabilité adaptée.

Contexte du travail de thèse

Situé au sein du laboratoire Informatique et Distribution (ID UMR 5132 CNRS-INPG-INRIA-UJF) appartenant à la fédération d'Informatique et Mathématique Appliquées de Grenoble (IMAG), le projet APACHE développe un environnement de programmation parallèle pour grappes et grilles de grappes appelé Athapascan qui offre un bon compromis entre la portabilité et la performance des applications. Au début de ce travail de thèse, le logiciel Athapascan été articulé en deux couches :

- *Athapascan-1* [Gal99, Cav99, Dor99, GRC+98, RGR03] est l'environnement de programmation parallèle pour le calcul parallèle à hautes performances. Il offre une interface de type macro *data-flow* pour la programmation parallèle asynchrone. *Athapascan-1* est conçu pour obtenir une exécution efficace en préservant la portabilité des codes. Cet environnement repose sur un modèle de programmation par tâches asynchrones et des objets partagés avec un calcul automatique des précédences entre tâches. Ce graphe de tâches peut être ordonnancé efficacement sur une architecture de type grappe [Dor99]. *Athapascan-1* gère automatiquement le placement et le mouvement des données entre les ressources de l'architecture. Il exploite la généricité du langage C++ [Str00] et se présente comme une bibliothèque générique C++. Cette première version d'*Athapascan*-1[GRC+98] s'appuie sur *Athapascan-0* pour la gestion de l'ordonnancement local des processus légers, ainsi que pour la communication entre processus.
- *Athapascan-0* [BGPP97, Gin97, Car99, Pas99] est un noyau exécutif de multi-programmation et communication. *Athapascan-0* est construit au dessus de la bibliothèque standard de passage de message *MPI* [SOH96] et de processus légers (*threads*) *POSIX Thread 1003c* [IEEE94]. Le modèle de programmation d'*Athapascan-0* est un réseau de processus légers qui communiquent entre eux. *Athapascan-0* permet d'exploiter deux types de parallélisme : le parallélisme intra-nœuds qui est géré par les processus légers qui communiquent via la mémoire partagée ; le parallélisme inter-nœud qui est géré l'envoi et la réception de message via *MPI*. La première limitation de *Athapscan-0* est sa dépendance vis-à-vis *MPI*. De plus, une perte de performance est due aux techniques utilisées pour permettre l'avancement des communications de MPI. Cette dégradation se traduit par l'utilisation d'une ressource de calcul pour l'avancement des communications, au détriment du calcul.

Objectifs du doctorat

L'objectif de ce travail de thèse se situe au niveau de l'articulation entre une application et une bibliothèque de communication (ou intergiciel de communication). D'une part, il s'agit d'étudier et de proposer un interface de bas niveau permettant de garantir une certaine portabilité des performances à l'exécution au-dessus de différentes bibliothèques pour la multiprogrammation légère et la communication ; d'autre part, il s'agit d'évaluer les performances d'une implantation de cet interface afin d'aider, par exemple, à choisir une bibliothèque de communication particulière sur une grappe donnée. Le travail de cette thèse se place dans le contexte du développement de l'intergiciel INUKTITUT, un intergiciel de portabilité générique adapté aux caractéristiques d'*Athapascan-1* :

- Sur les architectures actuelles de grappe, le parallélisme au sein d'un nœud multiprocesseur est efficacement exploité par l'utilisation de plusieurs processus légers partageant un même espace d'adressage. Le système d'exploitation ordonnance les processus légers localement en permettant de recouvrir une partie des latences dues aux entrées-sorties.
- Dans *Athapascan-1* [Rev04], le rôle des communications entre nœuds est, d'une part, de signaler un événement (typiquement, le fait qu'un processeur soit inactif), et, d'autre part, d'écrire les valeurs associées aux résultats d'une tâche de calcul dans la mémoire d'un nœud.

De ces contraintes et de l'étude des interfaces pour la communication de bibliothèques précédemment citées, il a été décidé qu'INUKTITUT contiendrait des fonctions pour la multiprogrammation légère (un sous ensemble de la norme POSIX) et pour les communications à base de message actif [ECGS92]. L'ensemble des fonctions d'interfaces est réduit afin de faciliter le portage sur les bibliothèques de communication précédemment citées. INUKTITUT a été porté avec succès sur des systèmes aussi différents que *Linux, Unix, MacOS X, Windows* au dessus de TCP/IP, Myrinet ou CORBA utilisé comme couche de transport de données.

L'évaluation des performances est réalisée à travers des mesures des différents portages d'INUKTITUT sur différentes bibliothèques pour la multiprogrammation légère et la communication. Les mesures sont réalisées à travers des jeux de test et des applications de simulation répandus dans la littérature. L'estimation et l'analyse des résultats se basent sur des indicateurs de performance et le modèle de communication de Hockney [HB94].

Les contributions de cette thèse sont : la définition de l'intergiciel INUKTITUT ; l'implantation des programmes pour l'évaluation des performances d'une implantation d'INUKTITUT ; l'évaluation et l'analyse des performances d'INUKTITUT sur différentes architectures.

Plan de la thèse

La thèse se compose de 5 chapitres principaux.

Le chapitre suivant *« Environnement exécutif pour les grappes : État de l'art »* introduit quelques concepts de base et les problèmes posés dans cette thèse : la définition d'un interface de communication pour la programmation parallèle des grappes de noeuds multiprocesseurs. Cet interface doit permettre de gérer les ressources d'une grappe (les processeurs, le réseau) à travers des bibliothèques et des services du système d'exploitation. Nous présentons aussi des projets existants sur les couches de communication et les intergiciels comme *MPI*[SOH96], *PVM*[KGPS97], *PM*^{2}[Nam01, ABMN02, Aum02], *Nexus*[FKT96], *PadicoTM*[DPP02], et *Athapascan-0*[BGPP97, Gin97, Car99, Pas99].

Dans le troisième chapitre *« INUKTITUT : un interface générique pour un environnement de programmation parallèle »*, nous décrivons minutieusement INUKTITUT en commençant par son architecture. Nous décrivons ensuite de sa bibliothèque de processus légers, son modèle et son interface. Nous présentons également son interface pour l'accès au réseau. Différentes implantations d'INUKTITUT sont ensuite abordées. La présentation des applications qui se basent sur INUKTITUT termine ce chapitre.

Le quatrième chapitre *« La performance du processus léger dans INUKTITUT »* est dédié à l'analyse des performances des fonctions de gestion des processus légers dans INUKTITUT. Nous étudions d'abord les coûts fondamentaux tels que la création de processus léger et la commutation de contexte. Nous évaluons ensuite le coût des primitives de synchronisation. Pour terminer, nous analysons comparativement les résultats de l'implantation des algorithmes de test dans INUKTITUT à ceux de l'interface de *threads* du standard *Posix* [IEEE94], appelée *Pthreads.*

Dans le cinquième chapitre *« Performances des communications bipoints »*, nous évaluons en particulier la bibliothèque de communication bipoint d'INUKTITUT à travers les expériences et les analyses de micro benchmarks. Nous comparons différentes implantations d'INUKTITUT au dessus de TCP/IP, au dessus de CORBA ainsi qu'au dessus de Myrinet. Nous présentons d'abord le modèle de communication Hockney [HB94] et les indicateurs de performance utilisés. Nous présentons ensuite nos expérimentations et nous analysons les résultats obtenus sur deux types d'architectures. Nous présentons en particulier nos mesures pour l'évaluation du potentiel de recouvrement des communications par du calcul de la bibliothèque.

Le sixième chapitre *« Évaluation des performances de la communication collective »* présente et évalue les fonctions fournies pour la communication collective en tant que bibliothèque au dessus d'INUKTITUT. Nous nous intéressons essentiellement aux fonctions de diffusion et réduction sur un groupe de processus. Nous présentons d'abord les méthodes d'évaluations puis nous analysons les résultats pour chaque algorithme de communication collective utilisé. Nous étudions la performance du service de barrière aussi offert.

Ce manuscrit se termine par le chapitre *« Conclusion »* qui fait le bilan de ce travail de doctorat et qui présente les perspectives qui nous semblent importantes.

2

Environnement exécutif pour les grappes : État de l'art

Ce chapitre présente les environnements exécutifs pour les grappes. Nous commençons par introduire brièvement des composants de base pour les grappes. Nous étudions ensuite deux concepts de base qui sont exploités pour le calcul parallèle à haute performance sur la grappe : processus légers et communication. Enfin, nous présentons et comparons quelques projets existants sur les couches de communication et de processus légers ainsi que sur les intergiciels tels que *OpenMP*[Open], *MPI*[SOH96], *PVM*[KGPS97], *PM*^{2}[Nam01,ABMN02,Aum02], *Nexus*[FKT96], *PadicoTM*[DPP02] et *Athapascan0* [BGPP97, Gin97, Car99, Pas99].

2.1 Les composants de base pour les grappes

Une grappe est un type de machine parallèle qui se compose d'un ensemble d'ordinateurs multi-processeurs autonomes et interconnectés afin de pouvoir travailler ensemble comme une ressource de calcul intégré. Dans cette section, nous étudions les composants de base qui sont utilisés pour construire une grappe. Ces composants sont les matériels ainsi que les logiciels. Nous commençons par présenter les matériels.

2.1.1 Les matériels d'un nœud[1] de grappe

Processeurs

Le processeur (*CPU_Central Processing Unit*, en anglais) est le cerveau de chaque ordinateur [Tan87a]. Chaque calcul et processus fait par un ordinateur est exécuté par le processeur. De puis quelques années, la puissance de calcul des processeurs fait des progrès à pas de géant (50% ou plus par an [Buy99a]). Par exemple, les processeurs d'Intel qui sont utilisés couramment dans les stations de travail ne cessent pas de croître : Pentium II (1997, 266Mhz), Pentium III (1999, 750Mhz), Pentium 4 (2001, 1,7GHz) et Itanium (IA-64 : 2002, 900Mhz) aujourd'hui. De nos jours, l'architecture 64 bits (IA-64, AMD64 [AMD], IBM Power [MSSW94], Sun Sparc64 [Sun], MIPS, Alpha) permet de gérer des espaces d'adressage plus important.

Mémoire et cache

L'accès mémoire est beaucoup plus lent que la vitesse du processeur. Les architectures utilisent des caches [Tan87b] d'accès rapide pour stocker les blocs de mémoire qui sont accédées régulièrement par le processeur. Celle-ci est très chère par rapport à la mémoire. Les architectures actuelles possèdent donc une hiérarchie mémoire importante : du registre processeur à la mémoire distribuée, les temps d'accès augmentent. Par exemple, le temps d'accès pour un registre processeur est d'inférieur à 10ns ; pour le cache est accédé entre 10ns à 100ns ; la mémoire principale entre 200ns à 500ns ; et pour la mémoire distribuée en 10ms [Don01] (suivant le réseau d'interconnexion).

Disque et E/S (entrée/sortie)

La capacité et la vitesse d'accès au disque augmentent rapidement d'année en année. Les disques permettent non seulement de stocker des applications parallèles ainsi que

[1] Un nœud est un ordinateur membre d'une grappe dans le contexte de section 2.1

les données et résultats. Ils peuvent aussi servir de mémoire secondaire pour les applications nécessitant de stocker de très gros volumes de données d'exécution.

Nous venons d'étudier les principaux matériels d'un nœud de la grappe. La performance globale d'une grappe dépend non seulement de chaque nœud mais aussi de l'interconnexion entre eux. La section suivante va aborder des problèmes de connexion pour la grappe.

2.1.2 Réseau d'interconnexion

Les nœuds dans la grappe communiquent à travers des protocoles de communication (par exemple *TCP/IP* [Ste94]). Les communications se basent sur les réseaux dont la latence et le débit [Mita] ont des impacts importants sur la performance. Les technologies de réseau usuels dans les grappes sont : *Ethernet, Fast Ethernet, Gigabit Ethernet* et *Myrinet* ou encore l'hyper transport (Infinity Band).

Ethernet

Ethernet standard (10Mbps[2]) est utilisé largement sur les réseaux de stations de travail. Néanmoins, son débit (10Mbps) est une limitation pour le calcul à haute performance sur les grappes qui ont toujours besoin de transférer une large quantité de données. *Fast Ethernet* est une amélioration d'Ethernet standard qui offre un débit de 100Mbps. Actuellement *Gigabit Ethernet* permet de communiquer à une vitesse d'environ 1 Gbps[2] [Mitb]. La technologie Gigabit Ethernet est utilisée pour l'interconnexion des nœuds dans la plupart des grappes.

Myrinet

Myrinet est un réseau d'interconnexion qui est fabriqué par Myricom [Myr]. Les cartes *Myrinet* sont intégrées avec deux composants importants : un processeur RISC [DS98] appelé LANai et une mémoire SRAM allant de taille de 2 à 8Mo (pour les dernières générations Myrinet-2000). LANai contrôle le transfert de données entre la

[2] Mbps: Megabit par second, Gbps: Gigabit par second

machine hôte et le réseau. Ces composants ont la possibilité d'opérer des transferts DMA (*Direct Memory Access*) [Tan87c] vers et depuis la mémoire de la machine hôte, le lien entrant et le lien sortant. L'avantage de transférer de données via des opérations DMA est de décharger le processeur de la machine hôte d'une charge de travail de communication. De plus, LANai est programmable ce qui permet d'implanter différentes stratégies de communication [PT98]. Le réseau *Myrinet*(GM) offre un débit d'environ 2 Gbps et une latence faible d'environ 5,7µs [Myr]. Il est de plus en plus utilisé largement dans les grappes à hautes performances [Top500].

Les composants pour une grappe sont non seulement les matériels mais aussi les logiciels qui se basent sur les systèmes d'exploitation pour permettre l'exploitation du matériel par les applications des utilisateurs. Quelques systèmes d'exploitation répandus sur les grappes sont présentés dans la section suivante.

2.1.3 Système d'exploitation

Les concepts de système d'exploitation sont présentés minutieusement dans [Tan92a][Tan01]. Dans cette section, nous étudions le système d'exploitation utilisé sur la plupart des nœuds des grappes : Unix et ses dérivés.

Unix

Unix [unix] reste aujourd'hui l'un des systèmes d'exploitation les plus populaires, surtout dans les milieux de la recherche et développement. La première version d'Unix est sortie en 1969 au sein du Laboratoire BELL, USA. Unix est un système d'exploitation robuste, fiable et conforme à POSIX [Pos]. Quelques caractéristiques principales de Unix sont :

- Multitâche préemptif, multi-utilisateurs, supportant des multiprocesseurs (SMP). Il permet l'exécution de plusieurs processus en parallèle. Il permet aussi le

contrôle et la répartition des ressources (mémoire, processeurs, espace disque, programmes et utilitaires) entre les utilisateurs et les tâches.

- L'exécution sur de nombreuses plates-formes comme Intel x86, IA-64, AMD, AMD64, Alpha, ARM, MIPS, Power PC, SPARC, SPARC64, etc.

- Son architecture fournit un grand choix de protocoles de réseau et de fonctionnalités dans le domaine du réseau.

2.1.4 Environnements logiciels pour grappes

L'objectif de ces logiciels est de permettre d'exploiter au mieux les capacités de l'architecture des grappes pour offrir des environnements de programmation efficaces et portables.

Dans la programmation parallèle, un problème donné est divisé en plusieurs sous problèmes qui sont résolus simultanément. Chaque sous-problème est exécuté par un processus différent. La résolution de ce problème nécessite de la part des processus qu'ils puissent échanger des données et se synchroniser. La coopération et synchronisation se font à travers d'une mémoire partagée ou par échange de message. Dans la plupart des environnements de grappe, à cause de l'absence d'une mémoire commune (physique ou partagée), les processus communiquent en échangeant des messages à travers le réseau d'interconnexion. Un programme parallèle sur la grappe peut être vu comme un réseau de processus communicants.

Afin d'atteindre leurs objectifs, les environnements logiciels doivent fournir une interface qui permet de réaliser facilement et efficacement des communications indépendamment de l'architecture sous-jacente. C'est répondre à ces besoins que plusieurs bibliothèques de communication sont apparues : *MPI* [SOH96], *PVM* [KGPS97], *Madeleine* [Nam01, ABMN02,Aum02], *Nexus*[FKT96], *PadicoTM*[DPP02] et *Athapascan0* [BGPP97, Gin97, Car99, Pas99].

Les processus légers sont utilisés largement pour la programmation concurrente sur les machines mono-processeurs ou multi-processeurs. La multiprogrammation légère permet aussi d'exploiter efficacement un parallélisme à grain fin dans les applications irrégulières. La création d'un processus léger est moins coûteuse que celle d'un processus lourd et la gestion d'un processus léger est aussi plus efficace que celle d'un de processus [Sin97a]. Des nombreuses bibliothèques de processus légers sont disponibles : *POSIX Threads 1003.1c* [IEEE94][MN95], *Java Threads* [LB00], et *Windows Threads* [Hel94].

Afin d'atteindre la portabilité et l'efficacité, les environnements de programmation parallèle reposent sur des noyaux exécutifs (ou supports d'exécution) qui se basent sur les protocoles ou les bibliothèques standards de communication et de processus légers. Ces noyaux permettent de « cacher » aux utilisateurs les différentes mises en œuvre sur des protocoles et des interfaces particuliers. Quelques bibliothèques de communication ainsi que noyaux exécutifs sont présentés et discutés dans la section 2.4 (page 28). Dans les deux sections suivantes, nous présentons deux éléments importants qui permettent d'exploiter efficacement le potentiel offert par les grappes pour le calcul parallèle à hautes performances : les processus légers et les bibliothèques de communication.

2.2 Processus légers

Cette section est consacrée aux processus légers, un élément fondamental pour la multiprogrammation efficace. Nous commençons par les concepts de base ainsi que les caractéristiques importantes d'un processus léger. Trois modèles d'implantation répandus des processus légers sont aussi présentés. Enfin, nous présentons quatre interfaces standards.

2.2.1 Concepts et Caractéristiques

Définition

Un processus lourd est une entité d'exécution de base fourni par le système d'exploitation qui garantit une isolation et une protection du processus des autres processus du système. Les systèmes d'exploitation aujourd'hui permettent de créer dans un processus lourd plusieurs flots d'exécution qui partagent les mêmes ressources que le processus. Un tel flot d'exécution est un processus léger (Figure 2.1). Le terme « multi programmation légère » est utilisé pour décrire la situation dans laquelle plusieurs processus légers sont présents dans le même processus lourd.

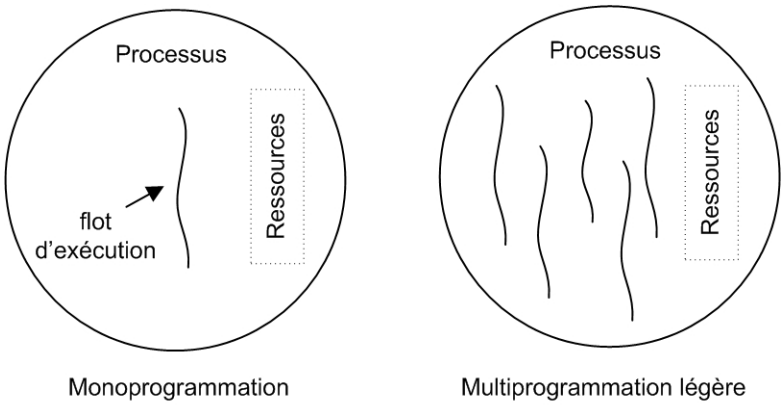


Figure 2.1 Processus et processus légers

Caractéristiques

Chaque processus léger a son propre compteur ordinal, un registre d'état, une pile. Tous les processus légers d'un même processus lourd partagent le même espace d'adressage. Les communications se font à travers cette mémoire. De plus, ils partagent aussi un même ensemble de ressources du système d'exploitation comme les fichiers ouverts, les autorisations, les quotas, etc.

Les opérations qui peuvent manipuler des processus légers sont la création, la terminaison, la synchronisation et l'ordonnancement. Un processus lourd est commencé

avec un processus léger simple. Ce premier fils d'exécution peut en créer d'autres. La terminaison d'un processus léger est en quelque sorte semblable à la terminaison d'un processus lourd. Il se détruit quand le flot se termine ou lorsqu'il est interrompu par d'autres processus légers.

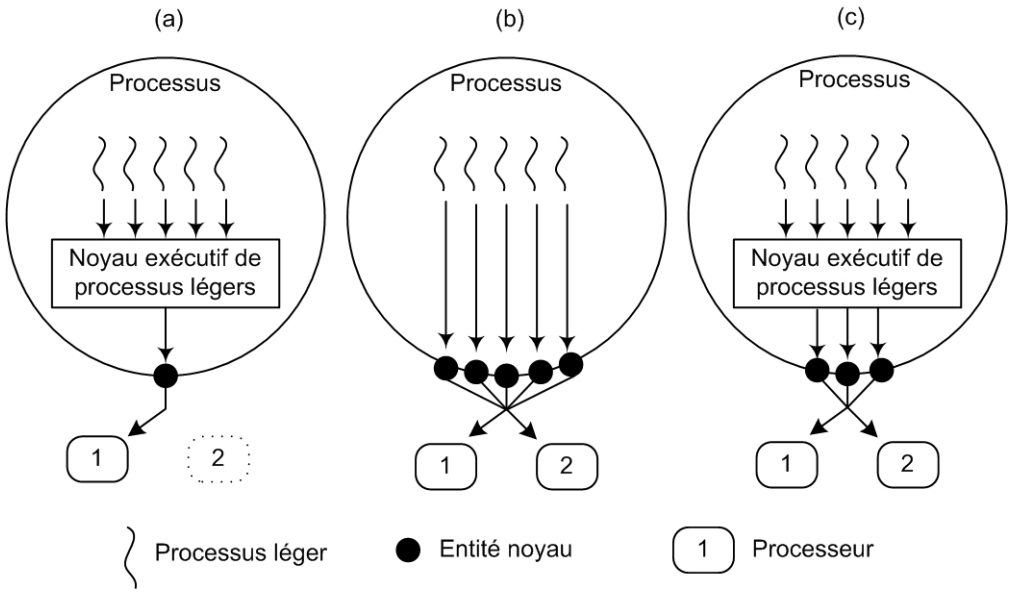
Parce que tous les processus légers d'un processus lourd partagent un même espace d'adressage, ils doivent synchroniser leurs accès à des données partagées afin de maintenir une certaine cohérence. Des primitives classiques pour la synchronisation sont le verrou, la variable de condition et le sémaphore [Tan92b].

Les stratégies d'ordonnancement et d'exécution des processus légers sont normalement décidées par l'utilisateur pour passer des paramètres appropriés aux noyaux. Quelques stratégies qui sont présentées dans [Sin97b] sont l'affectation de priorité, la flexibilité (changer la taille de quantum dynamiquement), *handoff scheduling*.

L'utilisation de processus légers procure plusieurs avantages comparés aux processus lourds. D'abord, le coût de création d'un processus lourd est plus grand à celui de la création d'un processus léger. De plus, la commutation de contexte entre les processus légers est moins couteuse que la commutation entre les processus lourds. Enfin, les processus légers permettent d'utiliser efficacement dans les machines de type SMP [EKB+92].

2.2.2 Modèle d'implantation

Il existe plusieurs façons d'implanter des processus légers dans les systèmes d'exploitation. Ils sont implantés soit dans l'espace utilisateur ou dans le noyau du système. Ils sont aussi implantés en combinant ces deux niveaux. Ces différentes façons d'implantation créent trois modèles de processus légers : le modèle N:1, le modèle 1:1 et le modèle M:N. Ces modèles d'implantation ont un impact sur la performance des processus légers.



(a) Modèle N :1 (b) Modèle 1 :1 (c) Modèle M :N

Figure 2.2 Les modèles différents de processus légers

N:1

Dans ce modèle, les processus légers sont placés dans l'espace utilisateur [FCL93]. Le noyau du système ne sait rien de leur existence. Les processus légers d'un processus lourd se partagent la même entité noyau pour leur exécution. C'est-à-dire qu'ils sont en compétition entre eux-mêmes pour les ressources allouées à ce processus lourd (Figure 2.2a). Ils sont ordonnancés, synchronisés par un noyau exécutif de processus légers (*threads runtime system*, en anglais) qui est une partie de code de ce processus lourd. Ce modèle est aussi appelé *« processus légers au niveau utilisateur »*.

Les avantages de ce modèle sont :

- la souplesse car il peut être réalisé sous forme de bibliothèques de niveau d'utilisateur, sans modifier le noyau.

- Le surcoût de la commutation de contexte entre processus légers est faible parce que toutes les opérations de contrôles des processus légers sont faites dans l'espace utilisateur, sans appel coûteux au système.

Ses inconvénients sont :

- On ne peut pas exploiter plusieurs processeurs physiques d'une architecture SMP pour les processus légers d'un même processus car son unique entité noyau est placée sur un processeur physique donné. Cette restriction limite la capacité de parallélisme disponible.
- Le problème d'appel système bloquant : par exemple, quand un processus léger fait une E/S, il bloque le processus portant ce processus léger. Tous les processus légers de ce processus sont aussi bloqués.

1:1

Dans ce modèle, chaque processus légers est ordonnancé comme une entité noyau. C'est-à-dire que le noyau du système « connaît » tous les processus légers. L'implantation des processus légers se fait totalement dans le noyau du système (Figure 2.2b). Ce modèle est appelé « processus légers au niveau système » [EKB+92].

Ce modèle résout les deux inconvénients du modèle exposé ci-dessus qui deviennent ses avantages :

- L'exploitation du parallélisme d'architecture SMP ; puisque le système d'exploitation peut placer des processus légers d'un même processus sur différents processeurs physiques.
- Les attentes sur un appel système ne bloquent que le processus léger concerné.

Pourtant ce modèle a certains inconvénients :

- Les opérations de contrôle des processus légers sont plus coûteuses que le modèle 1:1 parce qu'elles sont prises en charge par le noyau du système.
- La consommation des ressources limitées du système : le nombre de processus légers crées est limité dans les systèmes d'exploitation.

- L'utilisateur n'a nullement ou peu la possibilité d'agir sur l'ordonnancement des processus légers.

M:N

Ce modèle est aussi appelé « hybride » [EKB+92][PKB+91] parce qu'il est une combinaison des deux modèles précédents. Ce modèle a des avantages à la fois de modèle N:1 et de modèle 1:1 en offrant deux niveaux d'ordonnancement. Dans ce modèle, M processus légers d'un processus lourd au niveau utilisateur se partagent N (N<M) entités noyau pour leur exécutions (Figure 2.2c). Ce modèle permet non seulement d'exploiter le parallélisme d'architecture SMP ainsi que d'avoir un surcoût faible des opération de contrôle des processus légers mais aussi d'éviter le problème d'appel système bloquant. L'inconvénient de ce modèle est la complexité de sa mise en œuvre parce qu'il faut gérer à la fois des processus légers et des entités au niveau du système.

2.2.3 L'interface

Il y a plusieurs interfaces de processus légers. Les trois interfaces les plus connues sont les processus légers de *POSIX 1003c* [IEEE94], les processus légers de *Microsoft Windows* [Hel94] et le processus légers à la *Java* [LB00]. Bien que ces styles soient différents par les implantations et les APIs, les concepts de base restent essentiellement les mêmes. L'interface de POSIX est définie par le standard IEEE POSIX 1003.1 [IEEE94] et la plupart des constructeurs implantent sur leurs systèmes UNIX.

L'interface des processus léger de Microsoft est utilisée dans les systèmes d'exploitation de Windows.

L'interface Java est intégrée dans la JVM (*Java Virtual Machine*).

L'avantage de l'interface de POSIX est sa portabilité et ses performances car il se présente comme des interfaces applicatifs C/C++, et il est implanté dans la plupart de système d'exploitation de type Unix comme Linux.

OpenMP [Open] est un standard de l'industrie pour la programmation parallèle sur des architectures à mémoire partagée. Basée sur les techniques de la multiprogrammation légère, OpenMP peut être considéré comme l'un des grands standards au service du calcul scientifique. Cependant, OpenMP ne supporte pas des architectures distribuées et ne traite que partiellement les programmes ayant des structures de contrôle non régulières.

2.3 Communication

Dans cette section, nous introduisons les concepts de base du paradigme de programmation par échange de messages. Ensuite, nous étudions deux classes d'opérateurs de communication : la communication bipoint et la communication collective. Enfin, nous parlons des avantages de l'intégration des communications et de la multiprogrammation légère.

2.3.1 Introduction

La programmation parallèle sur la grappe s'appuie sur la communication par échange de message (*cf.* section 2.1.4, page 11). Dans ce paradigme, la coopération entre des processus se fait par émission et réception de message. Deux participants principaux de la communication sont l'émetteur et le récepteur. Toutes les données doivent être explicitement transférées de la mémoire de l'émetteur à la mémoire du récepteur. Il faut garantir aussi que le récepteur reçoit exactement les données dont il a besoin.

De nombreux protocoles sont définis pour implanter la communication point à point (bipoint) et la communication collective. Dans la communication bipoint, il n'y a qu'un

émetteur et un récepteur. Dans la communication collective, à l'inverse, un certain nombre de processus effectue une opération de communication des émissions et un autre nombreux effectue des réceptions.

2.3.2 Communication bipoint

Les communications bipoints possèdent des sémantiques différentes.

Synchrone

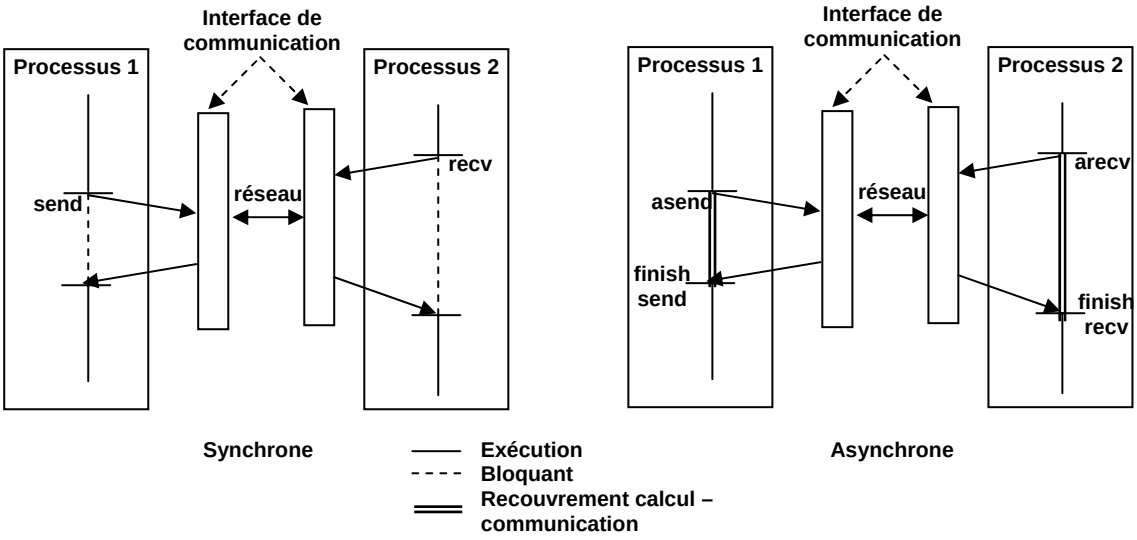
Cette sémantique garantit que l'émetteur et le récepteur doivent demander à communiquer (Figure 2.3a). Les délais d'attente sont maximaux. Du côté de l'émetteur, le processus qui envoie le message est bloqué jusqu'à ce que le message soit complètement envoyé au récepteur, ou bien mémorisé par la couche de communication. La zone émettrice peut stocker de nouvelles données. A l'autre côté (récepteur), le processus de réception est bloqué jusqu'à l'arrivée du message. On ne peut pas recouvrir la communication avec le calcul dans le cas de l'envoi/réception bloquant parce que les flots d'exécution des deux côtés sont dédiés à la communication. Ils sont bloqués jusqu'à la fin d'opération de communication. Les calculs ne peuvent pas être exécutés par ces flots pendant cette période.

Asynchrone

Dans ce genre de communication, les processus émetteurs et les processus de réception ne sont pas bloqués. Les opératuers de communication sont des requêtes pour initier les communications. Celles-ci sont réalisées ultérieurement par la bibliothèque. Les processus peuvent continuer d'autres calculs jusqu'à ce qu'un signal lui soit envoyé pour confirmer que la requête est bien transférée (ou arrivée). Le processus peut aussi attendre explicitement la fin d'une communication

Dans les deux sémantiques, la bibliothèque de communication doit implanter un contrôle de flux [AS83] [BST88] :

La première solution est de définir un protocole de retransmission. Dans cette solution, le message est conservé chez l'émetteur et il faut gérer la récupération et la retransmission du message.



(a) (b)

Figure 2.3 La communication synchrone et asynchrone

La deuxième solution est d'enregistrer le message dans un tampon intermédiaire (*« buffering »*) jusqu'à ce que le récepteur exécute la commande de réception. Le problème de la gestion du tampon est posé dans cette stratégie. Une autre solution est la signalisation à l'émetteur quand le récepteur est prêt : elle augmente le coût de communication par un aller/retour dans le réseau. Il existe aussi une solution intermédiaire qui consiste à utiliser des tampons du côté de récepteur pour recevoir des messages d'intention d'envoi. A la réception de cette intention, le récepteur exécute la commande de réception et l'émetteur envoie le message. Néanmoins, il n'est pas toujours nécessaire de payer la synchronisation sur les deux participants par l'utilisation des communications asymétriques.

Asymétrique

Dans ce cadre, la communication est découpée en des opérations de communication et de synchronisation entre deux participants (émetteur et récepteur). Nous étudions

quelques mécanismes de ce protocole : l'accès aux mémoires distantes - *RMA* (*Remote Memory Access*) [BK94], l'appel procédure à distance - *RPC* (*Remote Procedure Call*) [BN84] , les *messages actifs* (Active Message) [ECGS92] et *Fast Message* [LPC98].

RMA

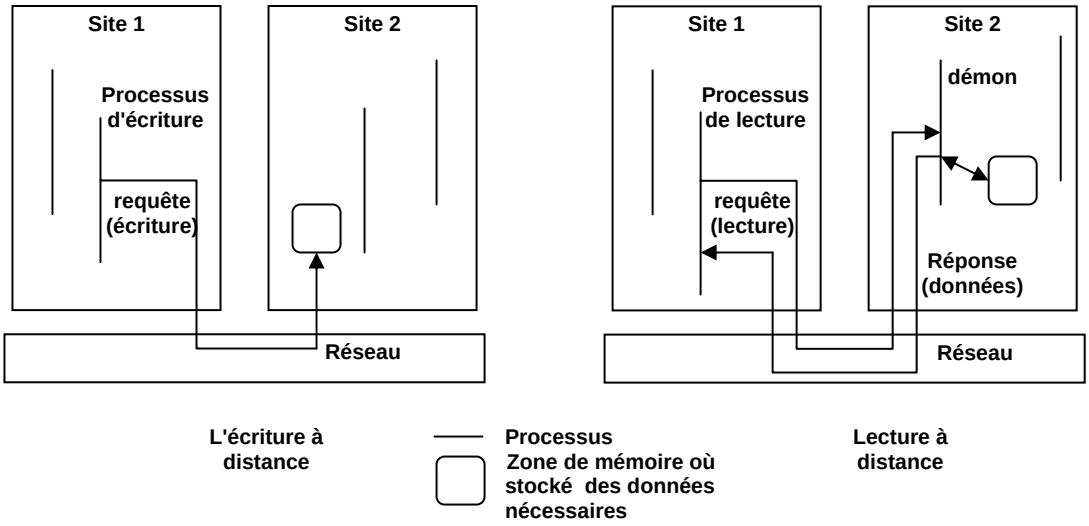


Figure 2.4 Le mécanisme RMA

La communication se base sur le concept de transfert des messages vers une mémoire distante à une adresse conventionnelle (Figure 2.4). Deux opérations fondamentales sont la lecture à distance et l'écriture à distance. Dans l'écriture à distance, un processus d'un site qui veut écrire une donnée dans la mémoire d'un autre site, envoie un message au site contenant la zone destinataire. Ce message contient l'adresse de la zone destinataire et les données. Les données sont écrites dans cette zone dès la réception de ce message. La lecture à distance fait un aller/retour dans le réseau. Le processus du site demandeur envoit un message au site qui porte les données à lire. Ce message contient l'adresse des données à lire, l'identification du site demandeur et l'adresse où les données vont être écrites. À la réception du message de demande, un message de réponse (une écriture à distance) qui contient les données concernant est envoyé en retour. Ces données sont écrites chez le lecteur à l'adresse reçue. Dans ce

modèle de communication, la cohérence des accès à la mémoire relève normalement de la responsabilité de l'utilisateur.

RPC

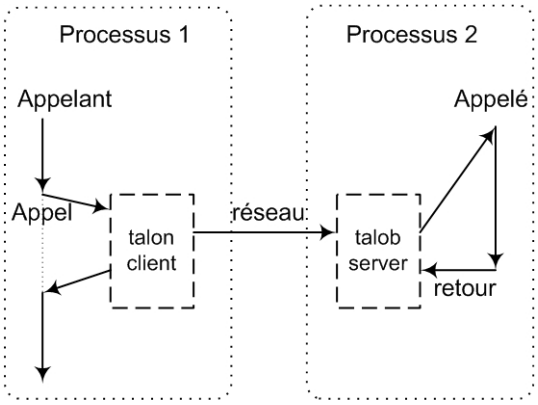


Figure 2.5 Le modèle de RPC

Cette méthode permet à un programme d'un processus 1 d'appeler une procédure sur un autre processus 2 (Figure 2.5). Le processus 1 est bloqué pendant cet appel et l'exécution de cet appel se passe sur processus 2. Les données sont communiquées de l'appelant à l'appelé dans les paramètres. L'échange de message est invisible au niveau du programme. Le noyau de communication prend en charge le transfère des données via le talon client (*client-stub*) et le talon serveur (*server-stub*). Le talon client est responsable de :

- La collection et assemblage des paramètres dans un message ;
- Du transfert des données avec le protocole de transport ;
- Du décodage des résultats de retour transmis à l'appelant lors du retour d’appel à la procédure.

Le talon serveur est responsable :

- Du désassemblage des paramètres et appel de procédure correspondante ;
- De l’emballage des résultats retournés grâce au protocole de transport sur le réseau.

Message actif

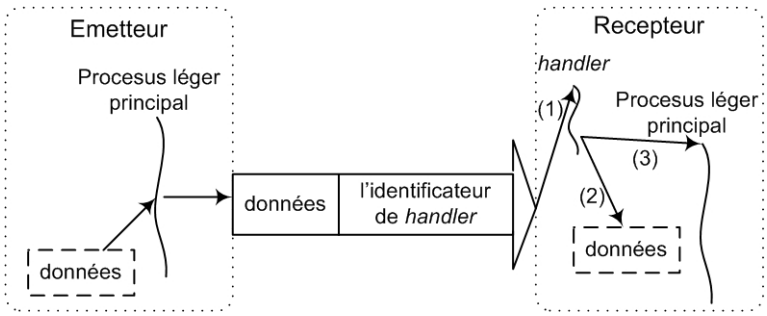


Figure 2.6 Le modèle de message actif

« Message actif » [ECGS92] est un protocole de communication unidirectionnelle qui n'a que des opérations d'envoi explicites (Figure 2.6). Un message actif contient l'identification de l'émetteur, de la fonction à exécuter (*handler*) et des paramètres d'appel à la fonction. Le rôle principal des messages actifs est d'extraire les messages du réseau et de les ajouter le plus vite possible dans le processus de calcul courant. Dès qu'un message arrive, il active la fonction exécutée correspondante du processus destinataire (Figure 2.6 : (1)). Cette fonction est un processus léger qui traite le message arrivé. En général, la tâche de traitement du message dans la fonction exécutée comprend deux étapes principales :

- Extraire les données arrivées du réseau et les intégrer dans la zone mémoire du processus (Figure 2.6 : (2)).
- Signaler l'arrivée du message au processus (Figure 2.6 : (3)).

La sémantique particulière des messages actifs permet de résoudre le problème de *« buffering »* dans le protocole envoi/réception *« traditionnelle »* décrit ci-dessus (*cf.* page 21). Effectivement, le protocole envoi/réception a besoin de tamponner des messages en attendant l'appel de l'opération de réception pour traitement ultérieur.

2.3.3 Communication collective

Un autre opérateur de communication est la *communication collective* [BI99] qui est utilisé lorsque des données sont partagées entre plusieurs processus. Les opérateurs de communication collective sont un des outils le plus puissants pour le traitement parallèle. Ils offrent les fonctionnalités nécessaires à la manipulation efficace des données sur les architectures distribuées dans le développement des programmes parallèles. À la différence des communications bipoints, lors d'une communication collective plusieurs (supérieure à 2) processus participent à la communication. Ce modèle permet d'échanger des messages entre processus d'un groupe pour coordonner des calculs.

Primitives

Les primitives de communication collective sont la diffusion (*broadcast*), la collection (*collection*) et l'échange total (*all-to-all communications*). Il y a deux types de diffusion, la **diffusion totale** (*broadcast*) et la **distribution** (*scatter*). Dans la diffusion totale, un processus d'un groupe propage un même message à tous les autres processus dans le même groupe. La distribution, suppose que les données sont coupées en n parties sur le processus émetteur, chaque partie est envoyée à chacun des autres processus dans le même groupe. L'opération de collection est divisée en deux types : le **regroupement** (*gather*) et la **réduction** (*reduction*). Le regroupement est l'inverse de la distribution. Il permet à un processus de récupérer un ensemble de données à partir d'autres processus. Dans la réduction, un processus récupère aussi des données mais de type accumulation, c'est-à-dire les valeurs reçues sont combinées en une seule valeur par une opération généralement associative et commutative. L'échange total est la combinaison d'une opération de diffusion et d'une opération de collection. Les processus d'un groupe envoient et reçoivent les données de tous les autres processus du groupe. La synchronisation entre les processus et la gestion de groupe de processus est source de problèmes dans la conception et implantation des primitives de communication collective. Chaque bibliothèque a sa propre stratégie de synchronisation. Par exemple, elle est asynchrone pour PVM [KGPS97], ou bien synchrone dans les cas de

MPI[SOH96] où toutes les opérations collectives sont des barrières de synchronisation. La gestion d'un groupe est statique ou dynamique. Néanmoins, la gestion d'un groupe qui se crée et évolue dynamiquement est difficile parce que la création dynamique d'un nouveau processus nécessite l'agrément de tous les autres processus. Normalement, les bibliothèques de communication collective ne permettent pas la création dynamique des processus.

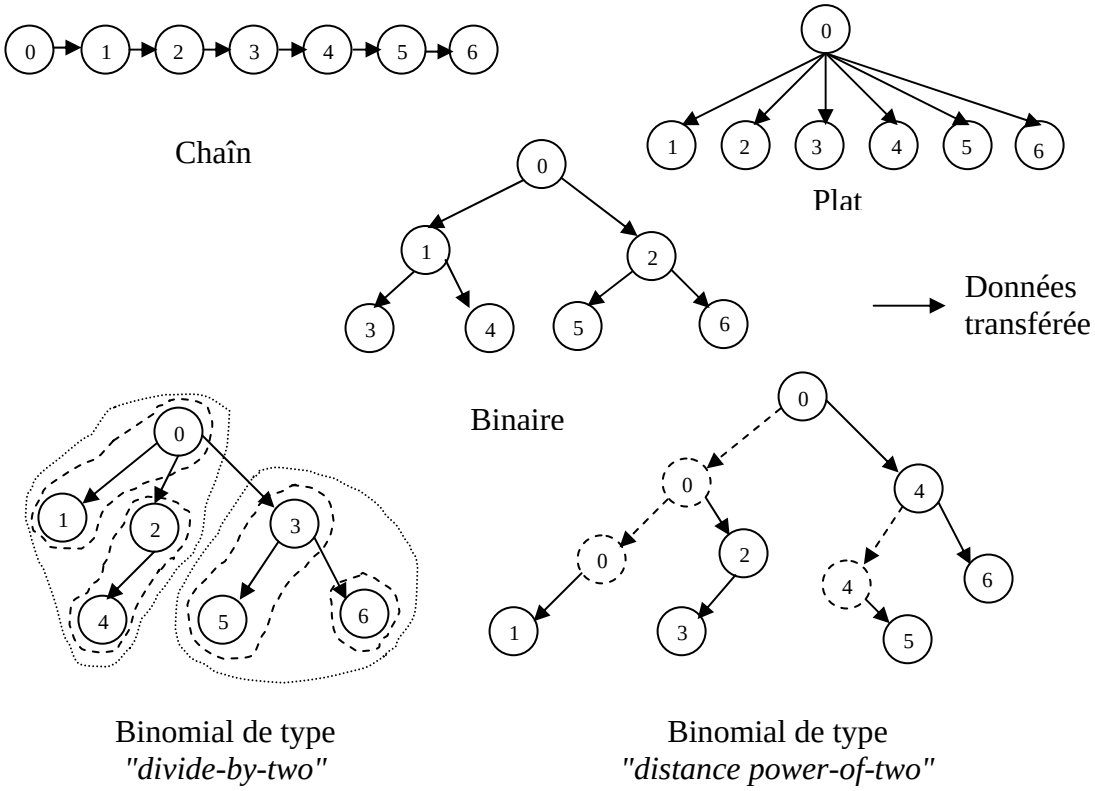


Figure 2.7 Différentes topologies de type d'arbre pour la diffusion

Algorithmes

Chaque opération de communication collective possède un ensemble d'algorithmes codant un schéma de communication entre processus. Les algorithmes les plus utilisés sont l'arbre plat, la chaîne, l'arbre binaire, l'arbre binomial et l'arbre de type α-tree [BI99]. Ces algorithmes et les autres sont expliqués dans [Hus99][VFD00]. Quelques algorithmes peuvent être utilisés dans plusieurs primitives de communication collective

comme l'arbre plat, la chaîne, l'arbre α. Néanmoins il existe d'autres algorithmes pour les opérations de communication collective, par exemple l'algorithme de Rabenseifner [Rab97] pour la réduction ou la réduction totale, l'algorithme de la réduction-distribution [Ian97] qui se base sur le modèle LogGP [AISS95], l'algorithme circulaire [VFD00] pour l'échange total. Il y a aussi d'autres algorithmes qui concernent la barrière [VFD00]. La relation entre l'opération primitive de communication collective et les algorithmes appropriés est présenté dans [VFD00][VFD01]. Le tableau 2.1 présente une synthèse de quelques algorithmes de type d'arbre de communications collectives. La Figure 2.7 présente quelques différentes topologies pour la diffusion.

Algorithme	**Type de communication collective** [VFD00]	**Complexité dans le modèle *pLogP*** [VFD00][Pig03]	**Avantage** [VFD00] [Pig03]
L'arbre plat	tous les types	$O(Pg(m)+L)$	quand L est élevée
La chaîne	tous les types	$O(P(L+g(m)))$	performance avec la méthode du pipeline
L'arbre binaire	tous les types	$O((g(m)+L)\lceil \log_2 P \rceil)$	minimiser le nombre d'étapes de communication
L'arbre binomial	diffusion totale et réduction	$O((g(m)+L)\lceil \log_2 P \rceil)$	comme l'arbre binaire mais dans la plupart de cas, il est plus performance que l'arbre binaire
α-tree	tous les types	$O((g(m)+L)\left\lceil \log_{\frac{1}{1-\alpha}} P \right\rceil)$	selon la valeur α, on a le performance de l'arbre plat ou l'arbre binomial.

Tableau 2.1 La comparaison des algorithmes de communication collective

2.3.4 Multiprogrammation légère et communication

L'intégration des communications et des processus légers a pour objectif de faciliter la programmation d'applications parallèles efficaces [FKT96]. Les avantages principaux de cette intégration sont :

- le surcoût faible, c'est-à-dire le coût de lancement, d'exécution et de gestion du processus léger est plus faible que celui du processus lourd. Alors, elle permet d'exploiter efficacement un parallélisme à grain fin dans les applications irrégulières [BGPP97].

- Elle permet également de recouvrir les attentes de communication d'un flot d'exécution par des calculs d'un autre qui cache de ce fait les durées de communication. En bref, elle supporte de réaliser efficacement le recouvrement de calcul-communication [FN92][Nam01].

En outre, d'autres avantages des processus légers tels que l'exploitation d'architecture SMP [EKB+92], l'évitement d'inter-blocages [Tan92], la facilité de programmation [FHM95] peuvent être exploités pour développer des applications parallèles et communicantes.

Les noyaux exécutifs implantent l'intégration des processus légers et des communications pour fournir un interface portable et performant à l'application de l'utilisateur. Elle assure aussi la coopération efficace entre les noyaux de communication et de processus léger. À présent, la plupart des noyaux exécutifs intègrent des bibliothèques de processus léger existants (par exemple *POSIX Threads* 1003c [IEEE94]) à des protocoles ou des bibliothèques de communication usuelles (par exemple *« socket »* [Ste94] ou *MPI* [SOH96]). Nous présentons donc quelques noyaux dans la prochaine section.

2.4 Bibliothèques d'échange de messages et noyaux exécutifs existants

La programmation parallèle a besoin d'outils efficaces de communication et portables pour garantir les performances sur des plates-formes différentes [CFT+94]. Actuellement, les systèmes d'exploitation fournissent des fonctions de communication entre ordinateurs sous forme d'un interface générique (*sockets*) [Tom91] et des protocoles d'Internet (TCP/IP) [Pos81][Gor95][Com91]. Ces fonctions atteignent un bon niveau de maturité et se présentent dans un grand nombre de systèmes. Il est néanmoins difficile de porter directement des programmes parallèles ou des environnements de programmation parallèle de haut niveau sur ces interfaces. L’objectif

des bibliothèques et noyaux exécutifs est de simplifier le portage de ces environnements sur ces protocoles et d'offrir un bon niveau de performance à l'exécution.

Du côté des bibliothèques de communication, il existe à l'heure actuelle plusieurs interfaces dont la bibliothèque *PVM* (Parallel Virtual Machine) [KGPS97] et le standard *MPI* (Message Passing Interface) [SOH96] sont les plus connus. Quant aux noyaux exécutifs, nous passons en revue certains noyaux courants comme *Nexus* [FKT96], *PM*^{2} [NM96], *PadicoTM* [DPP02] et *Athapascan-0* [BGPP97].

2.4.1 PVM

PVM [GBD+94][KGPS97] fournit une abstraction de machine parallèle qui comprend toutes les machines hétérogènes et homogènes différentes dans un réseau. Elle offre des fonctions de communication point à point et de communication collective entre processus. *PVM* propose le concept de groupe dynamique de processus, c'est-à-dire un processus peut s'insérer dans un groupe ou en sortir.

PVM comprend deux composants principaux : le démon *PVM* et la bibliothèque de routines qui implante l'interface de programmation de *PVM*. Dans l'approche *PVM*, l'utilisateur peut décider du déploiement des processus sur les machines. Les processus de *PVM* sont implantés par des processus lourds. La communication entre processus est basé sur l'envoi (PVM_SEND) et réception (PVM_RECV) de message.

2.4.2 MPI

MPI [SOH96] est un standard de communication par message qui définit un interface et les sémantiques des protocoles d'échange de messages. Il fournit un ensemble abondant et flexible d'opérations pour l'échange de message : communication bipoint bloquante et non bloquante, communication collective et communication asymétrique (lecture et écriture à distance). Néanmoins, comme *PVM*, les processus de *MPI* sont implantés par des processus lourds. *MPI* n'offre pas de messages actifs et ne supporte pas la multiprogrammation légère de manière claire.

MPICH [mpich] et *LAM* [LAM65] sont des implantations les plus répandues de ce standard.

MPICH

MPICH est développé au Laboratoire Argonne à Chicago, Etat-Unis. Il fonctionne sur plusieurs plates-formes, des architectures SMP aux grappes de multi processeurs.

Dans l'architecture de *MPICH*, il y a deux couches importantes : l'interface de programmation (API) et l'interface d'abstraction du matériel (ADI, *abstract device interface*). Cette première implante les fonctions de *MPI* comme des primitives de communication bipoint et de communication collective, de types de données et la gestion des groupes de communication. La couche ADI permet la portabilité des fonctions de *MPI* sur différentes architectures.

LAM/MPI

LAM/MPI, développé par l'université de l'Ohio aux Etats-Unis, permet d'exécuter des programmes sur différentes plates-formes de grappes homogènes ou hétérogènes. On trouve dans cette bibliothèque une implantation complète du standard *MPI 1.2* et des portions importantes du standard *MPI-2* comme la création et la gestion de processus, la communication asymétrique, les sorties/entrées parallèles de *MPI*.

La bibliothèque de communication *LAM* (version 6.5.9[3]) se divise en deux couches : l'interface de programmation (API) offrant des primitives *MPI* indépendantes du système de communication et la couche RPI (*Request Progression Interface*) implantant des modules qui permettent de réaliser des communications sur les environnements différents. Cette dernière fournit en outre deux types de transport : par démon et par *client-to-client(c2c)*. Le premier type utilise un démon à chaque nœud pour réaliser les communications. Ses avantages d'utilisation sont la simplicité du contrôle de l'évolution

[3] La plupart d'expérimentation dans cette thèse ont été réalisées avant la disponibilité de la nouvelle version 7.x

dynamique du nombre de processus participant à la communication, la gestion simplifiée de l'avancement des communications et la réduction du nombre de connexions réseaux gérées par nœud. Néanmoins, l'inconvénient le plus grave de cette approche est le surcoût des recopies des messages du processus utilisateur au démon et inversement. Quant au transport par *client-to-client*, il est implanté sous trois modes : *tcp*, *usysv* et *sysv*. Le premier mode utilise les *« socket »* du protocole TCP pour toutes les communications interprocessus. Par contre, les deux derniers utilisent les mémoires partagées offertes par SYSV de UNIX dans des communications interprocessus sur un même nœud physique et les *« sockets »* pour les autres communications entre différents nœuds physiques. La différence entre ces deux derniers modes est la manière dont sont de synchroniser les messages : l'utilisation de sémaphores SysV dans *sys-c2c* et de *spin-locks* dans usys-c2c.

2.4.3 Nexus

Le noyau exécutif *Nexus* [FKT96], développé par l'Argonne National Laboratory, Illinois (USA) est un exécutif parallèle destiné à des compilateurs plutôt que des programmeurs. L'objectif de ce noyau est de supporter l'exécution d'applications parallèles sur des environnements distribués hétérogènes.

Nexus est utilisé comme couche de compilateur et de communication par CC++ [CK93], Fortran M [FOT93], MPICH [GLDS96], nPerl [FO95], CAVEcomm [CLP+93] et Globus [FK97] qui sont des langages de programmation parallèle ou des environnements pour la programmation parallèle et distribuée. Les variétés du niveau d'utilisation de *Nexus* se basent sur ses différents niveaux d'abstraction.

Il est à noter que *Nexus* intègre la multiprogrammation légère et les communications en se basant sur les concepts de communication asynchrone et asymétrique, la création et la destruction de processus légers, un modèle d'espace d'adressage global entre processus.

Architecture logicielle

Nexus se compose de cinq concepts : ***nœud, contexte, processus léger, pointeur global*** et ***appel de service à distance*** (RSR, *remote service request*). Dans cette couche, un calcul s'exécute sur un ensemble de ***nœuds*** dont chacun est considéré comme une machine virtuelle et on peut attacher plusieurs contextes à chaque nœud. En principe, un ***contexte*** est comme un processus lourd et se compose d'un ensemble de processus légers exécutés dans un espace d'adressage du contexte.

Un ***pointeur global*** est un nom global à un ensemble de nœuds. C'est une structure de données qui représente un point final (*« endpoint »*) de communication et spécifie un destinataire auquel une opération de communication peut être adressée par une demande de service à distance. *Nexus* utilise les informations des protocoles de communication fournit par le pointeur global pour déterminer celui qui est le plus adapté à une demande de service à distance. Après sa création dynamique, le pointeur global peut être communiqué entre des nœuds via une demande de service à distance. En réalité, les calculs peuvent utiliser des pointeurs globaux pour construire des structures de données distribuées et complexes.

Un ***processus léger*** d'un nœud peut activer une action dans un contexte éloigné en utilisant un ***appel de service à distance*** qui est la seule primitive de communication de *Nexus*. L'*appel de service à distance* exécute une fonction spéciale (*« handler »*) dans le contexte indiqué par un pointeur global. Ce *« handler »* est invoqué de façon asynchrone dans ce contexte. L'*appel de service à distance* n'est pas un RPC (*Remote Procedure Call*) parce qu'il n'y a pas d'accusé de réception et de valeur de retour. De plus, le processus léger qui lance l'appel n'est pas bloqué.

Implantation

Lorsqu'un processus veut créer un autre processus d'exécution à distance, il utilise un *appel de service à distance* en y passant un pointeur global, un identificateur de *« handler »* et un tampon de données. La conversion et le codage de donnée sont exécutés en utilisant le pointeur global donné. Lors de l'exécution d'un *appel de service*

à distance, le pointeur global sert à déterminer le protocole de communication de bas niveau à utiliser. Ensuite, le *« handler »* s'exécute dans le contexte de destination avec l'adresse locale enregistrée dans le pointeur global et le tampon de message comme paramètres.

Au cas où *Nexus* est utilisé sans processus léger, son implantation s'appuie sur la disponibilité d'une *interruption* à l'arrivée d'un message de *RSR*. Sinon *Nexus* propose d'autres implantations pour détecter l'arrivée d'un message de *RSR* selon les propriétés du noyau de processus léger. Si le noyau ne permet pas de bloquer un processus léger d'exécution sur une communication, *Nexus* utilise un processus léger spécialisé qui vérifie de l'arrivée des messages de *RSR*, c'est donc la *scrutation par un processus léger spécialisé*. Par contre, un processus léger particulier est utilisé dans chaque contexte pour traiter des messages RSR qui viennent d'autres contextes. Le processus léger reste bloqué jusqu'à l'arrivée d'un de ces messages. De ce fait, cette méthode est appelée la *réception bloquante*. *Nexus* propose par ailleurs la méthode de *scrutation à la demande* dans laquelle la vérification d'arrivée des messages de RSR se réalise par un appel explicite à une procédure.

Nexus peut se baser sur des bibliothèques standards de processus légers comme POSIX et utiliser le protocole TCP/IP ou des bibliothèques d'échange de message comme MPI, PVM pour faire la communication dans les environnements distribués.

Nexus fournit un environnement efficace pour les langages parallèles et les bibliothèques de communication de haut niveau. Plus précisément, en intégrant la multiprogrammation légère et la communication, il permet de réduire le coût de communication, en particulier celui dû à la latence de réaliser la communication asynchrone. Grâce aux pointeurs globaux, on peut faire des communications avec des structures de données complexes et déterminer également le protocole de communication adapté. En implantant plusieurs types de demande à service distance, *Nexus* permet à l'application de choisir les actions appropriées de traitement des messages. Bien que *Nexus* procure la simplicité d'implantation et la portabilité, comme

montré dans [FKT96], *Nexus* vise notamment les applications qui opèrent avec des messages de grande taille.

2.4.4 PadicoTM

PadicoTM [DPP02], conçu au sein de projet PARIS - IRISA, est un environnement ouvert pour l'intégration de plusieurs exécutifs qui partagent des ressources réseaux. L'objectif de *PadicoTM* est de permettre d'exécuter des applications basées sur les modèles de programmations distribuée et parallèle, sur les grappes et les grilles de grappes, de façon indépendante aux infrastructures de réseaux sous-jacentes. En effet, *PadicoTM* fournit un environnement qui permet la cohabitation de plusieurs exécutifs dans un même processus. Cet environnement est utilisé dans le projet PACO++ d'objets *CORBA* parallèles [DPP01].

Architecture logicielle

La définition de *PadicoTM* s'inspire des composants logiciels et son architecture est constituée de trois types de module : module de noyau, module de service et module applicatifs. Le module de noyau se compose respectivement de *Puk*, *TaskManager*, *NetAccess* qui s'occupent respectivement de la gestion des modules, du multiplexage d'accès au réseau, et de la gestion des processus légers. Le deuxième module implante ses services à travers des interfaces d'accès aux réseaux (*VSock et Circuit*).

Implantation

La gestion des processus légers se base sur la bibliothèque de multiprogrammation légère *Marcel* [MN95][Nam01] alors que celle des communications de réseau réalisés par *NetAccess* s'appuie sur *Madeleine* [BMN99]. *NetAcess* exécute le multiplexage d'accès au réseau en gérant une scrutation (*polling*) unique pour expédier des messages arrivés aux modules qui ont besoins d'accéder aux réseaux sous-jacents. Ces modules enregistrent des procédures de rappel qui réagissent aux réceptions de messages. Ce mécanisme réalise un multiplexage logique. Il n'y a aucun conflit d'accès parce que tous

les modules utilisent le réseau à travers *NetAccess*, et il n'y a pas de concurrence grâce à cette scrutation unique.

VSock est une implantation d'un sous-ensemble de l'API standard des *sockets*. Cette implantation ne fournit qu'un transport de datagrammes en mode « zéro copie ». VSock est une couche de communication multi-protocole avec la sélection automatique (TCP/IP standard ou *NetAccess*/Madeleine).

PadicoTM offre des mécanismes efficaces pour intégrer les exécutifs de programmation parallèle et de programmation distribuée tels que *MPI* et *CORBA*. Il permet à ces exécutifs d'exploiter les performances des réseaux rapides actuels. Néanmoins, l'objectif de *PadicoTM* est de fournir une plate-forme ouverte pour intégrer des exécutifs plutôt que pour implanter directement des environnements de programmation parallèle à haute performance tels que *Athapascan* [GRC+98].

2.4.5 PM^2

Le noyau exécutif *PM^2 High-Perf* [NM96, Nam01, ABMN02, Aum02] est un projet qui a commencé aux laboratoires LIP (École Normale Supérieure de Lyon, France) et LIFL (Université des Sciences et Technologies de Lille, France) et qui se poursuit actuellement dans laboratoire LABRI à Bordeaux (projet RUNTIME). *PM^2* est un environnement de multiprogrammation distribuée conçu pour supporter efficacement des applications parallèles irrégulières sur des architectures distribuées. L'interface de *PM^2* fournit des fonctionnalités pour la gestion de haut niveau de parallélisme et l'exécution des applications sur des systèmes distribués hétérogènes.

PM^2 est utilisé par de nombreux projets en France et dans le monde. En particulier, *PadicoTM* [DPP02] utilise *PM^2* comme bibliothèque pour la gestion de processus légers et les communications. Nous présentons ces deux composants importants de *PM^2* : *Marcel* et *Madeleine*.

Marcel

Marcel est une bibliothèque de processus légers qui offre un large sous-ensemble d'interfaces *POSIX-Threads 1003c* [IEEE94] et qui propose des extensions par de la migration des processus légers et le mécanisme de « déviation » des processus légers [Nam01].

À présent, *Marcel* a trois implantations différentes : *MarcelMono*, *MarcelSMP* et *MarcelAct*. La version *MarcelMono* implante le processus léger en espace utilisateur (modèle N:1, *c.f* 2.2.2) avec un ordonnancement préemptif à priorité [DMN99]. Cette version ne tire aucun profit des différents processeurs d'une machine SMP. De plus, un appel système bloquant interrompt l'exécution du processus lourd et donc des processus légers. *MarcelSMP* est une implantation hybride (modèle M:N, *cf.* 2.2.2) qui peut utiliser plusieurs processus du noyau pour exécuter les différents processus légers de l'utilisateur. L'architecture SMP peut être exploitée mais il n'y a aucun support pour les appels systèmes bloquants [Nam01]. Dans *MarcelAct* [Nam01, DN03], on trouve une intégration d'un serveur pour gérer des événements de réseau. Ce mécanisme se base sur une extension du système d'exploitation (les activations) pour gérer les appels systèmes bloquants. Cette version ne fonctionne que sur Linux et nécessite un noyau Linux particulier.

Marcel est compatible avec plusieurs architectures de processeur tels que Intel, Sparc, Alpha, Power PC et IA-64, et avec plusieurs systèmes d'exploitation comme Linux, SunOS, AIX, Solaris, OSF, Unicos, IRIX et MS Windows.

Madeleine

Madeleine [ABMN02, Aum02] est un interface de communication conçue pour fournir un environnement portable, efficace et multi-protocoles aux applications utilisant la multiprogrammation légère sur les grappes et grilles de grappes.

L'interface de communication multi-protocole

Madeleine vise à permettre une utilisation efficace des fonctionnalités des logiciels et des matériels de communication. Cette bibliothèque fournit un contrôle explicite des

communications sur chaque protocole de réseau particulier et l'utilisateur peut changer dynamiquement entre ces protocoles. Ce contrôle se traduit par deux objets : le canal qui est associé à un protocole de réseau et la liaison qui représente une connexion bipoint fiable entre deux nœuds de communication.

Son interface de programmation fournit un petit ensemble de primitives pour la communication. Ces primitives sont l'envoi et la réception de message qui permettent à l'utilisateur d'insérer ou d'extraire des données. Un message, à ce niveau, est considéré comme une séquence de plusieurs zones de mémoire localisées dans l'espace d'adressage utilisateur. L'envoi d'un message se réalise en deux étapes : déterminer le processus destinataire (*mad_begin_packing*), emballer des données dans le message et envoyer ce message vers le destinataire (*mad_pack et mad_end_packing*). Les primitives d'emballage (*mad_pack*) offrent aussi différents modes pour garantir l'optimisation de la transmission des données selon le réseau utilisé et le type de données envoyées. La réception de message se déroule en trois étapes : initialisation d'un canal donné (*mad_begin_unpacking*), extraction des données (*mad_unpack*) et attente de fin de réception des données (*mad_end_unpacking*) [ABMN02]. L'extraction des données doit être effectuée dans le même ordre que leurs emballages et en utilisant le même mode.

La structure interne

Madeleine se compose de deux couches : l'une de bas niveau qui est une couche de portabilité et l'autre de haut niveau qui est une couche de gestion de tampon. La couche base se compose de modules de gestion des protocoles, chacun est spécifique à un protocole réseau particulier. Un module de gestion des protocoles contient d'un certain nombre de modules de transmission, chacun est responsable d'exploiter une stratégie de communication adaptée au réseau. La couche haute est indépendante des protocoles et elle se compose des modules de gestion des tampons. Chaque module est caractérisé par les types de tampon manipulés (dynamique ou statique). Il implante de plus un schéma d'agrégation spécifique pour regrouper des données afin d'exploiter la capacité de *scatter/gather* du protocole sous-jacent.

Support des grilles de grappes hétérogènes [Aum02]

Madeleine permet aussi d'exploiter souplement et efficacement des grilles de grappes hétérogènes au travers le concept de « canaux virtuels » qui permettent de réaliser des communications inter-grappes de manière transparente. Un canal virtuel se compose de plusieurs canaux réels et désigne un ensemble de nœuds passerelles. Madeleine utilise un mécanisme de réexpédition des messages sur les nœuds passerelles pour acheminer des messages sur un canal virtuel. Ce mécanisme se traduit par l'utilisation d'un module de transmission générique et d'un pipeline mulithreads.

Madeleine est déjà implanté sur les couches de communication VIA [Dun+98], BIP [PT98], SBP [BEMP93], Dolphin SCI [IEEE93], TCP/IP [Ste94], PVM [GBD+94] et MPI [SOH96]. De plus, il existe plusieurs logiciels qui se basent directement sur Madeleine tels que : MPICH/Madeleine [AMN01], Hyperion[ABH+01], 3D PM2-POV[BBD+99], PadicoTM [DPP02].

Madeleine fournit un interface de communication portable et efficace pour les environnements multiprogrammation légère qui permet d'exploiter efficacement les grappes et des réseaux rapides. Madeleine établi une passerelle entre les fonctionnalités offertes par les protocoles réseaux de bas niveau et le besoin d'abstraction de plus haut niveau. Son interface de programmation est de type d'envoi/réception explicite.

2.4.6 Athapascan-0

Athapascan-0 [Gin97, Car99, Pas99] est développé au sein du laboratoire ID-IMAG. C'est un noyau exécutif qui permet l'exécution de programmes parallèles irréguliers. Ce noyau a pour but d'intégrer la multiprogrammation légère et la communication. Athapascan0 fournit des opérateurs efficaces pour exploiter les différentes formes de parallélisme sur les nœuds SMP ou sur les réseaux de nœuds SMP. Il joue le rôle d'une couche assurant la portabilité de l'environnement de programmation Athapascan [GRC+98] [RGR03].

Athapascan-0 est utilisé comme couche de portabilité d'Athapascan-1 [Cav99] [Dor99] [Gal99], qui est un interface de programmation parallèle de type flot de données et qui gère le partage de charge. Il est aussi utilisé comme une couche de support aux applications : Takakaw [BGT99] et AHPIK [CCP99].

Modèle d'exécution et la réalisation

Le noyau *Athapascan-0* [Gin97] est une bibliothèque C basée sur une bibliothèque standard de processus léger POSIX 1003.c [IEEE94] et une bibliothèque standard de communication MPI. Le modèle de programmation d'Athapascan-0 est celui des réseaux dynamiques de processus légers communicants. Un programme Athapascan-0 s'exécute sur une machine parallèle virtuelle composée d'une ou plusieurs tâches. Les tâches *Athapascan-0* correspondent à des processus lourds. On peut lancer à la fois plusieurs processus lourds sur chaque cette machine. Le nombre de processus lourds est déterminé au moment du lancement de l'application et est invariable jusqu'à la terminaison de programme. Un processus lourd se compose d'un ensemble de processus légers dont chacun peut être créé localement ou à distance. Les processus légers locaux communiquent par la mémoire partagée. *Athapascan-0* fournit les primitives de synchronisation pour les processus légers locaux comme les verrous, les sémaphores et les variables de condition [Gin97].

La communication pour le partage de données entre les processus légers à distance est réalisée par échange de message. Les processus légers distants sont créés par appel à des services [BGPP97]. Un service est démarré par un appel de fonction d'un processus léger. L'exécution d'un service est réalisée selon le type *popup* [Car99] et celle de service urgent est réalisée selon le type *upcall* [Car99]. On utilise un tampon pour contenir le nom et les paramètres passés du service et du service urgent. De plus, *Athapascan-0* fournit le concept de format de données qui permet de définir le type et l'organisation en mémoire des données [Pas99].

Architecture logicielle

Athapascan-0 [Gin97, BGPP97, Car99, Pas99] est implanté en deux couches : l'interface de programmation et l'interface de portabilité. La première interface fournit les fonctionnalités aux utilisateurs et la deuxième est une couche nommée *Akernel* [Gin97] qui réalise l'intégration de MPI avec une bibliothèque de processus léger POSIX. *Akernel* permet aux processus légers d'exécuter, d'envoyer et de recevoir des messages. Cette couche offre un interface *thread-aware* pour la communication et elle se base sur deux sous-couches : API-MPI et API-Threads. La sous-couche API-MPI fournit un sous ensemble des fonctions MPI et la sous-couche API-Threads offre une interface qui homogénéise les différentes interfaces des bibliothèques de processus légers courants comme *POSIX, Marcel, DCE, Solaris*.

Implantation

La couche *Akernel* [Gin97] implante deux démons pour supporter l'exécution des deux types de service (service et service urgent). Elle implante aussi des mécanismes qui permettent d'utiliser des primitives de communication bipoint de *MPI*. De plus, elle fournit des primitives de communication *thread-aware* à partir des implantations non *thread-safe* de *MPI* via un verrouillage de protection. Les requêtes d'opérations non bloquantes sont gérées dans une liste baptisée *« liste de requêtes »*[Gin97, Car99]. Une condition *POSIX* [IEEE94] est associée à chaque requête. Les processus légers accèdent la *« liste de requêtes »* pour insérer ou extraire des requêtes. Si une requête n'est pas encore terminée, le processus léger est mis en attente sur sa condition. *Akernel* assure aussi les accès concurrents à cette liste via un *verrou unique*. Cette couche utilise un démon de communication qui réalise, périodiquement, la scrutation pour assurer la progression et la détection de terminaison des opérateurs asynchrones. Lors de la terminaison de la requête, ce démon extrait cette requête et envoie un signal au processus léger qui attend sur sa condition [Gin97].

De plus, *Athapascan-0* implante plusieurs modes d'activation du démon de communication pour gérer le problème de compatibilité du noyau de processus léger utilisé [Gin97].

Athapascan0-SMP

Il existe une version d'*Athapascan-0* sur multiprocesseurs nommé *Athapascan0-SMP* [Car99]. Cette version permet une exécution plus efficace sur les machines SMP. Elle résout des difficultés liées au parallélisme réel et à la synchronisation des processus légers. Cette version offre une solution pour les conflits d'accès au verrou de la liste des requêtes et à la mise en place de politiques de scrutation adaptées aux machines SMP.

Exclusion mutuelle

Afin de réduire la contention sur un *verrou unique* implanté dans la version *Athapascan-0, Athapascan0-SMP* [Car99] propose trois réglages :

- l'accès exclusif aux appels à MPI pour le rendre *thread-safe* par un *verrou spécifique*.
- Un verrou par requête de communication pour la phase de synchronisation à la fin de communication. L'avantage de cette suggestion est la diminution de la contention sur le *verrou unique* dans la version précédente d'*Athapascan-0* dans le cas où on a un certain nombre de processus légers communicants dans l'application. Cette solution utilise plus de mémoire par rapport à *Athapascan-0*.
- La gestion de la liste de requêtes. Dans l'implantation d'*Athapascan-0*, la gestion de communication est basée sur trois listes : la liste de requêtes déjà présentée, la liste des requêtes libres qui gère les requêtes avant la demande de communication et la liste des requêtes terminées qui gère les requêtes une fois la communication terminée [Gin97]. Dans l'implantation *Athapascan-0*, il y a un seul verrou pour assurer l'accès exclusif à toutes les listes. Alors, *Athapascan-SMP* réduit la contention sur ce verrou en utilisant un verrou par liste [Car99].

En outre, *Athapascan-SMP* remplace les verrous standard de *POSIX* par des verrous de type attente active (*spin lock*). L'attente active permet à un processus léger de tester de façon répétitive la disponibilité du verrou sans changement couteux de contexte.

Politiques de scrutation

Athapascan-SMP [Car99] propose aussi quatre modes de déclanchement de la scrutation pour détecter l'arrivée des messages. Le premier mode utilise un processus

léger spécialisé (démon de communication). L'inconvénient de cette approche est que ce processus léger est ordonnancé comme un processus léger de calcul. Ce mode provoque un problème de performance dans les applications dont la fréquence de communication est faible. Dans le deuxième mode, la scrutation est réalisée au niveau de l'application. La performance de ce mode dépend de la façon dont l'application exécute cette scrutation. Le troisième mode est la scrutation périodique qui est réalisée en fonction d'un délai fixe. Ce mode se base sur un mécanisme d'interruption périodique. Il est difficile de déterminer un délai approprié pour une application concrète. Dans le dernier mode, il n'y a pas de scrutation. Ce mode est utilisé dans le cas où l'application ne fait aucune communication.

Athapascan-0 et *Athapascan0-SMP* utilisent un mécanisme de scrutation pour assurer la progression et la détection de terminaison des opérateurs asynchrones. Dans Athapascan-0 et dans l'implantation par défaut de *Athapascan0-SMP*, cette scrutation est réalisée par démon de communication. L'inconvénient de cette approche, comme nous l'avons abordée ci-dessus, est le problème du surcoût d'ordonnancement pour ce démon. De plus, dans la plupart de systèmes d'exploitation disponibles dans le domaine public comme Solaris, Linux, le noyau de processus léger *POSIX* ne permet pas à l'utilisateur de maîtriser son ordonnancement. C'est pourquoi ce démon est ordonnancé comme un processus léger de calcul. Cette méthode cause un problème d'efficacité.

Le tableau 2.2 (page 43) suivante illustre la comparaison des systèmes présentés ci-dessus.

2.5 Bilan

Dans ce chapitre, nous avons présenté les concepts principaux des processus légers et des communications. Nous avons discuté des avantages de l'intégration de la communication avec la multiprogrammation légère pour la programmation des applications parallèles. Nous avons comparé quelques noyaux exécutifs courants qui supportent des environnements de programmation parallèles. L'utilisation de

bibliothèques de communication standard (par exemple *MPI*) donne quelques facilités dans l'implantation de ces noyaux. Néanmoins, l'utilisation d'une telle bibliothèque ne permet pas d'exploiter efficacement les processus légers dans les communications. De plus, l'utilisation des mécanismes de scrutation dans les noyaux exécutifs comme *Athapascan-0* (ou *Athapascan0-SMP*), provoque un problème de performance. Nous posons ici une question : comment peut-on marier le calcul et la communication en évitant le surcoût de scrutation ?

Nous constatons également qu'il existe de plus en plus de protocoles et des bibliothèques de communication pour exploiter les grappes et grilles de grappes. Les concepteurs doivent faire face à la question du développement des interfaces portables qui permettent d'implanter efficacement des environnements de programmation de haut niveau sur ces plates-formes cibles.

Dans le chapitre suivant, nous présentons INUKTITUK, un interface portable pour le langage *Athapascan*. Il fournit des primitives qui permettent de combiner efficacement des processus légers et des communications de type message actif.

	PVM	MPI	Nexus	PadicoTM	PM^2	Athapascan0
Niveau	bibliothèque d'échange de message	bibliothèque d'échange de message	noyau exécutif	noyau exécutif	noyau exécutif	noyau exécutif
Objectif	abstraction de machine virtuelle parallèle	standard de communication par échange de message	support d'exécution pour les applications parallèles sur architectures distribuées hétérogènes	environnement ouvert pour l'intégration de plusieurs exécutifs qui partagent des ressources réseaux	environnement de programmation parallèles portable pour la réalisation d'applications parallèle irrégulières sur des architectures distribuées	noyau exécutif intégrant la multiprogrammation légère et la communication qui permet l'exécution de programmes parallèles irréguliers
Architecture et Implantation	- démon PVM et interface de programmation - réseau dynamique de processus lourds communicants - envoi (synchrone) / réception (synchrone/asynchrone) explicites	- interface avec un ensemble riche et flexible de fonctions pour l'échange de message - groupe / contexte / communicateur/topologies de processus - envoi/réception explicites avec des modes variétés de communication. - *thread-unsafe*	- Pointeur global, RSR, nœud, contexte, processus léger - Génération de parallélisme: création locale de processus légers ou RSR. - RSR est la seule primitive de communication asynchrone. - Plusieurs implantations pour RSR: bloquant, scrutation ou basé sur interruption	- module de noyau, module de service et module applicatifs - gestion de processus légers se basant sur la bibliothèque Marcel. - gestion de la communication réseau s'appuiant sur Madeleine et exécute le multiplexage d'accès au réseau - interface virtuelle de *socket*	- noyau de processus légers Marcel et noyau de communication Madeleine - Marcel implante plusieurs stratégies d'ordonnancement : N:1 ; M:N ; « activation » - L'interface de Madeleine est envoi/réception explicite et asynchrone - Le modèle d'exécution de Madeleine se base sur les messages actifs.	- interface et Akernel - Akernet réalise l'intégration de MPI avec une bibliothèque de processus léger - création de processus légers locale et distant - primitives de communication de type échange explicite de message entre processus légers.
Avantages	- échange de message pour des milieux hétérogènes et homogènes - gestion dynamique des processus lourds	- standardisation - approprié aux calculs réguliers ou irréguliers à gros grain.	- support explicite de la multiprogrammation légère - simplicité d'implantation et portabilité au-dessus de plusieurs plates-formes de communication et de processus légers - pas de copies supplémentaires des messages	- support des exécutions simultanées de plusieurs exécutifs varées. - support des exécutions basées sur les paradigmes de programmation parallèle et distribuée - portabilité sur grappe et grille de grappes.	- support explicite de la multiprogrammation légère - portabilité sur grappe et grille de grappes. - pas de copies supplémentaires des messages - support des grilles de grappes hétérogènes	- support explicite de la multiprogrammation légère - portabilité sur les grappes - pas de copies supplémentaires des messages outre celles de MPI - adapté aux calculs irréguliers
Inconvénients	- inadapté aux calculs à grain fin. - tampon (« buffering ») - envoi/réception explicite	- pas de support explicite de la multiprogrammation légère . - inadapté aux calculs irréguliers à grain fin. - envoi/réception explicite - tampon (« buffering »)	- surcoût de l'envoi de messages de petite taille - inconvénients dus à l'implantation concrète des RSR	- inadapté à l'implantation directe des environnements de programmation parallèle à haute performance de type asynchrone tels que Athapascan	- envoi/réception explicite	- dépendance de MPI - scrutation pour assurer l'avancements des communications.

Tableau 2.2 Comparaison des bibliothèques de communication et des noyaux exécutifs

3

INUKTITUT : un interface générique pour un environnement de programmation parallèle

Ce chapitre est consacré à la présentation de l'interface INUKTITUT. Nous commençons par présenter les objectifs puis abordons son architecture logicielle. Nous étudions ensuite deux concepts principaux d'INUKTITUT : les processus légers et les réseaux. Nous discutons également de quelques modules qui ont été implantés comme SOCKNET, GMNET, ICS. Enfin, nous présentons les principales applications d'INUKTITUT.

3.1 Introduction

INUKTITUT est un intergiciel de portabilité pour la gestion des processus légers et pour l'exécution des communications pour un environnement de programmation parallèle sur grappes et grilles de grappes. Plusieurs types de protocoles et d'interface de communication peut être utilisés sur ces architectures : *TCP/IP* [Ste94], *MPI* [SOH96], *Corba* [Vin98], *Madeleine* [Nam01, ABMN02, Aum02]. L'objectif d'INUKTITUT est d'offrir un ensemble minimale d'interfaces assurant la portabilité des applications à hautes performances sur des grappes.

INUKTITUT définit deux classes d'interfaces pour exploiter au mieux des grappes de machines SMP (Symmetric MultiProcessor). L'une, gère le parallélisme intra-nœud entre les processeurs d'une machine SMP et se base sur des processus légers. L'autre, gère parallélisme inter-nœud entre les machines SMP et se base sur l'échange de messages actifs. INUKTITUT offre un interface qui permet de manipuler des processus légers, appelé PThread qui est inspirée à la fois de l'interface des processus légers de *POSIX 1003c* [IEEE94] et de Win32 [Hel94]. Les protocoles de communication définis dans INUKTITUT sont basés sur le principe des messages actifs [ECGS92]. De plus, INUKTITUT définit une bibliothèque de communication collective qui utilise des algorithmes variables comme l'arbre chaîne, l'arbre plat et l'arbre α et qui est basé au dessus des message actifs.

INUKTITUT est construit sous la forme de modules. INUKTITUT définit essentiellement des interfaces dont l'implantation sur une machine concrète (un noyau de processus légers ou une bibliothèque de communication) s'appelle un module. Les différents modules interagissent au travers de leurs interfaces. Cette articulation permet à la bibliothèque INUKTITUT d'être indépendante de l'architecture sous-jacente et réutilisable dans des domaines différents. Les modules ont été portés sur des systèmes d'exploitation plus répandus dans le cadre des grappes de calcul comme Linux, Unix, MacOSX et Windows. *Athapascan* [Rev04] et KaTools [Mar03] sont les principales applications d'INUKTITUT. La section suivante présente l'organisation logicielle d'INUKTITUT.

3.2 Architecture logicielle

3.2.1 Interfaces

Le niveau le plus haut de cet intergiciel est constitué des interfaces de programmation (API) génériques en C++. Ce niveau fournit l'abstraction d'une bibliothèque de processus léger et d'un réseau virtuel complètement maillé.

Processus légers

L'interface abstraite des processus légers offre la possibilité de créer/synchroniser des processus légers via des primitives simples et indépendantes du noyau des processus légers réellement utilisé dans une implantation concrête.

Réseaux

Les communications d'INUKTITUT se basent sur un réseau virtuel de processus complètement maillé. Ce réseau virtuel offre des primitives de communication simples basées sur l'échange des messages actifs [ECGS92] entre processus. Les communications sont unidirectionnelles et asynchrones. L'utilisateur peut programmer son application avec ces primitives indépendamment de la couche de communication réellement exploité. Le concept de réseau d'INUKTITUT sera étudié en détail dans la section 3.4.

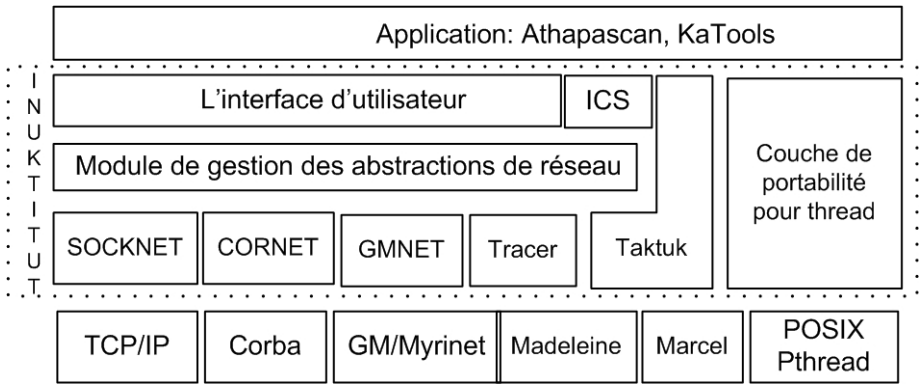


Figure 3.1 L'architecture logicielle d'INUKTITUT

3.2.2 Modules

Les différentes fonctionnalités d'INUKTITUT sont regroupées en modules. Les modules de réseau offrent une implantation des interfaces de type « réseau » afin d'offrir à une application INUKTITUT l'exploitation d'un matériel de communication particulier ou d'une bibliothèque spécifique. INUKTITUT implante en outre des modules pour

d'autres fonctionnalités comme l'utilisation des processus légers, le traçage d'événements, le déploiement des applications parallèles, etc. Pour l'instant, INUKTITUT contient les modules suivants : PThread, SOCKNET, CORNET, GMNET, ICS, Taktuk, Tracer. Nous présentons en bref la fonctionnalité de chaque module et quelques modules de réseau sont détaillés dans la section de l'implantation.

PThread est une bibliothèque de processus légers portables sur les APIs de processus légers comme *POSIX Threads* 1003c, *Marcel*, etc. L'interface de PThread est fondamentale et est utilisée non seulement par l'utilisateur mais aussi par les autres modules d'INUKTITUT.

SOCKNET est une implantation de communication qui se base sur l'interface *socket* SysV en utilisant *TCP/IP*. De nos jours, les *sockets* sont supportées dans tous les systèmes d'exploitation. De plus, le protocole *TCP/IP* est actuellement le protocole fédérateur des réseaux tel qu'Internet. Ce module sert à gérer l'ensemble des nœuds de calcul d'un programme INUKTITUT. Cette implantation permet de porter facilement INUKTITUT sur les réseaux de stations par example.

CORNET est une autre implantation de l'interface « réseau » d'INUKTITUT qui se base sur l'ORB *CORBA* omniORB3 de ATT [Vin98] comme couche de transport.

GMNET [1] [Her04] est une implantation de l'interface réseau d'INUKTITUT sur le réseau rapide Myrinet. L'objectif est d'exploiter au mieux les performances de ce type d'architecture.

ICS [2] [Pig03] est un interface de communication collective pour INUKTITUT qui offre des primitives non-bloquantes de type message actif « généralisé » de la diffusion totale et de la réduction. Ces primitives sont bien adaptées aux fonctions collectives du

[1] C'est un travail du stage de 3e année ENSIMAG d'Everton Hermann (Juin 2004)
[2] C'est un travail du stage DEA-IMAG de Laurent Pigeon (Juin 2003)

langage de haut niveau *Athapascan* [Rev04]. ICS est aussi un module qui implante cette interface au dessus de l'interface réseau et des messages actifs de base d'INUKTITUT.

Taktuk [MR01,Mar03] est un module qui implante l'interface réseau d'INUKTITUT. C'est aussi un ensemble d'outils qui permettent le déploiement parallèle de processus et la diffusion de fichier sur un grand nombre de processeurs. L'originalité du module Taktuk et que le lancement des processus sert aussi à établir les connexions entre les différents processus d'un programme INUKTITUT : les arbres de lancement des processus servent à construire les connexions initiales d'un réseau INUKTITUT. Cette bibliothèque offre de trois fonctionnalités : un lanceur, un routeur et une gestion des entrées/sorties dont les services consistent à démarrer un ensemble de processus interconnectées par un réseau logique, permettre aux différents processus de communiquer entre eux par le routage logique, et à gérer d'entrées/sorties et de signaux. L'implantation du réseau Taktuk se base sur l'interface de socket de SysV en utilisant *TCP/IP*.

Tracer [Gui04] est un module qui permet de tracer des événements logiciels et les performances d'application. Lors de l'exécution d'un programme INUKTITUT, ce module permet de gérer des fichiers stockant des événements générés à l'exécution des programmes. Les enregistrements réalisés par le module sont optimisés de façon à ne pas trop perturber le programme. La prise des traces peut être désactivée pour supprimer le surcoût dû à leurs enregistrements.

L'architecture logicielle en couche d'INUKTITUT avec une interface abstraite pour chaque module permet de masquer au programmeur les différentes mises en œuvre des APIs sur différentes bibliothèques (POSIX Thread, GM/Myrinet, SOCKNET TCP/IP).

Les sections 3.3 et 3.4 suivantes présentent les concepts principaux d'INUKTITUT : les processus légers et les réseaux (au sens INUKTITUT).

3.3 Les processus légers

L'un des objectifs d'INUKTITUT est d'offrir un ensemble d'interfaces assurant la portabilité des applications qui utilisent la multiprogrammation légère. Afin d'atteindre ce but, INUKTITUT défini une couche de processus léger portable **PThread** (*Portable Thread Package*) qui s'appuie sur les bibliothèques standard répandues comme *POSIX 1003.c Thread* [IEEE94] et *Windows Thread* [Hel94]. Nous présentons son modèle et son interface dans cette section.

3.3.1 Modèle

Comme nous l'avons présenté dans le chapitre 2 (*cf.* section 2.2.3, page 17), la bibliothèque standard *POSIX 1003.c* Thread est largement acceptée aujourd'hui et la plupart des implantations dans les systèmes de type Unix (Linux) se conforment à un sous ensemble de cette norme. Dans le cadre de notre travail, nous sommes particulièrement intéressés par la portabilité. C'est pour cela que la couche **PThread** d'INUKTITUT est construite au dessus de standard *POSIX 1003c Thread* et elle profite de la disponibilité et des fonctionnalités d'un noyau *POSIX*. Elle hérite aussi des caractéristiques et des contraintes d'implantations sur une architecture et un système d'exploitation visé. Cette approche permet de porter **PThread** sur différents noyaux de processus légers qui implantent le standard *POSIX*.

Bien qu'il y ait trois modèles d'implantation de processus légers (*cf.* section 2.2.2, page 15) : l'implantation des processus légers de **PThread** offre le modèle 1:1. L'avantage de cette approche est que l'implantation est simplifiée et qu'il est très portable. De plus, elle permet d'exploiter les architectures SMP. L'inconvénient est que le coût de gestion des processus légers dans PThread sera plus important qu'un modèle de type (M :N) et des processus légers gérés dans l'espace utilisateur.

3.3.2 Interface

L'interface **PThread** (PTH) se base sur un sous ensemble des APIs (*Application Programming Interface*) de *POSIX 1003c* et de *Win32 Thread*. **PThread** fournit des fonctions de base comme la création de processus légers, la libération d'une ressource (*yield*), la mise en sommeil du processus léger courant, la terminaison ainsi que les fonctions de synchronisation comme le verrou (*mutex*), les objets de conditions.

Les fonctions de base

Les primitives de l'interface **PThread** peuvent être utilisées entre l'initialisation et la terminaison de cet environnement comme suit :

```
// Intialisation
PTH::initialize(&argc, &argv);
……
// Terminaison, le module PThread n'est plus utilisable
PTH::terminate();
```

Il y a deux façons de créer un processus léger. La première utilise la méthode *create* de l'interface de **PThread** :

```
PTH::Thread::create(id du processus léger, fonction exécutée, paramètres,type);
```

La seconde façon crée un processus léger est par activation d'un objet de type *PTH::ThreadModel* qui est un emballage minimal d'objet C++ au processus léger natif. N'importe quelle classe avec la méthode *« run »* peut être exécutée comme un processus léger (Exemple 3.1) si elle dérive de la class PTH ::ThreadModel

Exemple 3.1

```
class   TOTO : public PTH::ThreadModel<TOTO> {
   ……
   void   run(){  ……  }
}
TOTO * th = new TOTO();
th->start; //l'activation un processus léger dérivé
```

Les fonctions de synchronisation

La bibliothèque **PThread** fournit aussi des fonctions de synchronisation comme le verrou (*mutex*) pour garantir l'exclusion mutuelle, et des objets conditions pour la signalisation entre processus légers :

Mutex:	*Condition:*
PTH::Mutex::init(); *// prise du verrou* PTH::Mutex::lock(); … *//libération du verrou* PTH::Mutex::unlock(); PTH::Mutex::destroy();	PTH::Condition::init(); … *// attente sur la condition* PTH::Condition::wait(PTH::Mutex &mutex); *// signalisation de la condition et réveil d'au moins 1* *//processus légers* PTH::Condition::signal(); *// signalisation réveil tous les processus legers en* *// attente* PTH::Condition::broadcast();

De plus, la bibliothèque **PThread** a un ensemble de fonctions pour faciliter la programmation concurrente : le compteur atomique *PTH::AtomicCounter*, le compteur de synchronisation *PTH::CountDown*, le sémaphore *PTH::Semaphore*, etc. Ces fonctions permettent à l'utilisateur de synchroniser plus efficacement des processus légers. La primitve *PTH::AtomicCounter* est un compteur dont les différentes implantations utilisent des instructions spécifiques aux processeurs exécutant un programme (Exemple 3.2).

Exemple 3.2

```
PTH::AtomicCounter   ct;
++ct;              //augmentation du compteur par 1
ct+=<value>;  // augmentation du compteur par une <valeur>
```

Le compteur *PTH::CountDown*, quant à lui, fournit un compteur décroissant qui permet de bloquer un processus légers jusqu'à la valeur zéro. Cette primitive est très efficace dans le cas de synchronisation de plusieurs processus légers avec un processus léger principal. Dans l'exemple 3.3, le processus léger principal initialise d'abord la valeur de *cd* par *n* et puis il crée *n* processus légers qui exécutent la fonction *ThreadMem*. En suite, il attend des terminaisons de tous les processus légers membres

par appel *cd.wait()*. À la fin de l'exécution de chaque processus léger, on réduit la valeur de *cd* par un : *--cd*. Au moment où la valeur de *cd* arrive à zéro, le processus léger principal est débloqué. L'utilisation de cette primitive dans ce cas pourrait être simulé par une utilisation des primitives de *mutex* et de *condition* mais plus complexe.

Exemple 3.3

```
PTH::CountDown  cd;
void ThreadMem(…){
      …
      --cd;
}
void ThreadsMain(…){
      …
      cd.init(n);
      //créer n processus légers qui exécute la fonction Threads
      cd.wait();
      …
}
```

Afin de bien gérer des régions critiques, INUKTITUT fournit la classe *PTH::CriticalGuard* qui permet de verrouiller la région critique durant la portée lexicale de la définition d'un objet (Exemple 3.4).

Exemple 3.4

```
// - déclaration mais sans d'initialisation
static PTH::Mutex mon_mutex;

void toto() {
 …….
  {
      PTH::CriticalGuard  guard(mon_mutex);
      // On veut que ce bloc soit exécuté dans l'accès exclusif
       .......
  } // Ici le mon_mutex est libéré quand « guard » est détruit
 …….
}
```

En conclusion, l'utilisation de la bibliothèque de processus léger d'INUKTITUT est plus facile que celle de POSIX grâce à l'utilisation de construction C++ (classe, méthode). Les performances de la bibliothèque **PThread** sont présentées dans le chapitre suivant.

3.4 Les réseaux

Le réseau est un concept dominant dans INUKTITUT. Dans cette section, nous présentons en détail son organisation et son interface.

3.4.1 Organisation d'un réseau

Un réseau d'INUKTITUT est un ensemble de nœuds qui sont logiquement complètement connectées (Figure 3.2). Un nœud est un processus lourd. Chaque nœud dans le réseau possède un entier unique. Un nœud est connecté à plusieurs canaux de communication. La communication entre deux nœuds est faite au travers d'un canal de sortie appelé *« OutChannel »*. Par exemple, un nœud *i* qui veut envoyer un message vers un nœud *j* doit tout d'abord récupérer un des canaux de sortie vers le nœud *j*.

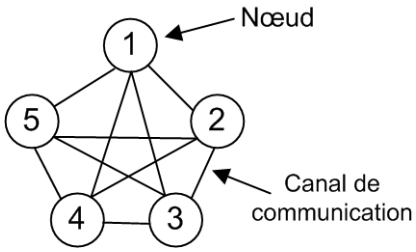


Figure 3.2 L'organisation d'un réseau d'INUKTITUT

Un canal de sortie garantit la fiabilité de la transmission et l'ordre FIFO des messages. Un objet *« OutChannel »* contient la description d'un nœud distant. Cette description dépend de l'implantation d'INUKTITUT sur un type de réseau particulier. Par exemple, dans le cas d'utilisation des *sockets*, il s'agit d'une adresse IP et un port. Les nœuds, les canaux de communication forment le réseau dans INUKTITUT.

3.4.2 Initialisation d'un réseau

L'initialisation d'un réseau est liée au lancement des applications parallèles. Le lancement d'un programme parallèle est l'opération de transition entre un état où une

requête de lancement est émise et l'état où ce programme commence son exécution sur tous les nœuds. Ce lancement se déroule en 4 étapes principales :

- La première étape est le lancement des nœuds. Chaque nœud va exécuter l'initialisation de ses paramètres. (Figure 3.3a)
- La seconde étape consiste à établir les liaisons de communication de contrôle. (Figure 3.3b)
- La troisième étape consiste à établir les liaisons de communication de calcul. Ces liaisons peuvent être utilisées sur un réseau physique différent du réseau des liaisons de communication de contrôle.
- La dernière étape démarre du programme parallèle lié à chaque nœud.

Par exemple, dans la figure 3.3 le réseau de contrôle peut être établi sur un réseau *CORBA* mais le réseau dédié au calcul peut être établi sur un réseau *socket* de type *TCP/IP*.

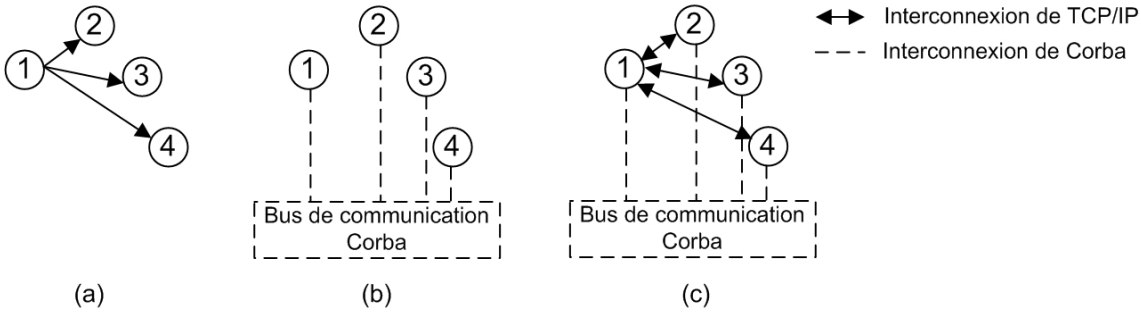


Figure 3.3 Lancement d'une application parallèle et illustration d'une initialisation avec deux modules réseaux

Afin d'atteindre la portabilité et l'efficacité dans le lancement et l'exécution des applications parallèles, les réseaux virtuels INUKTITUT sont séparés en deux catégories : les réseaux primaires et les réseaux secondaires. Les réseaux primaires regroupent les environnements qui permettent de déployer des nœuds et d'établir les liaisons de contrôle entre eux. Les réseaux étant capables de s'initialiser à partir d'un ensemble de nœuds communicants existants sont baptisés réseaux secondaires. Par

exemple, dans la figure 3.3, le réseau primaire est de type *CORBA*, et le réseau secondaire est de type *socket* de *TCP/IP*. Un réseau peut être à la fois primaire et secondaire. Dans ce cas, il peut s'initialiser par la création d'un nouveau réseau virtuel ou par la duplication d'un réseau virtuel existant.

3.4.3 Communication

La communication entre deux nœuds du réseau INUKTITUT par échange de messages actifs est asynchrone et unidirectionnelle, c'est-à-dire que l'envoi des ***messages*** est explicite dans le code utilisateur de l'émetteur (Figure 3.4). Du côté du récepteur, la réception des messages est implicite, c'est-à-dire qu'il n'attend pas l'arrivée des messages au niveau utilisateur. L'application ne déclare qu'un ***service*** de réception des messages. Dès qu'un message arrive au récepteur, le système d'INUKTITUT le met dans une zone mémoire allouée et il exécute ce service. La gestion de la mémoire des données dépend des différents ***protocoles*** d'INUKTITUT. Il n'existe pas de synchronisation entre deux participants à la communication. Durant l'envoi, l'émetteur doit garantir l'intégrité des données du message jusqu'à la fin de l'émission. À la fin de l'émission du message, une procédure de l'utilisateur est activée, c'est la fonction crochet (*callback*) qui signale qu'on peut réutiliser les données du message. Elle sert aussi à transmettre à l'application un code d'erreur possible de la communication.

La communication inter-nœud d'INUKTITUT par échange de messages actifs possède quelques avantages. D'abord, nous pouvons éviter le problème de *buffering* dans les modèles *envoyer/recevoir* traditionnels qui ont besoin d'allouer des tampons pour stocker les données reçues si l'appel de réception correspondant n'est pas encore réalisé (*cf.* section 2.3.2, page 19). Nous pouvons en outre éviter le problème de la copie mémoire. En effet, du côté de la construction d'un message actif, nous n'avons aucune copie mémoire pour les données envoyées de l'utilisateur (sauf les données immédiates qui sont normalement de petite taille). Du côté de l'extraction des données arrivées, nous avons de 0 à 1 copies mémoires selon le protocole choisi. La communication asynchrone et unidirectionnelle permet d'intégrer aisément la communication au calcul.

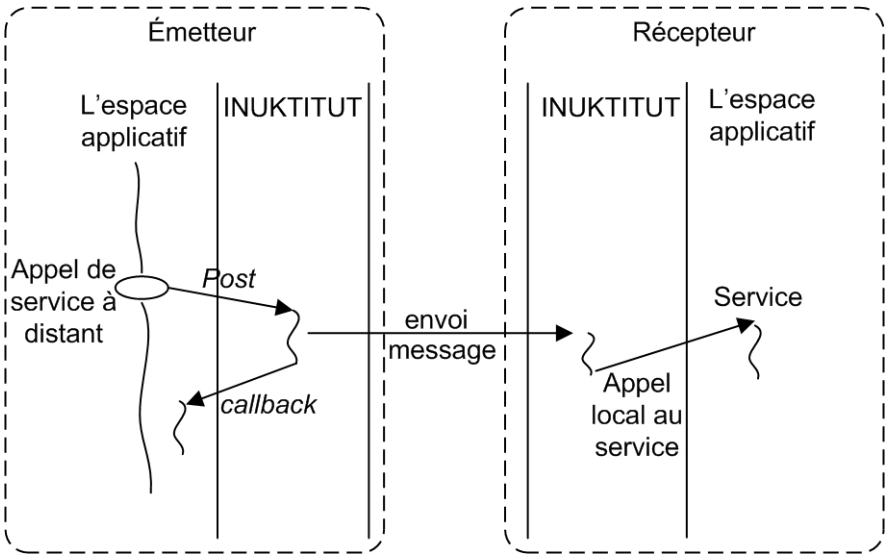


Figure 3.4 Principe de la communication par échange de message actif dans INUKTITUT

3.4.4 Message

Structure

Un message actif INUKTITUT a deux parties : une partie entête et une partie donnée de l'utilisateur. Dans la partie entête, il y a une partie fixe qui enregistre les informations nécessaires pour la livraison du message et une partie variable (*Extended Header*) utilisé par certaines implantations des réseaux. La partie donnée est aussi constituée de deux parties : l'une pour les données immédiates (IA) et l'autre pour les données différées (DA). La partie IA utilise le tampon du message pour stocker par recopie les données de l'application. La partie DA gère des pointeurs vers de régions mémoires d'application où sont stockées les données qui seront envoyées ultérieurement au moment de l'émission (Figure 3.5).

Formation d'un message

D'abord l'utilisateur définit un message avec des paramètres comme le type du protocole de message actif, le service qui devra être activé sur l'arrivé du message, etc. Ces informations sont enregistrées dans l'entête du message. Ensuite l'utilisateur décrit une séquence d'instruction d'emballage des données. Deux types de données sont

possibles : les données immédiates et les données différées. Les données immédiates sont copiées dans la partie IA via la primitive *pack*. Les adresses et les tailles des données différées sont enregistrées dans la partie DA via la primitive *put*.

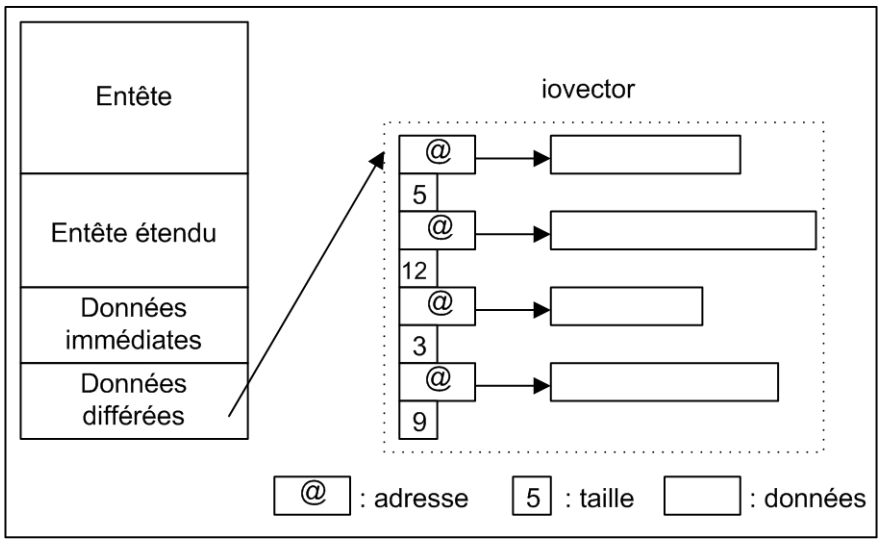


Figure 3.5 La structure d'un *message*

L'envoi et la réception d'un message

L'envoi d'un message est exécuté à travers un canal de sortie (*cf.* section 3.4.1, page 54). Nous étudions l'ordre d'envoi et de réception d'un message par réseau d'INUKTITUT (Figure 3.6).

Du côté d'émetteur, on envoie d'abord l'entête de message et puis son entête étendu (1) et (2). Ensuite, le bloc des tailles des données différées (DA) qui stocke la taille de chaque donné différée est émis (3). Ce bloc est construit grâce à des informations dans l'entête du message. Ces tailles de données différées seront utilisées dans l'allocation des zones mémoires de réception. Après cela, la partie des données immédiates (IA) de message est expédiée (4). Enfin, les données différées sont envoyées grâce aux informations (adresse, taille) dans la partie DA du message (5).

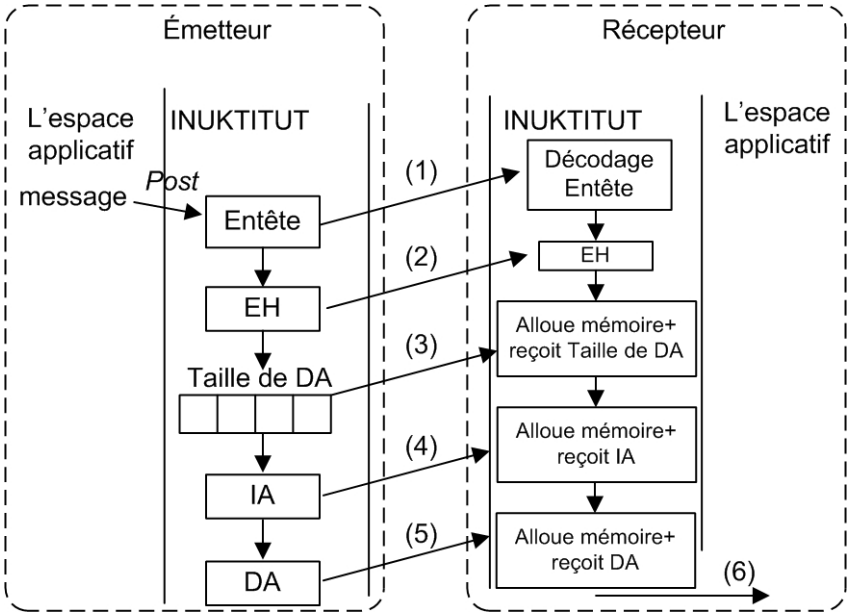


Figure 3.6 L'ordre d'émission et de réception

Du côté de récepteur, le prologue de la réception du message est l'admission de l'entête du message. Cette entête est décodé pour obtenir les informations nécessaires aux réceptions des autres parties du message. On reçoit ensuite l'entête étendu suivi par le bloc des tailles des données différées. Le réseau INUKTITUT alloue la mémoire pour enregistrer les données immédiates qui sont reçues par la suite. Après cela, on construit des *iovecteurs* grâce aux informations de l'entête et au bloc des tailles des données différées. Les zones de mémoire où seront stockées les données différées sont alors allouées. La manière d'allouer et de gérer ces zones (par le réseau INUKTITUT ou par l'application) dépendent du protocole choisi et sera détaillé dans la section 3.4.6. Enfin, c'est la réception des données différées qui sont extraites du réseau.

À la fin du processus de réception d'un message, le réseau INUKTITUT appelle le service (6) associé dont l'identificateur est enregistré dans l'entête du message.

3.4.5 Service

Un service est un objet défini au niveau utilisateur. Les services sont gérés par un système de gestion de service d'INUKTITUT. Chaque service a un identificateur unique

qui est donné lors de son enregistrement au système. Un service contient deux méthodes : ***call*** et ***get_iovect***. La méthode ***call*** d'un service est appelée par le sytème d'INUKTITUT après la réception d'un message qui contient l'identificateur de ce service. Les paramètres d'appel de la méthode ***call*** sont les données contenues dans le message. La méthode ***get_iovect*** est utilisée dans le cas du protocole *« write and signal »* pour décoder les adresses de définition des zones mémoires de réception pour les données différées.

3.4.6 Protocoles de message actif

INUKTITUT définit trois protocoles différents de communication des messages actifs pour le décodage des données reçues. Ils permettent de préciser la manière d'extraire et de gérer la durée de vie de ces données. Ce sont *« active message »* AM, *« write and signal »* WS, *« allocate, write and signal »* AWS. Ses différences sont présentées dans le tableau 3.1. L'allocation du tampon pour extraire ces données peut être prise en charge par l'utilisateur ou par le système INUKTITUT. Ce tampon est désalloué à la fin de l'exécution de la fonction du service ou par l'utilisateur. Dans ce tableau, un durée de vie *utilisateur* signifie que la donnée reçue est contrôlée (allouée ou libérée) par l'utilisateur.

Protocole	Allocation	Durée de vie
active message	Système d'INUKTITUT	Service
write and signal	Utilisateur	Utilisateur
allocate, write and signal	Système d'INUKTITUT	Utilisateur

Tableau 3.1 Caractéristiques des différents protocoles INUKTITUT vis-à-vis des données du message reçu.

Protocole *« active message »*

En utilisant le protocole *« active message »* (Figure 3.7), toutes les données reçues d'un message sont stockées dans des zones de mémoire gérées par le système

INUKTITUT (1). L'appel du service est réalisé dès que tous les composants de ce message sont arrivés (2). À la fin d'exécution de ce service, ces zones de mémoire sont désallouées par le système (3). Le contenu du message n'est valide que pendant l'exécution du service. Afin d'utiliser ces données après cette exécution, il faut les copier dans des mémoires gérées par l'utilisateur. Dans ce cas, on a une copie de mémoire. Il faut noter que cette durée de vie des données peut être utilisée afin d'offrir un mécanisme efficace d'allocation mémoire par pile.

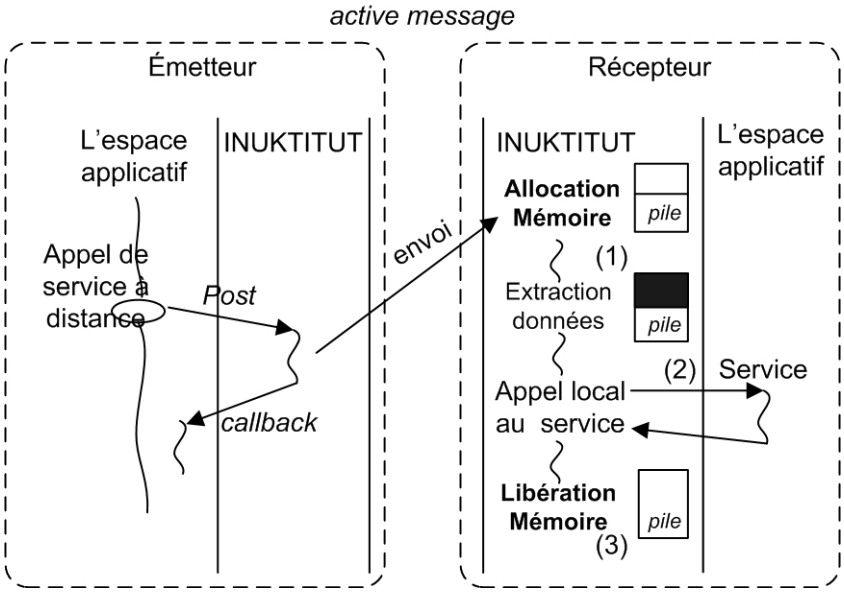


Figure 3.7 Le protocole *« active message »*

Protocole *« write and signal »*

Dans le protocole ***« write and signal »*** (Figure 3.8), les données différées extraites sont déposées dans des zones de mémoire qui sont allouées par l'utilisateur au lieu du système INUKTITUT. Après la réception de l'entête du message ainsi que des données immédiates, une fonction utilisateur est appelée (*get_iovect*) pour décoder les adresses où devront être stockées les données différées reçues (1). Ensuite, le système INUKTITUT extrait ces données dans ces zones (2) et appelle de service (call) (3). Ces zones seront désallouées par l'utilisateur par la suite. En bref, ce protocole permet à

l'utilisateur de contrôler la durée de vie des zones de mémoire stockant des données différées, on a donc zéro copie de mémoire au niveau d'INUKTITUT.

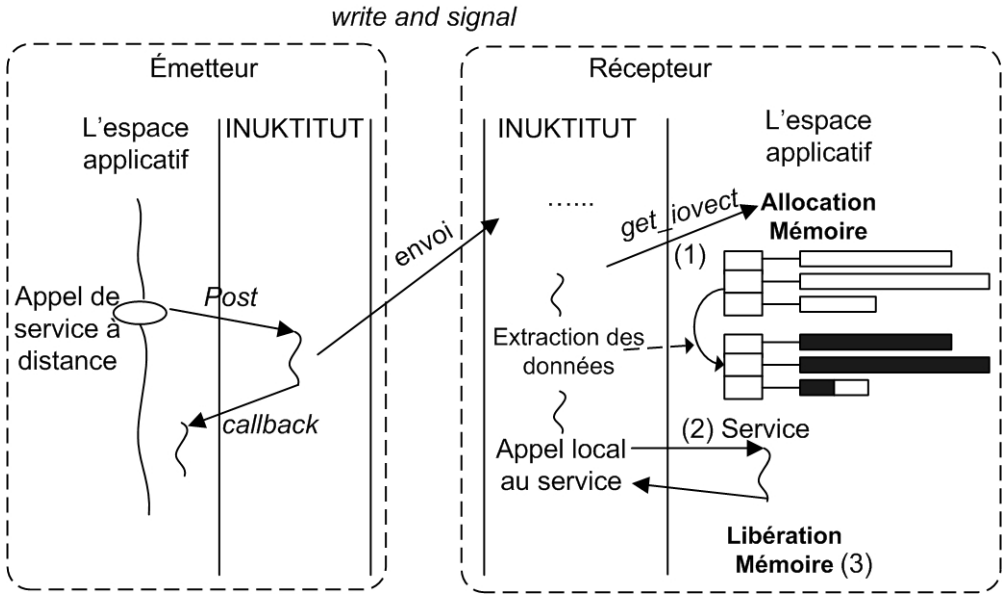


Figure 3.8 Le protocole *« write and signal »*

Protocole *« allocate, write and signal »*

Le protocole *« allocate, write and signal »* est un mélange des deux protocoles au-dessus. Dans ce protocole, les zones de mémoire pour des données différées sont allouées par le système INUKTITUT comme le protocole *« active message »*. Néanmoins, cette zone est gérée par l'utilisateur comme le protocole *« write and signal »*. C'est-à-dire la libération de ces zones est de la responsabilité de l'utilisateur et aucune copie mémoire n'est faite dans INUKTITUT.

3.4.7 Interface

L'utilisation d'un réseau d'INUKTITUT commence par la phase d'initialisation de ce réseau comme :

```
ACOM::Network::initialize(&argc, &argv);  // initialisation d'un réseau primaire
ACOM::Network::initialize(&argc, &argv, primary_net); // initialisation d'un réseau secondaire
```

L'envoi d'un message actif est réalisé via l'appel d'un service distant. La déclaration et l'inscription d'un service sont données dans l'exemple 3.5 suivant.

Exemple 3.5

Déclaration

```
class PongService: public AC::ServiceWS {  // ou ServiceAM, ServiceAWS
public:
void* get_iovect(
        AC::Network::Receipt* receipt,
        AC::Network::Header* header,
        AC::IOVectEntry* iov,
        AC::IOVectSize iov_sz
        )
 {
    iov[0].data = (void*) new char  [iov[0].len]; // pour WS, par exemple
   return 0 ;
 }

 void call (
       AC::ComFailure::Code e,          /* code d'erreur */
       AC::Network::Receipt* receipt,  /* utilise pour extraire les données immédiates */
       AC::Network::Header* h,          /*l'entête du message*/
       AC::IOVectEntry* iov,            /* vecteur de iov des données différées */
       AC::IOVectSize   iov_sz,         /* taille du vecteur iov */
       void* ctx                        /* contexte */
       ) throw (AC::ComFailure)
 {
     /* faire quelques chose ici, c'est le code de traitement du message */
     ……
 }
};
static PongService    pong;
```

Inscription

```
AC::Service::Manager::bind (2,  &pong);
```

Deux étapes principales pour préparer l'envoi d'un message vers un nœud destinataire est l'ouverture d'un canal de sortie vers ce nœud et la formation d'un message. Ensuite, un message est envoyé sur le canal de sortie (Exemple 3.6) :

Exemple 3.6

```
ACOM::Network::get_default_route(1); //ouverture d'un canal
// formation d'un message
ACOM::Network::Request    message (
s_net,                          /* réseau utilisé pour communiquer */
&pong,                          /* service appelé dans la machine distante */
callback,                       /* fonction crochet */
(int) n                         /* nombre de fois que le message sera envoyé */
);
ACOM::Network::Request::pack(message,len);  /* données immédiates, contenu copié */
ACOM::Netwrok::Request::put(message,len);   /*données différées, pointeur enregistré, pas de copie */
```

```
// envoi d'un message sur le canal
ACOM::Network::OutChannel::post(message);
```

3.5 Implantation

Dans cette section, nous présentons quelques implantations des interfaces virtuelles de type « réseau » afin d'offrir à une application INUKTITUT l'exploitation d'un matériel de communication particulier.

3.5.1 SOCKNET

Comme nous avons présenté dans 3.2.2, l'implantation du réseau SOCKNET se base sur l'interface *socket* SysV en utilisant *TCP/IP*. *TCP/IP* est non seulement un protocole qui est utilisé comme un standard pour la communication mais aussi un protocole orienté de connexion et *« byte streaming »*. Cette implantation offre un réseau portable, fiable et FIFO.

Le déploiement des applications parallèles avec ce réseau offre un ensemble de processus complètement connectés. L'établissement de connexion est dynamique. C'est-à-dire que ce réseau crée une liaison bipoint entre deux nœuds si ceux-ci ont besoin de communiquer. L'implantation SOCKNET utilise aussi un cache des dernières liaisons utilisées qui permet de n'ouvrir qu'un nombre restreint de descripteurs de fichier et *sockets* par processus.

Le réseau SOCKNET a deux modes de gestion des liaisons. Dans le premier mode, chaque liaison est gérée par un processus léger propre. Chaque processus léger attend l'arrivée d'un message sur la liaison correspondante et traite ce message. L'attente est bloquante. Dans le deuxième mode, un processus léger est utilisé pour gérer toutes les liaisons, c'est-à-dire que tous les messages arrivés sont traités par ce processus léger. La réception d'un message dans ce mode se base sur l'utilisation d'une fonction système appelée *select()* [Ste98]. Ce processus léger est bloqué sur la fonction *select()* en attente

de message. Lors d'un changement d'état des descripteurs, ce processus est réveillé pour traiter le message arrivé.

Le réseau SOCKNET peut être utilisé soit comme un réseau primaire soit comme un réseau secondaire (*cf.* 3.4.2). L'initialisation primaire de ce réseau se passe par la diffusion des identificateurs du port de réception de chacun des nœuds en utilisant un script de lancement. L'initialisation secondaire se base sur l'utilisation d'un réseau d'INUKTITUT existant (par exemple CORNET) pour faire la même opération.

3.5.2 GMNET

L'implantation du réseau GMNET a fait l'objet du stage de 3^{e} année ENSIMAG d'Everton Hermann (Mars-Juin 2004). L'objectif de cette implantation est le portage du réseau d'INUKTITUT sur une architecture à base de processeur Itanium (IA64) et de réseau rapide Myrinet. Les objectifs importants de cette implantation sont :

- L'exploitation aux mieux des performances des architectures cibles. Par exemple, en profitant la capacité de l'adressage mémoire sur 64 bits d'Itanium ainsi que des spécificités du réseau Myrinet.
- La communication est à un niveau le plus proche possible de la carte réseau pour éviter tous les surcoût logiciel et donc pour profiter des avantages de ce genre de réseau.
- Portage se base sur des bibliothèques existantes.

Bien qu'il ait eu quelques bibliothèque qui aient été portées sur les réseaux Myrinet comme PM2 [BMN96], BIP [PT98] et GM [Myr], l'implantation courante s'appuie sur la bibliothèque GM distribuée par la société Myricom. Pour l'instant (en Mars 2004), la bibliothèque PM2 ne concernait que les architectures IA32, elle ne se compilait pas sur l'architecture IA64.

Dans l'architecture générale de GMNET, l'envoi et la réception des messages à chaque nœud sont réalisés par un processus léger. On a deux systèmes de liste de message : la liste d'envoi et la liste de réception. Les messages envoyés sont ajoutés

dans la liste d'envoi. Un processus léger de GMNET prend les éléments de cette liste pour envoyer des messages sous forme de paquets GM. Du côté du récepteur, les paquets sont regroupés pour reconstruire des messages. L'appel du service de l'utilisateur est réalisé dès que tous les paquets nécessaires pour un message sont arrivés.

Cette implantation propose aussi des optimisations pour les cas particuliers comme le cas des échanges de message de taille très petite ou très grande. Dans le premier cas, GMNET propose de copier de tout le contenu du message dans un seul paquet de GM. Dans le deuxième cas, l'entête et les données immédiates sont envoyées par paquets de taille fixe. Du côté du récepteur, après l'allocation de toutes les zones mémoires pour les données différées, il renvoi des adresses mémoires de chacune des zones où l'émetteur peut alors écrire au travers des écritures distantes pour envoyer des données différées. De plus, elle fournit aussi l'opération de DMA qui s'appuie sur la possibilité offert par GM de faire des écritures dans les régions mémoires distantes.

L'initialisation de ce réseau est équivalente à celle du réseau SOCKNET : soit par l'utilisation d'un script de lancement, soit par l'utilisation d'un réseau d'INUKTITUT existant.

3.5.3 ICS

La bibliothèque ICS (INUKTITUT Communication Services) est une bibliothèque pour les opérations de communication collective. Le développement de ce cette bibliothèque est le travail du stage DEA-IMAG de Laurent Pigeon (Mars-Juin 2003) [Pig03] que son objectif est la définition d'interfaces asynchrones et l'implantation des opérations de communication collective. Cette définition est de prendre en compte les caractéristiques d'*Athapascan* [Rev04] pour une implantation fine de ces fonctions, plus particulièrement des fonctions non bloquantes.

Les caractéristiques remarquables d'ICS vis-à-vis de MPI sont :

- Multiprogrammation légère : les fonctions d'ICS sont ré-entrantes et utilisables dans un environnement avec plusieurs processus légers. La plupart des implantations du « domaine publique » de MPI, par contre, ne sont pas ré-entrants.

- Asynchrone : aucune opération d'ICS n'est bloquantes jusqu'à la fin locale de la communication. Dans MPI, les interfaces et implantations des communications collectives sont bloquantes. Tous les flots d'exécution qui participent à l'opération de communication sont bloqués jusqu'à la fin locale de l'opération.

Pour bien supporter l'exécution d'Athapascan, la bibliothèque ICS fournit une extension des messages actifs pour les opérations collectives : diffusions (1->N) et réductions (N->1). Ces opérations sont utilisées dans Athapascan pour la diffusion du graphe de flot de données et lors de l'exécution des tâches de communication des données. (Le modèle d'exécution d'Athapascan est présenté dans la section « Application » 3.6.1 suivante).

La bibliothèque ICS permet aussi à l'utilisateur de choisir plusieurs algorithmes de communications tels que l'arbre plat, l'arbre alpha et l'arbre chaîne pour atteindre la meilleure performance possible sur un environnement spécifique et pour une taille de message donnée.

Interface

Les primitives de communication collective sont réalisées à travers des réseaux définis dans ICS. Par exemple, pour exécuter la diffusion totale (nous prenons ce type de communication comme un exemple de la communication collective dans cette section), nous utilisons l'interface réseau de *« broacast » ICS::BcastNetwork*. Les réseaux d'ICS s'appuient sur les interfaces des réseaux de communication d'INUKTITUT. À l'exécution ils reposent sur les implantations concrètes comme SOCKNET, CORNET, etc. L'initialisation d' *ICS::BcastNetwork* se compose aussi du

choix de l'algorithme de communication et de ses paramètres. La diffusion totale de message est réalisée via l'appel du service sur tous les nœuds participants. Pour faire cela, nous ouvrons un canal de sortie de type diffusion dont ses paramètres comprennent son étiquette, le nombre de nœuds participants, l'ensemble des indices des nœuds et le nœud racine. La formation d'un message de diffusion est identique à celle de la communication bipoint. Ensuite, pour diffuser les messages par ce canal, nous utilisons le primitive *post()* du canal de diffusion de même que pour les messages bipoints. L'arrivée de ce message sur chaque nœud participant active le service adéquat pour le traiter.

L'avantage de cette interface est de pouvoir réutiliser les concepts de communication bipoint comme l'interface réseau, le canal, le message, le service dans des opérations collectives. Le traitement du message arrivé chez le récepteur est identique à celui d'une communication bipoint (*cf.* 3.3.1) et ne nécessite pas de revoir le code des services.

3.6 Applications

Ces sections présentent les différentes applications qui utilisent l'interface d'INUKTITUT présentée dans ce chapitre : un environnement de programmation parallèle à haut niveau (*Athapascan* [GRCD98, Cav99, Dor99, Gal99, RGR03, Rev04]) et un ensemble d'outils pour les grappes (*Katools* [MR01, Mar03]).

3.6.1 Athapascan

Comme nous l'avons abordé dans le chapitre 1, *Athapascan* est une interface de type macro *data-flow* pour la programmation parallèle asynchrone. Cette interface permet la description du parallélisme entre les tâches de calcul qui se synchronisent sur des accès à des objets dans une mémoire globale distribuée. Le parallélisme est explicite de type fonctionnel, la détection des synchronisations est implicite. La sémantique est de type séquentielle et un programme écrit avec la bibliothèque *Athapascan* est indépendant de l'architecture cible.

Modèle et Exécution

Le modèle d'exécution d'*Athapascan* se base sur un graphe de flot de données. Ce graphe représente les précédences entre les tâches de calcul et les données partagées. Les modes d'accès d'une donnée par une tâche sont : lecture, écriture, accumulation et modification. Ces modes d'accès sur les données partagées sont définis au moment de la création d'une tâche. Ceci permet à *Athapascan* de construire automatiquement le graphe de flots de données du calcul. Les états d'une tâche sont :

- Attente : au moins un des paramètres de la tâche n'est pas encore prêt.
- Prêt : cette tâche est prête à être exécutée.
- Exécution : elle est en train d'être exécutée.
- Terminé : l'exécution de la tâche est terminée.

Les états possibles d'une donnée sont :

- Attente : l'une des tâches prédécesseurs n'a pas terminé son exécution.
- Prêt : toutes les tâches prédécesseurs ont terminé leur exécution.
- Terminé : toutes les tâches reliées à cette donnée ont terminé leur exécution.

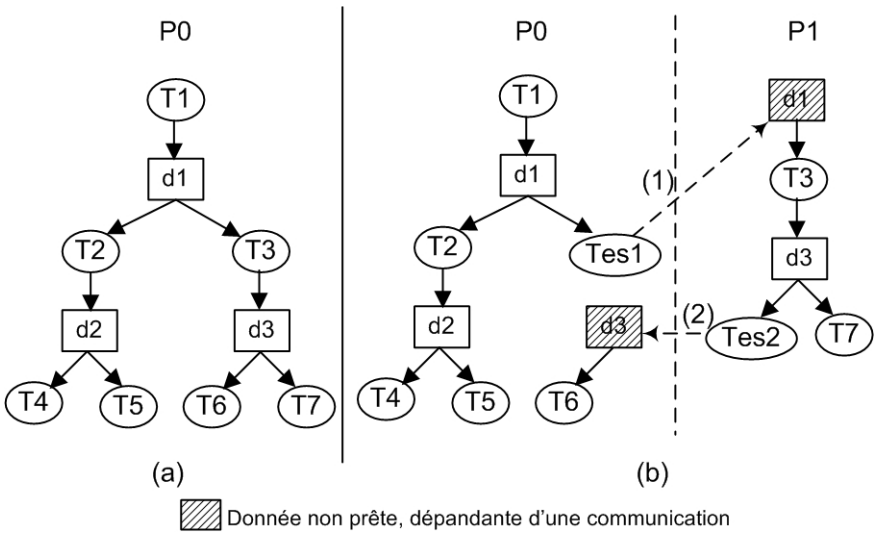
Il existe deux manières d'exécuter ce graphe selon l'ordonnancement dynamique ou l'ordonnancement statique qui se basent sur la communication unidirectionnelle via l'exportation des tâches.

Ordonnancement dynamique

Dans l'ordonnancement dynamique, l'exécution s'appuie sur le principe du vol de tâche. Un processeur commence par construire le graphe de flot de données. Lorsqu'un processeur devient inactif, c'est-à-dire qu'il n'a plus de tâche prête à l'exécution, il passe dans l'état « voleur » et cherche à récupérer du travail chez les autres processeurs. L'exécution selon ce modèle possède quatre étapes principales suivantes :

1. Le processeur « voleur » demande une tâche au processeur « volé ».
2. Cette tâche du processeur « volé » est exportée.
3. À la fin son exécution, le processeur « voleur » envoie le résultat.
4. Retour à 1

Pour assurer les nouvelles synchronisations et communications de données nées dans le processus « volé », les tâches de type « d'envoi et de signalisation » sont insérées aux endroits nécessaires dans le graphe de processeur « volé » et « voleur ». Etudions un exemple dans la figure 3.9, nous avons sept tâches (de T1 à T7) sur le processeur P0 et puis le processeur P1 vole deux tâches T3 et T7. La tâche T3 n'est pas prête au moment où elle est volée. Une tâche « d'envoi et de signalisation » Tes1 est insérée à la place de la tâche T3 dans le graphe de P0 (b). Dès que la donnée d1 est prête, la tâche Tes1 l'envoie à P1 (b1) et puis signale P1 pour exécuter la tâche T3. La tâche T6 est exécutée de la même manière (b2).



(a) graphe avant le vol (b) graphe après le vol par P1

Figure 3.9 Un exemple d'ordonnancement dynamique d'Athapascan

Ordonnancement statique

Dans l'ordonnancement statique, un graphe de flot de données est entièrement construit sur un seul processeur. Puis, ce graphe est analysé par l'algorithme d'ordonnancement statique afin d'affecter les tâches et les données aux différents processeurs. Les étapes principales d'exécution selon ce modèle sont :

1. Le processeur construit entièrement le graphe de flot de données.
2. Ce graphe est ordonnancé selon une stratégie choisie.
3. Le processeur diffuse ce graphe à tous les autres processeurs participants.

4. Tous les processeurs détectent les schémas de communication contenus dans le graphe et génèrent les tâches adéquates.
5. L'exécution parallèle débute.

De plus, chaque tâche est associée à un compteur du nombre de données en entrée produit à distance. Lorsqu'une donnée produite à distance est réceptionnée, ce compteur est décrémenté. Lorsque ce compteur a la valeur zéro, la tâche est considérée comme prête à s'exécuter.

La Figure 3.10 présente un exemple de génération statique de tâche. Dans cet exemple, le graphe est construit par un processeur, puis il est diffusé à P1, P2, P3. La détection de schémas de communication sur chaque Pi insère les tâches « d'envoi et de signalisation » TesPi appropriées dans le graphe. Par exemple, la tâche T2 dépend des données (d1 et d3) qui viendront de P1 et P3. La tâche TesP2 est ajoutée dans le graphe sur P1 pour envoyer la donnée d1 au processeur P2 et le signaler. La valeur du compteur associé à la tâche T2 est réduite et si elle est égale à zéro, c'est-à-dire que d1 et d3 sont arrivées, la tâche T2 est prête à l'exécution.

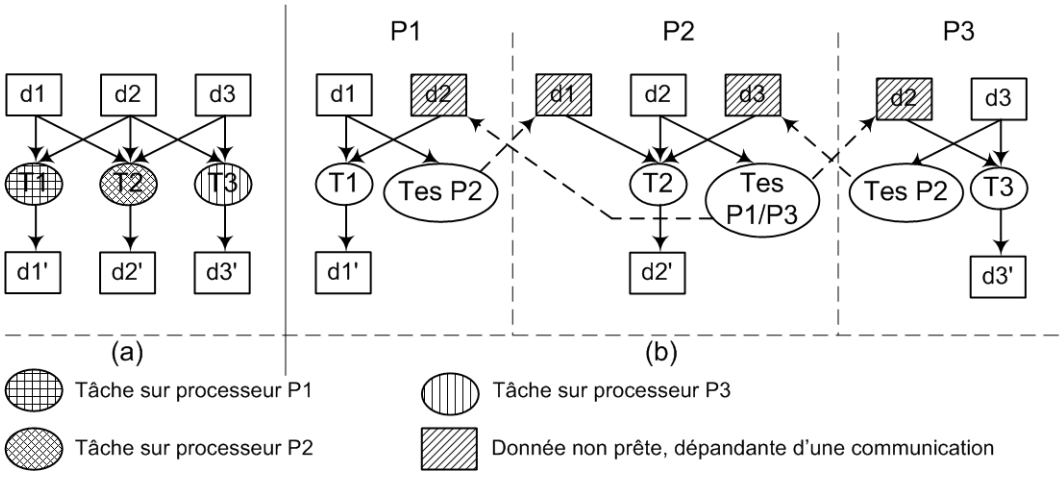


Figure 3.10 Un exemple d'ordonnancement statique d'Athapascan

Implantation

L'interface INUKTITUT permet d'implanter efficacement le modèle d'exécution d'*Athapascan* sur l'environnement de grappe. Reprenons les exemples dans la section ci-

dessus. Nous supposons que le graphe de flot de données est construit et exécuté de manière distribuée sur plusieurs nœuds INUKTITUT. L'exécution des tâches de type « d'envoi et de signalisation » dans toutes les stratégies d'ordonnancement sont basées sur les primitives de communication par échange de message actif d'INUKTITUT. Dans la figure 3.9b la tâche T3 sur le nœud P1 dépend de la donnée d1 qui sera écrite (protocole WS d'INUKTITUT, *cf.* section 3.4.6 page 62) par la tâche Tes1 sur le nœud 0. De la même manière la tâche Tes2 sur P1 écrit la donnée d3 sur le nœud P0.

Dans le cas de l'ordonnancement purement dynamique l'opération de vol est réalisée par un message actif INUKTITUT de type AM. De plus, dans l'ordonnancement statique, les schémas de communication identifiés sont les réductions et les diffusions. La réduction correspond à une séquence d'écriture concurrentes et la diffusion à une séquence de lecture concurrentes. De même pour l'ordonnancement dynamique, des tâches de diffusion de 1 vers N nœuds et de réduction sont détectés dans le graphe lors du calcul de l'ordonnancement et le partitionnement du graphe. La figure 3.10 montre le résultat du partitionnement du graphe sur 3 noeuds (a) et la génération des tâches « d'envoi et de signalisation » correspondant à l'ordonnancement calculé (b).

Performance

Nous présentons ici une comparaison des performances entre deux versions différentes d'*Athapascan* : l'ancienne version (1) basée sur Athapascan-0 [Cav99, Dor99, Gal99] et la nouvelle version (2) basée sur INUKTITUT [Rev04]. L'application choisie ici concerne le calcul récursif des éléments de la suite :

$$F(n) = F(n-1) + F(n-2) + ... + F(n-k)$$
$$F(0) = 0 \quad F(1) = 1$$

La plate-forme de test est les machines d'I-Cluster1 (*c.f.* Annexe A). Les valeurs de *n* et *k* dans la formule ci-dessus sont respectivement égales à 35 et 10. Les seuils d'arrêt de la découpe récursive sont de *n* égal à 10 et de *n* égal à 15. En deçà, le calcul est effectué séquentiellement. La figure 3.11 présente le gain de temps de version 2 par rapport à version 1.

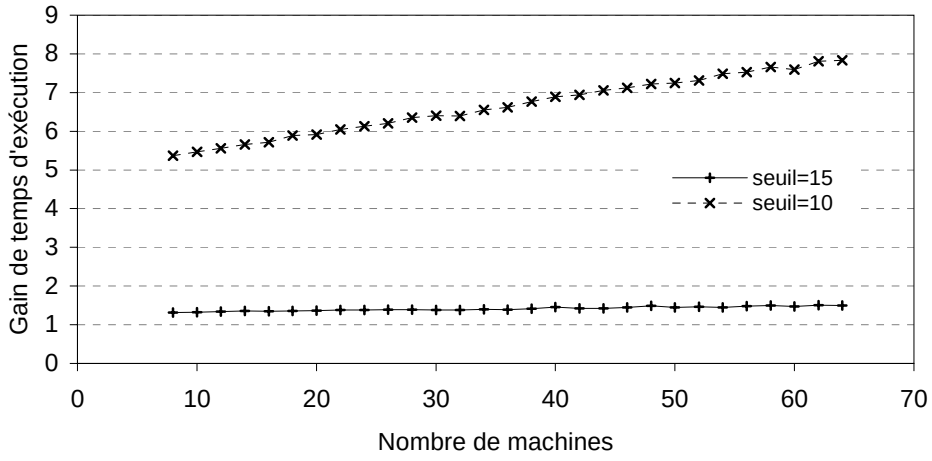


Figure 3.11 *Le gain de temps de la version 2 par rapport à la version 1*

Nous observons que la version 2 a un gain environs 7,8 pour seuil d'arrêt de n égal à 10 (grain fin). À gros grain (seuil d'arrêt de n égal 15), le gain est d'environs 1,5. Une partie de ces gains viennent du moteur exécutif Athapascan qui a été changé, la seconde partie était due à une meilleure intégration des communications dans Athapascan grâce à INUKTITUT.

3.6.2 Taktuk et KaTools

Du côté de l'interface, *Taktuk* (*cf.* 3.2.2) implante un réseau comme les autres réseaux d'INUKTITUT. Ce réseau permet la communication par échange des messages actifs. Ce réseau possède aussi la capacité de déployer une application INUKTITUT, c'est-à-dire que les applications parallèles développées avec INUKTITUT peuvent se déployer d'elles mêmes si elles utilisent le réseau offert par Taktuk. *KaTools* [Mar03] est un outil qui est développé au dessus de l'environnement *Taktuk. KaTools* se compose de *Rshp* et *Mput*.

Rshp est un outil d'exécution de commandes parallèles d'Unix. Il permet non seulement de lancer l'exécution d'une commande sur l'ensemble des nœuds d'une grappe mais aussi de disposer d'une console sur chaque nœud.

Mput est un outil de diffusion de fichier qui se base sur l'interface INUKTITUT. D'abord il est déployé en parallèle via l'implantation de *Taktuk* sur l'ensemble des nœuds et puis il utilise une autre implantation au dessus du protocole standard *TCP/IP* pour initialiser un réseau chaîné permettant la diffusion les fichiers.

3.7 Bilan

L'architecture de l'interface INUKTITUT a été présenté et discuté dans ce chapitre. Son modèle de communication par échange de messages actifs est implanté sous trois protocoles différents : *« active message », « write and signal », « allocation write and signal »* qui permettent à l'utilisateur d'avoir plusieurs méthodes de gestion de mémoire. INUKTITUT fournit aussi un ensemble d'interfaces génériques C++ simples et efficaces de communication et de processus légers qui sont construits par des modules indépendants. La conception de l'interface en C++ permet de réutiliser facilement des codes ou bibliothèques existantes par encapsulation. Le petit nombre d'abstractions nécessaires donne une simplification importante par rapport aux autres bibliothèques comme *MPI* (plus de 200 fonctions d'interface) ou *Athapascan-0* (environs 100 fonctions d'interface). INUKTITUT non seulement évite les inconvénients d'Athapascan-0 mais aussi permet d'implanter efficace l'environnement de programmation parallèle asynchrone *Athapascan* [Rev04]. De plus, son architecture par couche permet la portabilité de l'environnement de programmation par l'homogénéisation des différentes bibliothèques de communication utilisées. Il permet à l'utilisateur de développer aisément ses applications en masquant les problèmes de mise en œuvre sur des protocoles de processus léger (*POSIX Pthread, Win32 Thread*) et de communication (*TCP/IP, Corba, GM/Myrinet*). A ce moment, on peut donc se poser une question : quelles sont les performances d'INUKTITUT ? Les chapitres suivants se préoccupent de l'évaluation des performances offertes par les implantations d'INUKTITUT.

4

Performances des processus légers avec INUKTITUT

Dans ce chapitre nous évaluons les performances de la bibliothèque de processus légers INUKTITUT. Nous étudions d'abord les fonctions de base telles que la création de processus léger et la commutation de contexte de processus léger. Nous évaluons ensuite les fonctions de synchronisation. Nous analyserons les résultats en comparant l'implantation des algorithmes de test entre INUKTITUT et l'interface native des Pthreads du standard *POSIX* (*POSIX Threads*). Ces expériences sont exécutées sur différents noyaux de systèmes et implantations des bibliothèques des processus légers.

4.1 Fonctions de base

L'évaluation des performances d'un processus léger s'appuie sur certains indicateurs qui sont les coûts d'exécution des fonctions principales : la gestion d'un processus léger et la réalisation leur synchronisation. Nous utilisons donc deux types d'indicateur de performance : l'un se base sur les fonctions de gestion et l'autre se base sur les fonctions de synchronisation. Plus précisément, le premier type se compose *des coûts de création*

d'un processus léger et de *la commutation de contexte* que nous étudions dans cette section, et le deuxième concerne *le coût d'exécution des fonctions de synchronisation* qui est présenté dans la section 4.2 (page 89).

4.1.1 Coût de création d'un processus léger

Méthode d'évaluation

En général, le coût de création d'un processus léger est évalué par la durée de sa création qui dépend de l'implantation et de son ordonnancement. Les méthodes utilisées pour déterminer ce coût sont de mesurer la durée de l'appel de la primitive de création [GF00], celle d'un cycle de vie d'un processus léger [Bra02a][Car99] et celle de création d'un processus léger [GF00].

La première méthode n'est pas vraiment fiable parce qu'une partie de l'initialisation du processus légers est exécutée au moment où il est activé pour la première fois dans certains noyaux [Car99]. La deuxième méthode s'appuie sur le cycle de vie d'un processus léger. C'est-à-dire, qu'on mesure le temps entre le début de l'appel de la création d'un processus léger qui exécute une procédure vide et la détection de sa terminaison. Pourtant, nous ne pouvons pas utiliser cette méthode dans le cas où la bibliothèque n'a pas de primitive qui détecte la terminaison d'un processus léger comme INUKTITUT. Dans ce cas, on peut s'en remettre à la troisième méthode dans laquelle le coût est évalué par la durée entre un appel de la création d'un processus léger jusqu'à ce qu'il exécute sa première instruction. Pour éviter l'accumulation d'un quantum de temps d'ordonnancement, le processus léger qui appelle la création, rend la main à l'ordonnanceur processeur [GF00]. Cette méthode a été étendue par la mesure du temps de création récursif de plusieurs processus légers [SM99].

La bibliothèque de processus léger d'INUKTITUT ne fournit pas la primitive qui notifie la fin d'un autre processus léger, par exemple *« pthread_join »* de *POSIX Threads*. Nous ne pouvons donc pas utiliser la méthode d'évaluation basée sur le cycle de vie de processus légers [Bra02a][Car99]. Ce coût est évalué par la troisième méthode

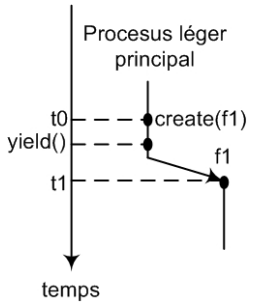
ci-dessus, cette méthode est donnée par *Algorithme 4.1* [GF00]. Nous avons d'ailleurs mesuré le coût de la création des processus légers par récursivité par un autre algorithme illustré dans *Algorithme 4.2* [SM99].

Algorithme

Algorithme 4.1 Estimation du coût de création d'un processus léger

```
f1(){
t1<-GetTime();
exit;
}

CreateurThread{
  t0<-GetTime();
  create(f1);
  yield();
  ……
}
```

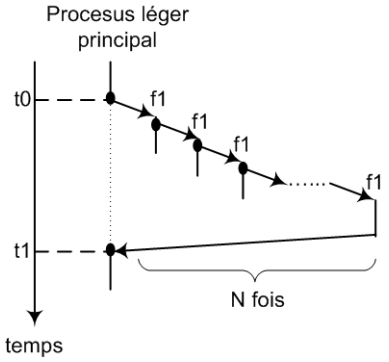


La durée *d* mesurée est égale à t1 – t0. Nous utilisons *d* pour évaluer le coût de la création. Il faut faire en outre attention au problème d'intervalle de temps (*time slice*, en anglais) dans l'utilisation de cet algorithme. C'est-à-dire que le processus léger principal continue à monopoliser le processeur après l'instruction de création du processus léger. Afin d'éviter ce problème, il doit rendre le processeur juste après cet appel par appel [GF00], par exemple, à la fonction *« yield() »* de *POSIX 1003c* [IEEE94]. Le coût de création d'un processus léger par cette méthode peut donc comprendre le surcoût d'exécution de cette primitive dans le cas où le processus léger principal est encore dans son quantum de temps d'ordonnancement.

Algorithme 4.2 Estimation du coût de création récursive des processus légers

```
f1(){
  count<-count +1;
  if (count=N)
    signal();
  else
   create(f1);
  exit;
}

CreateurThread{
   t0<-GetTime();
   create(f1);
   wait();
   t1<-GetTime();
   …
}
```



Le coût de la création récursive est évalué par la durée d = (t1 – t0)/N. N est le nombre de processus légers créés.

Résultats et Analyses

Afin d'évaluer les performances des processus légers, nous lançons des expériences qui utilisent l'interface PTH d'INUKTITUT au dessus de l'interface de processus légers de *POSIX 1003.1c (Pthreads)* [IEEE94]. Le nombre de processus légers créés dans l'algorithme 4.2 est ***N=10000***. Les plates-formes d'expérience sont une machine de la grappe ***I-Cluster2*** (I2, en bref) et la machine ***Opata*** avec deux noyaux de Linux : ***2.4.26*** (Opata_2.4, en bref) et ***2.6.8*** (Opata_2.6, en bref) (*cf.* Annexe A). Les valeurs affichées dans le tableau 4.1 (ainsi que les autres tableaux dans ce chapitre) sont les valeurs moyennes (*cf.* Annexe B.2). Nous utilisons *« gettimeofday »* (*cf.* Annexe B.1) comme la fonction de prise de temps (*GetTime()* dans les algorithmes) dans ce cas parce qu'on n'a pas besoins de la précision de moins d'1µs. De plus, nous utilisons cette fonction dans toutes les expériences dans ce chapitre.

Plate-forme	Bibliothèque de *threads*	*d* (µs) de l'Algorithme 4.1		*d* (µs) de l'Algorithme 4.2	
		INUKTITUT	**POSIX**	**INUKTITUT**	**POSIX**
I2	NPTL-0.60	9,89±0,92	9,47±0,80	13,71±0,04	13,57±0,05
Opata_2.4	linuxthread-0.10	71,62±2,73	70,83±2,00	80,85±0,12	80,52±0,19
Opata_2.6	NPTL-0.60	32,94±1,30	32,93±0,76	9,54±0,02	9,32±0,02

Tableau 4.1 *Comparaison de la durée de création du processus léger entre INUKTITUT et POSIX Threads*

(I2 : IA64-biproc 900Mhz; Opata : Athlon-biproc 1,2Ghz)

Nous étudions maintenant les résultats de ces algorithmes dans le Tableau 4.1.

Les conclusions sont :

(i) Sur les mêmes architectures et algorithmes de test, la durée de création du processus léger dépend de l'interface utilisé. Pourtant cette durée d'INUKTITUT est proche de celle de *POSIX*.

(ii) Par ailleurs, l'algorithme de test a aussi un impact sur mesure de la durée de création du processus léger. En observant les deux premières plates-formes dans le tableau, nous trouvons que cette durée de l'algorithme 4.1 est plus faible que celle de l'algorithme 4.2. Dans l'algorithme 4.2, nous créons récursivement des processus légers qui causent un surcoût de leur gestion. Pourtant, dans la troisième plate-forme, le rapport de cette durée entre deux algorithmes est inverse par rapport aux deux premières. De plus, cette durée en utilisant noyau 2.6.8 est plus faible par rapport au noyau 2.4.26 sur le mêmes algorithme de test et architecture matérielle (*Opata*) : *9,32µs* vs. *80,52µs* (POSIX, algorithme 4.2, par exemple). La raison de ce phénomène est que le noyau 2.6 de Linux réalise certaines améliorations sur modèle de processus légers par rapport au noyau 2.4 [San03]. Ces améliorations concernent notamment :

- L'augmentation de la quantité de processus légers gérés : le noyau 2.6 peut manipuler un nombre arbitraire de processus léger et jusqu'à deux billions PIDs sur IA32. Un des mécanismes qui supporte cet avantage est l'introduction de l'appel système TLS (*Thread Local Storage*) qui permet d'allouer un ou plusieurs GDTs (*Global Descriptor Table*). Un descripteur de segment de mémoire accédée d'un processus léger peut être enregistré dans chaque entrée

d'un GDT. Dans le noyau 2.4, les descripteurs sont stockés dans le LDT (*Local Descriptor Table*) de chaque processus lourd, et le nombre maximum de processus légers est donc limité à 8192. Par contre, cette limite dans noyau 2.6 ne dépend que la limite de ressource du système (par exemple la mémoire).

- Le changement de contenu du répertoire /proc : dans le noyau 2.4, l'information de chaque processus léger est enregistrée dans ce répertoire. Pourtant, celui de noyau 2.6 ne contient aucun de processus léger sauf l'initiateur. Cet avantage permet par conséquence d'enlever le coût d'accès du répertoire /proc qui est un impact sur la performance de gestion dans le cas de plusieurs processus légers existants. En particulier cela devient très important pour les systèmes de millier de processus légers.

Nous présentons maintenant les résultats principaux sur le coût de création d'un processus léger dans d'autres travaux. D'abord, le résultat d'algorithme 4.2 dans [Brad02a] est environs de *99µs* pour *POSIX* sur le noyau 2.4.2 et Pentium III 500 Mhz. La cause d'écart entre son résultat et le nôtre de *POSIX* ci-dessus (*80,52µs*) est la différence d'architecture (*Pentium III 500 Mhz vs. Athlon 1,2Ghz*).

Nous citons aussi les résultats de performance de *Marcel* [MN95, Nam01, DN03], une autre bibliothèque de processus légers. D'après [Nam01], son coût de création d'un processus léger sur Pentium II 650 MHz, Linux noyau 2.4.x est de *95µs* pour *Pthread*, *3µs* pour *Marcel SMP* et d'*1µs* pour *Marcel mono*. La différence entre ce coût pour Pthread et le nôtre pour Pthread ci-dessus (*95µs vs. 70,83µs*) dépend des deux facteurs : l'architecture (*Pentium II 650 MHz vs. Athlon 1.2GHz*) et la méthode de test. La durée de création d'un processus léger de Marcel est mesurée à travers d'un test « création+terminaison ». Dans ce test, le temps est compté juste avant l'appel à *pthread_create()* et juste après le retour de *pthread_join()*. Ce test est similaire à la deuxième méthode d'évaluation abordée ci-dessus (page 76) que nous n'utilisons pas dans nos expériences.

Nous remarquons aussi que le surcoût de toutes les versions de *Marcel* est plus faible que celui de *POSIX Threads*. *Marcel mono* est une bibliothèque de processus léger au niveau utilisateur. Il n'est pas nécessaire de passer en mode noyau pour l'opération de création de processus léger. Son inconvénient est qu'il ne peut pas être exploité sur les architectures SMP. *Marcel SMP*, quant à lui, est une bibliothèque hybride (*cf.* section 2.2.2, page 17). Elle peut utiliser plusieurs processus légers au niveau noyau pour exécuter son ordonnanceur utilisateur.

En conclusion, la bibliothèque de processus léger utilisée a un impact sur le coût de création d'un processus léger. Pourtant, le surcoût dû à l'interface d'INUKTITUT est négligeable par rapport à celui de *POSIX* (inférieur à 1µs dans nos expériences). En outre, l'architecture du système (matériel et système d'exploitation) a un autre impact sur ce coût. Particulièrement, le noyau 2.6.x de Linux fournit des avantages importants sur ce coût par rapport au noyau 2.4.x. Enfin, le choix de la méthode de mesure influence également ce coût. En effet, chaque méthode réalise sa propre stratégie d'évaluation : création d’un ou plusieurs processus légers, création avec ou sans terminaison.

4.1.2 La commutation de contexte

Méthode d'évaluation

La multiprogrammation d'une ressource processeur par plusieurs processus légers utilise plusieurs stratégies d'ordonnancement (pré-emptive, non pré-emptive) basées sur un changement du contexte « processeur » des processus légers. Le processus léger courant est interrompu et un nouveau s'est activé qui est choisi dans la queue de priorité d'ordonnancement. La commutation de contexte est volontaire ou involontaire. La cause de la commutation de contexte involontaire est normalement due à l'expiration de quantum de temps. La commutation de contexte volontaire se passe quand un processus léger attend sur une opération d'entrée/sortie ou sur une variable de synchronisation (verrou, sémaphore). La commutation volontaire sera utilisée pour évaluer le coût de la commutation de contexte parce qu'elle est souvent intégrée dans les applications de multiprogrammation de processus légers. De plus, la commutation involontaire se base

sur la valeur de quantum de temps qui est définie comme un paramètre du système d'exploitation et non maîtrisable dans les expériences. Il existe une procédure de « passage de main » (*yield()*, en anglais) qui peut être utilisée par un processus léger pour s'interrompre et permettre au système d'activer un autre processus léger. Nous pouvons ainsi estimer le coût de commutation de contexte par mesure du temps de « passage de main » mutuellement entre deux processus légers [SM99]. Néanmoins, la procédure de « passage de main » n'est pas définie dans le standard *POSIX*, mais dans l'une des extensions « realtime ».

Une autre méthode qui consiste à mesurer le temps de commutation entre deux processus légers par blocage alternatif à l'aide des variables de condition [GF00] [Car99]. Une méthode qui mesure aussi le temps de commutation contexte entre plusieurs processus légers est proposée dans [Bra02c]. Cette méthode n'utilise que la primitive de verrouillage (*lock* et *unlock*). Chaque processus léger a son propre verrou et il essaye de prendre le verrou du processus léger voisin. Nous utilisons les deux dernières méthodes pour évaluer le coût de commutation de contexte entre deux et entre plusieurs processus légers.

Algorithme

La commutation de contexte entre 2 processus légers

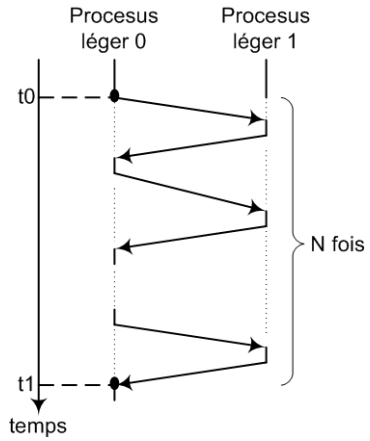
Cet algorithme est pour évaluer le coût de la commutation de contexte entre deux processus légers par le blocage alternatif entre eux [GF00][Car99]. Nous utilisons deux primitives de synchronisation: le verrou et la variable de condition pour faire la commutation de contexte. Nous avons deux processus légers : 0 et 1. Chaque processus léger exécute alternativement la routine *Thread()* comme dans Algorithme 4.3. Nous n'avons qu'un processus léger qui s'exécute (processus léger 0, par exemple). Il envoie un signal à l'autre processus léger qui attend sur une condition (processus léger 1, par exemple). A chaque tour (ou itération), les rôles sont inversés. Nous prenons le temps après N tours et nous avons 2 x N commutations de contexte entre deux processus

légers qui se bloquent mutuellement. Le temps de commutation *t* est calculé par [Car99] :

$$t = (t1 - t0) / (2*N) \quad (4.1)$$

Algorithme 4.3 Commutation de contexte entre deux processus légers

```
Thread(tid){
count <- 0
t0 <- GetTime()
lock(m)
while count < N do
  if tid <> 0 then
    tid <- 0
    cond_signal(c)
  else
    tid <- 1
    count <- count +1
    cond_wait(c,m)
  endif
endwhile
cond_signal(c)
unlock(m)
t1 <- GetTime()
}
```



La commutation de contexte entre n processus légers

Nous étudions ensuite la deuxième méthode d'évaluation qui estime la commutation de contexte entre n processus légers. Cette méthode est présentée par [Bra02c]. On a N processus légers et M verrous (M>N). L'exécution de chaque processus léger est illustrée dans Algorithme 4.4. D'abord, chaque processus léger se bloque sur son propre verrou. C'est-à-dire que le processus léger i ($1 \leq i \leq N$) bloque le verrou i ($1 \leq i \leq M$). Il

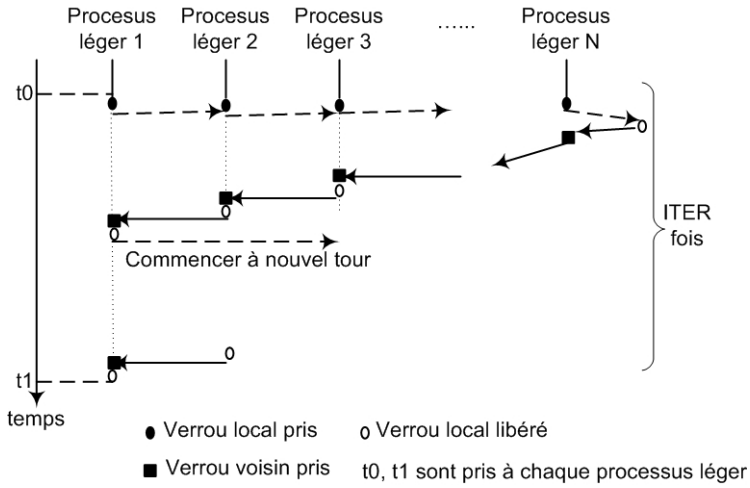
essaie ensuite de bloquer le verrou voisin (verrou i+1) et s'il réussit il débloquera son propre verrou.

Algorithme 4.4 Commutation de contexte entre N processus légers

```
Lock(k);

t0 <- GetTime()

for i=1 to  ITER do
   k1 = k+1;
   if  (k1 >=M) then
     k1 = 0;
   endif
   Lock(k1);
   Unlock(k);
   k = k1;
endfor
Unlock(k);
t1 <- GetTime()
t <- t1 – t0
```



Nous mesurons le temps *t* de chaque processus léger. Nous prenons le temps minimum t_{min}, le temps maximum t_{max} et le temps moyen t_{aver}.

$$t_{min} = min(t_k);\ \ t_{max} = max\ (t_k)\ \ \text{où}\ \ k = 1..N \quad \text{et} \quad t_{aver} = \frac{1}{N}\sum_{k=1}^{N} t_k \qquad (4.2)$$

N est le nombre de processus légers participants, t_k est le temps d'exécution mesuré d'un processus léger (égale t1 – t0 dans l'algorithme 4.4).

Résultats et Analyses

La commutation de contexte entre 2 processus légers

Les plates-formes des expériences de cet algorithme sont I2 (*cf.* la section 4.1.1, page 80) et une machine de la grappe Idpot (*cf.* Annexe A) (Xeon, en bref). Le nombre d'itérations choisi est ***N=10000***. Le Tableau 4.2 présente la comparaison de cet algorithme de test entre l'interface INUKTITUT et *POSIX*. En observant ce tableau, on constate que le temps *t* (*cf.* l'équation 4.1, page 83) de *POSIX* est proche de celui de d'INUKTITUT. Notons que c'est normal parce que les fonctions de synchronisation d'INUKTITUT appellent les fonctions correspondantes de POSIX. Le surcoût de cet appel est faible. Le surcoût dû à INUKTITUT est négligeable. Ce surcoût *s* est déterminé par :

$$s = \frac{\left| t_{INUKTITUT} - t_{POSIX} \right|}{t_{POSIX}} \tag{4.3}$$

	Bibliothèque de *threads*	INUKTITUT	POSIX	*surcoût s*
I2	NPTL-0.60	6,79±0,09 µs	6,67±0,03 µs	1,7%
Xeon	linuxthread-0.10	7,76±0,03 µs	7,73±0,01 µs	0,3%

Tableau 4.2 *Comparaison du temps t (µs) de commutation de contexte entre deux processus légers entre INUKTITUT et POSIX Thread*

(I2 : IA64-biproc 900Mhz; Xeon : Xeon-biproc 2,5Ghz)

La commutation de contexte entre n processus légers

Les expériences sont exécutées en utilisant ***10, 100*** et ***500*** processus légers. Dans chaque expérience, l'itération ITER (*cf.* l'algorithme 4.4) est égale ***10000***. Via des études expérimentales, nous constatons que la différence entre les trois valeurs t_{min}, t_{max}, t_{aver} (*cf.* l'équation 4.2, page 84) est inférieure à 0,1%. Nous utilisons donc t_{aver} comme le temps d'exécution de cette expérience. Les expériences sont lancées sur les mêmes plates-formes que celles utilisées dans la section 4.1.1 (page 78).

La comparaison entre deux interfaces de processus léger (INUKTITUT et *POSIX Thread*) est présentée dans le tableau 4.4. On se rend compte de plus que la performance de commutation entre plusieurs processus légers de l'interface d'INUKTITUT est très proche de celle de *POSIX Thread*. Le surcoût dû à INUKTITUT est aussi négligeable. Ce surcoût *s* est déterminé comme dans l'expérience précédente (*cf.* l'équation 4.3, page 85). Nous constatons en outre que le temps d'exécution par cet algorithme de test varie linéairement en fonction du nombre de processus légers (Figure 4.1).

Plate-forme	Nombre de processus léger	INUKTITUT	POSIX		
		t(ms)	*t(ms)*	*surcoût s*	*surcoût s/processus léger*
I2	10	552,33±2,12	546,86±2,20	0,91%	0,091%
	100	7299,41±12,01	7228,66±15,01	0,96%	0,0096%
	500	43532,05±92,44	43032,63±65,91	1,16%	0,0023%

Tableau 4.3 *Comparaison du temps (milliseconde) de commutation de contexte entre N processus légers entre INUKTITUT et POSIX Thread*

(I2 : IA64-biproc 900Mhz, NPTL-0.60)

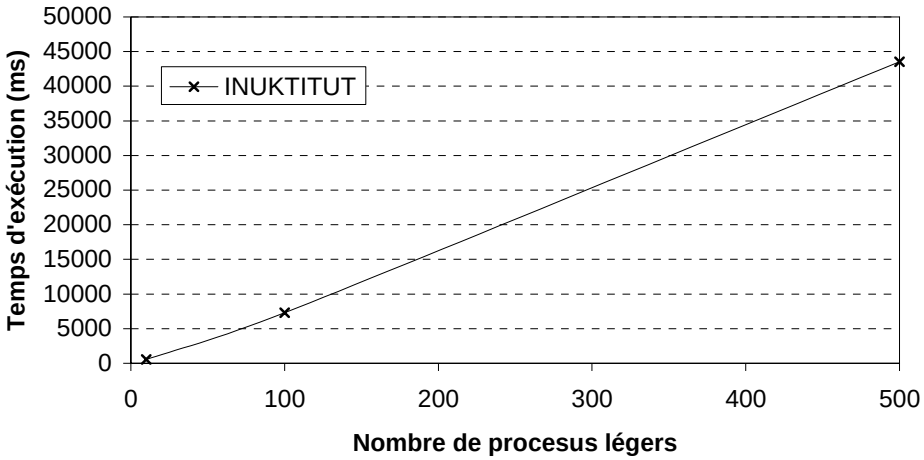


Figure 4.1 *Temps de commutation de contexte par INUKTITUT selon le nombre de processus légers*

Nous avons aussi comparé les temps de commutation de contexte de 100 processus légers sur le noyau 2.4.26 (linuxthread-0.10) et le noyau 2.6.8 (NPTL-0.60) comme dans le Tableau 4.4 suivant :

Plate-forme	Interface	Linux noyau 2.4.26	Linux noyau 2.6.8
		t(ms)	*t(ms)*
Opata	INUKTITUT	12193,94±30,46	10879,62±223,13
	POSIX	11308,71±12,58	10444,40±94,46

Tableau 4.4 *Comparaison du temps (milliseconde) de commutation de contexte par 100 processus légers entre les noyaux différents*

(Opata : Athlon-biproc 1,2Ghz, linuxthread-0.10 pour le noyau 2.4.26 et NPTL-0.60 pour le noyau 2.6.8)

En étudiant ce tableau, nous remarquons que le surcoût de commutation de contexte sur le noyau 2.6 est plus faible que celui sur le noyau 2.4 : on a un gain de 10% du temps d'exécution. L'effet du nouvel ordonnanceur du noyau 2.6 explique ce gain [San03]. Il n'a plus de synchronisation globale. La commutation de contexte entre des processus légers sur un processeur peut être exécutée en parallèle avec la commutation de contexte entre des processus légers sur un autre processeur.

Nous présentons ici les performances de la commutation de contexte exécutée par la bibliothèque *Marcel* [Nam01, DN03]. Le temps de la commutation de contexte dans [Nam01] est de *830ns* pour le noyau processus léger de POSIX, *320ns* pour *Marcel SMP* et *130ns* pour *Marcel Mono* sur Pentium II 650 MHz avec Linux 2.4.x. La différence entre ce coût pour *POSIX Thread* et le nôtre pour *POSIX Thread* ci-dessus (*830ns vs. 11,31µs*) dépend principalement de la méthode de test utilisé. La méthode utilisée dans le test de *Marcel* effectue des *« ping-pong »* entre deux processus légers en utilisant la primitive *pthread_yield*. Cette méthode est similaire à la méthode *« passage de main »* abordée ci-dessus (page 82). De plus, on utilise dans le test de Marcel une version modifiée du noyau Linux garantissant une commutation de contexte effective à chaque appel à *sched_yield*.

Nous constatons que l'utilisation de la bibliothèque *Marcel* donne un coût de commutation de contexte bien plus faible que celui du noyau de processus léger de Linux à cause de la différence de modèle de processus léger comme nous l'avons abordé dans 4.1.1 (page 80). *Marcel* possède de plus une politique d'ordonnancement efficace de type *« Round-Robin »* [DMN99].

En conclusion, les impacts sur le coût de commutation de contexte sont dus : à la bibliothèque de processus légers choisite, le nombre de processus légers participant à cette opération et l'architecture du système (matériel et noyau du système d'exploitation). En outre, le surcoût de commutation de contexte en utilisant l'interface d'INUKTITUT est négligeable par rapport à l'utilisation de l'interface de *POSIX Threads*.

4.2 Coût de la synchronisation

Méthode d'évaluation

Le coût de synchronisation peut être évalué par le coût minimum de synchronisation de chaque objet c'est-à-dire quand un blocage n'est pas nécessaire. Ce coût est estimé par la mesure du temps d'exécution de chaque primitive à évaluer [GF00] [Car99].

Algorithme

Nous évaluons ensuite le coût de synchronisation d'INUKTITUT et de *POSIX Threads* par l'Algorithme 4.5. Nous mesurons le temps des primitives *lock/unlock* et *trylock/unlock*. Il est obtenu par l'exécution en N itérations d'un bloc contenant la primitive à évaluer. Toutes les primitives sont non bloquantes, c'est-à-dire que le verrou n'est pas pris. Les méthodes similaires sont aussi proposées dans [GF00]. La durée d calculée par la formule $d = (t1 – t0)/N$ est utilisée pour évaluer ce coût.

Algorithme 4.5 Estimation du coût minimum de synchronisation

```
t0 <- GetTime()
for  i=1  to  N do
  lock(m)
  unlock(m)
endfor
t1 <- GetTime()
```

Résultats et Analyses

Les plates-formes expérimentales de cet algorithme sont les mêmes que celles utilisées dans la section 4.1.1 (page 78). Le nombre d'itérations choisi est ***N=10000***. Les résultats sont dans les Tableau 4.5 et Tableau 4.6.

Plate-forme	Bibliothèque de *threads*	**INUKTITUT**	**POSIX**
I2	NPTL-0.60	0.13±0.0001 µs	0.13±0.0001 µs
Opata_2.4	Linuxthread-0.10	0.10±0.0001 µs	0.10±0.0001 µs
Opata_2.6	NPTL-0.60	0.08±0.0004 µs	0.08±0.0004 µs

Tableau 4.5 *Comparaison du temps d de lock/unlock entre INUKTITUT et POSIX Threads*

(I2 : IA64-biproc 900Mhz; Opata : Athlon-biproc 1,2GHz)

Plate-forme	Bibliothèque de *threads*	**INUKTITUT**	**POSIX**
I2	NPTL-0.60	0.12±0.0001 µs	0.12±0.0001 µs
Opata_2.4	Linuxthread-0.10	0.09±0.0001 µs	0.09±0.0001 µs
Opata_2.6	NPTL-0.60	0.07±0.0004 µs	0.07±0.0003 µs

Tableau 4.6 *Comparaison du temps d de trylock/unlock entre INUKTITUT et POSIX Threads*

(I2 : IA64-biproc 900Mhz; Opata : Athlon-biproc 1,2GHz)

L'observation des résultats dans ces tableaux montre que les primitives de synchronisation d'INUKTITUT ont une performance équivalente à celles-ci de *POSIX*. C'est-à-dire que le surcoût d'exécution de ces primitives d'INUKTITUT sur l'interface

de *POSIX Threads* est très faible. Nous constatons de plus que le coût d'exécution de ces primitives en utilisant le noyau 2.6 est un peu plus faible qu'en utilisant le noyau 2.4.

4.3 Bilan

Nous avons étudié dans ce chapitre la performance de l'interface de processus léger d'INUKTITUT (PTH) qui se base sur l'interface *POSIX Thread*. Via les comparaisons de quelques indicateurs de performance entre PTH et *POSIX Thread* sur les plates-formes différentes, nous notons que le surcoût d'exécution des primitives de PTH est faible : il est inférieur à 2% pour la création, à 1,7% pour la commutation de contexte et à 1% pour la synchronisation. Les surcoûts dus à l'utilisation de PTH sont négligeables par rapport à l'utilisation de *POSIX*.

De plus, nous avons exécuté des expériences sur deux noyaux différents de Linux (2.4.x et 2.6.x). Nous remarquons que le noyau 2.6.x est plus efficace que le noyau 2.4.x pour l'ensemble des opérations concernant les processus légers. En particulier, le surcoût de création d'un processus léger en utilisant le noyau 2.6.x est très faible par rapport celui obtenu en utilisant le noyau 2.4.x. Ces avantages proviennent des différentes améliorations du noyau 2.6.x [San03][Pran03] ainsi que de l’implantation NPTL de la bibliothèque de processus légers. INUKTITUT est capable d'exploiter ce gain de performance par son surcoût faible.

Les chapitres suivants présenteront les performances des autres bibliothèques d'INUKTITUT concernant la communication.

5

Performances des communications bipoints

Dans ce chapitre nous allons évaluer la bibliothèque de communication d'INUKTITUT. Cette bibliothèque permet de réaliser des communications par échange de messages actifs entre les processus du système. Les expériences et ses analyses se basent sur trois implantations au dessus de trois couches de transport (*Socket SysV* [Ste94], *Corba* [Vin98] et *GM*[Myr]). Nous présentons d'abord des indicateurs de performance pour évaluer ces implantations. Les sections suivantes présentent les évaluations de performance de ces implantations sous différents régimes d'opération de communication. Dans chaque section, nous présentons d'abord la méthode d'évaluation et puis nous analysons des résultats obtenus. Nous étudierons la performance du recouvrement calcul/communication offert par l'interface de communication d'INUKTITUT dans la dernière section. Enfin une conclusion termine ce chapitre.

5.1 Indicateurs de performance

Pour chaque étude de performance, il faut se baser sur certains indicateurs. L'objectif de cette section est donc de présenter les indicateurs caractéristiques du comportement de la communication entre processus. Nous étudions d'abord deux

indicateurs de base pour la communication bipoint et puis nous abordons des indicateurs de Hockney [HB94] utilisés dans nos évaluations.

5.1.1 Latence et débit

Une communication bipoint est caractérisée par deux paramètres de performance de base : la **latence** (*communication latency*, en anglais) et le **débit** (*throughput*, en anglais). La **latence** est l'intervalle de temps mesuré entre le moment où l'émetteur demande à envoyer un message et celui où le récepteur termine de recevoir ce message. De façon générale, si on suppose que le récepteur est toujours prêt et que la charge du réseau est nulle, la **latence** *t*, selon le modèle linéaire, est en fonction de taille *n* de message envoyé :

$$t(n) = \alpha + \beta n \qquad (5.1)$$

α : le coût fixe pour appeler les fonctions de l'envoi ou de la réception d'un message indépendamment de sa taille. Il est aussi appelé le coût d'amorçage.

n : la taille du message en octets

β : le temps de transmission d'un octet

Le **débit** δ est le taux de transfert (en octets par seconde) que la bibliothèque de communication est capable d'offrir :

$$\delta = \frac{n}{t} \qquad (5.2)$$

où t est la latence de transfert d'un message de taille n.

La latence et le débit sont utilisés comme deux indicateurs de base dans notre évaluation de performance des communications entre les processus. Hockney [HB94] propose en outre des indicateurs calculables à partir de latence mesurée. Ces indicateurs qui sont présentés dans la section suivante permettent d'évaluer la performance d'une bibliothèque de communication.

5.1.2 Indicateurs de performance de Hockney

Hockney [HB94] propose trois indicateurs principaux de performance : r_∞, $n_{½}$, t_0. L'indicateur r_∞ est le débit asymptotique de communication approchée quand la taille de message tend vers l'infini. Nous pouvons présenter cet indicateur en utilisant l'équation (5.2). En remplaçant (5.1) dans (5.2), nous obtenons :

$$\delta = \frac{n}{\alpha + \beta n} = \frac{1}{\frac{\alpha}{n} + \beta} \tag{5.3}$$

L'indicateur r_∞ est donc déterminé par :

$$r_\infty = \lim_{n->\infty} \delta = \lim_{n->\infty} \frac{1}{\frac{\alpha}{n} + \beta} = \frac{1}{\beta} \tag{5.4}$$

L'indicateur $n_{½}$ est la taille de message exigée pour atteindre la moitié du débit asymptotique. Cet indicateur peut être déterminé grâce aux équations (5.3) et (5.4) :

$$\delta_{n_{1/2}} = \frac{r_\infty}{2} = \frac{1}{2\beta} \text{ et } \delta_{n_{1/2}} = \frac{1}{\frac{\alpha}{n_{1/2}} + \beta} \tag{5.5}$$

De ces deux équations, nous obtenons $n_{1/2} = \frac{\alpha}{\beta}$ (5.6)

La latence décrite en (5.1) peut être réécrite en utilisant r_∞ et $n_{1/2}$, on a alors :

$$t(n) = \frac{\alpha}{\beta} \times \beta + \beta \times n = n_{1/2} \times r_\infty^{-1} + r_\infty^{-1} \times n \tag{5.7}$$

L'indicateur t_0, est le coût d'amorçage α, il est également représenté par $n_{½}$ et r_∞ comme :

$$t_0 = n_{1/2} \times r_\infty^{-1} \tag{5.8}$$

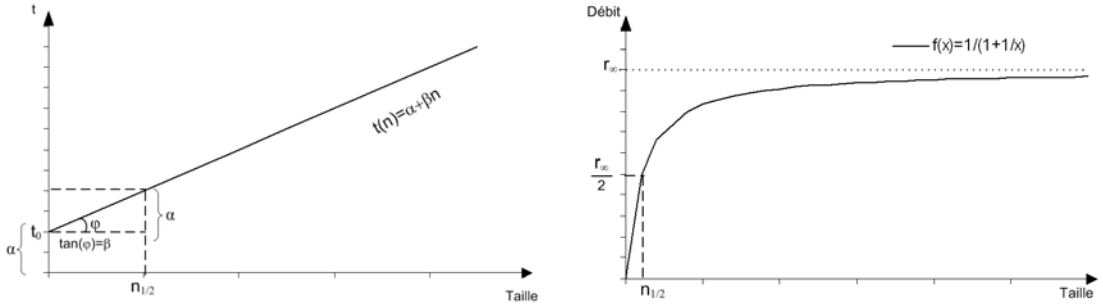


Figure 5.1 *Interprétation des paramètres r_∞, $n_{½}$ et t_0*

Ces trois indicateurs r_∞, $n_{½}$ et t_0 permettent d'évaluer des aspects différents de la performance d'un système de communication comme : le débit maximum offert par ce système, le coût d'amorçage. De plus, ces indicateurs peuvent être calculés grâce à l'indicateur de base : la latence *t* par l'utilisation de la technique de régression des moindres carrés. Nous utilisons ces indicateurs pour évaluer les performances des bibliothèques de communication dans ce chapitre. En outre, Hockney défini aussi un autre indicateur π_0, avec $\pi_0 = t_0^{-1}$, utilisé pour évaluer le débit d'une communication de petite taille de message. Néanmoins, nous n'utilisons pas cet indicateur parce que ce débit peut être estimé via les mesures de latence des messages de petite taille.

5.2 L'envoi et la réception d'un message

Dans cette partie, nous présentons d'abord deux suites de tests pour mesurer des métriques principales de la communication bipoint: la latence et le débit. Nous faisons ensuite des expériences avec l'interface de communication d'INUKTITUT et de *LAM MPI*. Nous analysons ensuite les résultats en utilisant les indicateurs des performances de Hockney [HB94].

5.2.1 Les suites de tests

Le test « Ping-pong » (COMMS1)

Nous utilisons le test *« ping-pong »* (COMMS1 [HB94]) pour mesurer la latence de la communication en fonction de la taille d'un message envoyé d'un processus *« ping »* à d'un processus *« pong »*. La latence est théoriquement égale (t_b - t_a) (Figure 5.2a). Généralement, nous ne pouvons pas obtenir directement cette latence sur un réseau de stations de travail parce que ces stations n'ont pas d'horloges synchronisées. Une façon de contourner ce problème est de mesurer la durée $t_{(a,c)}$ d'un aller-retour (*RTT - Round Trip Time*) d'un message entre ces deux processus : $t_{(a,c)}$ = (t_c - t_a) (Figure 5.2a). Supposons que le temps d'envoi et le temps de réception sont équivalents, nous obtenons la latence en divisant le temps $t_{(a,c)}$ par deux. De cette manière, il est possible de mesurer le temps sur une seule machine. Le test « ping-pong » consiste donc à envoyer un message d'une taille donnée d'un processus *« ping »* à un processus *« pong »* qui renvoie ce message au processus *« ping »* dès qu'il le reçoit.

Afin d'éliminer des brouillages de système, il faut répéter l'opération de communication [SCP97]. C'est pour cela que la latence (temps d'envoi) t d'une expérience est déterminée grâce à la durée $t_{(0,1)}$ composé de i itérations du *RTT* d'un message (Figure 5.2b) :

$$t = \frac{t_{(0,1)}}{2 \times i} = \frac{(t_1 - t_0)}{2 \times i} \tag{5.9}$$

Nous pouvons en outre évaluer l'effet de la taille du message sur la latence en faisant varier la taille des messages envoyés. Ce test permet aussi de déterminer le débit δ de l'envoi d'un message de taille n en utilisant la latence t correspondante estimé et l'équation 5.2 (page 92). L'évaluation du débit est réalisée par le test de l'échange des messages de différentes tailles.

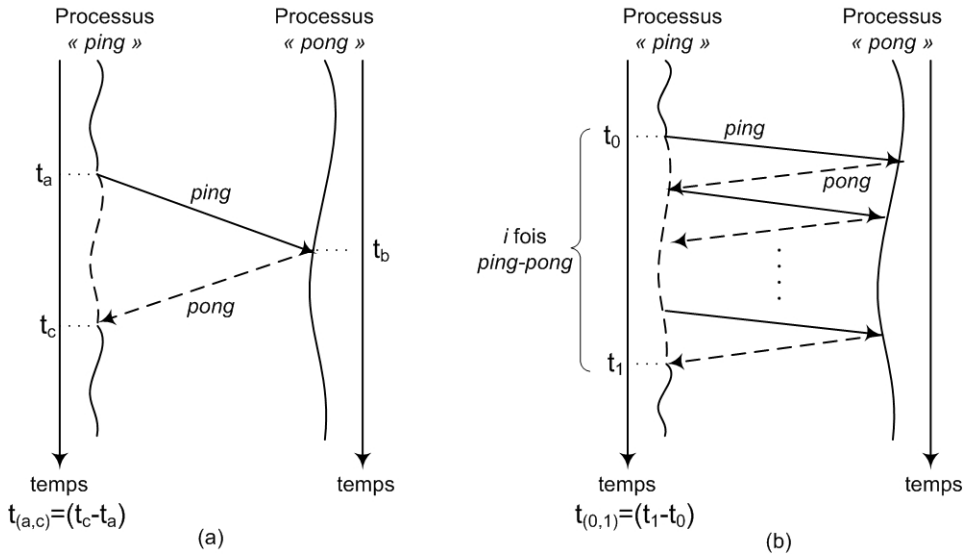


Figure 5.2 *Le test « ping-pong »: (a) Un RTT; (b) i iterations de RTT*

Dans INUKTITUT, le traitement d'un message arrivé est réalisé via un service (*cf.* la section 3.4.3, page 56) qui est généralement implanté par un processus léger. Les durées $t_{(a,c)}$ d'un *RTT* et $t_{(0,1)}$ de *i* itérations du *RTT* sont mesurées comme dans la Figure 5.3 :

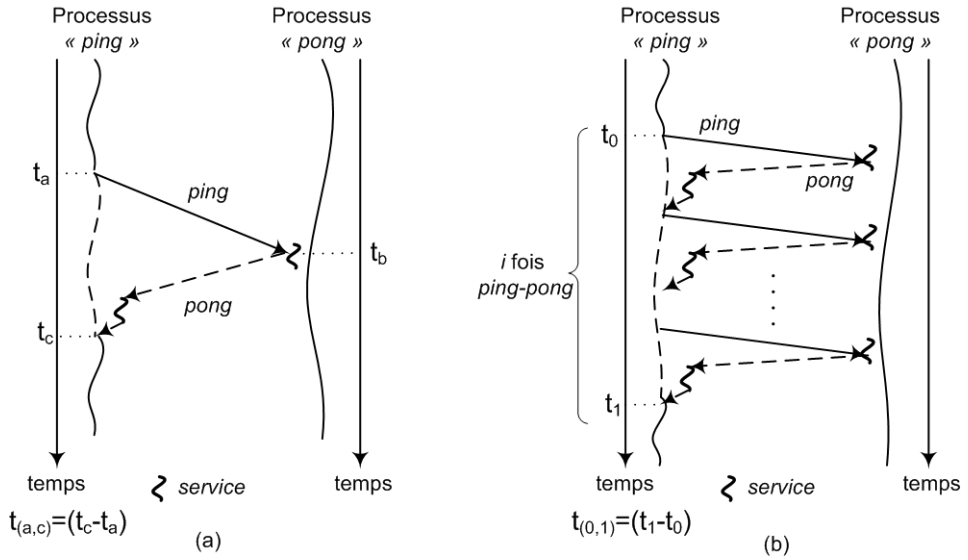


Figure 5.3 *Mesure de temps dans le test « ping-pong » pour INUKTITUT : (a) Un RTT; (b) i iteration de RTT*

Le test « Ping »

Nous avons un autre test permettant de mesurer le débit de l'envoi de message entre deux processus du système. Nous baptisons ce test *« Ping »* dans ce chapitre. Dans ce test, un processus (processus *« ping »*) envoie *i* messages de taille *n* à un autre processus (processus *« pong »*). Le processus *« pong »*, après avoir reçu les *i* messages, envoi une réponse *ACK* (*acknowledgement*) au processus *« ping »* (Figure 5.4a). Cette méthode est proposée dans [KKJ01] [ILM98].

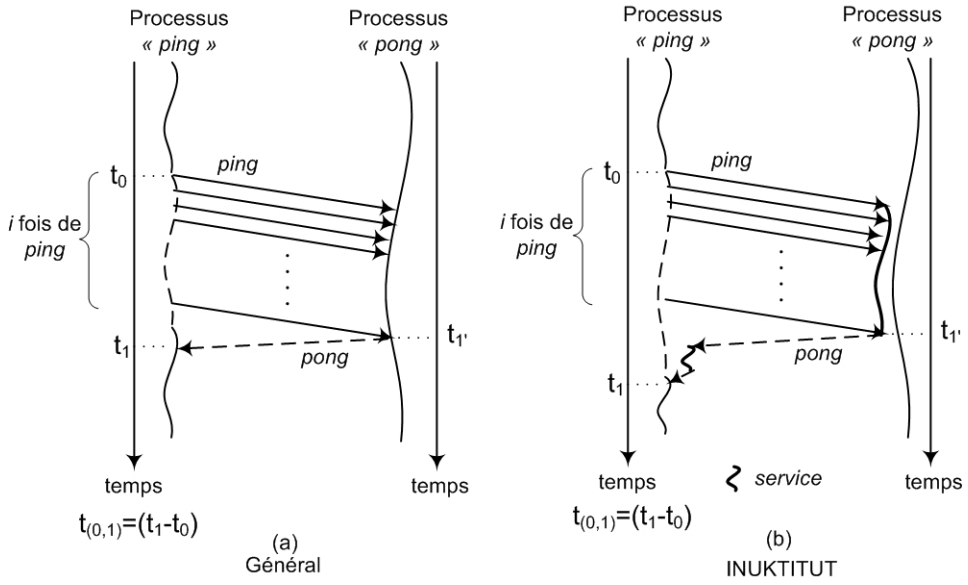


Figure 5.4 *Mesure de temps dans le test « ping »*

Supposons que le temps $t_{(1',1)}$ (égale $t_1 - t_{1'}$) d'une réponse ACK est négligeable par rapport à la durée totale $t_{(0,1)}$ mesuré pour le processus *« ping »*. Cela est possible si le nombre *i* est grand. La durée $t_{(0,1)}$ considérée est le temps d'envoi de *i* messages de taille *n* d'un processus *« ping »* à un processus *« pong »*. Le débit de transfert est donc déterminé en divisant le nombre d'octets envoyés pendant ce laps de temps [ILM98] :

$$\delta(n) = \frac{i \times n}{t_{(0,1)}} = \frac{i \times n}{(t_1 - t_0)} \qquad (5.10)$$

De même que pour le test *« ping-pong »* ci-dessus, nous remarquons que la durée $t_{(0,1)}$ mesurée dans les expériences d'INUKTITUT prendre en compte le temps de traitement des messages à l'arrivée (Figure 5.4b).

5.2.2 Résultats et analyses

Les expérimentations de ces tests sont réalisées pour deux machines sur I-Cluster1 (*cf.* Annexe A). Chaque processus (*« ping »* ou *« pong »*) est placé sur une machine différente. Les tailles des messages utilisées dans ces expériences sont comprises entre 0 octet et 8M octets. Le temps mesuré est en microseconde et nous utilisons *« gettimeofday »* (*cf.* Annexe B.1) comme fonction de prise de temps dans toutes les expériences dans ce chapitre.

Bien que le nombre d'itération pour l'opération de communication soit 20 pour éliminer des brouillages dû au système [SCP97], le nombre *i* d'itération pour chaque expérience élémentaire est choisie selon la taille du message: 5000 pour les tailles de message de 0 à 65536 octets, 200 pour celles supérieures ou égales à 128Koctets.

Le Test *« Ping-pong »* avec SOCKNET et CORNET

Nous étudions d'abord le comportement du test « ping-pong » avec deux implantations différentes de INUKTITUT : SOCKNET et CORNET. Les intervalles de confiance (*cf.* Annexe B.2) pour toutes les mesures sont inférieurs à 1% du temps d'envoi *t* (latence *t*, *cf.* l'équation 5.9, page 95)

En observant la Figure 5.5 et la Figure 5.6, nous constatons que :

- SOCKNET et CORNET donnent des résultats similaires au niveau de la forme des courbes.

- SOCKNET donne une meilleure performance au niveau de temps d'envoi *t* par rapport au réseau CORNET : gain de 40% à 50% pour les messages de taille inférieure ou égale 1024 octets et de 7% à 23% pour les messages de taille supérieure à 2048 octets.

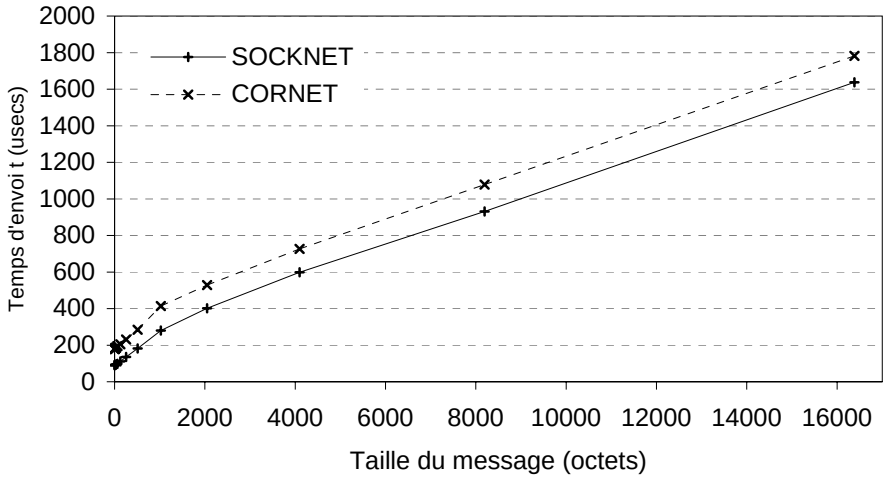


Figure 5.5 *Temps d'envoi t pour les messages jusqu' à 16Koctets*

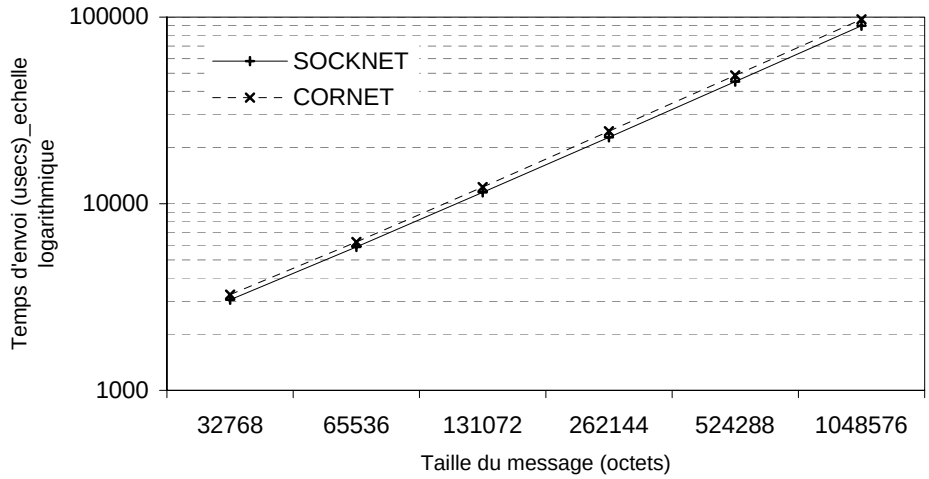


Figure 5.6 *Temps d'envoi t pour les messages de 32Koctets à 1Moctets*

- Nous avons une différence de la pente des courbes entre le début d'envoi de messages et la suite des envois pour les deux implantations. Le point de

changement est aux environs de 1500 octets ce qui correspond au MTU[1] [Ste98] du protocole Ethernet sur les machines considérées.

	Débit r_∞(1Moctets)	**Amorçage (t_0)**	**Mi-débit ($n_{½}$)**
SOCKNET	11,14 Mo/s	169,75 µs	1,94 Ko
CORNET	10,30 Mo/s	225,42 µs	2,38 Ko

Tableau 5.1 *Les indicateurs r_∞, t_0, $n_{1/2}$ pour SOCKNET et CORNET*

Le Tableau 5.1 compare les indicateurs de performance de SOCKNET et ceux de CORNET. Il s'avère que SOCKNET offre une meilleure performance par rapport à CORNET.

Nous obtenons à ce niveau une conclusion partielle sur les deux implantations : le temps de transmission d'un message par CORNET est plus élevé qu'en utilisant SOCKNET dans les mêmes tailles de message. La raison est que le protocole CORNET utilise l'ORB CORBA qui rajoute une pile de protocole pour émuler GIIOP de CORBA au dessus de TCP/IP, à la différence de SOCKNET qui exploite directement la pile TCP/IP.

Le Test « Ping-pong » avec SOCKNET et MPI LAM 6.5.9

Nous étudions la latence et le débit de l'implantation de SOCKNET avec le test *« ping-pong »*. Nous les comparons à ceux obtenus avec *MPI LAM* version 6.5.9. Pour *LAM*, la version de RPI (*Request Progression Interface*) utilisée est *c2c* (*client-to-client*) au dessus de TCP/IP. Les primitives employées pour ces implantations sous *MPI* sont de type *Send/Receive* synchrone.

Les comparaisons sont réalisées au niveau de temps et au niveau des indicateurs de performance r_∞, t_0 et $n_{1/2}$. Le Tableau 5.2 compare les indicateurs de performance

[1] *Maximum Transfer Unit, en anglais*

d'INUKTITUT à ceux de *MPI* dans deux cas : les messages de taille inférieure ou égale à 64K octets et celles supérieures à 64 K octets (jusqu'à 8M octets).

En observant la Figure 5.7 et la Figure 5.8, nous constatons que :

- La courbe du temps *t* d'envoi d'un message en utilisant SOCKNET se présente sous la forme d'une fonction linéaire. Cela confirme la dépendance trouvée dans la section 5.1.1 (*cf.* l'équation 5.1, page 92). Néanmoins, la courbe obtenue en utilisant *LAM* a une différence de pente en un point de taille de 64K octets (Figure 5.7). La raison est que *LAM* a deux protocoles différents pour envoyer les messages : *short* et *long*. Le message est envoyé en une seule opération de transfert dans le protocole *short*. Il est utilisé dans le cas de transfert des messages de *« petite »* taille. Le protocole *long*, quant à lui, est utilisé pour envoyer des messages de grande taille. Deux opérations de transfert sont réalisées dans ce cas : l'envoi d'une enveloppe et après l'envoi du message. Le point de changement de protocole par défaut est une taille de 64K octets [LAM65].

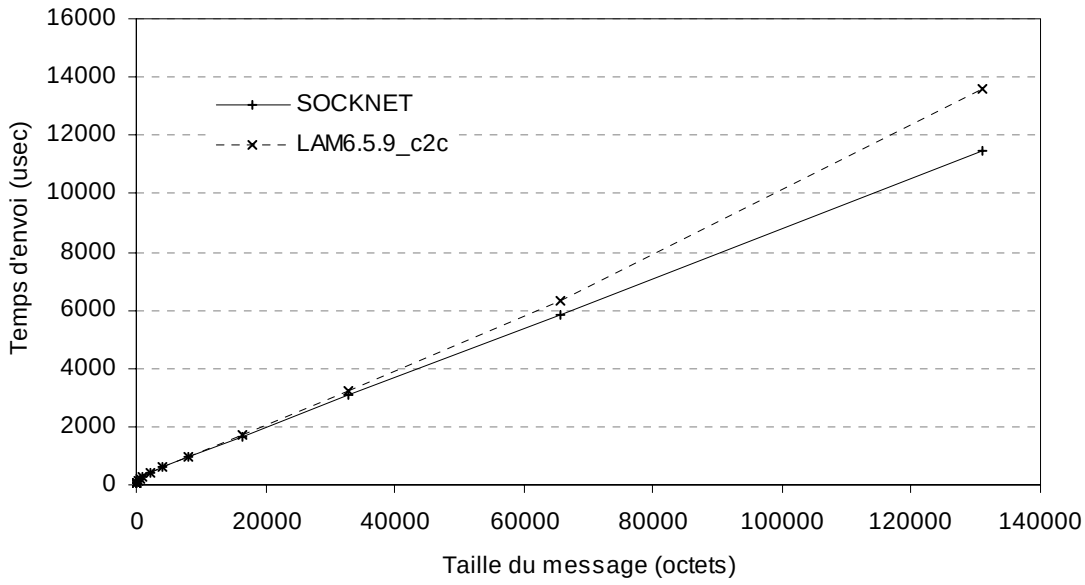


Figure 5.7 *Comparaison du temps d'envoi t entre SOCKNET et LAM6.5.9 pour les messages de taille de 0 à 128K octets sur I-Cluster1*

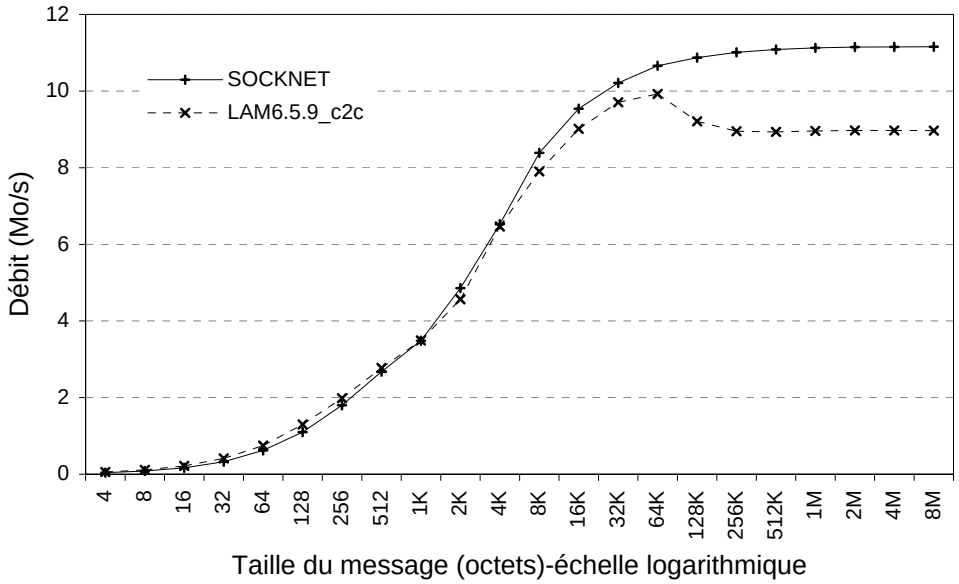


Figure 5.8 *Comparaison du débit δ entre SOCKNET et LAM6.5.9 pour les messages de taille de 4 octets à 8M octets sur I-Cluster1*

- *LAM* donne un meilleur temps d'envoi *t* par rapport à SOCKNET pour les tailles de message inférieures à 1024 octets (le débit correspondant de *LAM* est plus grand que celui de SOCKNET, Figure 5.8). C'est pour quoi *LAM* a un temps d'amorçage t_0 plus faible que celui de SOCKNET (Tableau 5.2). Cela s'explique aussi par une augmentation de la taille de message nécessaire pour atteindre le mi-débit $n_{1/2}$. La raison est que *LAM* a un surcoût d'envoi de message plus faible que INUKTITUT pour les messages de petite taille : un message est envoyé par une seule requête qui se compose d'une entête et d'un contenu de message. Du côté d'INUKTITUT, l'entête et le contenu du message sont envoyés par des requêtes différentes. (*cf.* 3.4.4, page 58)

- SOCKNET, à l'inverse, est plus efficace que *LAM* pour les messages de taille supérieure à 1024 octets. Nous avons un gain d'environ de 15% à 20% sur le temps d'envoi *t* pour des messages de taille supérieure à 64 K octets. Le gain de débit correspondant est de 18% à 24 % par rapport à *LAM* (Figure 5.8). Ici le

pourcentage est plus haut que dans le cas précédent parce que *LAM* utilise le protocole *long* pour envoyer les messages de taille supérieure à 64 K octets (comme nous abordons ci-dessus). Dans ce protocole, les messages sont envoyés réellement après que le processus de réception ait lancé le *receive* correspondant. On a ici un surcoût de synchronisation entre l'expéditeur et la réception. Du côté de INUKTITUT, la communication est toujours asynchrone pour toutes les tailles de message envoyé.

- La différence de la pente au début de la courbe de temps d'envoi (Figure 5.7) est due à la limitation de MTU du protocole TCP/IP (MTU = 1500 octets sur les machines considérées). L'envoi des messages de taille supérieure à cette limitation provoque un surcoût de fragmentation.

	Débit r_∞(64Koctets)	**Amorçage (t_0)**	**Mi-débit ($n_{½}$)**
SOCKNET	10.82 Mo/s	138.32 µs	1.53 Ko
LAM6.5.9_c2c	10.06 Mo/s	121.99 µs	1.26 Ko

	Débit r_∞(8Moctets)	**Amorçage (t_0)**	**Mi-débit ($n_{½}$)**
SOCKNET	11.15 Mo/s	138.32 µs	1.58 Ko
LAM6.5.9_c2c	8.97 Mo/s	121.99 µs	1.12 Ko

Tableau 5.2. *Les indicateurs r_∞, t_0, $n_{1/2}$ pour SOCKNET et LAM 6.5.9 (mode c2c) sur I-Cluster1*

À partir du Tableau 5.2, bien que le temps d'amorçage de SOCKNET soit plus important que celui de *LAM*, nous constatons aussi qu'il donne un meilleur débit asymptotique par rapport à *LAM* dans tous les cas. Un gain d'environ 7,5% sur ce débit est constaté pour le cas de taille inférieure ou égale à 64K octets et un gain d'environ 24% est constaté quand la taille est supérieure à 64K octets.

Nous avons en outre des expérimentations de ces tests réalisées pour deux machines sur **I-Cluster2**, une grappe de réseau haut débit (*cf.* Annexe A). Ces expérimentations de l'implantation SOCKNET et de LAM6.5.9_c2c utilisent le protocole TCP/IP au

dessus de Myrinet [Myr] pour échanger des messages. Les autres paramètres sont semblables aux expérimentations ci-dessus (*cf.* page 100)

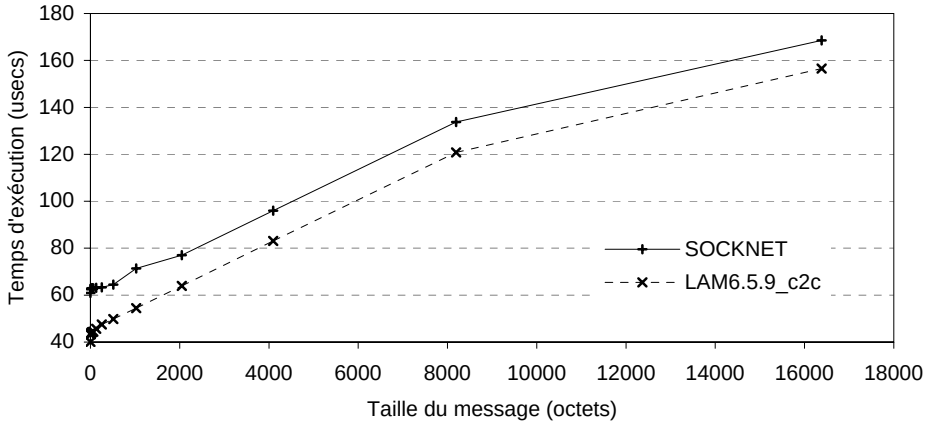


Figure 5.9 *Comparaison du temps d'envoi t entre SOCKNET et LAM6.5.9 pour les messages de taille de 0 à 16K octets sur I-Cluster2*

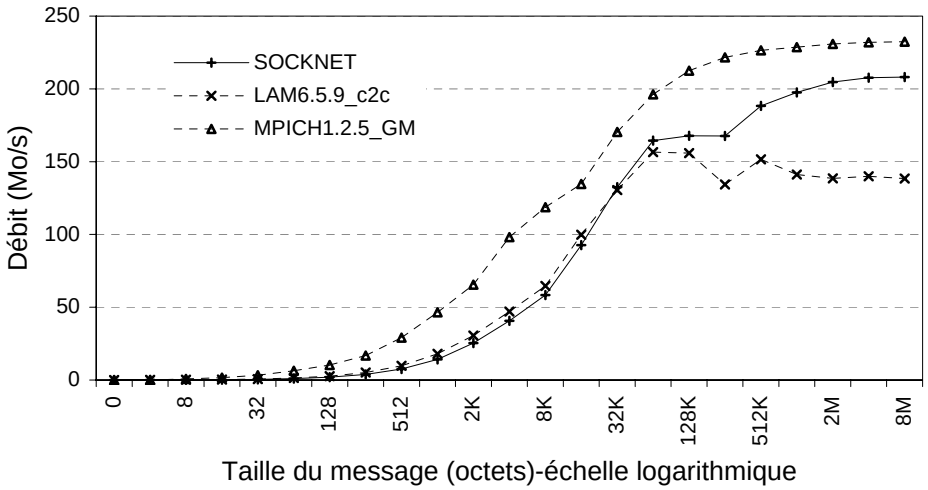


Figure 5.10 *Comparaison du débit δ entre SOCKNET, LAM6.5.9 et MPICH1.2.5(GM) pour les messages de taille de 4 octets à 8M octets sur I-Cluster2*

En observant la Figure 5.9 et la Figure 5.10, nous constatons que :

- *LAM* donne un meilleur temps d'envoi *t* par rapport à SOCKNET pour les tailles de message inférieures à 32K octets. Nous avons expliqué la cause de cette

phénomène dans le cas sur I-Cluster1 ci-dessus (*cf.* page 102) : *LAM* a un surcoût d'envoi de message plus faible que INUKTITUT pour les messages de petite taille

- SOCKNET est plus efficace que *LAM* pour les messages de taille supérieure à 32K octets. Nous avons un gain d'environ de 2% à 50% sur le débit par rapport à *LAM* pour les messages de taille supérieure à 64K octets (Figure 5.10). *LAM* utilise le protocole *long* pour envoyer les messages de taille supérieure à 64 K octets (comme nous abordons ci-dessus).

- Nous obtenons aussi la différence de la pente au début de la courbe de temps d'envoi (Figure 5.9) due à la limitation de MTU du protocole TCP/IP (MTU est égal 9000 octets sur les machines de I-Cluster2).

	Débit r_∞(64Koctets)	**Amorçage (t_0)**	**Mi-débit ($n_{½}$)**
SOCKNET	194.63 Mo/s	67.48 µs	13.45 Ko
LAM6.5.9_c2c	173.40 Mo/s	49.75 µs	8.83 Ko

	Débit r_∞(8Moctets)	**Amorçage (t_0)**	**Mi-débit ($n_{½}$)**
SOCKNET	211.93 Mo/s	67.48 µs	13.45 Ko
LAM6.5.9_c2c	138.21 Mo/s	49.75 µs	8.83 Ko

Tableau 5.3 *Les indicateurs r_∞, t_0, $n_{1/2}$ pour SOCKNET et LAM 6.5.9 (mode c2c) sur I-Cluster2*

Le Test « Ping »

SOCKNET et LAM sur I-Cluster1

Nous étudions d'abord le débit de SOCKNET sur I-Cluster1 par le test *« ping »*. Nous les comparons aussi à ceux de *LAM*. La Figure 5.11a présente le comportement du débit en fonction de la taille du message pour deux couches. Via l'étude de ces courbes, on constate que :

- Pour les tailles de message inférieur à 64 K octets, peu de différence existe entre les deux courbes.
- Une dégradation de performance au niveau du débit pour les messages de taille supérieure à 64 K octets. La raison est que *LAM* change du protocole *« short »* au protocole *« long »* à partir de 64K octets.

- Le débit maximum reçu en utilisant l'interface SOCKNET dans ce test est de 11,15 Mo/s. Il est équivalent au débit asymptotique r_∞ dans le test précédent.

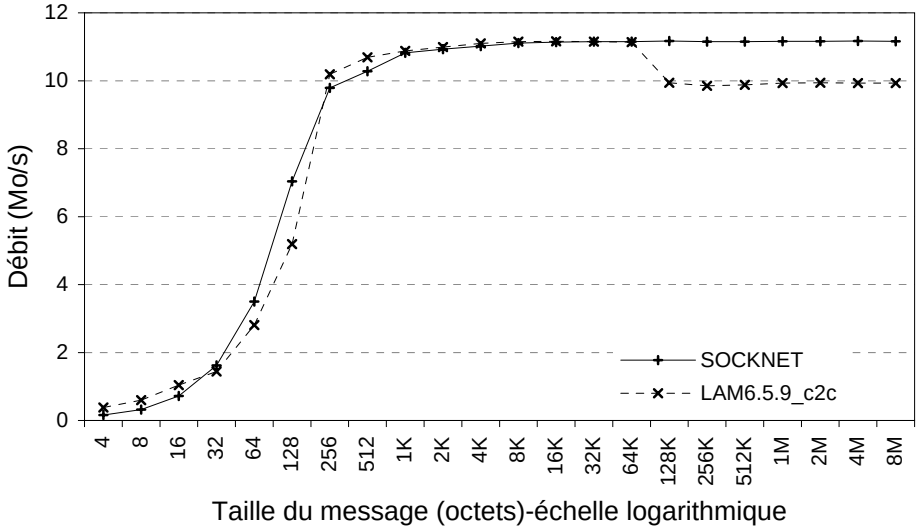


Figure 5.11a *Comparaison du débit δ entre SOCKNET et LAM6.5.9 par le test « ping » sur I-Cluster1*

GMNET, MPICH[2] et GM[2] sur I-Cluster2

Nous présentons ensuite la comparaison de débit entre GMNET (*cf.* section 3.5.2, page 65), *MPICH* et *GM* sur ***I-Cluster2*** (*cf.* Annexe A) par ce test dans la Figure 5.11b[3]. Ces débits sont fonction de la taille du message de 0 à 20K octets. Dans cette comparaison, nous observons que le débit de GMNET est plus faible que celui de *MPICH* et de *GM*. Ce débit faible de GMNET est causé par le surcoût d'appel de fonction nécessaire à l'envoi d'un message INUKTITUT. De plus, le niveau de

[2] MPICH version 1.2.5.10, GM version 2.0.9
[3] Ce résultat est réalisé dans le stage de 3e année ENSIMAG d'Everton Hermann (Juin 2004)

complexité d'un message INUKTITUT (sous forme d'un tableau de (pointeur, taille)) explique aussi le temps de transfert plus important. Pourtant, dans INUKTITUT, les messages actifs déclanchent des appels de service dans un environnement multi-processus léger. Dans le cas de *MPICH* ou de *GM*, la réception d'un message est de manière explicite dans un environnement mono-processus léger.

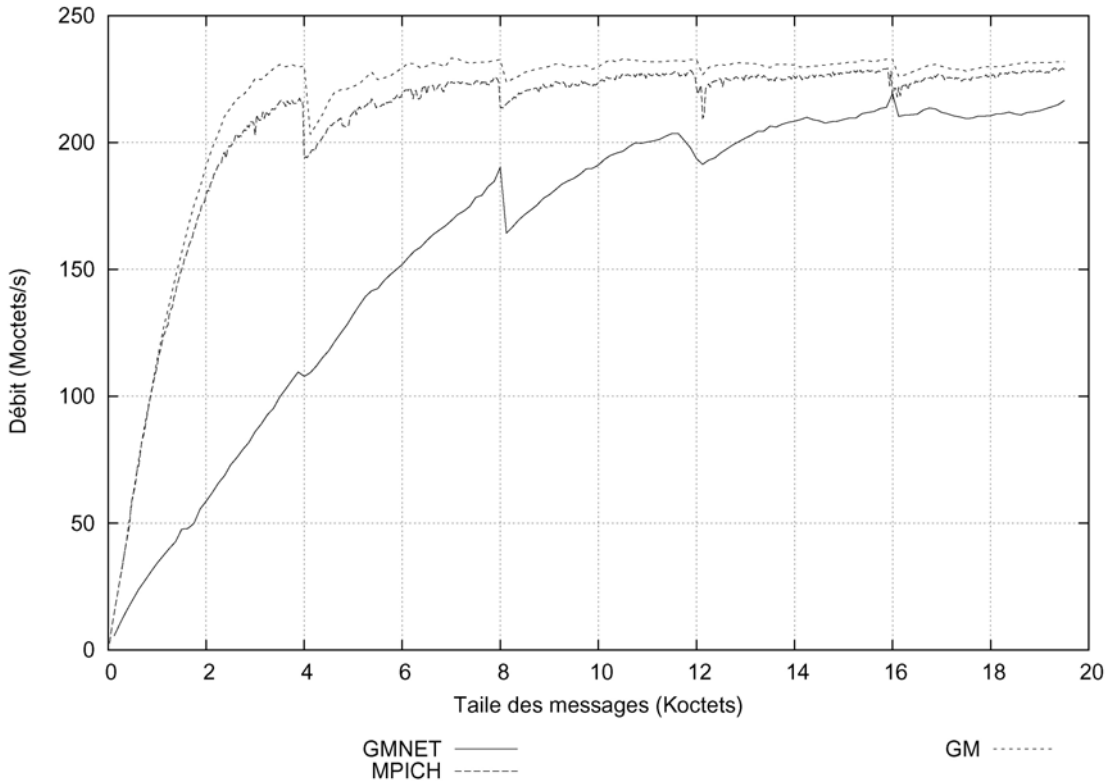


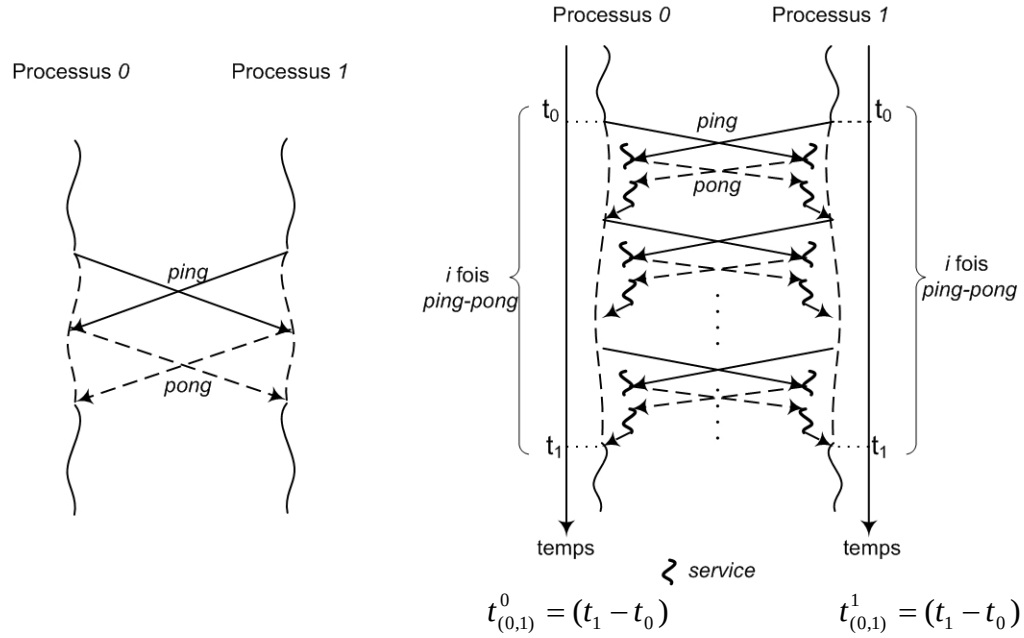
Figure 5.11b *Comparaison du débit (Mo/s) entre GMNET, MPICH et GM par le test « ping » sur I-Cluster2*

5.3 L'échange simultané de message

5.3.1 Le test COMMS2

Nous avons un autre test de communication. C'est le test COMMS2 [HB94] où chaque processus émet un message vers l'autre processus. Deux processus envoient simultanément le message entre eux et les renvoient (*« ping-pong »* simultané, Figure

5.12a). L'objectif du test COMMS2 est d'évaluer la latence et le débit des communications simultanées entre deux processus dans le système. L'opération d'échange simultané de message entre des processus est utilisée fréquemment dans le calcul parallèle, par exemple dans des applications de décomposition de domaine, deux processus s'échangent simultanément des valeurs aux frontières. Nous pouvons en outre obtenir dans le test COMMS2 un débit plus grand que celui possible dans le test COMMS1 [HB94], pour peu que les réseaux sous-jacents autorisent des communications bidirectionnelles.



(a) : Test COMMS2 (b) : Mesure de temps pour INUKTITUT

Figure 5.12 *Echange simultané de message*

Supposons que nous ayons deux processus : 0 et 1 qui échangent simultanément des messages, nous mesurons la durée $t^0_{(0,1)}$ de i *itérations* du *RTT* sur le processus 0 et celle $t^1_{(0,1)}$ sur le processus 1 (Figure 5.12b). Nous utilisons ces durées pour déterminer la latence *t* de chaque processus par l'équation 5.9 (page 95) comme dans le test *« ping-pong »* (section 5.2.1, page 95) et puis nous calculons aussi le débit δ de chaque

processus grâce à cette latence via l'équation 5.2 (page 92). L'évaluation de la latence t et du débit δ est donc réalisée par l'échange des messages de différentes tailles.

5.3.2 Les résultats de SOCKNET

Les expériences de ce test sont réalisées pour des tailles de message comprises entre 0 et 65536 octets. La plate-forme est l'I-Cluster1 (*cf.* Annexe A) et les autres paramètres sont semblables aux tests *« ping-pong »* (*cf.* section 5.2.2, page 98).

La Figure 5.13 présente la comparaison de débit entre deux processus sur deux machines qui envoient simultanément des messages. Nous constatons qu'il n'y a pas de différence entre les deux processus au niveau de temps mesurés, ceux-ci se confondent pratiquement. On se rend compte que l'envoi simultané de message entre deux processus par INUKTITUT est symétrique.

Les courbes représentant le temps d'envoi de messages simultanés par un processeur, comparé à la durée du *« ping-pong »* classique (COMMS1), sont présentées dans la Figure 5.14. Nous observons que le test *« ping-pong »* classique donne la meilleure performance par rapport au test d'envoi simultané. Ce test montre néanmoins que INUKTITUT profite de débit offert par un réseau parce que les messages arrivent et sortent de deux processeurs en même temps.

Nous calculons le débit asymptotique r_∞ sur chaque processeur par l'intermédiaire de ses résultats de temps d'envoi. Nous avons r_∞(64Ko, Processus 0) = 9.672 Mo/s, et r_∞ (64Ko, Processus 1) = 9.672 Mo/s. Nous avons aussi r_∞(Processus 0) + r_∞(Processus 1) = 19.344 Mo/s. C'est le débit asymptotique de communication simultanée dans les deux sens. Nous trouvons que ce débit est supérieur à celui de test *« ping-pong »* classique car le réseau *Fast Ethernet* utilisé autorise l'envoi et la réception simultanée de paquet (Tableau 5.2, page 103). En conclusion, la performance d'INUKTITUT en *« ping-pong »* simultané paraît raisonnable.

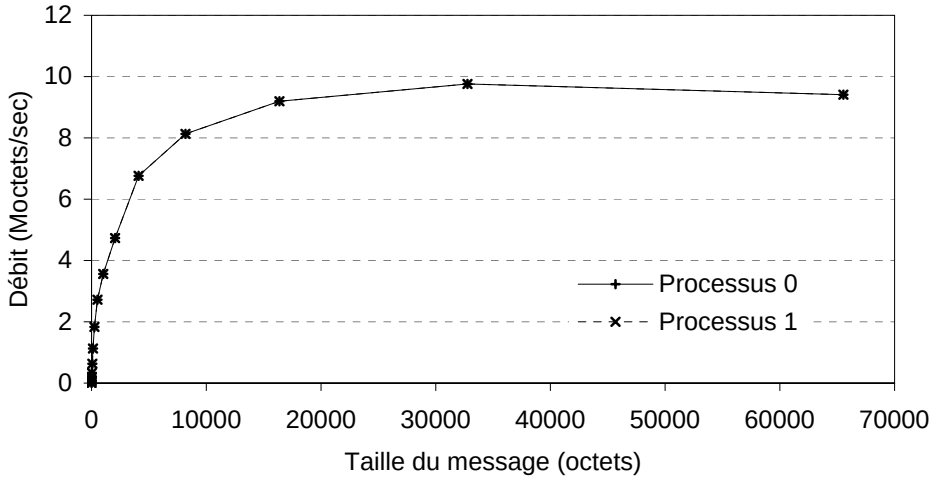


Figure 5.13 *Comparaison du débit δ entre processus 0 et processus 1*

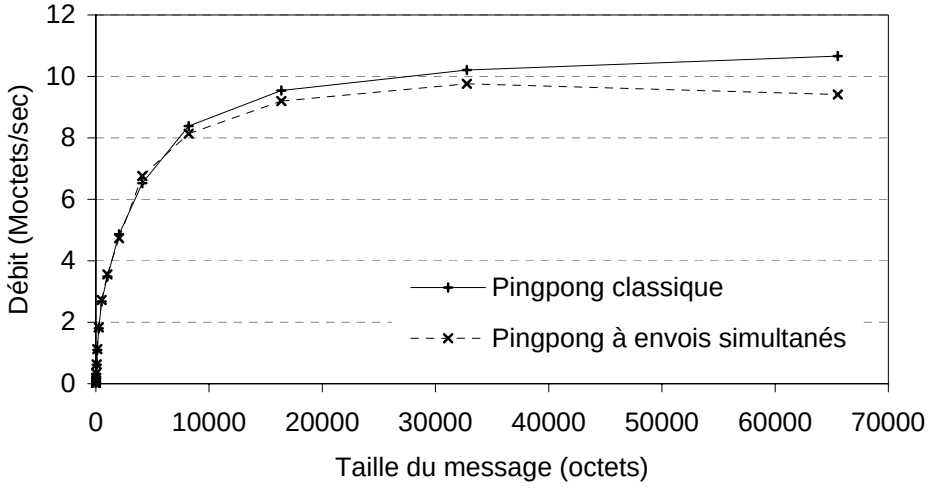


Figure 5.14 *Comparaison du débit δ entre le test « ping-pong » classique (COMMS1) et « ping-pong » dans lequel les messages sont envoyés simultanément*

5.4 L'envoi en parallèle

5.4.1 Objectif et Méthode d'évaluation

Ce test est une simulation d'une opération utilisée fréquemment dans le calcul parallèle, dans laquelle un calcul sur un processus est divisé en plusieurs sous calculs.

Dans ce test, le processus *ping* envoie des messages à tous les autres processus *pong* qui renvoient chacun leur réponse au processus *ping* (Figure 5.15) [Gin97]. L'objectif de ce test est d'évaluer des métriques de performance (*latence* et *débit*) dans le cas de l'envoi de plusieurs messages en parallèle. Ce schéma de communication ressemble à celui de la diffusion ou de la réduction dans une communication collective. Nous utilisons pourtant les primitives de communication bipoints pour exécuter les communications dans ce test. Les expériences utilisant les algorithmes parallèles de la bibliothèque de communication collective (ICS) sont présentées dans le chapitre suivant. Nous utilisons aussi ce test pour évaluer deux modèles de gestion de liaison (*cf.* 3.5.1, page 64) : un processus léger pour toutes les liaisons (1:n) et un processus léger pour chaque liaison (1:1).

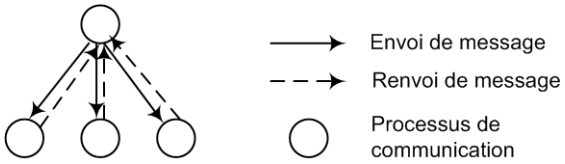


Figure 5.15 *Envoi en parallèle*

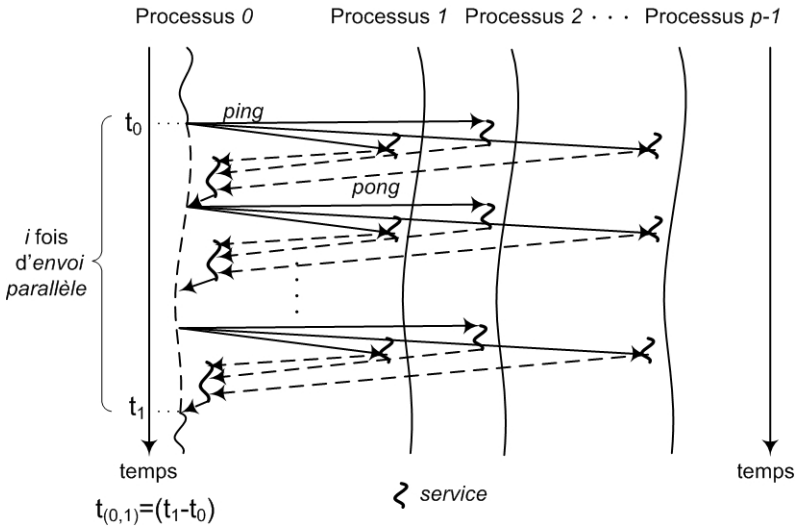


Figure 5.16 *Mesure de temps avec le test de l'envoi parallèle avec INUKTITUT*

La durée d'un *RTT* d'envoi parallèle est mesurée entre le temps d'envoi des messages depuis un processus (processus *0*) vers tous les autres processus (*p-1* processus) et le temps de réception de toutes leurs réponses. La latence *t* de cette opération est déterminée grâce à la durée $t_{(0,1)}$ de *i* itérations de ce *RTT* (Figure 5.16) et l'équation 5.9 (page 95).

5.4.2 Évaluation des résultats

Les expérimentations de ces tests sont réalisées en utilisant 4, 8, 16 et 32 processus sur les machines de la grappe I-Cluster1. Chaque processus est placé sur une machine différente. Les tailles des messages envoyés sont comprises entre 0 et 1M octets. L'interface de communication utilisée est SOCKNET et les autres paramètres de la plate-forme de test sont semblables au test précédent (*cf.* section 5.3.1, page 107).

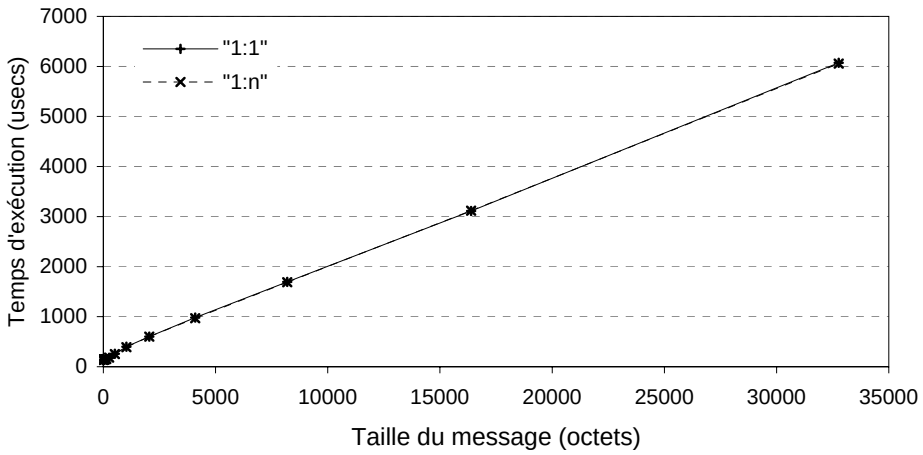


Figure 5.17 *Comparaison du temps d'exécution entre deux modes : (1:1) et (1:n) pour* ***4 processus*** *pour messages jusqu'à 32Ko*

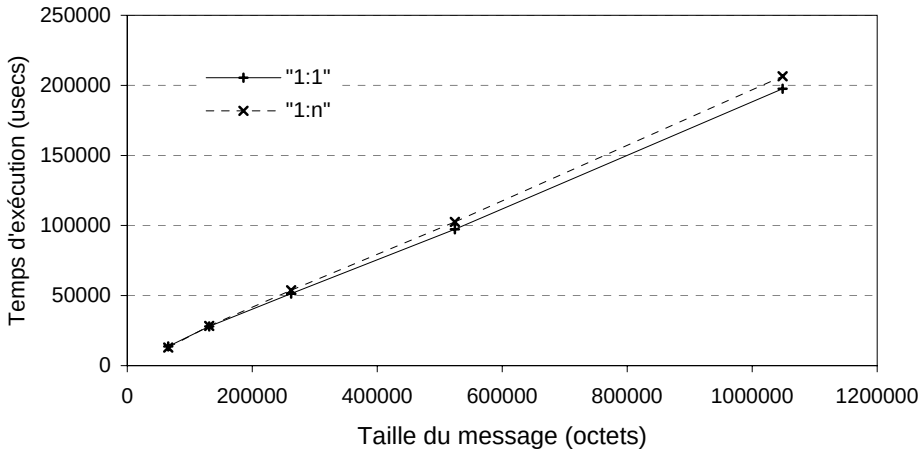


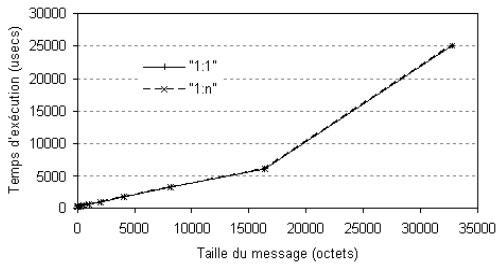
Figure 5.18 *Comparaison du temps d'exécution entre deux modes : (1:1) et (1:n) pour **4 processus** pour des gros messages*

Les Figure 5.17 et Figure 5.18 présentent le comportement de deux modes de connexion en fonction de la taille du message envoyé pour 4 processus. Nous constatons que :

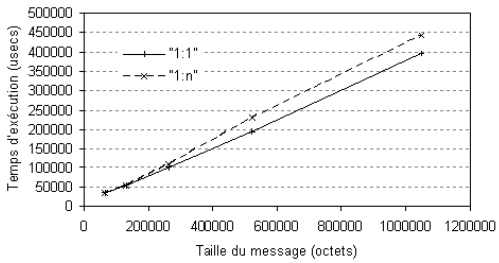
- Peu de différence existe entre les deux modes de liaison pour les messages de petite taille. Les points de valeur sont pratiquement identiques à chaque taille de message (Figure 5.17).

- Pour les messages de grande taille, le mode (1:1) donne une meilleure performance, gain de 2% à 5% par rapport au mode (1:n) (Figure 5.18).

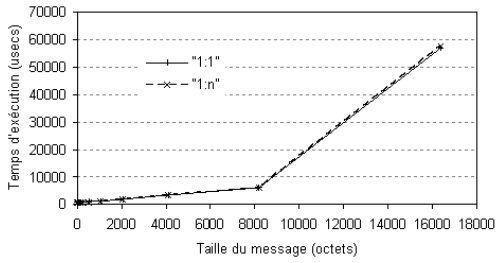
La Figure 5.19 présente aussi la comparaison du temps d'envoi des messages entre deux modes de connexion en fonction de la taille pour 8, 16, 32 processeurs. Nous avons les mêmes caractéristiques pour les messages de petite et de grande taille par rapport au cas de 4 processeurs. Néanmoins, en observant la Figure 5.19a, la Figure 5.19c et la Figure 5.19e, nous constatons que l'on a une grande différence de la pente entre le début et la fin des courbes. La taille *n* des messages au point de changement dépend du nombre *p* de processus communication et *n* x *p* = *constant* : 16 x 8 = 8 x 16 = 4 x 32 =128.



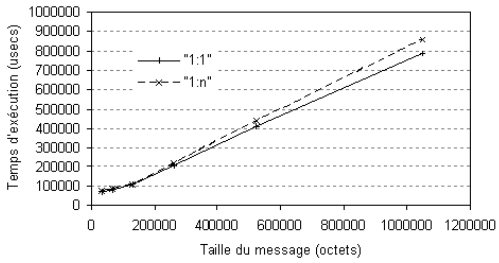
(a) 8 processeurs



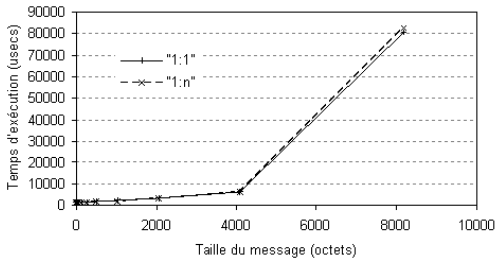
(b) 8 processeurs



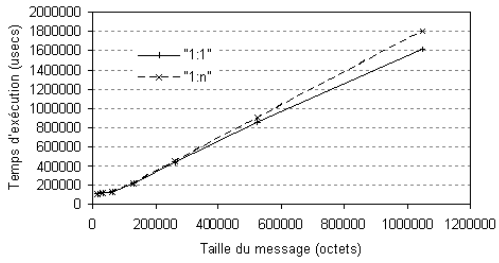
(c) 16 processeurs



(d) 16 processeurs



(e) 32 processeurs



(f) 32 processeurs

Figure 5.19 *Comparaison du temps d'exécution entre deux modes : (1:1) et (1:n) pour 8, 16 et 32 processeurs et pour des petits et des gros messages*

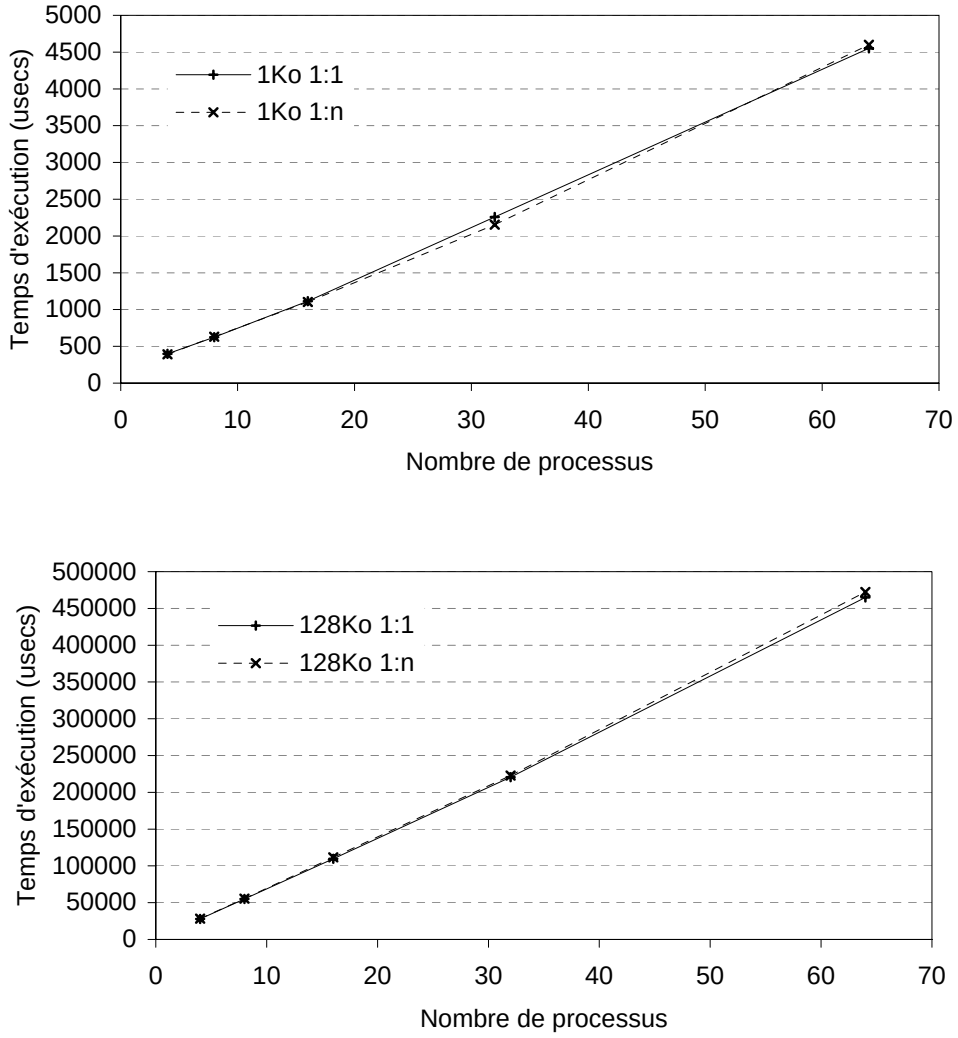


Figure 5.20 *Temps d'envoi de message en fonction du nombre de processus pour la taille de 1Ko et de 128Ko par deux modes de liaison*

5.5 Le recouvrement calcul-communication

Le recouvrement des communications par du calcul regroupe des stratégies à l'exécution qui permettent de masquer les délais de communication par du calcul de l'application. Ceci une des méthodes pour atteindre la meilleure performance au niveau d'application. En effet, l'objectif principal du recouvrement est de réduire le temps

d'exécution total d'une application en utilisant des architectures de matériel et de logiciel qu'offrent les progrès concourants de communication et de calcul.

Du côté de matériel, ce recouvrement peut être réalisé en utilisant le processeur au bas niveau dans la communication. Cette technique permet alors de libérer le processeur principal de la communication et d'utiliser plus de ses cycles pour le calcul utile [DS03]. Par exemple, le réseau Myrinet [Myr] avec sa carte LANai et son mécanisme de DMA permet de décharger le processeur principal de la machine hôte d'une charge de travail de communication (*cf.* section 2.1.2 page 10).

Les intergiciels de communication offrent également des capacités de réaliser ce recouvrement au niveau de la programmation, comme par exemple les opérations asynchrones, l'intégration de la multiprogrammation légère et de la communication.

Afin d'évaluer de potentiel de recouvrement des communications par du calcul dans INUKTITUT, nous utilisons deux tests de performance à deux niveaux différents. Le premier test est un *microbenchmark* qui s'appuie sur le test de latence (*« ping-pong »*, *cf.* section 5.2.1). Le deuxième test est de niveau applicatif et on fait une simulation qui se base sur une itération parallèle de Jacobi [BT89].

5.5.1 Le *microbenchmark*

Analyse du problème

Supposons que nous ayons un calcul qui s'exécute en t_{cal} unités de temps sur un processus et notons par t_{comm} le temps de communication nécessaire de ce processus. Alors, le temps d'exécution total t dans le cas de non recouvrement calcul/ communication est donné par la formule :

$$t = t_{comm} + t_{cal} \tag{5.11}$$

Pendant la durée de communication t_{comm}, il y a des périodes où le processeur est réservé aux opérations liées à la communication. Il ne peut donc pas être utilisé pour le calcul. Supposons que ces périodes sont représentées par le temps t_{nr} et que les autres périodes qui peuvent être recouvertes par du calcul sont notées par t_r, alors $t_{nr} + t_r = t_{comm}$ l'équation (5.11) devient :

$$t = t_{nr} + t_r + t_{cal} \tag{5.12}$$

C'est-à-dire la durée t_{comm} se compose de deux composants de temps t_{nr} non recouvrable et t_r recouvrable. Le temps d'exécution total dans le cas d'un recouvrement idéal t est donné par

$$t = t_{nr} + \max(t_{cal}, t_r) = \begin{cases} t_{nr} + t_{cal}, & t_{cal} > t_r \quad \text{(a)} \\ t_{nr} + t_r = t_{comm}, & t_{cal} \le t_r \quad \text{(b)} \end{cases} \tag{5.13}$$

En considérant l'équation (5.13), nous constatons que la croissance de temps t_{nr} réduit l'effet du recouvrement. A l'inverse, l'accroissement du temps t_r augmente le potentiel de recouvrement du calcul par les communications. Ainsi, nous définissons le taux de recouvrement R du temps de communication comme suit :

$$R = \frac{t_r}{t_{comm}} \tag{5.14}$$

La valeur de R varie dans l'intervalle [0, 1]. Nous avons de cas :

(i) $R = 0 <=> t_r = 0$ et $t_{nr} = t_{comm}$,
(ii) $R = 1 <=> t_r = t_{comm}$ et $t_{nr} = 0$.

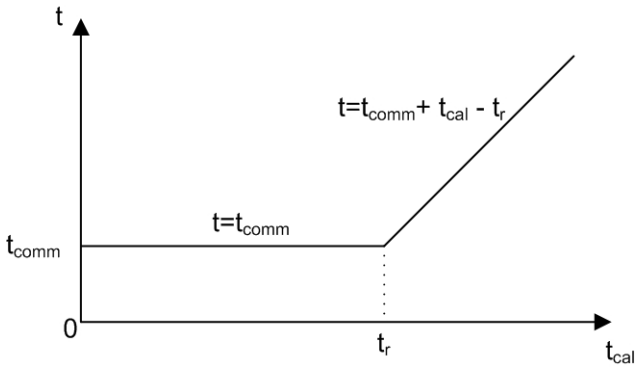
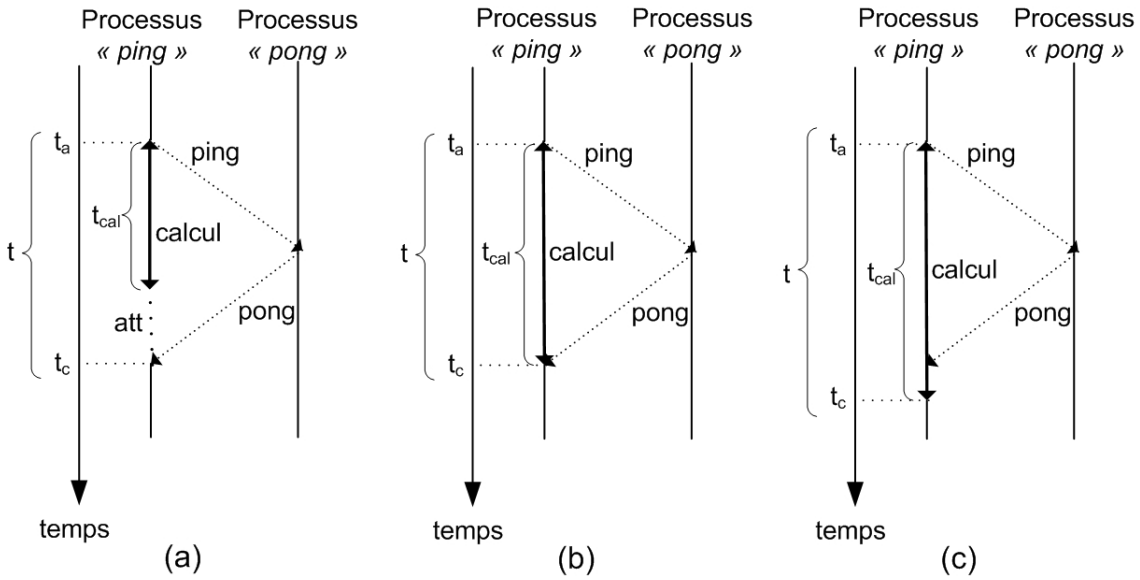


Figure 5.21 *Interprétation de relation entre t_r, t_{comm} et t_{cal}*

On peut calculer le taux *R* en mesurant directement les temps t_r et t_{comm} grâce à l'équation (5.13) (page 117). En plus, en observant la Figure 5.21, nous remarquons que le temps d'exécution *t* est invariant (égale à t_{comm}) dans l'équation (5.13b) où la valeur t_{cal} est variée de 0 à t_r. Si cette valeur est supérieure à t_r, le temps *t* est monotone croissant (5.13a). La valeur t_r est donc égale à la plus grande valeur de t_{cal} qui n'augmente pas la valeur de *t*. Ensuite, nous étudions une méthode basé sur le test *« ping-pong »* pour déterminer t_r. comme suit.

Détermination de t_r

Pour s'affranchir des problèmes dûs à l'absence d'horloge globale, nous reprenons le test de « *ping-pong* » (*c.f* section 5.2.1, page 95) pour ce *microbenchmark* afin d'évaluer le taux de recouvrement *R*. D'abord, le processus *« ping » (émetteur)* commence par l'envoi d'un message et puis exécute un calcul. Ensuite, il attend la réponse du processus *« pong » (récepteur)* pour se terminer (Figure 5.22). Quand le temps de calcul t_{cal} est inférieur ou égal à t_r, comme dans l'équation 5.13b, le temps *t* d'un *RTT* (= t_c - t_a) est constant (Figure 5.22a et 5.22b). Et puis le temps *t* est monotone croissant (Figure 5.22c) quand temps $t_{cal} > t_r$ comme dans l'équation 5.13a. Le temps du recouvrement t_r est donc le temps t_{cal} de calcul maximum estimé qui n'augmente pas *t*. Cette méthode est également abordée dans [LCW+03].



att : attente de la réponse du processus *« pong »*

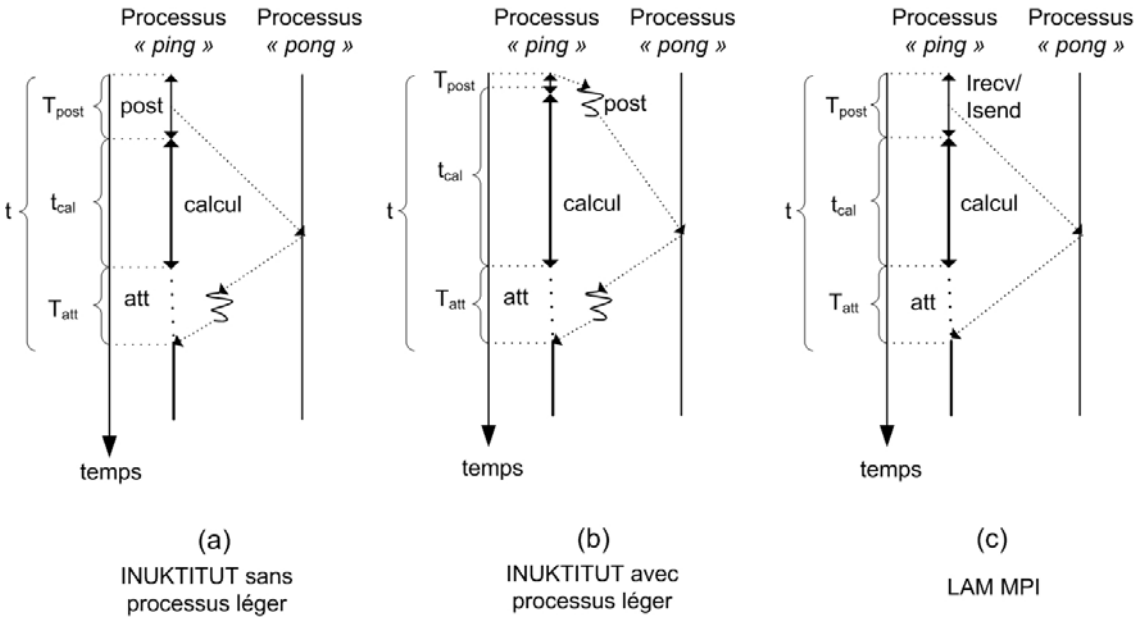
Figure 5.22 *Microbenchmark pour évaluer le potentiel de recouvrement*

Implantation

Nous avons trois implantations différentes de cette méthode : deux pour INUKTITUT et un pour *LAM MPI* 6.5.9_mode c2c (Figure 5.23).

La première implantation pour INUKTITUT, baptisé INUKTITUT sans processus léger, est un enchaînement de communication non bloquante et de calcul. Un processus « ping » réalise d'abord un appel de service à distance à envoyer un message et ensuite exécute le calcul. À la fin ce calcul, il attend la réponse du processus *« pong »* (Figure 5.23a). Dans la deuxième implantation nommé INUKTITUT avec processus léger, l'appel de service est mis dans un processus léger de communication. D'abord, le processus *« ping »* envoie un signal au processus léger pour activer l'envoi d'un message et puis exécute le calcul. Ensuite, il attend la réponse du processus *« pong »* (Figure 5.23b). En bref, les implantations d'INUKTITUT donnent deux exécutions différentes : soit en parallèle (le calcul est exécuté concurremment avec l'appel de service à distance), soit le calcul s'exécute après l'appel non bloquant du service à distance. En ce qui concerne l'implantation de *LAM*, nous utilisons dans processus *« ping »* les

fonctions non bloquantes *Isend/Irecv* pour commencer les opérations de réception et d'envoi, et puis ce processus exécute de calcul. Ensuite, il attend la terminaison de ces opérations d'envoi et de réception (Figure 5.23c).



att : attendre de la réponse du processus *« pong »*

Figure 5.23 *Les implantations différentes de microbenchmark*

Maintenant, nous mesurons les temps d'envoi T_{post}, de calcul t_{cal}, d'attente T_{att} (étant le temps écoulé entre la fin d'un calcul et l'arrivé de la réponse au fil de calcul) et d'exécution t pour toutes les implantations. Nous avons :

$$t = T_{post} + T_{att} + t_{cal} \tag{5.15}$$

De plus, nous utilisons le temps moyen de 5000 itérations de cette expérience au lieu d'une seule pour les mêmes raisons que celle abordées précédemment (*cf.* section 5.2.1, page 95).

Nous représentons le calcul t_{cal} par un certain nombre i de calcul élémentaire : $t_{cal} = i$ x t_{elem} où t_{elem} est fixé. Cela afin de faire varier le temps de calcul pour les valeurs de i=0 à une valeur noté *max*.

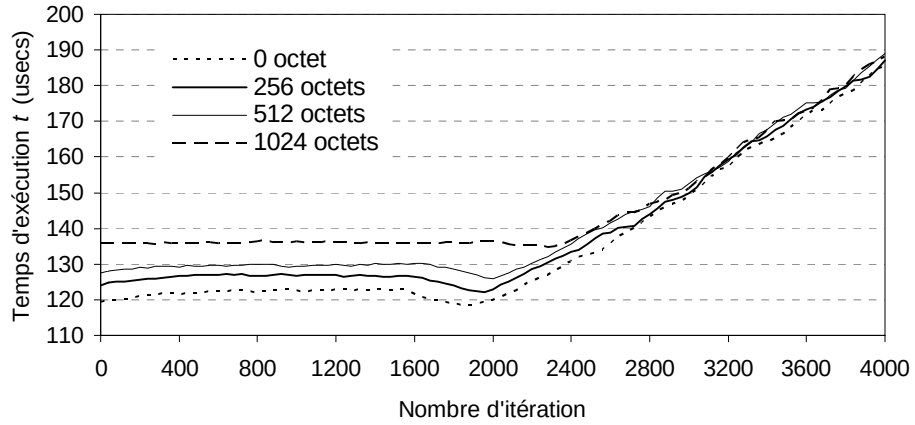
Résultats et analyse

Les expériences de ce test sont réalisées pour des messages de taille de 0 octets à 16384 octets. Nous utilisons SOCKNET comme le réseau de communication. La plate-forme d'exécution est se compose de deux machines d'I-Cluster2 (*c.f* Annexe A). Chaque processus (*« ping »* et *« pong »*) est placé sur une machine différente. La valeur *max* d'itérations d'un calcul choisie est égale 4000 pour les tailles de message ≤ 1K octets et égale 10000 pour ceux > 1K octets.

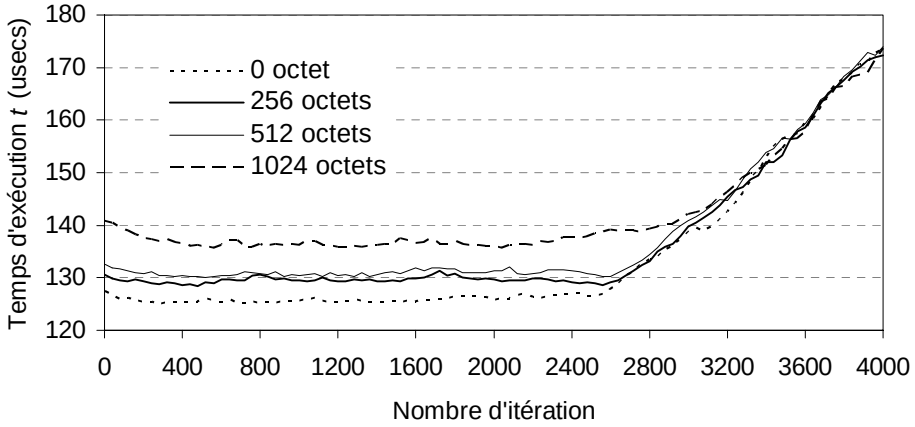
Les Figure 5.24 et Figure 5.25 présentent la variation de temps d'exécution *t* selon le nombre d'itérations du calcul pour ces trois implantations.

Grâce à ces figures, nous trouvons que :

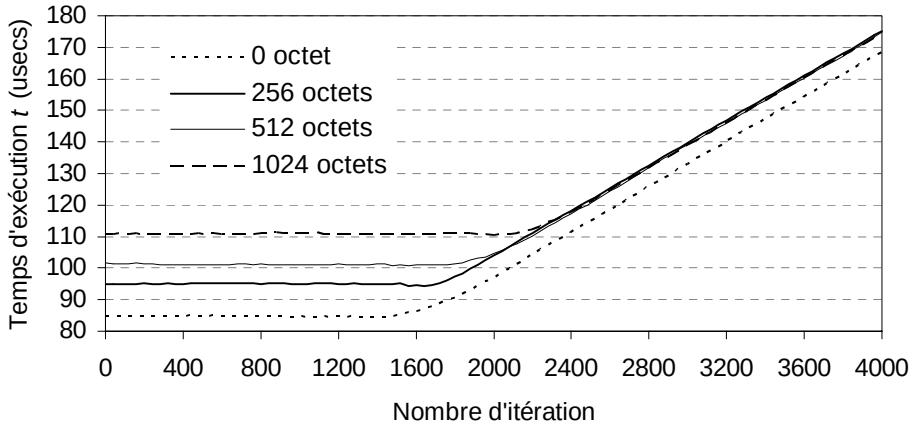
- toutes les courbes sont identiques au niveau de la forme. D'abord, la croissance du volume de calcul n'augmente pas le temps d'exécution *t*. Ceci est le premier cas (l'équation 5.13b, page 117) où le calcul est recouvert par la communication. Le temps de calcul est inférieur au temps de communication. Ensuite, la pente à chaque courbe est monotone croissant, c'est-à-dire que nous passerons déjà au deuxième cas (l'équation 5.13a, page 117). En ce cas, le temps d'exécution *t* est en fonction du volume de calcul. Le point de flexion des courbes délimite à gauche le temps de calcul qui peut être recouvert par la communication. Ce point est nommé *point de pivot*. Alors, le temps de calcul t_{cal} correspondant à ce point est égal au temps t_r.



(a) *INUKTITUT sans processus léger*

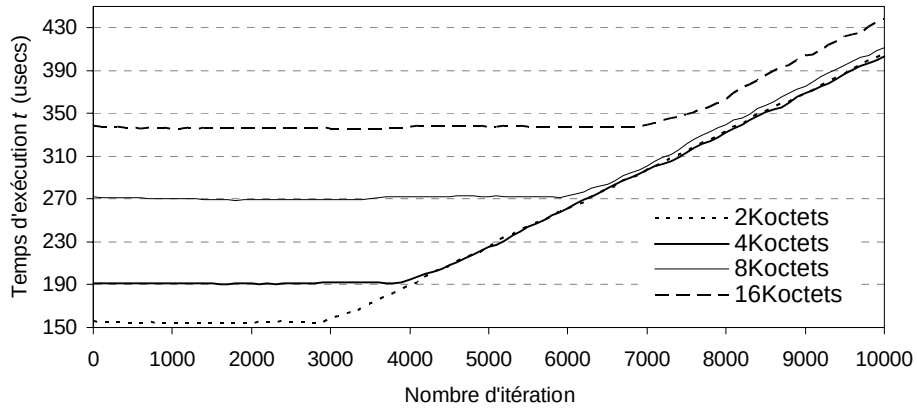


(b) *INUKTITUT avec processus léger*

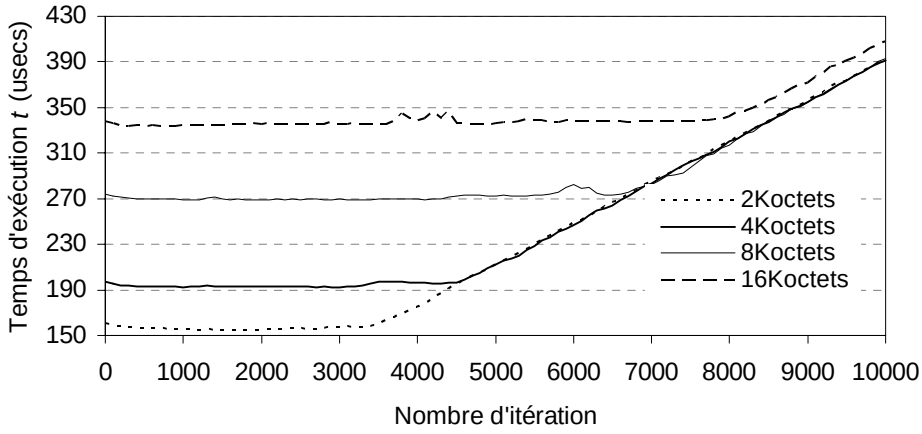


(c) *LAM MPI 6.5.9_c2c*

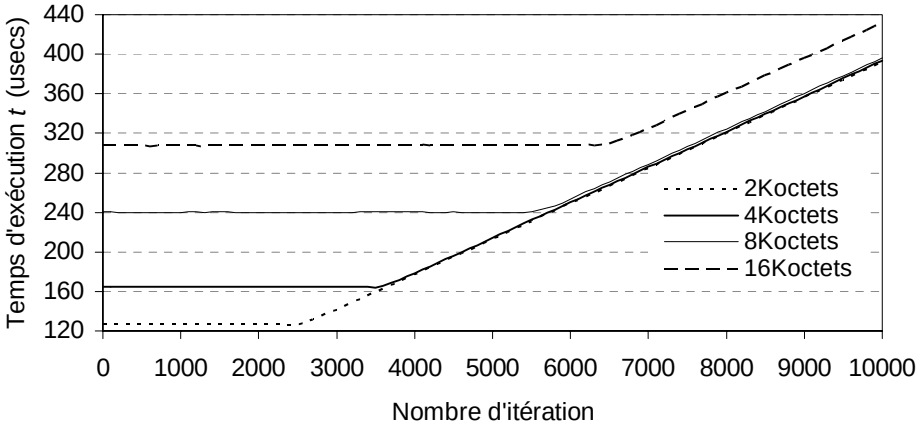
Figure 5.24 *Temps d'exécution selon l'augmentation du volume de calcul exécuté pour les tailles de message échangé de 0 à 1K octets*



(a) *INUKTITUT sans processus léger*

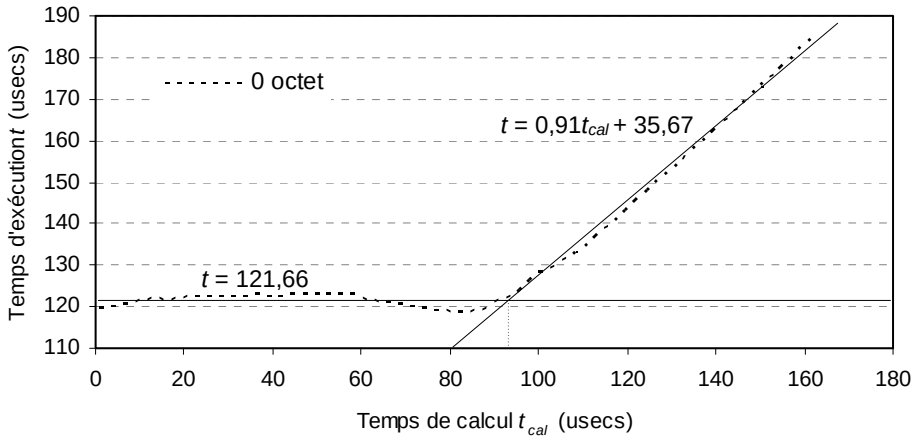


(b) *INUKTITUT avec processus léger*

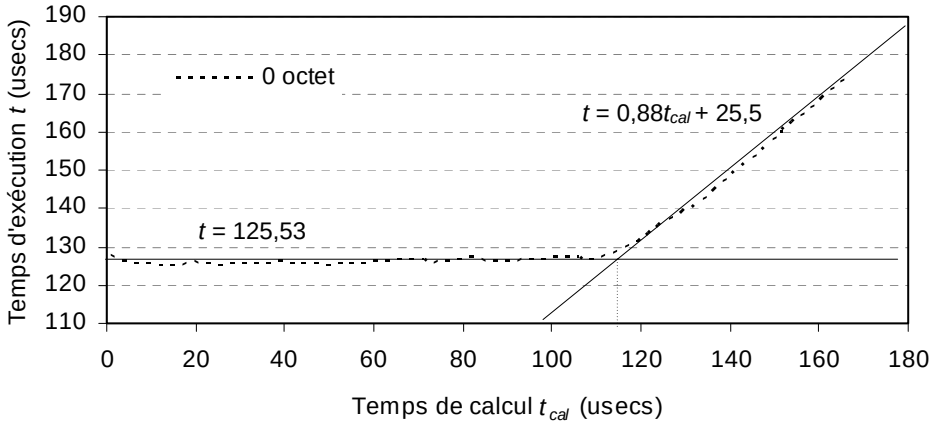


(c) *LAM MPI 6.5.9_c2c*

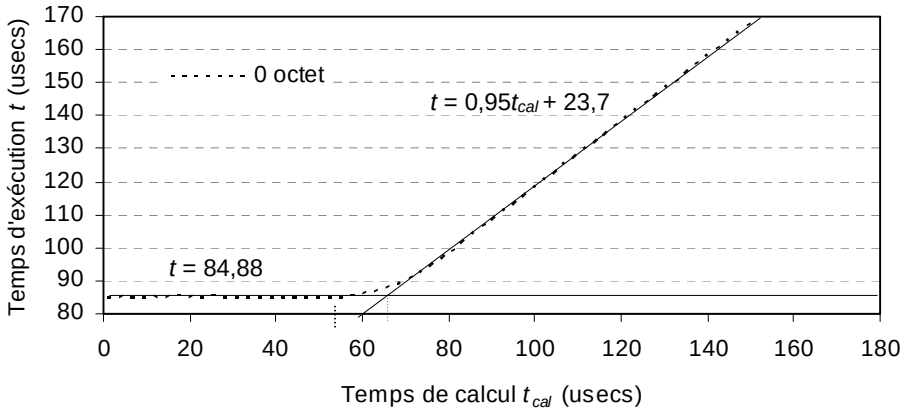
Figure 5.25 *Temps d'exécution selon l'augmentation du volume de calcul exécuté pour les tailles de message échangé de 2K à 16K octets*



(a) *INUKTITUT sans processus léger*



(b) *INUKTITUT avec processus léger*



(c) *LAM MPI 6.5.9_c2c*

Figure 5.26 *Détermination du temps de calcul exécuté au point pivot pour le message de taille nulle*

- Nous avons un décalage entre les courbes sur la partie gauche du *point pivot*. Ce phénomène est causé par la différence de temps d'envoi *t* pour les tailles différentes des messages échangés. Néanmoins, la différence entre ces courbes est petite sur la partie droite du *point pivot* dans le plupart des cas parce qu'on a le même temps de calcul et la différence de coût d'envoi est petite. Par contre, le coût d'envoi d'un message 16Koctets est plus grand que celui des autres parce que cette taille de message dépasse la limitation de MTU (Maximum Transfer Unit, = 9000 octets sur I-Cluster 2) et elle provoque un surcoût dû à la fragmentation du message échangé.

- L'augmentation de taille de message déplace le *point pivot* parce que le temps d'envoi *t* augmente et donc le temps de calcul recouvert croît aussi.

Afin de déterminer le temps t_{cal} au *point pivot*, nous nous appuyons sur l'équation (5.13) (page 117). En observant la Figure 5.21 (page 118), nous constatons que ce point pivot est l'intersection des courbes représentent deux fonctions t = *constant* (*cf.* l'équation 5.13b) est $t = \alpha t_{cal} + \beta$ (*cf.* l'équation 5.13a).

Nous avons donc estimé ces courbes et calculé leurs intersections pour déterminer le temps t_r dans nos expériences. Puis nous calculons le temps de calcul t_{cal} correspondant à ce point. Par exemple, nous étudions l'échange d'un message de taille nulle par implantations d'INUKTITUT et de *LAM* (Figure 5.26). Nous commençons par le cas d'INUKTITUT sans processus léger (Figure 5.26a), le temps t_{cal} au *point de pivot* est 94,53µs (t = 121,66 et $t = 0{,}91t_{cal} + 35{,}67$). Nous obtenons le taux de recouvrement selon l'équation (5.14) :

$$R = \frac{t_r}{t_{comm}} = \frac{t_{cal(pivot)}}{t_{(pivot)}} = \frac{94{,}53}{121{,}66} = 0{,}7769 = 77{,}69$$

Semblablement, le taux *R* de la deuxième implantation avec INUKTITUT et processus léger (Figure 5.26b) (t = 125,53 et $t = 0{,}88t_{cal} + 26{,}5 \Rightarrow t_{cal(pivot)} = 112{,}84$ et $t_{(pivot)} = 125{,}53$) est calculé :

$$R = \frac{t_r}{t_{comm}} = \frac{t_{cal(pivot)}}{t_{(pivot)}} = \frac{112{,}84}{125{,}53} = 0{,}8982 = 89{,}82\%$$

Enfin, le taux *R* pour l'implantation de *LAM MPI* (Figure 5.26c) (t = 84,88 et $t = 0{,}95t_{cal} + 23{,}7 \Rightarrow t_{cal(pivot)} = 64{,}1$, $t_{(pivot)} = 84{,}88$) est :

$$R = \frac{t_r}{t_{comm}} = \frac{t_{cal(pivot)}}{t_{(pivot)}} = \frac{64{,}1}{84{,}88} = 0{,}7552 = 75{,}52\%$$

Au *point pivot*, la somme des temps T_{post} et T_{att} est le temps de non recouvrement. Le taux *R* d'implantation d'INUKTITUT avec processus léger est meilleur par rapport aux autres parce qu'on réduit considérablement le coût T_{post} qui représente le coût d'intersection d'une requête dans une liste : *4,45*µs vs. *19,3 µs* (sans processus léger) et *9,74µs* (*LAM*).

De plus, nous comparons le taux de recouvrement *R* pour différentes tailles de messages échangés entre les différentes implantations dans la Figure 5.27. Cette figure montre que l'implantation avec INUKTITUT et processus léger donne le meilleur taux de recouvrement par rapport aux autres parce qu'on réduit fortement le temps de non recouvrement. Du fait que le fil d'exécution du calcul et de la communication sont exécutés en parallèle sur deux processeurs différents.

Nous trouvons en outre une croissance de taux *R* pour des messages de taille inférieure ou égale 8Koctets. L'augmentation du temps d'envoi t_{post} est plus lente que celle du temps d'envoi d'un message. Ce phénomène provoque alors la décroissance du potentiel de non recouvrement. Pour les messages de taille supérieure à 9000 (la

limitation de MTU), le coût d'envoi inclut un surcoût de fragmentation comme nous l'avons abordé ci-dessus. Nous n'avons donc plus cet avantage, et par conséquent le taux de recouvrement décroît.

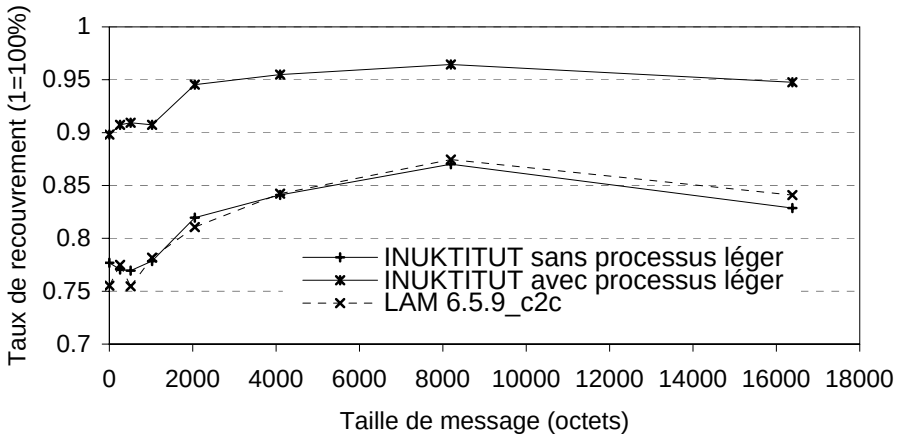


Figure 5.27 *Comparaison du taux de recouvrement entre les trois implantations*

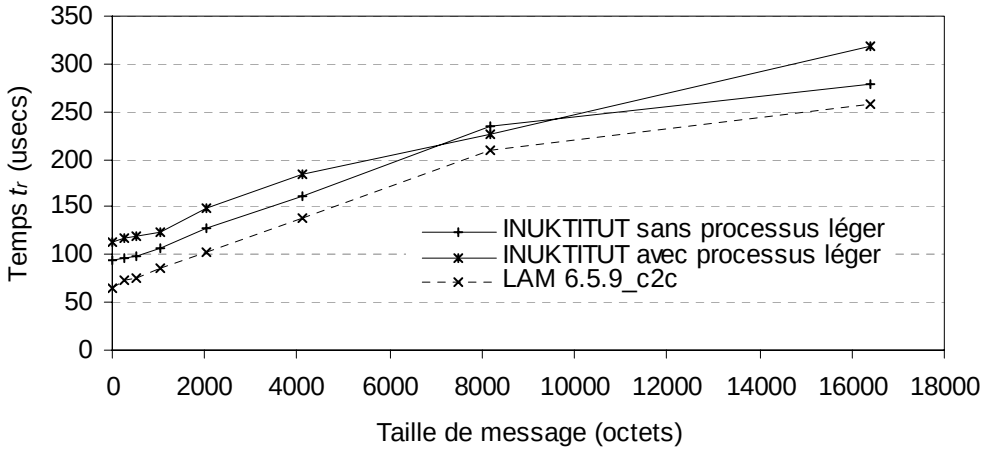


Figure 5.28 Comparaison du temps t_r entre les trois implantations

5.5.2 Évaluation de performance au niveau d'application

Dans cette partie, nous estimons la performance du recouvrement de calcul et communication par un test de niveau applicatif. Nous avons développé une simulation qui se base sur l'itération parallèle de Jacobi [BT89]. Nous évaluons par la suite la performance via la vitesse de calcul de cette simulation qui possède des possibilités de recouvrement.

Méthodologie

L'algorithme de cette application est du type décomposition de domaine. Il suppose la division d'un domaine de calcul (un plan, une matrice) en plusieurs sous domaines (en blocs, en lignes, en colonnes, en sous matrice). Nous réalisons un calcul moyen pour chaque point d'un sous domaine. Ce calcul se base sur les valeurs des points immédiatement à gauche, à droite, en haut et en bas [Les93] (Figure 5.29). Les points sur la frontière d'un sous domaine doivent être échanges avec les sous domaines voisins. Une implantation générale de cette application pour évaluer la performance en utilisant l'échange de message est donnée par l'algorithme 5.2

Algorithme 6.1 Mesure de temps pour l'algorithme de Jacobi

```
t0 <- GetTime()
While not (term) do
  E1. Envoyer les points frontières aux sous domaines voisins
  E2. Evaluer les points intérieurs (calcul1)
  E3. Recevoir les points frontières des sous domaines voisins
  E4. Evaluer les point frontières (calcul2)
Endo
t1 <- GetTime()
Temps d'exécution t <- (t1 – t0)
```

term est la condition de terminaison de la boucle. Elle est normalement évaluée par une condition de convergence. Nous pouvons profiter du recouvrement de calcul et communication par l'exécution en parallèle des étapes E1 et E2 [BBC04].

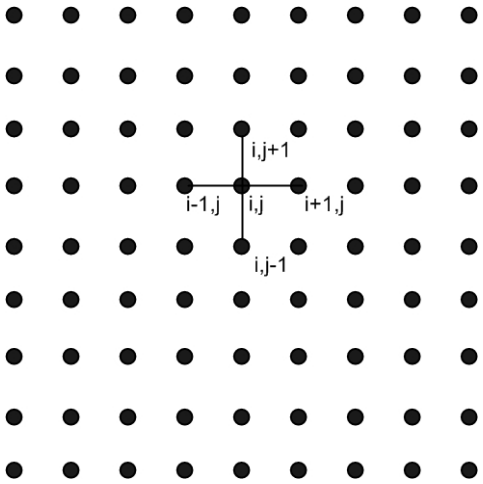


Figure 5.29

Implantation

Nous avons deux implantations de l'application de Jacobi : la décomposition en une dimension (1D) et la décomposition en deux dimensions (2D).

Dans l'implantation 1D, une matrice *M x N* est divisée par ligne. Chaque processeur a une sous matrice *m x N* de la matrice origine où *m = M/p* et *p* est le nombre de processeurs participants (Figure 5.30). Chaque sous matrice échange ses frontières avec deux voisins : la matrice en haut et la matrice en bas. La première matrice et la dernière n'ont qu'un voisin.

Nous avons trois versions différentes de cette implantation comme dans le *microbenchmark* : INUKTITUT sans processus léger, INUKTITUT avec processus léger et *LAM MPI* 6.5.9_c2c. Afin de profiter des possibilités de recouvrement calcul/communication, nous utilisons les fonctions non bloquantes *Isend/Irecv* pour envoyer et pour recevoir les points frontières dans le cas de *LAM MPI*. Une

implantation similaire est décrite dans [*mpich*]. La métrique de performance est Mflops [DS98]. Il est définit par

Mflops = ((Taille de problème x 4) / Temps d'exécution) / 1000000 (5.16)

4 est le nombre d'opérations d'arithmétiques qui est exécuté à chaque point de matrice.

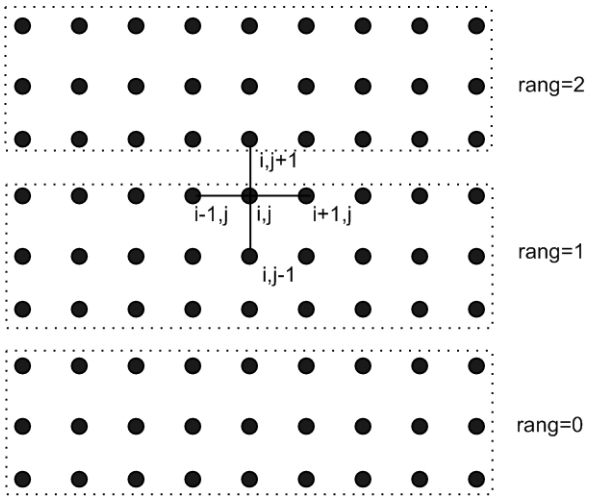


Figure 5.30 *Décomposition en 1D (rank est l'ordinaire de sous matrice)*

La deuxième implantation, quant à elle, est une extension de la première. Une matrice *M x N* est divisée en ligne et en colonne. Chaque processeur a une sous matrice *m x n* issu de la matrice origine, où *m = M/p*, *n = N/p*, *p* est le nombre de processeurs (Figure 5.31). Les calculs des points intérieurs et des points frontières sont similaires à ceux de la première implantation. L'échange des points frontières est de type torique [Les93]. C'est-à-dire une sous matrice échange toujours ses frontières avec ses quatre sous matrices voisines. L'avantage de cette implantation est que la taille de la frontière échangée par domaine est plus petite que dans la première implantation. On a pourtant globalement plus de communication par rapport à la première. Nous avons ainsi trois versions de cette implantation comme précédemment. Nous utilisons aussi l'équation 6.10 pour estimer la valeur Mflops.

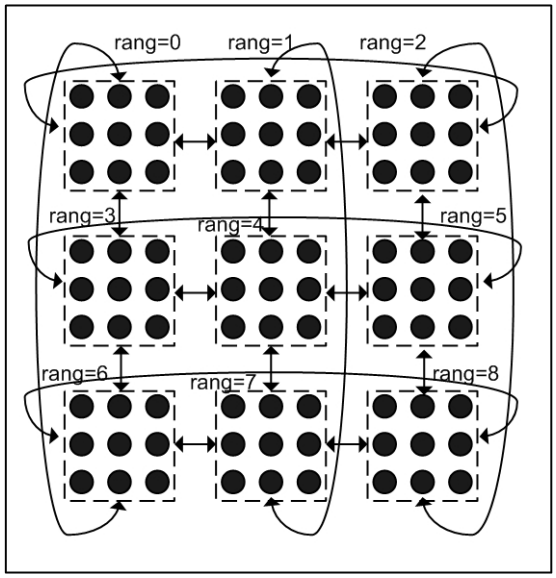


Figure 5.31 *Décomposition en 2D de type torique (rang est le numéro de la sous matrice)*

Résultats et analyse

Les expériences sont réalisées pour une taille de problème de 1024 x 1024, chaque élément est de type *« double »*. Nous utilisions SOCKNET comme le réseau de communication. La plate-forme d'exécution est le I-Cluster 2 (c.f Annexe A). Dans nos expériences, chaque sous matrice est placée sur une machine. Le nombre de machine participant varié de 2 à 16. Nous n'avons pas de test de convergence et nous supposons que l'algorithme s'arrête après 1000 itérations.

Le tableau 6.5 présente les résultats de ces deux implantations 1D et 2D de l'implantation. En observant ce tableau, nous constatons que :

Nombre de machine	LAM 6.5.9_c2c	Inuktitut_sans processus léger	Inuktitut_avec processus léger
2	209.14±0.07 Mflops	211.08±0.30 Mflops	211.12±0.21 Mflops
4	405.72±0.19 Mflops	405.86±0.34 Mflops	412.13±0.60 Mflops
8	782.75±2.42 Mflops	780.62±1.27 Mflops	808.16±2.52 Mflops
16	1604.29±1.55 Mflops	1637.13±2.46 Mflops	1688.96±5.95 Mflops

(a) L'implantation 1D

Nombre de machine	LAM 6.5.9_c2c	Inuktitut_sans processus léger	Inuktitut_avec processus léger
4	395.95±0.16 Mflops	404.53±0.43 Mflops	408.05±0.18 Mflops
9	891.48±1.67 Mflops	903.22±1.55 Mflops	931.11±1.74 Mflops
16	1640.71±1.35 Mflops	1678.06±8.19 Mflops	1715.02±7.10 Mflops

(b) L'implantation 2D

Tableau 5.5 Comparaison de Mflops selon de nombre de machines participantes

- La version d'INUKTITUT avec processus léger donne les meilleurs résultats dans tous les cas, gain de 1% à 5% par rapport aux autres versions. L'utilisation des processus légers permet de mieux profiter du recouvrement calcul/communication comme nous l'avons analysé pour le *microbenchmark*.

- L'implantation de 2D donne la meilleure performance par rapport à celle de 1D pour le cas d'utilisation de plusieurs machine (16 machines vs. 4 machines)

- Mflops est en fonction de nombre de machines participantes. C'est quand même logique parce qu'on a plus de processeurs qui calculent en parallèle avec la moindre taille de sous matrice.

5.6 Conclusions

Les performances des communications bipoints d'INUKTITUT sont analysés dans ce chapitre. Les comparaisons entre INUKTITUT et *LAM/MPI* sont présentées. Les résultats obtenus nous permettent de dire :

- Peu de différence existe entre l'utilisation des réseaux SOCKNET et CORNET de INUKTITUT. Bien que la latence de communication du réseau CORNET est plus élevée que celle du réseau SOCKNET.

- Bien que le surcoût d'envoi d'INUKTITUT pour les petites tailles de messages soit plus important que celui de *LAM*/*MPI*, il n'a pas de problème de changement de protocole qui cause une perte des performances. L'échange de message actif d'INUKTITUT donne de bonnes performances dans la communication point à point comme l'ont montré l'étude des indicateurs de performance et des comparaisons avec *LAM*/*MPI*.

- Le modèle d'un processus léger pour une connexion donne la meilleure performance par rapport au modèle d'un processus léger pour toutes les connexions dans le cas de grappes de monoprocesseur comme I-Cluster1 pour les messages de grande taille.

- Nous pouvons exploiter en pratique la capacité du recouvrement calcul/communications. La communication asynchrone par message actif permet de profiter bien de ce recouvrement. Le fil de calcul peut éviter la charge de réception du message. De plus, l'utilisation de processus léger pour envoyer des messages permet aussi d'augmenter la performance parce qu'on réduit le surcoût de génération de la communication dans fil du calcul.

En conclusion, la communication par échange de message actif d'INUKTITUT donne des résultats satisfaisants. L'application parallèle qui utilise INUKTITUT comme le noyau exécutif de communication peut bien profiter du recouvrement du calcul/communication qui est considéré comme un avantage important pour les applications.

6

Évaluation des performances de la communication collective

Dans ce chapitre nous allons évaluer les performances des opérations de communication collective qui sont implantées dans la bibliothèque ICS d'INUKTITUT. Les expériences et ses analyses se basent principalement sur l'implantation de SOCKNET (*cf.* section 3.2.2, page 48). Nous présenterons d'abord les méthodes utilisées pour mesurer les métriques de performance pour les opérations de diffusion totale et du service de barrière. Nous analyserons ensuite les résultats de chaque algorithme de communication collective (*cf.* section 2.3.3, page 24) utilisé dans le cas de la diffusion totale. Nous faisons en outre des comparaisons entre les algorithmes. Nous étudions la performance du service de barrière dans la dernière section. Une conclusion termine ce chapitre.

6.1 Méthodologie d'évaluation

6.1.1 Les méthodes utilisées pour prendre le temps des expériences dans la diffusion totale

La mesure de temps d'une opération de communication collective (OCC) est plus difficile que celle d'une communication bipoint à cause du nombre de processus participants. Le métrique de performance d'une OCC est le temps d'achèvement (*completion time*, en anglais). Le temps d'achèvement est normalement le temps écoulé entre le moment où une OCC commence et celui où cette même OCC se termine sur tous les processus participant. Par exemple, dans le cas de MPI, le temps d'achèvement est défini par le temps écoulé entre l'instant t_0 où tous les processus appellent simultanément un OCC et l'instant où ils mettent fin à cet appel [NN97]. Pourtant cette définition n'est utilisée que dans le cas de communication synchrone. Or dans INUKTITUT, la communication est asynchrone. Nous prenons par exemple le cas de la diffusion totale. Dans ce cas l'appel de service de diffusion est initié sur le processus racine (*cf.* 3.5.3, page 67). Le temps d'achèvement dans ce cas est le temps écoulé entre cet appel et le moment où ce service est exécuté par tous les autres processus. La détermination de ce moment est difficile à cause de l'absence d'une horloge globale. Il nous faut donc une solution qui consiste à mesurer le temps d'achèvement via la réponse des autres processus au processus racine. Une méthode qui permet de mesurer le temps de réponse en s'appuyant sur le flot de communication est déjà proposée dans [NN97]. Elle reste néanmoins compliquée pour l'algorithme de communication tel que l'arbre α. Nous pouvons donc assurer aussi la réponse des processus de réception en utilisant la barrière de synchronisation.

Nous présentons dans la suite deux méthodes de mesure de temps dans la diffusion. La première utilise une barrière de synchronisation après chaque diffusion, c'est-à-dire qu'elle prend en compte la réponse des récepteurs. La présence d'une barrière de synchronisation permet aussi de répondre à l'exigence d'exécution simultanée cette opération par tous les processus. Nous utiliserons cette méthode dans la plupart des

évaluations. La deuxième méthode, quant à elle, n'utilise pas de barrière à chaque opération de communication collective. Une méthode similaire pour MPI est proposée dans [NN97] [HWW97]. Bien que cette méthode cause un effet de *« pipeline »*, elle provoque aussi une saturation totale du réseau de communication. Nous pouvons l'utiliser pour évaluer le débit de communication ainsi que la capacité de segmentation des messages.

Avec barrière de synchronisation

Cette méthode permet d'évaluer le temps de diffusion totale en utilisant une barrière de synchronisation après chaque opération de communication. Dans cette méthode, on exécute *i* itérations de la diffusion d'un message de taille *n* depuis un processus racine (processus 0, par exemple) vers d'autres processus. Une barrière de synchronisation après chaque réception permet de garantir l'arrivée des messages. Nous mesurons le temps *t* (Figure 6.1) comme suite:

$$t = T_2 - T_1$$

Nous évaluons alors le temps moyen t_{averb} :

$$t_{averb} = \frac{t}{i} \tag{6.1}$$

Et nous utilisons t_{averb} comme le temps d'exécution de cette opération.

Algorithme 6.1 Evaluation du temps d'exécution en utilisant la barrière

E1. Exécution d'une barrière de synchronisation
E2. Prise du temps T_1 à processus *0*
E3. Exécution de *i* itérations
- L'opération de diffusion est réalisée
- Exécution d'une barrière de synchronisation (après réception locale sur les récepteurs)

E4. Prise du temps T_2 à processus *0*
E5. Le processus *0* réalise la moyenne t_{averb} de ces *i* itérations et affiche le résultat.

Dans cet algorithme le temps de synchronisation par la barrière n'est pas retranché au temps de l'opération une estimation de ce temps est donné dans la section 6.3. Le débit δ_{ab} est donc évalué par l'équation suivante :

$$\delta_{ab} = \frac{n}{t_{averb}} \tag{6.2}$$

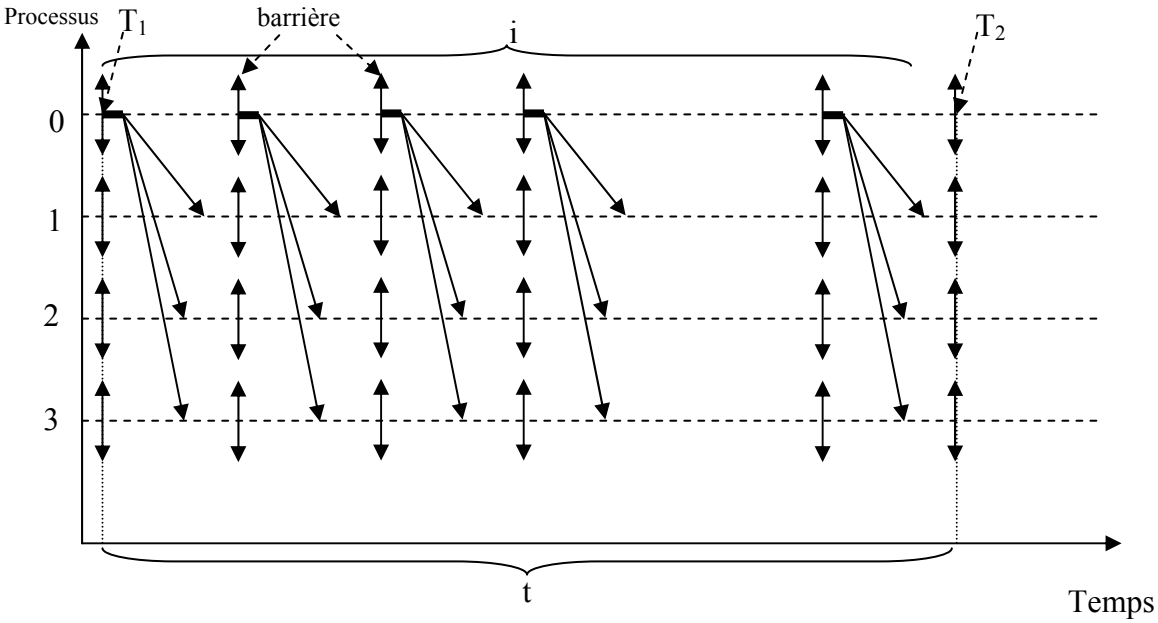


Figure 6.1 *La méthode de mesure de temps en utilisant une barrière de synchronisation*

Sans barrière de synchronisation

Cette méthode permet d'évaluer le temps de diffusion sans utiliser la barrière de synchronisation après chaque opération. On exécute consécutivement *i* itérations de diffusion d'un message de taille *n* de l'initiateur (processus 0, par exemple) aux autres processus participants dans cette opération. Nous prenons le temps T_1 quand l'initiateur exécute sa première diffusion. Ensuite, dès que chaque processus termine sa réception de *i* diffusions, il envoie un signal à l'initiateur. Si l'initiateur reçoit tous les signaux des autres processus, il prend le temps T_2. Nous mesurons donc le temps *t* (Figure 6.2):

$$t = T_2 - T_1$$

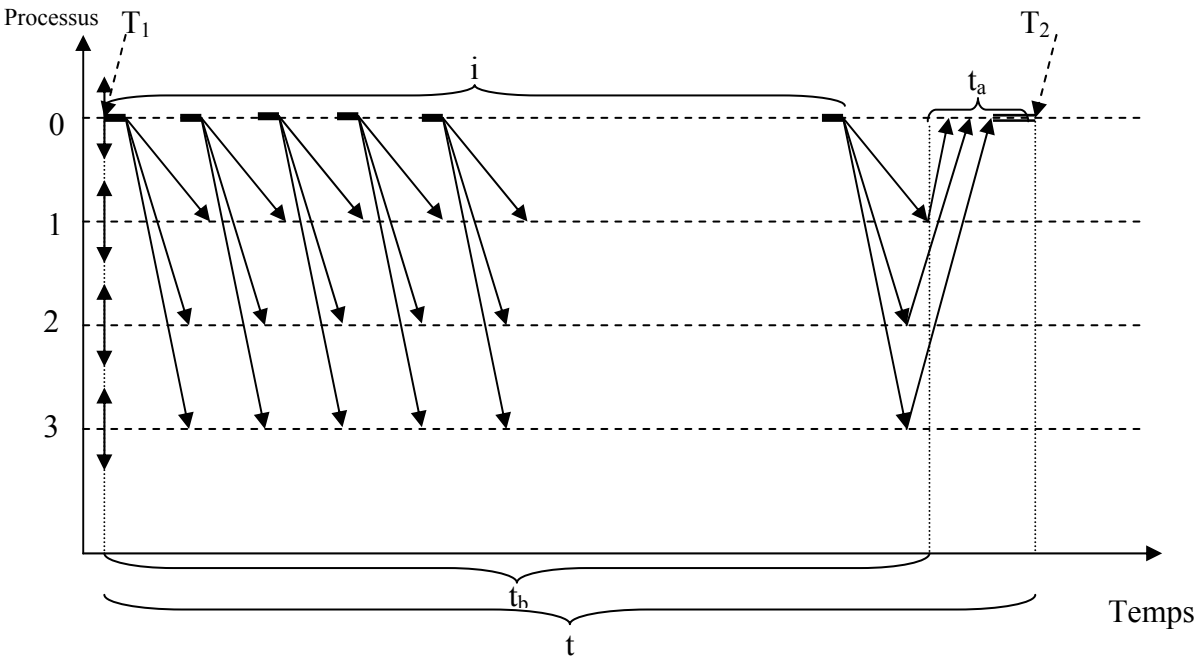


Figure 6.2 *La méthode de mesure de temps sans barrière de synchronisation*

Dans cette méthode, les données diffusées aux processus peuvent se chevaucher entre itérations (effet de pipeline). C'est-à-dire que le processus *j* peut recevoir les données de l'exécution *m* en même temps que le processus *k (k≠j)* reçoit les données de l'exécution *m+1*. Si l'itération *i* est grand, le temps $t >> t_a$. Nous pouvons utiliser *t* comme mesure du temps de diffusion d'un message de taille *n*. Cette méthode provoque une saturation des données transférées sur le réseau. Nous pouvons l'utiliser pour déterminer le débit δ_{sb} de communication par cette opération. Ce débit est évalué par l'équation suivante :

$$\delta_{sb} = \frac{n \times i}{t} \qquad (6.3)$$

Algorithme 6.2 Evaluation du temps d'exécution sans barrière

E1. Exécution d'une barrière de synchronisation

E2. Prise du temps T_1 à processus 0

E3. A processus 0 :

- exécution de i itérations de l'opération de diffusion est réalisée
- attendre de la réponse des autres processus

Pour les autres processus:

- compte des messages arrivés
- s'il a reçu i messages, il envoie une réponse au processus 0

E4. Prise du temps T_2 sur le processus 0

E5. Le processus 0 calcule la valeur $t_{aver} = (T_2 - T_1) / i$ et δ_{sb} grâce à l'équation (6.3)

6.1.2 L'algorithme d'évaluation du service de barrière

La bibliothèque d'ICS possèdent trois implémentations du *service de barrière* : *« α_barrier »*, *« flat_barrier »* et *« default_barrier »*. Les deux premières implantations utilisent les algorithmes de communication collective correspondants : l'arbre plat et l'arbre α. L'implantation de *« defaultbarrier »* se base sur l'algorithme arbre α où la valeur α change en fonction du nombre de processus. On évalue donc les deux premières implémentations par l'algorithme 6.3. Une méthode d'évaluation similaire pour MPI est proposée par [HWW97].

Algorithme 6.3 Evaluation du temps d'exécution du *service de barrière*

E1. Prise du temps t_1

E2. Exécution i itérations

- exécution d'une barrière de type *« service de barrière »*

E3. Prise du temps t_2

E4. Le processus racine réalise la moyenne de ces i itérations et affiche le résultat :

$$t_{serbarr} = \frac{t_2 - t_1}{i} \qquad (6.4)$$

6.1.3 Les paramètres utilisés pour les expériences et pour analyser les performances

Les expériences de ces méthodes sont réalisées pour {4, 8, 16, 32, 64} machines sur I-Cluster1 (*cf.* Annexe A). Chaque processus de nos algorithmes d'évaluation ci-dessus est placé sur une machine différente. De plus, elles sont aussi réalisées pour les différents algorithmes de communication : l'arbre plat, l'arbre α (avec la valeur α est égale 0.5, 0.4 et 0.3) et l'arbre chaîne. Les tailles de message pour les expériences varient de 4 octets à 8M octets.

Le nombre d'itérations *i* dans chaque expérience est choisit selon la taille des messages et le nombre de processus utilisés. Nous estimons que l'opération de communication collective est stable à cette valeur. La valeur du temps *t* et les tailles des messages utilisées dans les graphes sont en échelle logarithmique.

La barrière qui est utilisée dans les méthodes d'évaluation de la diffusion totale est la barrière par défaut d'INUKTITUT.

6.2 Etude expérimentale de la diffusion totale

6.2.1 L'algorithme de l'arbre plat

Nous étudions d'abord la performance de l'algorithme de l'arbre plat implanté dans ICS par la méthode de prise du temps avec la barrière de synchronisation. La Figure 6.3 présente le comportement de cet algorithme en fonction de la taille des messages sur 4 et 8 processus. En observant cette figure, nous constatons que:

- Les courbes sont identiques. Le décalage entre elles est quasiment stable pour toutes les tailles de message. La différence du nombre de processus participants explique l'origine de ce décalage.

- La différence du temps d'exécution est faible pour les messages de petite taille (inférieure à 1 K octets). La latence des communications prédomine le coût total.
- Les courbes pour les messages de taille à partir de 1 K octets se présentent sous la forme d'une fonction linéaire : le temps d'exécution $t = \alpha L + \beta$ (L est la taille du message envoyé).
- Il y a une différence de pente entre les petites tailles et les grandes tailles de message. Les messages de taille supérieure au MTU (*Maximum Transfer Unit*, égale 1500 octets pour I-Cluster1) [Ste97] du protocole Ethernet provoque un surcoût de fragmentation.

A cause du peu de différence entre le cas de 4 processus et de 8 processus, nous utiliserons soit le premier cas soit le deuxième pour afficher les comparaisons dans la suite de ce chapitre.

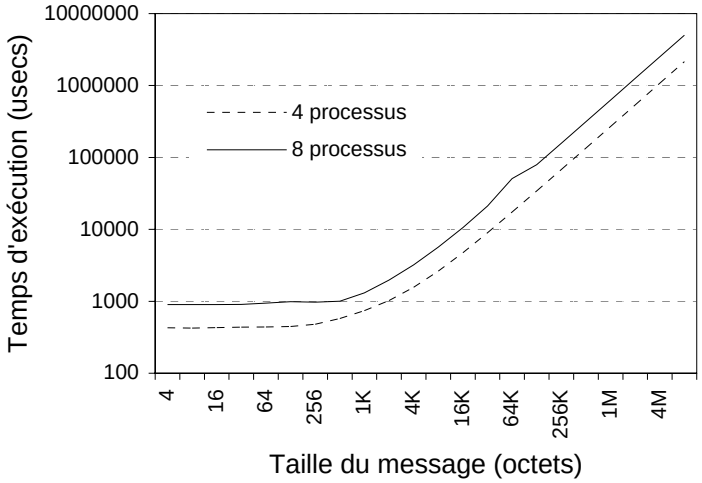


Figure 6.3 *Allure de l'arbre plat pour 4 et 8 processus*

La Figure 6.4 présente ensuite le temps d'exécution de cet algorithme en fonction de la taille du message sur 16, 32 et 64 processus. Nous remarquons que les courbes ont la même forme. Les décalages entre des courbes sont presque identiques à chaque point de mesure. On se rend compte que le temps d'exécution par cet algorithme pour une taille

de message est une fonction linéaire en fonction du nombre de processus, comme précédemment.

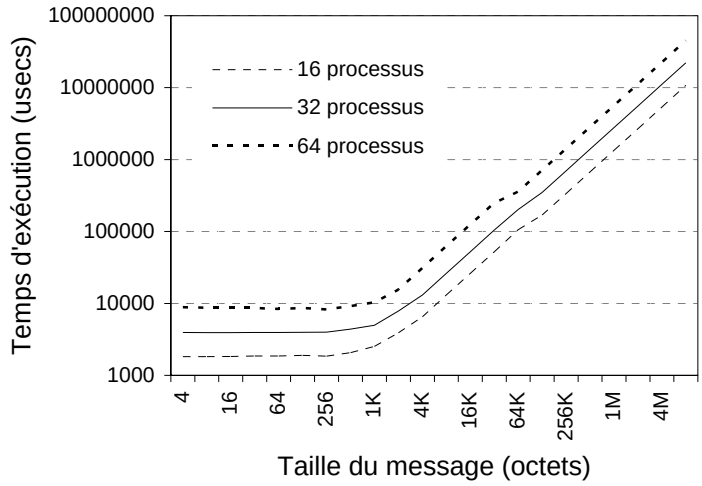


Figure 6.4 *Allure de l'arbre plat pour 16, 32 et 64 processus*

La figure 6.5 présente le temps, pour quelques tailles de message, en fonction du nombre de processus. Les courbes générées correspondent à la forme d'une fonction linéaire, soit $t = \omega P + \varphi$ (P est le nombre de processus). Ce qui corespond au modèle de coût de la section 2.3.3 (page 26).

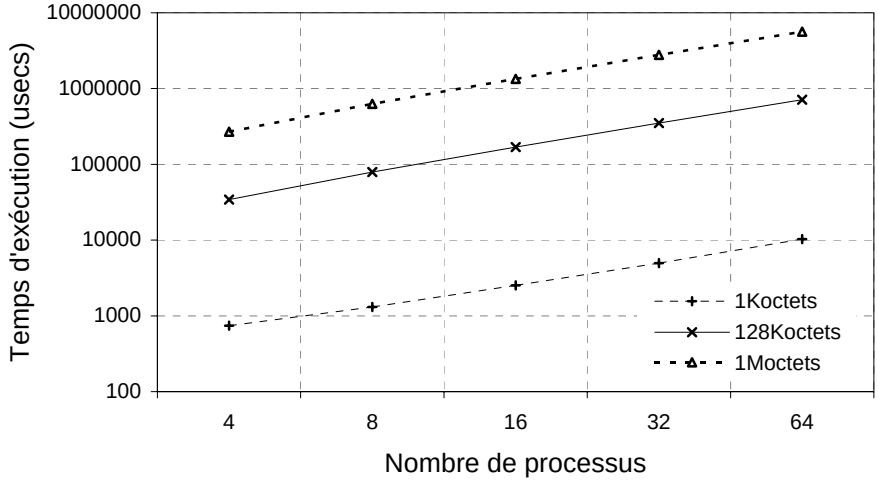


Figure 6.5 *Allure de l'arbre plat pour des messages de taille 1Ko, 128Ko et 1Mo en fonction du nombre de processus*

6.2.2 L'algorithme de l'arbre α

Comparaison des valeurs de α

Nous étudions dans cette partie le comportement de l'algorithme de l'arbre α. Nous utilisons la méthode de prise du temps avec la barrière de synchronisation comme dans l'algorithme de l'arbre plat.

La Figure 6.6 présente l'influence de la valeur de α sur l'envoi de messages de différentes tailles sur 8 processus. Nous constatons que peu de différence existe entre les trois valeurs de α pour des messages de petites tailles (inférieures ou égales 1K octets). Dans ce cas, l'α offrant le meilleur résultat est 0.3. Nous avons un gain de 8% à 14% par rapport à α égal 0.5. Puis entre 1K octets et 4K octets, le meilleur α est 0.4. Enfin, à partir de 8K octets le meilleur temps d'exécution est obtenu pour α égal à 0.5.

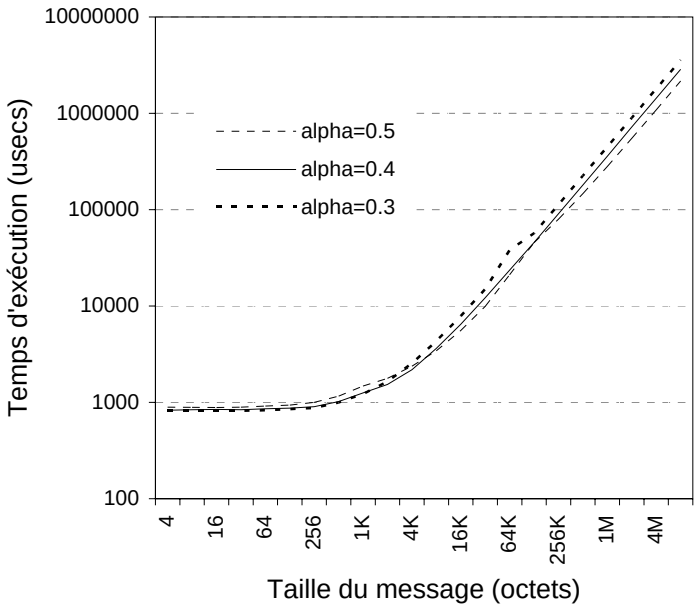


Figure 6.6 *Comparaison des valeurs de α sur 8 processus*

La Figure 6.7 présente la comparaison de l'algorithme de l'arbre α sur 16 processus. Nous avons les mêmes remarques sur la meilleure valeur de α qu'avec 8 processus. La

Figure 6.8 présente ensuite les comparaisons des valeurs de α sur 64 processus. Via étude de ces courbes, nous constatons que :

- Peu de différence entre trois valeurs de α pour les messages envoyés de taille inférieure à 16K octets.
- Au-delà de 16K octets le meilleur temps de communication est obtenu pour α égal à 0.5.

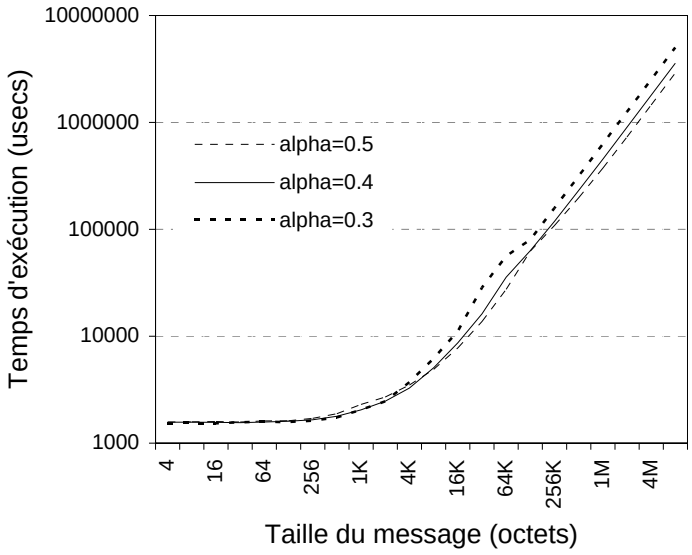


Figure 6.7 *Comparaison des valeurs de α sur 16 processus*

Nous avons ici une conclusion partielle : Dans le cas où le nombre de processus est petit (inférieur ou égal à 8), le meilleur α est 0.3 pour les petites tailles de message. La valeur α égale à 0.5 donne les meilleurs résultats pour les grandes tailles de message. Dans le cas du nombre de processus supérieur à 8, la valeur α choisit est 0.5 pour obtenir le meilleur temps de communication.

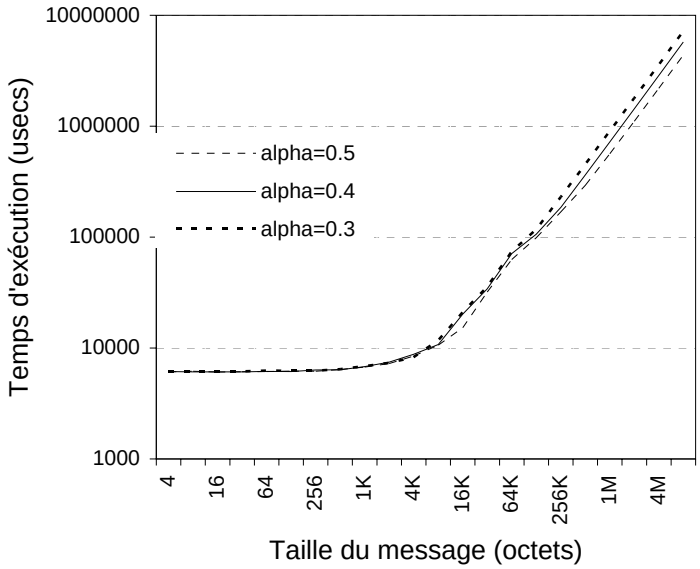


Figure 6.8 *Comparaison des valeurs de α sur 64 processus*

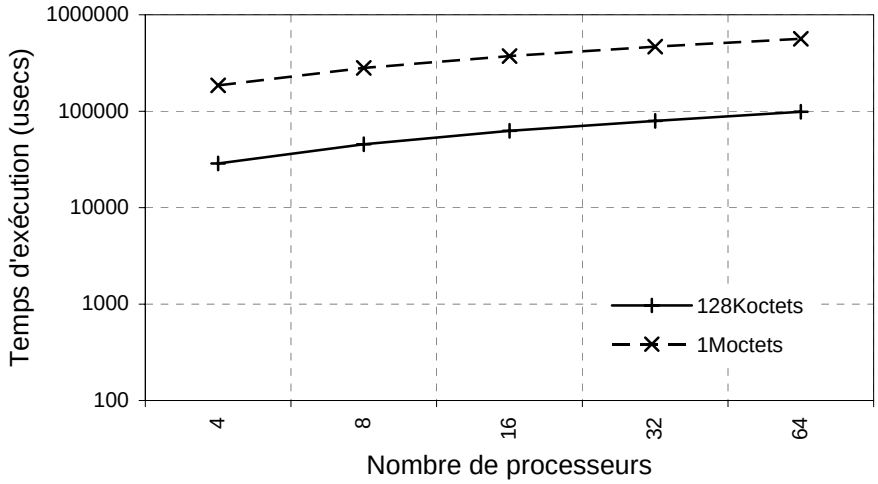


Figure 6.9 *Allure de l'arbre α pour des messages de taille 128Ko et 1Mo en fonction du nombre de processus*

La Figure 6.9 présente le temps de l'algorithme arbre α pour des messages de 128Koctets et de 1Moctets en fonction du nombre de processus avec α égal à 0.5. Les courbes sont de la forme $k \log_2 P$, P est le nombre de processus participant.

6.2.3 L'algorithme de l'arbre chaîne

Nous prenons la méthode de prise de temps avec la barrière de synchronisation pour évaluer l'algorithme arbre chaîne. La Figure 6.10 présente ensuite l'allure du temps de cet algorithme en fonction de taille des messages sur 4, 8, 16, 32, 64 processus. En observant cette figure, nous constatons que :

- Toutes les courbes ont la même allure.
- Après un seuil de taille de message, les courbes se présentent sous la forme de la fonction linéaire, soit $t = \alpha n + \beta$ (n est la taille du message envoyé). Ce seuil dépend au nombre de processus.

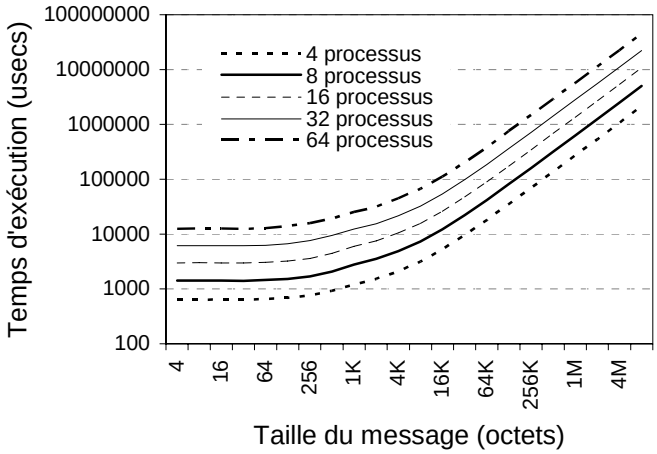


Figure 6.10 *Allure de l'arbre chaîne pour 4, 8, 16, 32 et 64 processus*

6.2.4 Comparaison des algorithmes

Dans cette partie, nous comparons la performance des trois algorithmes de diffusion totale d'ICS : l'arbre plat, l'arbre α et l'arbre chaîne. Nous utilisons la méthode de prise du temps avec la barrière de synchronisation pour comparer le temps d'exécution des algorithmes et le débit δ_{ab} (*cf.* l'équation 6.2, page 138) et la méthode de prise du temps

sans barrière pour comparer leur débit δ_{sb} (*cf.* l'équation 6.3, page 139). La valeur α utilisée dans l'algorithme arbre α est égale 0.5

Temps d'exécution

Les figures de 6.11 à 6.13 présentent la comparaison de temps d'exécution entre ces trois algorithmes de diffusion totale sur différents nombres de processus et la version de l'opération de diffusion de LAM. Nous observons que :

- L'algorithme arbre α est le meilleur algorithme dans tous les cas de figure. L'arbre α dans cette comparaison est l'arbre binomial (α est égal à 0.5).

- Le gain de temps d'exécution par l'algorithme α par rapport aux autres algorithmes est croissant en fonction du nombre de processus pour les messages de tailles supérieures au kilo-octet, ce qui vérifie les complexités présentées dans la section 2.3.3 (page 26).

- Le décalage des temps d'exécution entre l'algorithme α et les autres pour les messages de petite taille (inférieure à 1K octets) est plus petit que celle pour les messages de grande taille (prédominance du *startup* des communications sur le temps total). Ce décalage varie aussi en fonction du nombre de processus.

- *LAM6.5.9_broadcast* a un temps d'exécution plus faible qu'INUKTITUT α_0.5 pour des messages de petite taille. Cela est dû à son temps d'envoi également faible. Nous ne se tranchons en outre pas le temps de barrière dans le temps d'exécution et la barrière par défaut de *LAM* est implantée par l'algorithme d'arbre binomial par rapport de l'algorithme d'arbre plat d'INUKTITUT. Quand la taille de message est grande, le temps de barrière devient négligeable par rapport au temps d'exécution. Par conséquent, *LAM* n'a plus ce gain de temps. De plus, le changement de protocole de *« short »* à *« long »* (*cf.* section 5.2.2, page 103) de

LAM augmente son temps d'exécution jusqu'à être plus important qu'INUKTITUT.

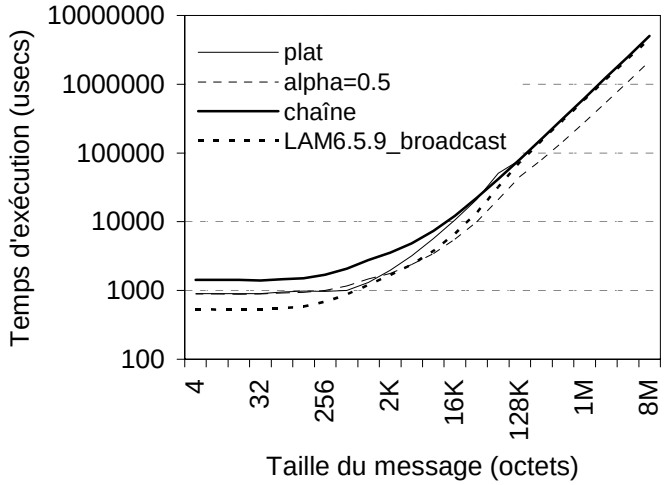


Figure 6.11 *Comparaison des temps d'exécution des algorithmes arbre plat, arbre α et arbre chaîne sur 8 processus*

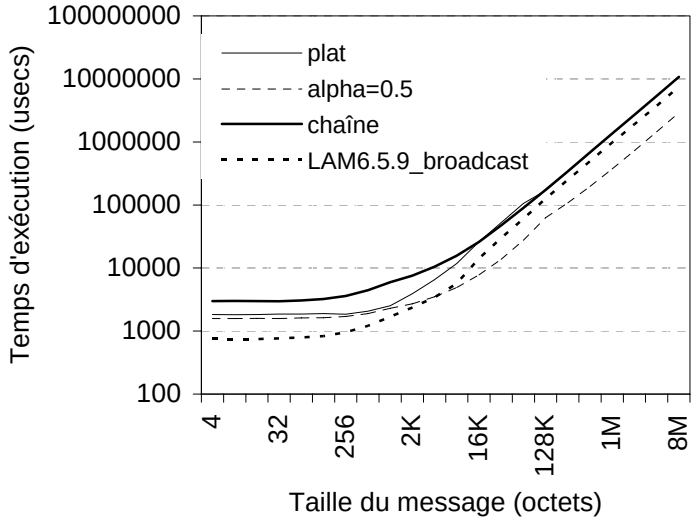


Figure 6.12 *Comparaison des temps d'exécution des algorithmes arbre plat, arbre α et arbre chaîne sur 16 processus*

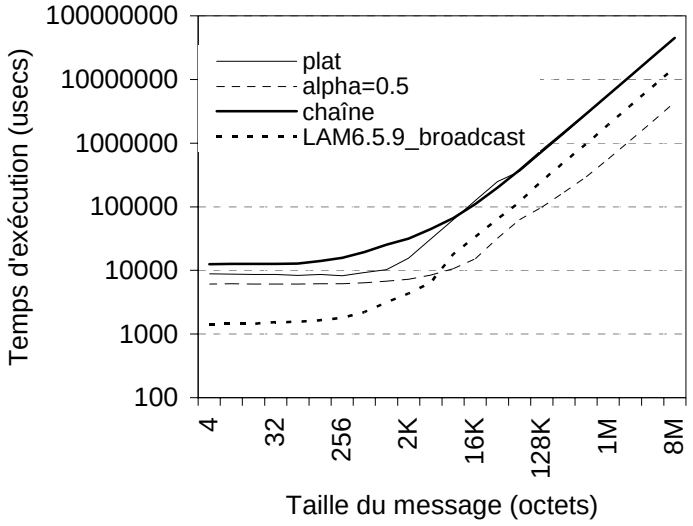
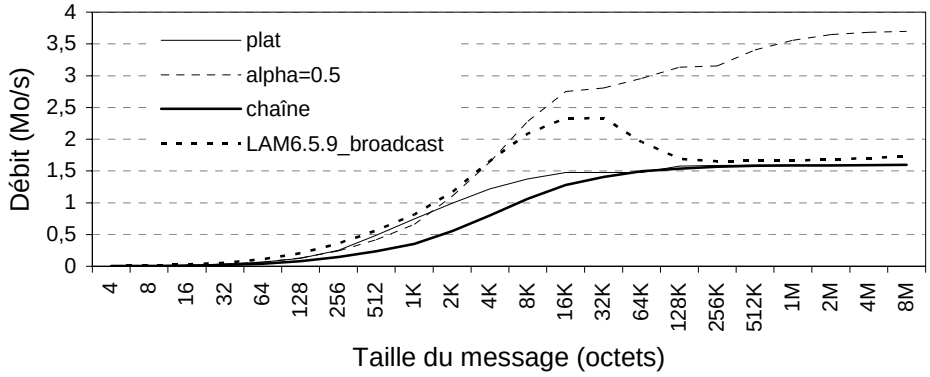


Figure 6.13 *Comparaison des temps d'exécution des algorithmes arbre plat, arbre α et arbre chaîne sur 64 processus*

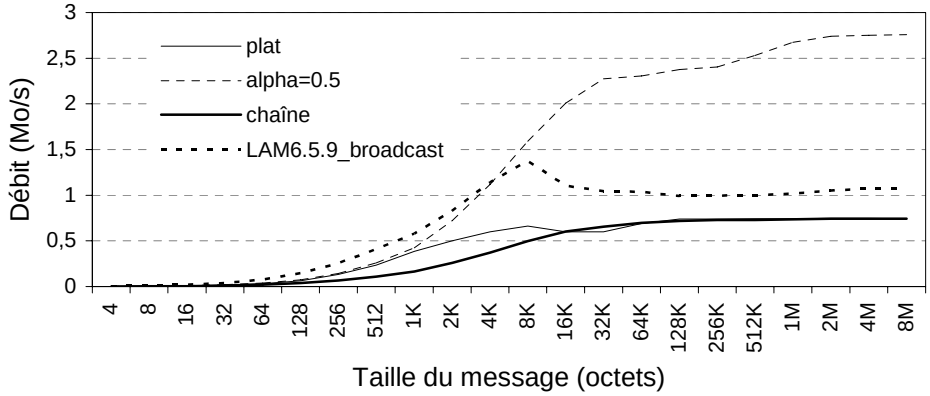
Débit δ_{ab}

La Figure 6.14 présente donc la comparaison de débit δ_{ab} évalué par l'équation 6.2 (*cf.* section 6.1.1, page 138) entre trois algorithmes. Via étude cette figure, nous remarquons que :

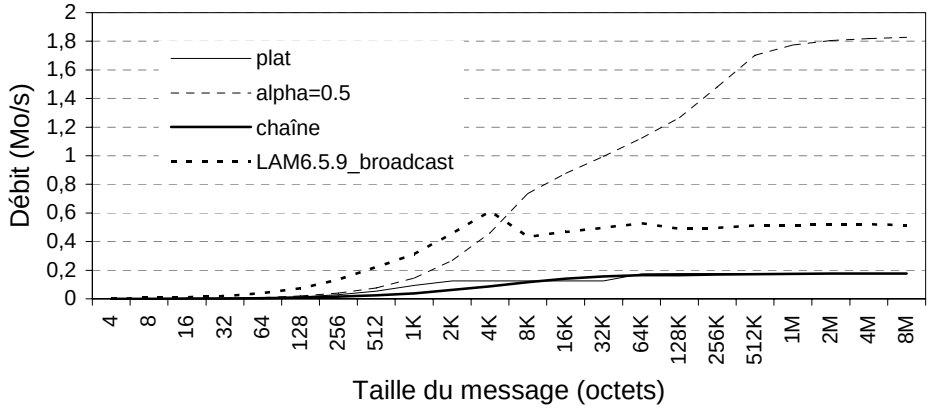
- L'algorithme arbre α donne les meilleurs débits par rapport aux autres algorithmes parce qu'il a des temps d'exécution plus faibles que les autres. L'algorithme chaîne, à l'inverse, a le débit qui est plus faible que les deux autres algorithmes. Dans cet algorithme, le processus initiateur n'envoie qu'un message à un autre processus qui continue à l'envoyer à un autre et l'initiateur attend la réponses du dernier processus avant commencer une nouvelle exécution. Dans l'algorithme arbre α (α=0.5), l'initiateur n'envoie que deux messages à ses deux autres processus et puis ces processus envoient parallèlement à ses fils (processus).
- Les débits maximums de l'algorithme arbre α pour 8, 16 et 64 processus égalent respectivement à 3,69 Mo/s, 2,75Mo/s et 1,83Mo/s.
- Peu de différence de débit entre l'arbre plat et l'arbre chaîne pour les grandes tailles de message (supérieure à 64K octets).



(a) 8 processus



(b) 16 processus



(c) 64 processus

Figure 6.14 *Comparaison des débits δ_{ab} des algorithmes arbre plat,α et chaîne et LAM6.5.9_broadcast*

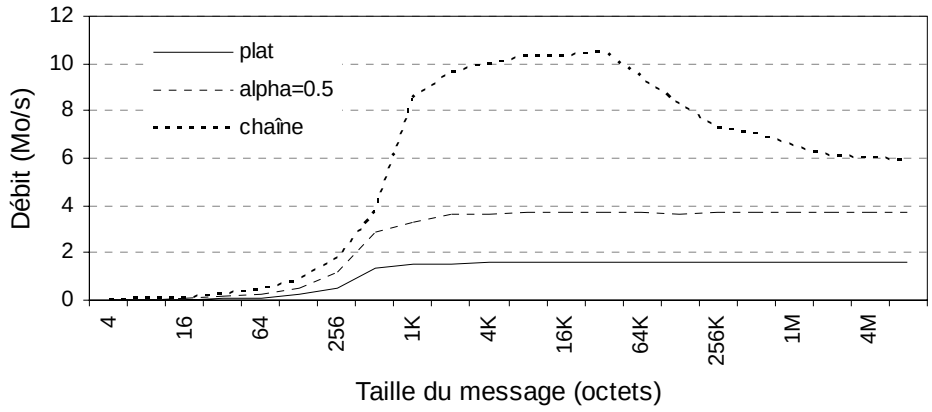
Débit δ_{sb}

La Figure 6.15 présente la comparaison des débits des trois algorithmes. Nous rappelons que le débit est évalué par l'équation 6.3 (*cf.* section 6.1.1, page 139) de la méthode de prise de temps sans barrière de synchronisation. L'utilisation de cette méthode cause un effet de *pipelining*. En observant cette figure, nous constatons que :

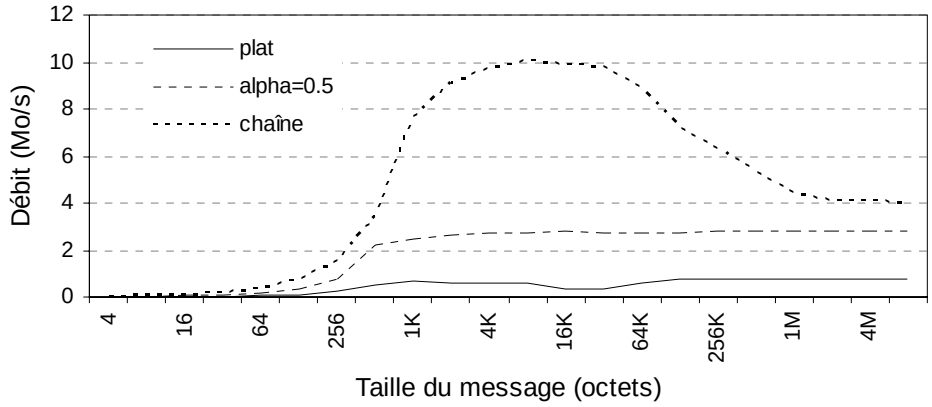
- L'algorithme arbre chaîne donne les meilleurs résultats par rapport aux autres algorithmes parce qu'il profite bien du mécanisme de *pipeline*. L'algorithme plat, à l'inverse, a le débit qui est plus faible que les deux autres algorithmes. Dans cet algorithme, le processus initiateur envoie directement un message à tous les autres à chaque exécution. Il faut attendre la fin de l'envoie au dernier processus avant commencer une nouvelle itération. Dans l'algorithme arbre chaîne, l'initiateur n'envoie qu'un message à un autre processus et il commence aussitôt une nouvelle itération.

- L'algorithme arbre α donne les meilleurs résultats pour les grandes tailles de message (supérieure à 1M octets) dans le cas d'un grand nombre de processus participants (supérieure à 32 processus).

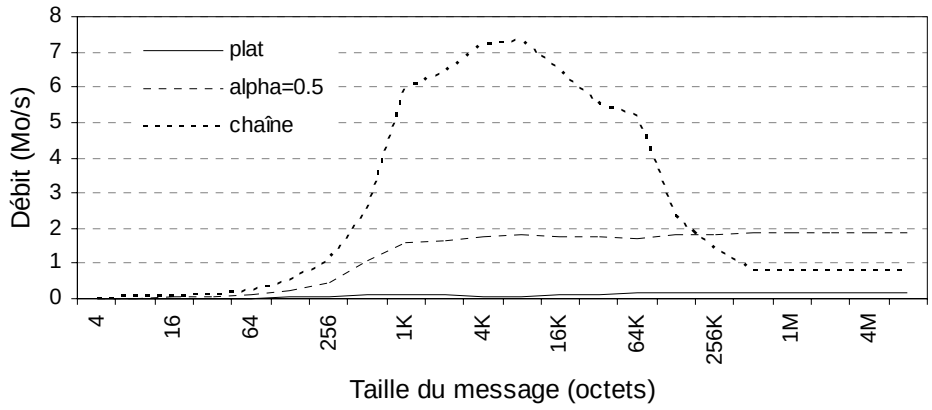
Ces résultats nous permettent de prédire que l'utilisation d'une méthode de segmentation des messages envoyés donnera de bonnes performances. Par exemple, une taille de 8Koctets donne un meilleur débit dans nos expériences.



(a) 8 processus



(b) 16 processus



(c) 64 processus

Figure 6.15 *Comparaison des débits δ_{sb} des algorithmes arbre plat, α et chaîne*

6.3 Évaluation des performances du service de barrière

Nous étudions dans cette partie le comportement du service de barrière. La Figure 6.16 présente l'allure de temps $t_{serbarr}$ (*cf.* l'équation 6.4, page 140) de chaque algorithme de communication utilisé en fonction du nombre de processus. Via étude de ces courbes, nous constatons que :

- La courbe de l'arbre plat se présente sous la forme de la fonction linéaire $t_{plat}(P) = \omega P + \varphi$. L'allure de la courbe de l'arbre α est une fonction logarithme $t_{\alpha}(P) = k \log_{\frac{1}{1-\alpha}} P$ (P est le nombre de processus participant).

- L'arbre plat donne le meilleur résultat par rapport à l'arbre α pour le nombre de processus inférieur à 8. Au-delà de 8 processus le meilleur temps d'exécution est obtenu pour l'arbre α. Le décalage entre deux courbes croît rapidement en fonction croissante du nombre de processus. Pour un nombre de processus égal à 64, par exemple, le temps d'exécution de l'arbre plat est trois fois plus important que celui de l'arbre α.

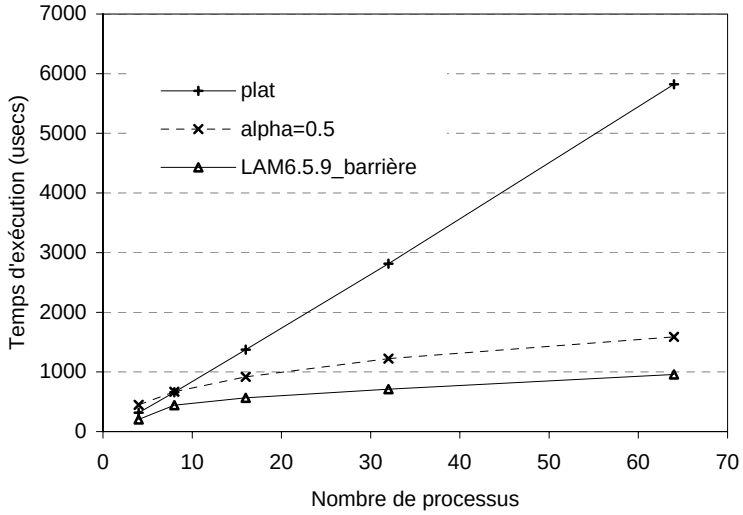


Figure 6.16 *Allure du temps d'exécution du service de barrière par l'algorithme arbre plat, l'algorithme arbre α(α=0.5) d'ICS et de la barrière de LAM 6.5.9 en fonction du nombre de processus.*

6.4 Conclusions

Dans ce chapitre, nous avons évalué les performances des opérations de communication collective implantées dans la bibliothèque ICS d'INUKTITUT. Deux méthodes d'évaluation ont été utilisées pour mesurer le temps d'exécution de diffusion totale. Nous avons aussi étudié le comportement de trois algorithmes de diffusion : l'arbre plat, l'arbre α et l'arbre chaîne sur les nombres différents de processus. Trois valeurs différentes de α ont été utilisées : 0.5, 0.4 et 0.3. Les résultats obtenus avec la première méthode d'évaluation et leurs analyses nous permettent de dire que le meilleur algorithme est l'arbre α avec α égal à 0.5 (l'arbre binomial) dans la plupart des cas de diffusion total. Pour les cas d'une petite tailles des messages (inférieure à 1Ko) et d'un nombre de processus inférieur ou égal à 8, l'α qui offre le meilleur temps est 0.3.

Une autre remarque est que le temps d'exécution de chaque algorithme de diffusion est fonction du nombre de processus et de la taille de message diffusé. Ceci vérifie la complexité de ces algorithmes présentée dans la section 2.3.3 (page 26).

Nous constatons aussi que le mécanisme de *« pipeline »* a des influences différentes suivant les algorithmes de diffusion totale. Nous pouvons exploiter cette capacité pour segmenter un message afin d'augmenter sa performance. Nous avons besoins d'études plus approfondies pour résoudre ce problème. Il existe aussi des travaux recherche similaires dans [VFD00].

L'étude des performances du service de barrière nous permet de conclure que l'utilisation des barrières avec l'arbre binomial donne la meilleure performance pour le cas d'un nombre de processus supérieur à 8.

7

Conclusion et perspectives

Bilan

De nombreux travaux concernent le développement de bibliothèques de communication [Myr, ABMN02, Aum02, FKT96] et de processus légers [Nam01, DN03, DM03] sur des architectures de type grappe. Ces bibliothèques offrent de meilleures performances que les standards de l'industrie comme POSIX Thread [IEEE94] ou MPI [SOH96]. Elles utilisent à travers des interfaces parfois propriétaires. Du fait de cette complexité, le portage sur grappe d'application ou d'environnement de programmation parallèle, comme *Athapascan*, est un travail difficile.

L'objectif de ce travail de thèse a été de définir et d'évaluer les performances d'INUKTITUT, un interface qui permet de porter efficacement et facilement sur ces bibliothèques des applications et des environnements de programmation parallèle de haut niveau comme *Athapascan*. En définissant un petit ensemble de fonctions d'interface, INUKTITUT fournit une abstraction des machines parallèles virtuelles basée sur des communications de type message actif [ECGS92]. Un processus peut être lié à un réseau virtuellement complètement maillé et ainsi communiquer avec les autres processus existants dans le réseau. INUKTITUT utilise deux concepts pour exploiter au mieux l'architecture des grappes de machines SMP : le parallélisme intra-nœuds des

processus est géré à l'aide des processus légers communiquant par la mémoire partagée ; et le parallélisme inter-nœuds est exploité via l'échange de messages actifs. De plus, INUKTITUT permet aussi le déploiement parallèle [Mar03] d'applications ainsi que l'initialisation d'une machine virtuelle et d'un réseau de communication initial. L'évaluation des performances d'INUKTITUT est réalisée par l'estimation d'indicateurs de performance mesurés au dessus des différentes implantations de l'interface.

Les résultats et les analyses du chapitre 4 montrent que l'utilisation de l'interface de processus légers PTH d'INUKTITUT permet d'obtenir des performances équivalentes à celles de la bibliothèque standard de *POSIX Pthreads*. Le surcoût dû à l'utilisation de l'interface est négligeable par rapport à l'utilisation native d'une bibliothèque *POSIX Pthreads* comme LinuxThread [Ler96] ou NPTL [DM03].

Nous avons mesuré les performances l'interface de communication d'INUKTITUT à travers des micro-benchmarks pour les communications bi-points et collectives. Une application synthétique d'un schéma numérique de type Jacobi a aussi été utilisée pour mesurer les capacités de recouvrement calcul/communication. Nous avons utilisé plusieurs grappes (architecture IA32 et IA64) et testé plusieurs implantations d'INUKTITUT (SOCKNET, CORNET, GMNET) sur différents réseaux et protocoles de communication (TCP/IP, Myrinet). Si l'interface d'INUKTITUT de message actif permet de cacher au programmeur les différentes mises en œuvre sur des protocoles de communication standards, l'estimation des indicateurs de performance d'Hockney (r_∞, $n_{1/2}$, t_0) nous permet de conclure que :

- La communication bipoint au-dessus de SOCKNET donne une meilleure performance au niveau de la bande passante r_∞, par rapport à la version 6.5.9 (mode c2c) de *LAM/MPI* sur les différentes plates-formes testées.
- Le modèle de communication asynchrone et unidirectionnelle d'INUKTITUT permet d'exploiter la potentialité de recouvrement calcul/communication sur des architectures de nœud qui le permet comme par exemple les nœuds SMP.
- À travers l'évaluation de l'interface de communication collective ICS présentée dans le chapitre 6, nous constatons que le meilleur algorithme pour la diffusion totale est l'arbre α où $\alpha = 0.5$. De plus, la technique de *pipeline* augmente la performance du débit. Les implantations offertes s'avèrent plus

performantes que l'implantation LAM/MPI à partir d'une taille de message de 8K octets.

Le bilan d'INUKTITUT est globalement positif, tant du point de vue des performances intrinsèques des opérateurs que des performances obtenues sur des applications et en particulier des performances d'*Athapascan* au-dessus d'INUKTITUT [Rev04]. En cela nous retrouvons partiellement les conclusions de [ECGS92] qui montrent que l'utilisation de message actif permet d'inclure efficacement des données provenant du réseau dans un calcul en cours d'exécution.

Nous notons que les performances d'INUKTITUT sur le réseau GMNET relève un surcoût important pour des messages de petites tailles. Nous suppons que cette implantation effectuée rapidement souffre d'une « erreur de jeunesse ». Ce résultat négatif serait à comparer aux performances d'INUKTITUT sur une couche de plus haut niveau que GM et aussi performante, telle que PM², disponible sur architecture IA64 à la fin de cette thèse, et offrant des fonctionnalités proches de celle d'INUKTITUT.

Travaux futurs et perspectives

INUKTITUT est en évolution pour devenir une interface portable et efficace pour des environnements de programmation parallèle sur les grappes et les grilles de grappes. Pour ce faire, les travaux futurs consistent à :

- Développer une implantation d'INUKTITUT au-dessus d'une bibliothèque performante sur réseau rapide comme PM² [Nam01][ABMN02][Aum02]. Il s'agira d'implanter efficacement l'allocation des données INUKTITUT, en particulier celles qui seront directement communiquées par le réseau.
- L'étude des applications avec Athapascan montre qu'il est important de pouvoir écrire un parallélisme à grain fin qui sera automatiquement ordonnancé en regroupant les calculs locaux sur un même processeur. Dans ce cas, les communications entre processeurs peuvent être fréquentes et les messages contiendront des données de petite taille. Il est donc important de pouvoir agglomérer les messages entre deux processeurs afin de réduire le temps de démarrage des communications (*cf.* section 3.6.1, page 72).

- Compléter les fonctions de réduction pour la communication collective dans la bibliothèque ICS et mesurer leurs impacts au niveau applicatif, par exemple à travers des applications en Athapascan qui permet de les exploiter automatiquement.
- Étendre l'interface d'INUKTITUT pour l'ajout dynamique de nœuds et l'exploitation des grilles de calcul. Cela nécessite en particulier d'étendre les algorithmes de diffusion pour exploiter pleinement les performances de ces architectures. Les résultats de l'ACI REDGRID [RedG] ainsi que la thèse de doctorat de Luiz Angelo Estefanel du laboratoire ID-IMAG constitue une base de travail à étudier.

Annexe A

La plate-forme d'expérimentation

Dans cette annexe, nous présentons deux plates-formes principales : le I-Cluster1 et le I-Cluster2 qui sont utilisées dans la plupart des expérimentations.

I-Cluster1

Le I-Cluster1 [RAM+01] est une grappe de 225 machines qui a été mise en œuvre par le laboratoire ID [LabID] et l'équipe *hp-labs* de Grenoble. Le I-Cluster1 est une association de PC classique (*personal computer*, en anglais) de type *hp e-PC* [Hpe] reliés par un réseau « Ethernet » classique (100Mb/s) à des commutateurs (*switch*, en anglais) que l'on trouve sur le marché. Les commutateurs sont reliés entre eux par un réseau « Gibabit-Ethernet » (Figure A.1). En Juin 2001, il a été classé à la 385^{ième} du TOP500 [Top500].

Un noeud du I-Cluster1 est un mono Pentium III à 733 MHz, 256Mo de mémoire, 15Go de disque dur. Son architecture logicielle au moment du lancement des expériences est :

- Système d'exploitation : Mandrake [Man] version 9.0 avec le noyau 2.4.18 de Linux.[Linuxnoy]
- Compilateur : g++ 3.2.2
- La bibliothèque de processus léger : LinuxThreads [Ler96]

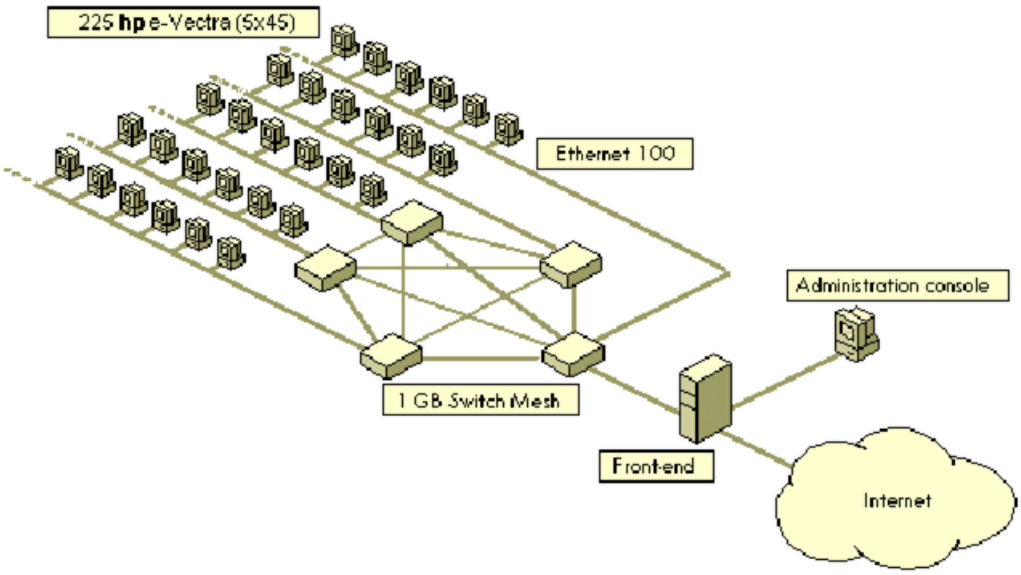


Figure A-1 Le I-Cluster1

I-Cluster2

I-Cluster2 [Iclus2] est une grappe de 104 bi Itanium-2. C'est le premier superordinateur en France construit à partir de processeurs Itanium-2. Chaque nœud est un HP rx2600 avec la configuration suivante : deux processeurs Itanium-2 à 900 MHz, 3Go de mémoire, 72Go de disque dur. Les nœuds sont reliés par un réseau Myrinet [Buy99]. Il y a aussi 5 commutateurs « Fast Ethernet » reliés par un commutateur « Gigabit Ethernet » (Figure A-2). En Mars 2003, il a été classé à la 283^{ième} du TOP500 [Top500].

L'architecture logicielle d'un nœud est :

- Système d'exploitation : RedHat [Red] version AS 3.0 avec le noyau 2.4.21 de Linux.[Linuxnoy]
- Compilateur : g++ 3.2.3
- La bibliothèque de processus léger : NPTL 0.6 [DM03]

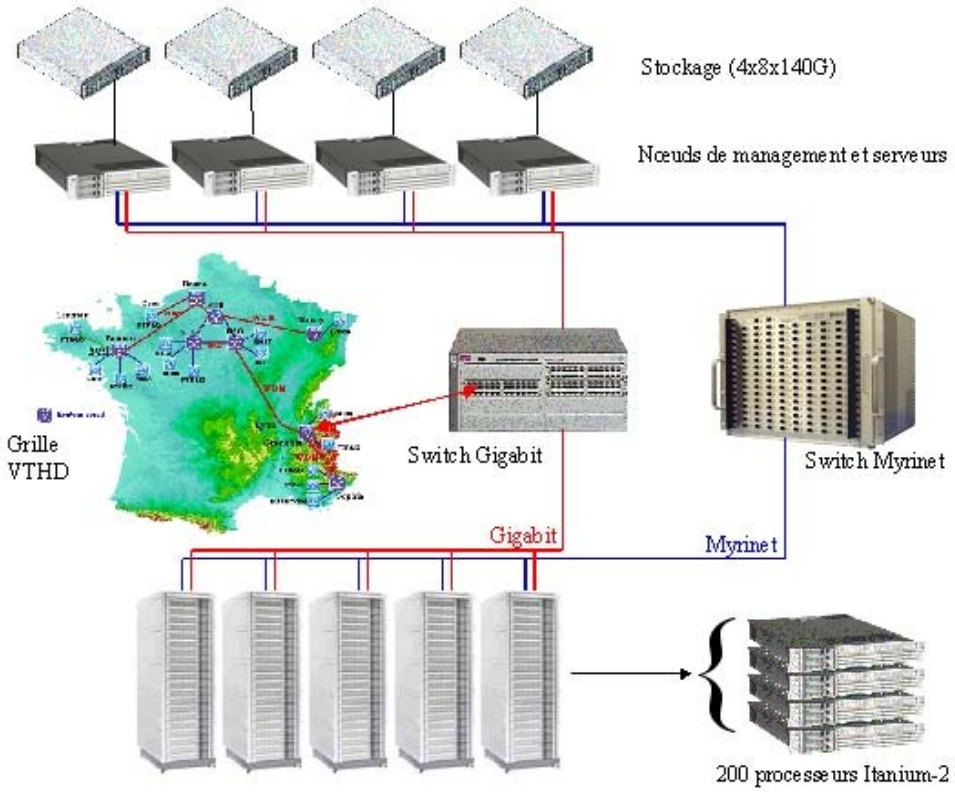


Figure A-2 Le I-Cluster2
(*http://www.inrialpes.fr/sed/i-cluster2/hardware.html*)

Opata

Opata est une machine biprocesseur Athlon 1,2 GHz, 1024Mb RAM. Son architecture logicielle est :

- Système d'exploitation : *Debian* avec deux noyaux de Linux [Linuxnoy] : le noyau 2.4.26 et le noyau 2.6.8.
- Compilateur : gcc 3.3.3
- La bibliothèque de processus léger : Linux Thread 0.10 [Ler96] pour le noyau 2.4.26 et NPTL 0.6 [DM03] pour le noyau 2.6.8

Idpot

Idpot est une grappe de 48 Xeon biproc 2.5Ghz, du laboratoire ID. L'architecture logicielle d'un nœud est :

- Système d'exploitation : Debian avec le noyau 2.4.24 de Linux.[Linuxnoy]
- Compilateur : gcc 3.3.3
- La bibliothèque de processus léger : Linux Thread 0.10 [Ler96]

Annexe B

Mesures

Dans cette annexe, nous étudions d'abord la méthode de datation utilisée dans des mesures de temps d'exécution de nos expérimentations et nous présentons ensuite le calcul des moyennes et d'un intervalle de confiance.

B.1 La datation

La prise de temps utilisée dans nos expérimentations se base sur la fonction système de Unix (et Linux aussi) suivante :

```
int    gettimeofday(struct timeval  *tv, struct  timezone  *tz)
```

qui renvoie la date tenue à jour à travers la structure struct timeval *tv. Cette structure se compose d'une partie *seconde* (long tv_sec) et d'une autre *microseconde* (long tv_usec)[1]. La résolution théorique maximum de cette fonction est donc 1µs.

Dans Linux, l'implantation de cette fonction dépend de l'existence d'un registre de compteur de cycle *TSC* (*« time stamp counter »* [She96]) dans l'architecture du processeur utilisé. La valeur de ce registre correspond au nombre de cycles effectués par le processeur depuis son démarrage. Dans le cas de l'existence de ce registre, la date est calculée et mise à jour à partir de cette valeur[2]. Sinon, c'est-à-dire en absence de registre, la date est mise à jour à partir d'un circuit spécialisé (puce 8254) qui procure le nombre de cycles écoulés par un oscillateur à quartz pour le calcul de temps [BC00].

[1] La valeur 0:0 (tv_sec:tv_usec) correspond donc au 01.01.1970
[2] Ce calcul s'explique en détail dans [BC00]

De plus, Linux permet aussi à l'utilisateur de consulter le temps à travers l'accès direct au nombre de cycle du processeur dans le registre *TSC*. Pour ce faire, il faut diviser ce nombre de cycle par la fréquence du processeur. La résolution de temps prise par cette méthode est plus fine que celle obtenue par la fonction *« gettimeofday »*. De ce fait, cette méthode est souvent utilisée dans les cas où on a besoin de précision inférieure à 1µs.

Bien que la résolution de *« gettimeofday »* soit 1µs, il est possible d'augmenter la durée mesurée, par exemple en répétant un certain nombre de fois une même expérience. Nous utilisons cette technique pour prendre le temps dans toutes nos expérimentations. Le tableau B.1 présente la résolution R et le coût moyen t_c d'utilisation de cette fonction sur quelques plates-formes dans nos expérimentations (*cf.* Annexe A). Les deux facteurs R et t_c sont mesurés par *l'algorithme B.1* [LinP] suivant.

Algorithme B.1

```
struct timeval t[N];
for i =0 to N do
    gettimeofday(&t[i]);
enddo
```

R = min{(t[i] - t[i-1])} i=1, 2,…N et (t[i] - t[i-1]) ≠ 0

$$t_c = \frac{1}{N}\sum_{i=1}^{N}(t[i]-t[i-1])$$

Plate-forme	***R(µs)***	***t_c(µs)***
I-Cluster 1	1	0,64 ± 0,017
I-Cluster 2	1	0,62 ± 0,014
Opata (2.4.26)	1	0,35 ± 0,013

Tableau B.1 Durée d'exécution de la fonction *« gettimeofday »* (N=10000)

Nous avons utilisé la fonction *« gettimeofday »* dans toutes les expériences de cette rédaction.

B.2 Estimation d'une moyenne et d'un intervalle de confiance

Chaque valeur utilisée dans les tableaux et les courbes de cette thèse est une valeur moyenne. Supposons que $\{x_1, x_2, ..., x_n\}$ soient les *n* observations de l'exécution d'une même expérience. Le moyenne $\bar{x}$ est alors estimé par :

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Nous avons choisit ***n = 100*** pour toutes les expériences de processus légers ainsi que de communication bipoint et ***n = 50*** pour ceux des communications collectives. Nous utilisons de plus l'intervalle de confiance IC. L'intervalle de confiance pour un échantillon « assez grand » (n>30) se calcule par la formule [Jai91] :

$$\text{IC} = \left(\bar{x} - \theta_\varepsilon \frac{\sigma_n}{\sqrt{n}}, \bar{x} + \theta_\varepsilon \frac{\sigma_n}{\sqrt{n}} \right)$$

où $\bar{x}$ est l'estimateur de la moyenne ci-dessus et σ_n est l'estimateur de l'écart type calculé par :

$$\sigma_n = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} (x_i - \bar{x}_i)^2}$$

et θ_ε dépend du niveau de confiance souhaité ε calculé par [CI] :

$$\varepsilon = \frac{1}{\sqrt{2\pi}} \int_{-\theta_\varepsilon}^{\theta_\varepsilon} e^{-\frac{x^2}{2}} dx$$

Par exemple, pour les niveaux de confiance de 95% et de 99%, θ_ε égale respectivement à 1,96 et 2,57. Ces chiffres sont fournis dans les tableaux statistiques.

Dans cette thèse, l'intervalle de confiance IC utilisé est ***99%***.

Bibliographie

[ABMN02] O. Aumage, L. Bougé, J-F. Méhaut, and R. Namyst. Madeleine II: A portable and efficient communication library for high-performance cluster computing. *Parallel Computing*, 28(4):607-626, April 2002

[ACGL98] M. Acacio, O. Canovas, J.M. Garcia, P.E. Lopez-de-Teruel. The Performance of MPI Parallel Jacobi Implementation on Workstation Clusters *IX Jornadas de Paralelismo*, Septembre 1998
http://www.ditec.um.es/~jmgarcia/papers/perf-mpi.pdf

[AGH+93] C. Addison, V. Getov, T. Hey, R. Hockney and I. Wolton. The {Genesis}Distributed-memory Benchmarks. *Journal Concurrency: Practice and Experience, vol.5,N°.1*,p.1-22,1993

[AGL+02] D. Ashton, W. Gropp, E. Lusk, R. Ross, B.Toonen. MPICH2 Design Document 0ct-2002,
http://www-unix.mcs.anl.gov/mpi/mpich/adi3/mpich2/mpich2.htm#Node0

[AISS95] A. Alexandrov, M.F. Ionescu, K.E. Schauser, C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation *Journal of Parallel and Distributed Computing, vol.44, N°1,p.71-79*, 1997

[AMD] http://www.amd.com/us-en/Processors/ProductInformation/

[Amd67] Amdahl Validity of the single processor approach to achieving large-scale computing capabilities *In AFIPS Conference Proceedings p.483* 1967

[AS83] Andrews, G. R. and Schneider, F. B., Concepts and Notations for Concurrent Programming, *Computing Surveys, Vol. 15, No. 1, pp. 3-43*, Mar. 1983.

[Aum02] O. Aumage. Heterogeneous multi-cluster networking with the Madeleine III communication library. In *Proc. 16th Intl. Parallel and Distributed Processing Symposium, 11th Heterogeneous Computing Workshop (HCW 2002)*, Fort Lauderdale, April 2002

[BBC04] S. B. Baden, G.T. Balls, P. Colella Applications, scalability and technological change p.50 *http://www.psc.edu/training/PPS_May04/talks/Baden.pdf* 04/2004

[BC00] D.P.Bovet, M.Cesati Understading the Linux Kernel *Chapter 5 p.145-p.146 O'Reilly* October 2000

[BEMP93] S. Bolis, E. G. Economou, D. Mouzakis, and G. Philokyprou. SBP-net : an integrated voice/data token ring LAN. *ACM Computer Communications*, 16, 8 :494–500, 1993.

[Ber97] P.-E. Bernard. Parallélisation et multiprogrammation pour une application irrégulière de dynamique moléculaire opérationnelle. *Thèse de doctorat en mathématiques appliquées, Institut National Polytechnique de Grenoble*, France, October 1997.

[BG03] J.Briat, T. Gautier. What is INUKTITUT ?
http://www-id.imag.fr/logiciels/inuktitut

[BGPP97] J.Briat, I. Ginzburg, M. Pasin, B. Plateau. Athapascan Runtime: Efficiency for Irregular Problems. *In: Proc. of EuroPar'97. Passau.* Aug. 1997.

[BI99] M. Bernaschi and G. Iannello. Collective Communication Operation : Experimental Results vs Theory. *Concurrency : Practical and Experience*, March 1999

[BK94] R. Bariuso, Allan Knies, SHMEM's User's Guide, *Cray Research, SN-2516*, 1994.

[BMN99] L. Bougé, J-F. Méhaut, R. Namyst. Madeleine: An Efficient and Portable Communication Interface for RPC-based Multithreaded Environments. *Rapport de recherche, Institut National De Recherche En Informatique Et En Automatique,* ISSN 0249-6399 ISRN INRIA/RR--3864—FR, N°3845 Decembre 1999

[BN84] A.D. Birrel and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39-59, February 1984

[Brad02a] E. G. Bradford. Setting up timing routines *RunTime: High-performance programming techniques on Linux and Windows 2000_IBM developerWorks :: Linux articles* April 2001 *ftp://www6.software.ibm.com/software/developer/library/l-rt1.pdf*

[Brad02b] E. G. Bradford. Managing processes and threads *RunTime*: *High-performance programming techniques on Linux and Windows _IBM developerWorks :: Linux articles* February 2002 *http://www-106.ibm.com/developerworks/library/l-rt7/*

[Brad02c] E. G. Bradford. Context switching *RunTime*: *High-performance programming techniques on Linux and Windows _IBM developerWorks :: Linux articles* July 2002 *http://www-106.ibm.com/developerworks/library/l-rt9/*

[Brad02d] E. G. Bradford. Context switching, Part 2 *RunTime*: *High-performance programming techniques on Linux and Windows _IBM developerWorks :: Linux articles* Septembre 2002 *http://www-106.ibm.com/developerworks/library/l-rt10/*

[BST88] Bal, H. E., Steiner, J. G., and Tanenbaum, A. S., Programming Languages for Distributed Systems, *IR-147*, Vrije Universiteit, Amsterdam, The Netherlands, Feb. 1988.

[BT89] D.P.Bertsekas and J.N.Tsitsiklis. Parallel and Distributed Computation, *chapter 5, 6.5 6.6 Prentice-Hall*, NJ 1989

[Buy99a] R. Buyya. High Performance Cluster Computing: Architectures and Systems, Volume 1 *p.14. Prentice Hall NJ* 1999

[Buy99] R. Buyya. High Performance Cluster Computing: Architectures and Systems, Volume 1 *p.235-p.238. Prentice Hall NJ* 1999

[Car99] A.S. Carissimi. Athapascan-0: Exploitation de la multiprogrammation légère sur grappes de multiprocesseurs. *Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble*, France Novembre 1999

[Cav99] G.G.H. Cavalheiro. Athapascan-1: Interface générique pour l'ordonnancement dans un environnement d'exécution parallèle. *Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble*, France Novembre 1999

[CCP99] A. S. Charão, I. Charpentier, and B. Plateau. Un environnement modulaire pour l'exploitation des processus légers dans les méthodes de décomposition de domaine. In *11ème Rencontres francophones du parallélisme, des architectures et des systèmes*, Rennes, France, June 1999.

[CFT+94] L.J. Clarke, R. A. Fletcher, M. Trewin, R. Alasdair, A. Bruce, A.G. Smith and S.R. Chapple. Reuse, portability and parallel libraries. *Technical Report TR9413, Edinburgh Parallel Computing Centre, University of Edinburgh*, 1994

[CI] Confidence Interval http://www.iut.dk/bibliotek/encyclopedia/math/c/c547.htm

[CK93] K.M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. *In Research Directions in Concurrent Object-Oriented Programming, MIT Press*, 1993

[CKP+93] D.E. Culler, Karp, D.A. Patterson, A. Sahay, K.E. Shauser, E. Santos, R. Subramonian and T. von Eicken. LogP : Towards a realistic model of parallel

computation. Procs. of the 4th SIGPLAN Symp. on Principles and Practices of Parallel Programming ACM , May 1993

[CLP+93] C.C.Neira, J. Leigh, M. Papka, C.Barnes et al. Scientists in wonderland: A report on visualization applications in the CAVE virtual reality environment. In Proceedings of the Symposium on Research Frontiers in Virtual Reality, p.59-66, San Jose, CA, USA, IEEE Computer Society Press. October 1993

[Com91] D. Comer Internetworking with TCP/IP Vol.1 : Principles, Protocols and Architecture. *Prentice Hall London*, 2 edition, 1991

[CT93] M. Cosnard et D.Trystram Algorithmes et architectures parallèles *Chapitre 2. InterEditions* 1993

[Den02] A. Denis. PadicoTM : un environnement ouvert pour l'intégration d'exécutifs communiquants. In *14èmes Rencontres Francophones du Parallélisme (RenPar'14)*, Hammamet, Tunisie, pages 99-106, April 2002.

[DM03] U. Drepper, I. Molnar. The Native POSIX Thread Library for Linux *http://people.redhat.com/drepper/nptl-design.pdf* January 2003

[DMN99] Y.Denneulin, J-F. Méhaut, R. Namyst Customizable Thread Scheduling directed by Priorities. *Research Report N° 1999-11 Laboratoire de l'Informatique du Parallélisme Ecole Normale supérieure de Lyon* February 1999
http://www.ens-lyon.fr/LIP/Pub/rr1999.html

[DN03] V. Danjean and R. Namyst. Controling Kernel Scheduling from User Space: an Approach to Enhancing Applications' Reactivity to I/O Events. In *Proceedings of the 2003 International Conference on High Performance Computing (HiPC '03)*, Hyderabad, India, December 2003

[Dor99] M.Doreille. Athapascan-1 Vers un modèle de programmation parallèle adapté au calcul scientifique, *Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble*, France 1999

[DPP01] A.Denis, C. Pérez and T.Priol. Portable parallèle CORBA objects : an approach to combine parallel and distributed programming for grid computing. *In Proceedings of the Intl. EuroPar'01 conf.*, pages 835-844, Manchester, UK, 2001, Springer

[DPP02] A. Denis, C. Pérez, and T. Priol. PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin, Germany, pages 144-151, May 2002. IEEE Computer Society.

[DPPR03] A. Denis, C. Pérez, T. Priol, and A. Ribes. Padico: A Component-Based Software Infrastructure for Grid Computing. In *17th International Parallel and Distributed Processing Symposium (IPDPS2003)*, Nice, France, April 2003. IEEE Computer Society.

[DS98] K. Dowd, C.R. Severance. High Performance Computing. Chapitre 1, Chapitre 6 *O'Reilly & Associates* Second Edition July 1998

[DS03] R.P.Dimitrov, a.Skjellum A Theoretical Framework for Overlapping of Communication and Computation and Early Binding, part I : BOUM Model and Overlapping Metrics *Preprint submitted to Elsevier Science* September 2003 *http://www.mpi-softtech.com/company/publications/files/parco-metrics-feb14.pdf*

[Dun+98] D. Dunning et alli. The virtual interface architecture. *IEEE Micro*, pages 66–76, March 1998.

[ECGS92] T. v. Eicken, D. E.Culler, S. C. Goldstein, K. E. Schauser. Active Message A Mechanism for Integrated Communication and Computation. *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, ACM Press

[EKB+92] J.R. Eykholt, S.R. Kleiman, S. Barton, J. Voll, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, M. Weeks, D. Williams Beyond multiprocessing : multithreading the SunOS kernel. *In Proceedings of the Summer 1992 USENIX Technical Conference and Exhibition, pages 11-18, San Antontio, USA,* Jun 1992

[FCL93] M. Feeley, J. Chase and E. Lazowska. User-level threads and interprocess communication. Technical Report UW-CSE-93-02-03, University of Washington, Departement of Computer Science and Engineering, Feb 1993.

[FHM95] T.Fahringer, M.Haines, and P. Mehrotra. On the utility of threads for data parallèle programming. *In ACM, editor, Conference proceedings of the 9th International Conference on Supercomputing*, Barcelona, Spain, July 3-7, 1995 pages 51-59. ACM Press, New York, NY 10036, USA, 1995

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. Solving Problems on Concurent Processors, vol.1 chapter 6 and 15, *Prentice Hall, Englewood Cliffs*, NJ, 1988

[FK97] I. Foster, C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115-128, Summer 1997

[FKT96] I. Foster, C. Kesselman, S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing* Vol.37, No.1, pp. 70-82. 1996

[FN92] E.W. Felten, D.McNamee Improving the performance of Message Passing Application by Multithreading *Proceedings of the Scalable High Performance Computing Conference*, pages 84-89, April 1992

[FO95] I. Foster and R.Olson. A guide to parallel and distributed programming in nperl. *Technical report, Argonne National Lab*, 95

[Fos95] I.Foster. Designing and Building parallel programs. Chapitre 2 *Addison-Wesley* 1995

[FOT93] I. Foster, R.Olson and S. Tuecke. Programming in fortran M. *Technical report, Argonne National Lab*, 93

[FST98] G. Folino, G. Spezzano and D. Talia. Performance Evaluation and Modeling of {MPI} Communications on the Meiko {CS}-2. *HPCN Europe,p.932-936*, 1998

[Gal99] F.Galilée. Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle. *Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble*, France Septembre 1999

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. PVM: Parallel Virtual Machine: A User's Guide and Turorial for Networked Parallel Computing. *Scientific and Engineering computation, MIT Press, Cambridge, MA, USA*, 1994

[GC01] D. Grove and P. Coddington. Precise MPI performance measurement using MPIBench. *In Proceedings of HPC Asia*, September 2001

[GF00] F.Garcia and J.Fernandez. POSIX Thread Libraries. *Linux Journal* February 2000. *http://www.linuxjournal.com/article.php?sid=3184*

[Gin97] I. Ginzburg. Athapascan-0b: Intégration efficace et portable de multiprogrammation légère et de communications. *Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble*, France Septembre 1997

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22 (6): 789-828, September 1996

[Gor95] W. Goralski. TCP/IP applications and protocols. *Computer Technology Research Corp*, 6N. Atlantic Wharf, Charleston, SC 29401-2150, USA, 1995

[GRC+98] F. Galilée, J-L. Roch, G. Cavalheiro and M. Doreille. Athapascan-1: Online building data flow graph in a parallel language. *In IEEE, editor, Pact 98', page 88-95, Paris, France*, October 1998

[Gui04] C. Guilloud Traçage flexible d'exécutions de programmes parallèles *These de doctorat en informatique,* Institut National Polytechnique de Grenoble, France Février 2004

[HB94] R.Hockney, M.Berry Public International Benchmarks for Parallel Computers, Feb 7 1994

[HCM94] M. Haines, D.Cronk and P. Mehrotra. On the design of Chant : A talking threads package. *In Supercomputing '94, pages 350-359,* November 1994

[Hel94] M. Heller Win32 Programmation *Sybex p.157* 1994

[Her04] E. Hermann Portage de la bibliothèque de communication INUKTITUT sur architecture Itanium et réseau rapide Myrinet. *Rapport de stage de 3ème année ENSIMAG*. Juin-2004

[Hus99] L.P. Huse. Collective Communication on Dedicated Cluster of Workstations. *Proceedings of the 6th European PVM/MPI User's Group Meeting, Barcelona, Spain,* September 1999. p(469-476).

[HWW97] K.Hwang, C.Wang, C.-L.Wang. *Evaluating MPI Collective Communication on the SP2, T3D, and Paragon Multicomputers.* HPCA 1997 p.106-115

[Ian97] Giulio Iannello. Efficient Algorithms for the Reduce-Scatter Operation in LogGP **.** *IEEE Transactions on Parallel and Distributed Systems* (Vol. 8, No. 9) September 1997

[Iclus2] I-Cluster2 Presentation *http://I-cluster2.inrialpes.fr*

[IEEE93] IEEE. *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE Computer Society Press, IEEE-Standard, New York,1993, 1993.

[IEEE94] IEEE 1003.1c-1994 Standard for Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Programming Interface (API) – Amendment 2: Threads Extension [C langage]. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994

[ILM98] G. Iannello, M. Lauria, and S. Mercolino. Cross-Platform Analysis of Fast Messages for Myrinet. *In Proc. Workshop CANPC'98, number 1362 in Lecture Notes in Computer Science,* pages 217-231, Las Vegas, Nevada, January 1998. Springer.

[Jai91] R. Jain The art of computer system performance analysis *John Wiley & Son, Inc.* 1991

[KBG00] T. Kielmann, H.E. Bal and S. Gorlatch. Bandwidth-efficient Collective Communication for Clustered Wide Area Systems. *In Proc. International Parallel and*

Distributed Processing Symposium (IPDPS2000), page 492-499, Cancun, Mexico, May 2000

[KBV00] T. Kielmann, H.E. Bal and K. Verstoep. Fast Measurement of {LogP} Parameters for Message Passing Platforms. *Lecture Notes in Computer Science, volume 1800*, p.1176--??,2000

[KGPS97] J.A. Kohl, G.A. Geist, P.M. Papadopoulos, and S.Scott. Beyond pvm 3.4 : What we've learned, what's next, and why. In Proceedings of Euro PVM-MPI 97, November 1997

[KKJ01] J.Kim, K. Kim and S. Jung. Building a High Performance Communication Layer Over Virtual Interface Architecture on Linux Clusters. *15th ACM International Conference on Supercomputing (ICS)*, June 16-21, Sorrento, Italy, 2001

[KSSS93] R.M. Karp, A. Sahay, E.E. Santos, K.E. Shauser. Optimal Broadcast and Summation in the LogP Model . *Procs. of the 5th Annual ACM Symp. on Parallel Algorithms and Architectures* June 1993 pp. 142-153.

[LAM65] LAM 6.5.9 Installation Guide *http://www.lam-mpi.org/6.5/install.php*

[LabID] Laboratoire Informatique et Distribution *http://www-id.imag.fr*

[LB00] B. Lewis D.J. Berg Multithreading programming with Java Technology *SunMicrosystem Press A Prentice Hall title* 2000

[LCW+03] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P.Wyckoff, and D. K. Panda. Performance comparison of MPI implementations over In_niBand, Myrinet and Quadrics. In *The International Conference for High Performance Computing and Communications (SC2003)*, November 2003.

[Les93] B.P. Lester. The art of parallel programming Chapter 6, 7.2 *Prentice-Hall*, NJ 1993

[Ler96] X. Leroy GNU LinuxThreads
http://pauillac.inria.fr/~xleroy/linuxthreads/index.html

[Lin] The Linux documentation Project *http://www.ibiblio.org/mdw/index.html*

[LinP] Performance Measurement of Linux Kernel
http://www.cs.wisc.edu/~dusseau/ Classes/CS736/CS736-S02/warmup.ps

[Linuxnoy] The Linux Kernel Archives *http://www.kernel.org/*

[Man] Mandrake Linux *http://www.mandrakelinux.com/fr/*

[Mar03] C. Martin Deploiement et controle d'applications paralleles sur grappes de grandes tailles. *These de doctorat en informatique*, Institut National Polytechnique de Grenoble, France December 2003

[Mita] B. Mitchell Bandwidth and latency
http://compnetworking.about.com.library/weekly/aa021902a.html

[Mitb] B. Mitchell Ethernet tutorial
http://compnetworking.about.com.library/weekly/aa102900a.html

[MN95] J-F. Mehaut and R. Namyst. Marcel : Une bibliothèque de processus légers. *Technical report, laboratoire d'Informatique Fondamentale de Lille*, 1995.

[mpich] MPICH - A portable MPI Implementation
http://www-unix.mcs.anl.gov/mpi/mpich/

[MR01] Cyrille Martin and Olivier Richard. Parallal launcher for cluster of pc. *In World Scientific, editor, Proceedings ParCo 2001, pages 473-480*, 2001.

[MSSW94] C.May, E.Silha, R.Simpson, H.Warren The Power PC Architecture *Morgan Kaufmann Publishers* 1994

[Myr] *http://www.myri.com/*

[Nam01] R.Namyst Contribution à la conception de supports exécutifs multithreads performants *Thèse d'Habilitation à diriger les recherches Ecole Normale Supérieure de Lyon* 2001

[NBF98] B. Nichols, D. Buttlar and J.P. Farrell. Pthread Programming. *O'Reilly Press* 1998

[NM96] R. Namyst and J.-F. Méhaut. PM2 : Parallel multithreaded machine. A computing environment for distributed architectures. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, editors, *Parallel Computing : Stateof-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*,pages 279–285, Amsterdam, Elsevier,North-Holland. February 1996.

[NN94] N. Nupairoj and L.M. Ni. "Performance Evaluation of Some MPI Implementations on Workstation Clusters," *Proceedings of the 1994 Scalable Parallel Libraries Conference (SPLC94*), October 1994, pp. 98-105

[NN97] N. Nupairoj and L.M. Ni, "Performance Metrics and Measurement Techniques of Collective Communication Services," *First International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'97), San Antonio, TX*, February 1997, pp. 212-226.

[Open] OpenMP *http://www.openmp.org*

[Pal00] C. Pallier. Linux et le temps
http://www.pallier.org/ressources/timing_linux/timing.html 2000

[Pas99] M. Pasin Mouvement de données efficace pour la programmation parallèle irrégulière. *Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble*, France Novembre 1999

[Pig03] L. Pigeon. Conception d'une bibliothèque pour les opérations de communication collective pour le langage de haut niveau Athapascan. *Mémoire DEA d'Informatique: Système et Communication, Ecole doctorale mathématique et informatique, IMAG* Juin-2003

[PKB+91] M.L. Powell, Steve R. Kleiman, Steve Barton, Devang Shah, Dan Stein, and Mary Weeks. SunOS multi-thread architecture. *In Proceedings of the Winter 1991 USENIX Technical Conference and Exhibition*, pages 65-80, Dallas, TX, USA, January 1991.

[Pos] The ISO POSIX Working Group *htpp://open-std.org/jtc1/sc22/WG15/*

[Pos81] J. Postel. Internet Protocol. *Internet Engineering Task Force*, Septembre 1981.RFC 791

[Pran03] J. Pranevich The Wonderful World of Linux 2.6
http://www.kniggit.net/wwol26.html

[PT98] L. Prylli and B. Tourancheau. BIP : A new protocol designed for high performance networking on myrinet. *Lecture Notes in Computer Science*,1388 :472–80, 1998.

[Rab97]R.Rabenseifner. A new optimized MPI reduce algorithm.
http://www.hlrs.de/structure/support/parallel_computing/models/mpi/myreduce.html(19 97)

[RAM+01] B. Richard, P. Augerat, N. Maillard, S. Derr, S. Martin et C. Robert. I-Cluster: Reaching TOP500 performance using mainstream hardware. *Rapport Techinque HPL-2001-206, hplabs*, 2001.
http://www.hpl.hp.com/techreports/2001/HPL-2001-206.html

[Red] RedHat *http://www.redhat.com*

[RedG] *http://graal.ens-lyon.fr/~desprez/REDGRID/*

[Rev04] R. Revire Ordonnancement de grappe dynamique de tâche sur architecture de grande taille. Régulation par dégénération séquentielle et distribuée. *Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble*, France Septembre 2004

[RGR03] J.-L. Roch, T. Gautier, R. Revire. Athapascan: API for asynchronous parallel programming. *Technical Report RT-0276, INRIA Rhône-Alpes, project APACHE*, February 2003

[San03] A. K. Santhanam Toward Linux 2.6 A look into the workings of the next new kernel. *http://www-106.ibm.com/developerworks/linux/library/l-inside.html*

[SCP97] L.P. Santos, V. Castro and A. Proença. "Evaluation of the Communication Performance on a Parallel Processing System" *4th European PVM-MPI Users Group Meeting --- Recent Advances in Parallel Virtual Machine and Message Passing Interface; Springer Lecture Notes in Computer Science 1332; ISBN 3-540-63697-8; Krakow, Poland*, 1997

[SG91] A. Silberschatz, P.B. Galvin. Operating Systems Concepts. *Addison Wesley Chapter 4 p.96*, 1991

[She96] J. Shemitz. Using RDTSC for benchmarking code on Pentium computers *June 17, 1996 http://www.midnightbeach.com/jon/pubs/rdtsc.htm*

[Sin97a] P.K. Sinha. Distributed Operating Systems Concepts and Design. *IEEE Computer Scociety Press Chapter 8 Sec. 8.3 p.399*, 1997

[Sin97b] P.K. Sinha. Distributed Operating Systems Concepts and Design. *IEEE Computer Scociety Press Chapter 8 Sec. 8.3 p.406*, 1997

[SM99] B. R. de Supinski, J. May. Benchmarking Pthreads Performance. *Parallel and Distributed Processing Techniques and Applications PDPTA Vol 4*. Nevada 1999

[SOH96] M. Snir, S.W. Otto, S.H.Lederman, D.W. Walker and J. Dongarra *MPI: the complete reference, MIT Press, Cambridge*, MA, USA, 1996

[SR00] M.S. Schlansker, B.R. Rau EPIC: Explicitly Parallel Instruction Computing *Computer IEEE Vol 32 N° 2 p.37-45* February 2000

[Ste94] W. Richard Stevens. TCP/IP Illustrated, volume 1+2. *Addison Wesley*, 1994.

[Ste98] W. Richard Stevens. Unix Network Programming, volume 1+2. *Addison Wesley*, 1998.

[Str00] Bjarne Stroustrup. The C++ Programming Language. *Addison-Wesley 3 edition* February 15, 2000

[Sun] *http://www.sun.com/processors/*

[Tan87a] A.S. Tanenbaum. Architecture de l'ordinateur *Chapitre 2 p.18 InterEdition* 1987

[Tan87b] A.S. Tanenbaum. Architecture de l'ordinateur *Chapitre 6 p.337 InterEdition* 1987

[Tan87c] A.S. Tanenbaum. Architecture de l'ordinateur *Chapitre 5 p.257 InterEdition* 1987

[Tan92a] A.S. Tanenbaum. Modern Operating Systems *Prentice Hall* 1992

[Tan92b] A.S. Tanenbaum. Modern Operating Systems *Prentice Hall Chapter 12 Sec. 12.1 p.507-523* 1992

[Tan01] A. Tanenbaum. Modern Operating Systems, *2nd edition Prentice Hall 2001*

[Tan01a] A. Tanenbaum. Modern Operating Systems, *2nd edition Prentice Hall Case Study N°1* 2001

[Tom91] M. Tomassini. Programming with sockets. *C User Journal*, 9(9):39-48, September 1991

[Top500] Top 500 supercomputer sites *http://www.top500.org/*

[unix] http://www.bell-labs.com/history/unix/

[VFD00] S.S. Vadhiyar, G.E. Fagg and J.Dongarra. Automatically tuned collective communication. *In Proceedings of SuperComputing2000*, November 2000

[VFD01] S.S. Vadhiyar, G.E. Fagg and J.Dongarra. Performance modeling for self adapting collective communication for mpi. *In LACSI Symposium, Santa Fe, USA*, October 2001

[Vin98] S. Vinoski. New features for CORBA 3.0. *Communications of the ACM* 1998

[XH96] Z. Xu and K. Hwang. Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2. *IEEE Parallel & Distributed Technology*, Spring 1996, pp.9-23

Résumé
Les grappes de calcul sont constituées par l'interconnexion de stations de travail par un réseau plus ou moins performant. Elles rencontrent un large succès dans le domaine du calcul scientifique. De nombreux protocoles et interfaces de programmation ont été développés pour exploiter ces grappes tels que *Posix Threads, Marcel, Open MP, Socket, MPI, Madeleine, GM/Myrinet, Corba, etc*. Pourtant la programmation d'une application ou le portage d'un environnement de programmation parallèle sur ces grappes est un travail difficile du fait de la complexité et la variété des caractéristiques de ces architectures et des bibliothèques disponibles.
L'objectif de cette thèse est de définir et d'évaluer les performances d'INUKTITUT, un interface qui permet de porter efficacement et facilement sur des grappes des applications et des environnements de programmation parallèle de haut niveau comme *Athapascan*. INUKTITUT contient des fonctions pour multiprogrammation légère et pour les communications à base de message actif : le parallélisme intra-nœuds des processus, est géré à l'aide de processus légers communiquant par la mémoire partagée ; le parallélisme inter-nœuds est exploité par des communications basées sur des messages actifs. INUKTITUT est porté avec succès sur des systèmes aussi différents que *Linux, Unix, MacOS X, Windows* au dessus de *TCP/IP, Myrinet* ou *Corba*.
Ce mémoire présente l'architecture d'INUKTITUT et les résultats d'évaluation des performances. Les deux applications principales utilisant INUKTITUT sont : Athapascan, un environnement de programmation parallèle et les KaTools, des outils pour le déploiement efficace de grandes grappes.

Mots clés
interface portabilité de communication, message actif, multiprogrammation légère, grappes de machines SMP, évaluation de performance.

Abstract
The computing clusters are established by the interconnection of workstations by a more or less performance network. They have a wide success in the scientific computing area. Numerous protocols and interfaces of programming have been developed to exploit these clusters such as *Posix Threads, Marcel, Open MP, Socket, MPI, Madeleine, GM/Myrinet, Corba, etc.* Nevertheless the programming of an application or the carrying of an environment of parallel programming on these clusters is a difficult work because of the complexity and the variety of the characteristics of these architectures and the available libraries.
The objective of this thesis is to define and to evaluate the performances of INUKTITUT, an interface which allows carrying effectively and easily the applications and the environments of parallel programming high-level as Athapascan on theses clusters. INUKTITUT contains functions for multithreading and for the communications based on active messages: the intra-nodes parallelism of the processes is managed by means of threads communicating by shared memory; the inter-node parallelism is exploited by communications based on active messages. INUKTITUT is successfully carried on systems so different as Linux, Unix, MacOS X, Windows above TCP/IP, Myrinet or Corba used as data transport layers.
This thesis presents the architecture of INUKTITUT, the results of evaluation of the performances as well as two important applications using INUKTITUT: Athapascan, an environment of parallel programming and the KaTools, tools for the effective deployment on large clusters and grids.

Keywords
interface portability of communication, active message, multithreading, clusters of machines SMP, performance evaluation.