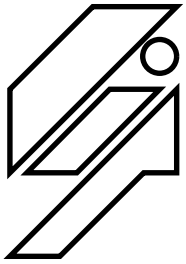
Utilisation de processus légers pour l'exécution de programmes a parallélisme de données : étude expérimentale.

Christian Perez



Laboratoire de l'Informatique du Parallélisme

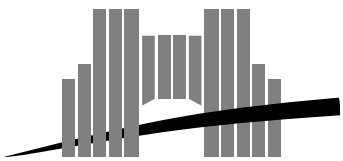
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Utilisation de processus légers pour l'exécution de programmes à parallélisme de données : étude expérimentale

Christian Perez

Avril 96

Research Report N^{O} 96-09



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Utilisation de processus légers pour l'exécution de programmes à parallélisme de données : étude expérimentale

Christian Perez

Avril 96

Abstract

This paper studies the use of threads to support the execution of data-parallel programs. The overhead induced by the multithreaded environment is experimentally studied : global synchronization, thread creation, communication, thread migration. We propose some simple criteria to determine the "right" size of threads with respect to the expected overhead. We use the PM^2 multithreaded environment which provides thread migration facilities.

Keywords: Data parallel languages, compilation, multithreaded environment, threads, experimental cost study.

Résumé

Ce rapport présente une étude de quelques aspects importants des processus légers pour leur utilisation à l'exécution de programmes à parallélisme de données. La surcharge de l'environnement d'exécution, le coût des synchronisations globales, de création de processus légers, le comportement des processus légers vis-à-vis des communications ainsi que le coût de la migration des processus légers sont étudiés. Nous en déduisons des critères simples pour la détermination de la "bonne" taille des processus légers en relation avec le surcoût engendré. L'environnement d'exécution choisi est PM^2 car il intègre la migration de processus légers.

Mots-clés: Langages à parallélisme de données, compilation, environnement d'exécution *multi-thread*, processus légers, étude expérimentale des coûts.

Utilisation de processus légers pour l'exécution de programmes à parallélisme de données : étude expérimentale

Christian Perez[†]

Avril 96

Table des matières

†. Financé par le projet CNRS-INRIA *ReMaP*.

1 Introduction

Dans la recherche de haute performance en calcul, le langage à parallélisme de données High Performance Fortran (HPF) [13] semble amené à jouer un rôle important. De nombreuses recherches se sont attaquées aux problèmes de trouver un ordonnancement et un placement statiques performants [9, 11]. Cependant, il semble intéressant d'insérer aussi des possibilités d'équilibrage de charge à HPF. En effet, les travaux de G. Utard et G. Cuvillier [27] ([10] pour une version plus détaillée) sur l'intégration d'un module d'équilibrage de charge au code produit par le compilateur C* [26] de l'Université du New Hampshire sont prometteurs. Comme C*, HPF est un langage à parallélisme de données et donc, suivant la *owner computes rule*, les calculs se font là où se trouvent les données. Par conséquent, équilibrer la charge revient à équilibrer les données. La notion de processeur virtuel dans HPF permet de considérer qu'équilibrer la charge d'un programme HPF revient à faire, comme dans le cas de C*, de l'équilibrage de processeurs virtuels. Il faut pour cela, bien sûr, disposer d'un nombre de processeurs virtuels suffisamment grand.

Le problème auquel ce rapport apporte des éléments de réponse est d'estimer les avantages et les inconvénients à l'utilisation de processus légers à l'exécution de programmes à parallélisme de données style HPF. En effet, nous voudrions déterminer si un code *multithread* est compétitif par rapport au parallélisme de processus classique. Nous nous attendons à avoir une surcharge due à la gestion des processus légers. Mais nous espérons que d'une part elle ne sera pas trop grande et que d'autre part cette surcharge pourra au moins être en partie masquée par un meilleur recouvrement des communications par les calculs. En effet, nous désirons introduire de l'équilibrage de charge dans HPF mais sans altérer sensiblement les performances de HPF pour du code régulier.

Unité de la migration Le premier problème est de savoir sous quelle forme migrer les données d'un processeur virtuel. En effet, nous pouvons migrer seulement les données, les données et le code (i.e. un processus) ou les données et la pile d'exécution (i.e. un processus léger ou *thread* [25]).

- La migration de processus est la plus facile à mettre en place. En effet, les compilateurs actuels comme Adaptor [4, 5] ou Portland Group HPF [22] génèrent un parallélisme de processus. En ajoutant un système de migration de processus comme MPVM [6] ou Cocheck [24], nous obtenons la capacité de migration. Malheureusement, cette approche n'est pas compatible avec la recherche de haute performance pour deux raisons principales. Tout d'abord, la migration d'un processus est une opération extrêmement coûteuse car la taille d'un processus se mesure en millions d'octets. Ensuite, le parallélisme de processus n'est guère performant du fait du coût élevé des changements de contexte dans un système Unix. Par conséquent, le nombre de processus présents sur un même processeur est fortement limité et donc le découpage du domaine est fait à un très gros grain.

- La migration de données est parfaite du point de vue finesse de l'équilibrage de charge. En effet, il est possible de calculer un équilibrage de charge dont l'ordre de grandeur de la précision est la donnée. Malheureusement, le prix à payer est la perte de la simplicité des fonctions de parcours du domaine. De surcroît, le domaine de recherche est grand. La recherche dynamique d'une bonne solution dans cet espace demande donc un temps non négligeable.

- Les processus légers se placent entre les deux. Par rapport aux processus, ils ont l'avantage d'être beaucoup moins coûteux à migrer car seules les données et la pile migrent. De plus,

le temps de changement de contexte est de plusieurs ordres de grandeur plus petit. Ces caractéristiques font que leur granularité est plus petite que celle des processus mais n'est pas aussi fine que celle de l'approche donnée.

Une étude détaillée sur des avantages et des inconvénients de ces trois approches peut être trouvée dans [6].

Régularité et granularité Notre but est d'introduire de la régularité au niveau du traitement des données. Cela revient à augmenter la granularité de l'application. D'une granularité au niveau de la donnée, nous voulons passer à une granularité supérieure. Nos motivations sont les suivantes.

- La régularité des données est primordiale pour avoir un parcours efficace des données. En effet, elle permet de limiter le coût du contrôle de flot qui peut vite devenir important. C'est en partie pourquoi la précision des découpes est rarement au niveau de la donnée. C'est le cas des bissections récursives, des partitions rectilinéaires [21], de la partition N4 [15], etc.

- Une granularité au niveau de la donnée apparaît comme inutile car la décision de rééquilibrer est prise lorsque le déséquilibre est suffisamment grand pour que le gain espéré dépasse le coût des communications. Les temps mis en jeu sont en général beaucoup plus grands que celui du calcul sur une donnée.

- Les communications étant encore coûteuses, il apparaît plus profitable de réduire les communications que de chercher un équilibrage parfait des données au détriment d'un coût de communication exorbitant. La régularité des blocs est un moyen qui permet de calculer assez rapidement et précisément le coût en communication d'un bloc de données. Elle permet donc de choisir un placement qui minimise le temps de calcul plus le temps de communication.

- En restreignant l'espace de recherche, la probabilité de trouver un bon placement en un temps réduit augmente.

Il s'avère donc qu'une granularité supérieure à la donnée est intéressante. Cependant, la gestion à la main des blocs élémentaires de données (i.e. une partie contiguë d'un tableau) apparaît assez lourde. En effet, il faut mettre en place un mécanisme de gestion des dépendances entre blocs sur le même processus. Or, un sur-ensemble des fonctionnalités utiles est présent dans les processus légers. Nous choisissons donc d'encapsuler un bloc de données régulier dans un thread.

Compilation HPF et processus légers Nous choisissons les threads pour les raisons suivantes : les environnements de processus légers arrivent à maturation, grâce notamment à la norme POSIX [16], ce qui leur confère une stabilité suffisante pour être utilisés sereinement [3, 7, 14, 17, 20]; la migration de processus légers apparaît un niveau adéquat pour notre approche; l'efficacité de certains environnements devient satisfaisante. Enfin, cette solution a l'avantage d'une plus grande simplicité d'implémentation pour une performance qui devrait être bonne. Nous avons choisi l'environnement *multithread* PM^2 [20] car il remplit ces critères en plus celui d'avoir ses sources disponibles.

Les threads commencent à être utilisés pour la compilation de programme HPF mais seulement dans le but d'avoir un meilleur recouvrement calcul communication. Dans [1], les threads sont utilisés afin d'avoir un thread de réception de messages qui, au travers d'un thread de gestion, donne du travail au thread de calcul quand toutes les données relatives à un bloc de travail sont

présentes. Dans [23] l'approche qui consiste à avoir un parallélisme de threads à l'intérieur d'un processus est étudiée. Cependant, ces deux papiers ne mènent pas une étude précise sur les différents coûts des threads intervenant à l'exécution de leurs programmes.

Plan du travail Le but de ce papier est d'étudier les différents aspects des threads. Ainsi, dans la section suivante, nous estimerons la surcharge générée par l'utilisation de threads dans le cas d'un programme de calcul simple. Nous étudierons dans la section 3 la question de synchronisation globale de threads et de temps de création d'un thread. Ensuite, dans la section 4, nous introduirons les communications entre threads. Nous aborderons la question de la migration de threads dans la section 5.

2 Parallélisme de processus lourds et légers

Le but de cette section est d'étudier le comportement d'un programme *multithread* sans communication. Nous regarderons plus précisément la surcharge engendrée par l'introduction des threads par rapport au même programme séquentiel et parallélisé par Adaptor [4, 5]. En effet, nous voudrions essayer de séparer la surcharge due à la parallélisation de la surcharge due aux threads.

Nous comparerons d'abord différentes approches afin d'estimer le coût de la parallélisation. Ensuite, nous considérons le parallélisme de processus pour nous donner un point de référence. Enfin, nous nous attaquerons au parallélisme de processus légers.

Nous considérons deux classes de machines : des stations Sparc ELC et des stations Sparc 5 afin d'avoir une idée de l'évolution des résultats sur des machines plus ou moins puissantes. Les Sparc 5 sont environ 4 fois plus rapides que les Sparc ELC.

Programme de test

Le programme de base pour cette section est un programme de calcul sans communication dont le cœur calculatoire est constitué de deux nids de boucles de profondeur 3. Afin d'éviter les débordements de calcul et les optimisations trop pointues des compilateurs (comme la suppression de code inutile), le premier nid de boucle calcule $A(I,J) = A(I,J) * (T+1) - A(I,J) * T$ sur un tableau de taille $400 * 400$ de flottants en double précision pendant 800 itérations, décrites par la variable T. Pour les mêmes raisons, le deuxième calcule $B(I,J) = A(I,J) * T * 3.5 - B(I,J) * 0.9$ sur un domaine de $200 * 200$ pendant 2400 itérations. Les valeurs 3.5 et 0.9 ont été choisies expérimentalement pour éviter les débordements. Notre but est seulement d'occuper le processeur.

Choix du temps mesuré

Nous avons choisi de mesurer les temps réels pour pouvoir tenir compte des temps de communication et du temps passé par nos programmes dans le noyau du système Unix. Par conséquent, tous les temps reportés sont légèrement perturbés par le système: processus systèmes parasites, changements de contexte entre processus, etc.

2.1 Surcharge générée par l'environnement de processus légers

L'expérience : description et mesures

Afin de mesurer les coûts induits par chaque exécution, nous mesurons les performances sur une machine inoccupée avec un processus contenant au plus un processus léger de calcul.

Les expériences ci-dessous sont exécutées sous Solaris 2.4. Les mesures n'intègrent pas les coûts d'initialisation des programmes. La version Adaptor est la version HPF de la version Fortran 77. Les versions sans thread et avec threads utilisent la même routine de calcul qui est la version C du code Fortran 77 produit par Adaptor. La différence entre ces 2 versions est que la deuxième est exécutée dans l'environnement PM^2. Adaptor transforme le tableau bidimensionnel en un tableau unidimensionnel. Par conséquent, les accès au tableau `A(i,j)` sont de la forme `*(A+j*S+i+C)` avec `A` la base du tableau, `S` le pas et `C` le décalage à l'origine. Seul la version Fortran 77 fait exception et utilise un tableau bidimensionnel.

Langage	Temps sur ELC (s)	Temps sur Sparc 5 (s)
Fortran 77	271.0	77.3
Adaptor	267.6	75.2
C sans thread	268.0	76.6
C + 1 thread (sans preemption)	279.05	77.60
C + 1 thread (avec preemption courte)	279.08	77.61
C + 1 thread (avec preemption longue)	279.14	77.64

TAB. 1 - *Temps d'un noyau de calcul pour différents compilateurs et différents environnements d'exécution.*

Analyse des résultats

Comme le montre le tableau 1, l'emploi d'un thread, avec ou sans préemption, induit une surcharge de 4% sur ELC et de 2% sur Sparc 5. Cependant, les effets de la préemption sont très faibles. Les auteurs de PM^2 estiment le coût de la préemption à moins de 1% ce qui est en accord avec nos résultats.

La version Adaptor est 1.5% sur ELC et 3% sur Sparc 5 plus rapide que la version Fortran 77 avec le même compilateur et les mêmes options de compilation. La cause doit être recherchée dans le fait qu'Adaptor linéarise les tableaux.

En conclusion, un code Adaptor *multithread* sur Sparc 5 devrait être 2% plus lent que le même code non *multithread* mais aussi rapide que du code Fortran 77. Le surcoût semble acceptable. Il trouve ses sources d'une part dans le mécanisme de préemption et d'autre part dans les perturbations du thread de communication. En effet, PM^2 emploie des appels de procédures distantes légères (LRPC) pour communiquer et utilise un thread comme *handler* pour activer l'appel de procédures distantes. Ce thread teste dès qu'il peut si des messages sont arrivés et consomme donc un peu de ressource processeur.

2.2 Parallélisme de processus

L'expérience : description et mesures

Les threads sont par définition plus légers que les processus. Afin d'avoir un repère de coût, nous allons mesurer maintenant le coût du parallélisme de processus pour un même programme de calcul. Le programme de test est celui utilisé précédemment avec une distribution des données de type `(BLOCK,BLOCK)`.

Dans cette expérience, nous faisons varier le nombre de processus sur une machine. Dans la version avec threads, chaque processus contient au plus un thread de calcul et utilise une préemption

courte. Les résultats sont résumés dans le tableau 2. Nous n'avons pas les chiffres pour plus de 16 processus car une exécution à 32 processus n'a pas été possible par manque de mémoire. Aucune exécution n'a généré d'échange entre la mémoire et la zone de swap sur le disque dur.

Nombre de processus	Temps sur ELC (s)		Temps sur Sparc 5 (s)	
	Adaptor	C + PM^2	Adaptor	C + PM^2
1	267.6	282.0	75.2	78.2
2	270.9	283.7	72.3	78.6
4	270.8	288.1	71.3	79.1
8	272.6	303.0	70.8	82.7
16	268.3	388.6	71.7	95.5

TAB. 2 - *Parallélisme de processus sur une machine.*

Analyse des résultats

Le tableau 2 montre un phénomène a priori surprenant : le temps total diminue quand le nombre de processus augmente pour les processus Adaptor alors que le temps augmente pour les processus PM^2. Comme nous le verrons par la suite, le temps des processus Adaptor baisse à cause des effets de cache. Pour l'instant, intéressons-nous au cas des processus PM^2.

La raison pour laquelle le temps augmente pour les processus PM^2 vient de la manière dont est implémentée la préemption [18, 19]. Toutes les 20 ms par défaut, le processus reçoit un signal qui arrête l'exécution du thread courant et donne la main à l'ordonnanceur : c'est la base du mécanisme de préemption. Or, recevoir un signal signifie faire un appel au noyau [2]. Comme l'ordonnanceur Unix est appelé à chaque retour du noyau, le processus PM^2 perd la main. Lorsque le nombre de processus ou la fréquence des signaux augmentent, le processeur passe de plus en plus de temps dans le noyau à effectuer des changements de contexte de processus du fait de l'augmentation du nombre de signaux. Par conséquent le calcul proprement dit avance moins vite : le temps réel total augmente.

Nous avons mené une expérience dans laquelle nous avons mesuré le temps réel et le nombre de changements de contexte en fonction de la tranche de temps pour une même somme de travail. Les résultats sont reportés à la figure 1. Nous pouvons y voir très clairement qu'une surcharge de signaux dégrade fortement le système.

Le tableau 2 montre aussi l'influence de la puissance des machines sur la préemption. Ainsi la différence entre 1 et 16 processus est de 37.8% sur ELC alors qu'elle n'est que de 22.1% sur une Sparc 5. Proportionnellement, une exécution sur Sparc 5 passe moins de temps dans le noyau qu'une exécution sur ELC.

Il y a donc un choix à faire entre la fréquence de préemption des threads et le nombre de processus par processeur si nous voulons garder un mécanisme de préemption. Cependant, l'intérêt d'un mécanisme de préemption dans le cadre de l'exécution de programmes data-parallèles n'est pas clair. En effet, comme l'exécution est fortement couplée, tous les threads sont obligés de progresser ensemble. Le changement de contexte se produit naturellement lorsqu'un thread atteint l'une des nombreuses barrières de synchronisation.

Il peut être quand même intéressant de garder la préemption afin de ne pas pénaliser le thread de réception des communications et donc les communications : l'ordonnanceur de PM^2 garantit qu'un thread aura bien la main de temps en temps grâce à un système de priorités entre threads. Il est ainsi possible de régler, via les priorités, le délai de prise en compte d'une communication. Nous

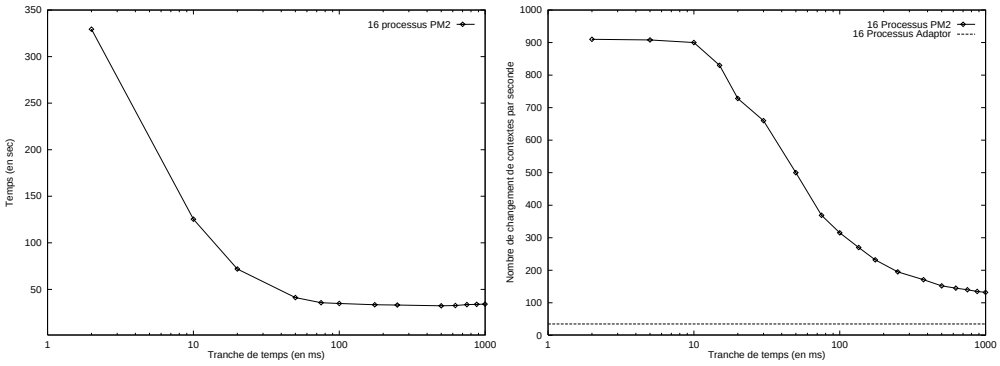


FIG. 1 - *Temps d'exécution et nombre de changements de contexte en fonction de la tranche de temps pour un programme qui a une somme de travail fixe et générant 16 processus PM^2. A titre de référence, un programme Adaptor à 16 processus génère environ 35 changements de contexte par seconde.*

garderons donc par la suite la préemption d'autant plus que son coût est petit pour un processus par processeur et que nous ne considérons justement qu'un processus par processeur.

2.3 Parallélisme de threads

Nous allons maintenant nous pencher sur le cas du parallélisme de threads dans la perspective d'une exécution d'un programme data-parallèle.

2.3.1 Première expérience : programme de test et threads

Cette expérience va de pair avec celle de la section précédente. Nous pourrons ainsi comparer la surcharge du parallélisme de threads par rapport au parallélisme de processus.

Description et mesure

Nous reprenons le même programme que précédemment, mais au lieu de varier le nombre de processus nous varions le nombre de threads à l'intérieur d'un processus unique. Les résultats ont été regroupés dans le tableau 3. La courbe pour différentes stratégies de partition et pour une plus grand plage de threads est présentée en figure 2.1.

Analyse des résultats

Tout d'abord, nous remarquons que le nombre de threads possible est évidement beaucoup plus grand que le nombre de processus. Nos expériences ont utilisé jusqu'à 2048 threads. Un parallélisme de threads consomme donc beaucoup moins de mémoire qu'un parallélisme de processus. Ensuite, comme le laissent supposer les temps sur la Sparc 5, la surcharge due au parallélisme de threads est petit. Nous obtenons environ le même temps pour 1 thread que pour 64 threads. Enfin, comme le montre la figure 2.1, la chute de la courbe dépend très peu de la forme du domaine affecté à chaque thread. Cette courbe est obtenue en faisant varier le nombre de threads pour différentes partitions. La quantité de données n étant constante, un nombre t de threads implique une quantité

Nombre de threads	Temps sur ELC (s)	Temps sur Sparc 5 (s)
1	280.9	78.2
2	280.5	78.2
4	280.7	77.9
8	281.4	77.9
16	273.6	78.0
32	239.6	78.0
64	215.0	78.6

TAB. 3 - *Parallélisme de threads à l'intérieur d'un seul processus Unix.*

de données par thread égale à n/t. La courbe dépend donc essentiellement de la quantité de données par thread. C'est donc bien le signe d'un effet de cache.

Les effets de cache nous gênent pour exploiter les résultats. Nous allons donc faire une expérience différente avec deux objectifs. Premièrement, analyser le phénomène du cache pour, deuxièmement, mesurer plus précisément le coût du parallélisme de threads.

2.3.2 Seconde expérience : coût de gestion des threads

Nous avons mis en évidence dans l'expérience précédente les effets du cache. Nous allons faire une expérience afin de mieux les maîtriser pour estimer le surcoût lié aux threads. Cette expérience ne sera menée que sur une station Sparc 5 car elle possède un cache pour les données et un cache pour les instructions contrairement à la station ELC qui n'a qu'un seul cache. Nous n'avons pas ainsi à estimer la quantité d'instructions qui se trouve dans le cache de la station ELC.

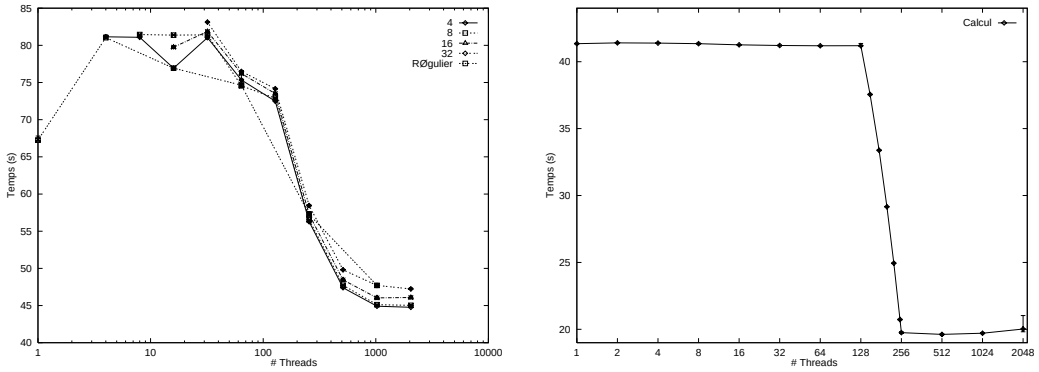
L'expérience : description et mesures

Notre programme consiste en 1024 itérations sur un tableau à une dimension de 262144 flottants en double précision (64 bits). Un bloc contigu de ce tableau est attribué à chaque thread, qui calcule ensuite dessus. Les résultats sont présentés à la figure 2.2.

Analyse des résultats

Notre principale hypothèse pour expliquer la forme de la courbe 2.2 est que le temps en fonction du nombre de threads est une courbe affine. En effet, la somme de travail reste constante mais la surcharge liée aux threads augmente. La surcharge comprend la partie contrôle de flot du programme qui est en partie répliquée sur tous les threads (le parcours de la boucle séquentielle mais pas le parcours de la boucle parallèle qui est bien distribué) ainsi que les changements de contexte. Une telle surcharge est donc proportionnelle au nombre de threads. Cette hypothèse se trouve vérifiée par les parties gauche et droite de la courbe qui sont des morceaux de droites.

La partie centrale de la courbe est due au cache des données du processeur. En effet, plus le nombre de threads augmente, plus la taille des données par thread diminue et donc plus la proportion de réutilisation du cache (*cache hit*) augmente. C'est la phase de transition durant laquelle la proportion de réutilisation des données présentes dans le cache passe de 0% à 100%.



(2.1) (2.2)

FIG. 2 - *Partition de données et effets de cache : 2.1 - Mise en évidence d'un phénomène singulier pour différentes partitions d'une quantité de données totale constante et d'un nombre variables de threads. 2.2 - Nombre de threads et effets de cache pour une quantité de données totale constante.*

Modélisation de l'expérience

Nous allons tenter de modéliser l'expérience afin d'en extraire des informations utiles comme le coût par thread.

Le temps est donc une fonction affine du nombre de threads t et de la taille des données n. D'où,

$$temps(t,n) = (\delta + I * c_1)t + I * n * c_2 \tag{1}$$

avec comme constante δ qui est le coût d'initialisation du thread, I qui représente le nombre d'itérations, c_1 qui modélise la surcharge par thread et par itération (i.e. les changements de contexte) et c_2 pour le temps de calcul sur une donnée. Nous devons distinguer en fait deux cas pour c_2. Soit le flottant est dans le cache processeur (*cache hit*) et nous noterons c_2 soit il est en mémoire (*cache miss*) et nous noterons alors c_2'. La partie gauche de la courbe a alors pour équation :

$$temps(t,n) = (\delta + I * c_1)t + I * n * c_2' \tag{2}$$

et la partie droite de la courbe a pour équation :

$$temps(t,n) = (\delta + I * c_1)t + I * n * c_2 \tag{3}$$

La partie centrale a besoin d'une hypothèse supplémentaire. Nous supposons en effet que, durant cette portion, le proportion de cache réutilisable est égale à

$$\text{Réutilisation du cache} = \left(\frac{x_B}{x_A} - \frac{n/t}{n_c}\right) \tag{4}$$

où $[x_A, x_B]$ représente le segment d'abscisse de la partie centrale (pour nous $x_A = 128$ et $x_B = 256$), n_c est la taille du cache et n/t est la quantité de données affectée à un thread. En effet, il faut que le cache des données soit utilisé à 0% au début de ce segment de droite et à 100% à la fin. L'équation du segment de droite central est alors :

$$temps(t,n,n_c) = (\delta + I * c_1 + I * l * c_2 + I * l' * c_2')t \tag{5}$$

avec l le nombre de données présentes dans le cache processeur et l' le nombre de données non présentes dans le cache processeur. Les variables l, l', n_c, n et t sont liées par les deux équations suivantes :

$$l = \%\text{ de réutilisation du cache} * \text{taille du cache} = \left(\frac{x_B}{x_A} - \frac{n/t}{n_c}\right) * n_c \quad (6)$$

et

$$l + l' = n/t \quad (7)$$

L'équation 5 se réécrit grâce aux équations 6 et 7 en

$$temps(t, n, n_c) = [\delta + I * c_1 - 2I * n_c * (c'_2 - c_2)]t + I * n * (2c'_2 - c_2) \quad (8)$$

L'application numérique à partir des résultats de la figure 2.2 nous donne :

$\delta + I * c_1$	=	178	μs	soit $c_1 < 174$ ns
$I * c_2$	=	74.3	μs	soit $c_2 = 72.6$ ns
$I * c'_2$	=	157.1	μs	soit $c'_2 = 153.4$ ns
n_c	=	1024	flottants	soit $n_c = 8$ ko

La différence entre les mesures expérimentales et notre modèle est en tout point strictement inférieur à 2% sauf pour 2048 threads ou l'erreur est de 3%. De plus, la taille du cache données sur les stations Sun Sparc 5 est bien de 8 ko [8]. Ces résultats valident ainsi nos hypothèses.

Retour à la première expérience

Nous pouvons maintenant comprendre pourquoi le phénomène apparaît plus vite sur la station ELC. En effet, son cache est de 64 ko. Le point de transition se situe donc après 9 threads. En effet, le domaine a une taille de $400 * 400$ flottants (soit $400 * 400 * 8$ octets). De plus, le point de transition se situe dès que les données affectées à un thread représentent deux fois la taille du thread. Or $400*400*8/(64*1024*2) = 9.77$ arrondi par défaut à 9. Nous n'observons pas d'effet de cache pour 8 threads. Par contre, nous l'observons pour 16 threads. Nos résultats ne contredisent donc pas la première expérience.

2.4 Discussion

Le parallélisme de processus se comporte somme toute assez bien. Cependant, nous nous sommes placés dans de bonnes conditions en ne générant pas de communications entre processus (voir la section 4 pour les communications). Pourtant, il souffre d'une très forte limitation : le nombre de processus. En effet, la duplication de code consomme trop de mémoire.

Le parallélisme de threads offre une plus grande extensibilité car seule la pile est dupliquée. L'utilisation de threads dans un programme de calcul intensif paraît avoir un coût très raisonnable. En effet, dans nos expériences, il nous en coûtait 178 μs par thread pour 1024 itérations. En supposant que le coût de lancement soit nul ($\delta = 0$), ce qui n'est pas le cas, nous obtenons un coût de 174 ns par thread. C'est à dire le temps de calcul de moins de 3 flottants dans le cache ou 1.1 flottants non présents dans le cache de notre exemple.

Grâce à ces estimations, nous pouvons donner deux critères pour le choix de la taille des threads (et donc leur nombre) en posant le système d'équation suivant :

$$n/t < n_c \quad (a)$$

$$\frac{c_1}{c_2 * n/t} < s \quad (b)$$

L'équation (a) exprime que la taille des données affectée à un thread doit être inférieure à la taille du cache afin de maximiser l'utilisation du cache. L'équation (b) propose de borner le surcoût des threads par itération. La quantité s représente la proportion maximale de temps qu'un thread peut perdre en surcoût par rapport au calcul qu'il effectue. Homogénéisons le système :

$$n/t < n_c \quad (a)$$

$$n/t > \frac{c_1}{c_2 * s} \quad (b')$$

Le défaut du système d'équation (a,b') est qu'il n'a pas toujours de solution. En effet, il ne possède de solution que si le surcoût $s > \frac{c_1}{c_2 * n_c}$. Par exemple, pour l'expérience précédente, le système a une solution si $s > 0.23\%$. Nous pouvons donc obtenir des solutions pour une surcharge tout à fait raisonnable dans notre exemple.

Nous remarquons aussi qu'il ne prend pas en compte la découpe du domaine. Il faut donc contraindre le système par des équations spécifiant la forme du domaine. Par exemple, pour un domaine à deux dimensions, la découpe peut être `(BLOCK,BLOCK)` ou `(BLOCK,*)`.

2.5 Conclusion

D'abord, nous avons vu les détails de l'implémentation de PM^2 qui en font un mauvais candidat pour le parallélisme de processus. Nous avons aussi constaté que le surcoût lié aux threads de calcul sans communication est faible. Nous avons revu l'importance du cache. Ces considérations nous ont mené à formuler deux critères pour déterminer une "bonne" taille des threads pour un problème donné.

Avant d'envisager d'utiliser les processus légers pour l'exécution de programme à parallélisme de données, il nous reste à déterminer le coût d'une synchronisation globale entre n threads d'un même processus, opération très courante en data-parallélisme. Ce problème est l'objet de la prochaine section.

3 Synchronisations globales de processus légers

3.1 Le problème

Nous allons maintenant tâcher d'estimer le coût de synchronisation globale de n threads. En effet, une exécution d'un programme data-parallèle génère un grand nombre de synchronisations. C'est donc une forte contrainte pour le nombre de threads par processus. La surcharge générée par la synchronisation des threads devra rester dans des proportions acceptables.

Notre problème est de synchroniser n threads. Nous distinguons deux niveaux de synchronisation : une synchronisation interne à un processus et une synchronisation de threads sur des processus différents.

La synchronisation interne est un problème de mémoire partagée. Elle est réalisée, de façon classique, en utilisant un compteur du nombre de threads qui ont appelé la fonction de synchronisation. L'accès au compteur est en exclusion mutuelle. Le dernier thread appelant la fonction réveille tous les threads endormis sur la barrière.

Si la synchronisation met en jeu plusieurs processus, nous sommes face à un problème de mémoire distribuée. Une fois la synchronisation interne réalisée, chaque processus participe à la syn-

chronisation des processus. C'est un problème très étudié et il existe quantité de protocoles plus ou moins performants [25].

3.2 L'expérience : synchronisation intra-processus

Nous nous intéressons qu'au cas de synchronisations entre threads à l'intérieur d'un unique processus. En effet, la synchronisation de threads sur des processus distincts revient à faire des synchronisations internes à chaque processus, puis des synchronisations de processus classique. Notre problème est d'estimer le coût d'une synchronisation interne qui, contrairement aux synchronisations de processus, est peu connu.

Description de l'expérience et mesures

L'expérience consiste donc en un programme qui effectue 1024 synchronisations entre tous les threads à l'intérieur d'un unique processus. Les effets de cache éventuels ont été supprimés. Le temps du programme en fonction du nombre de threads est présenté en figure 3. Il n'y a pas de mesures pour 2048 threads pour la station ELC car la station ne disposait pas de suffisamment de mémoire.

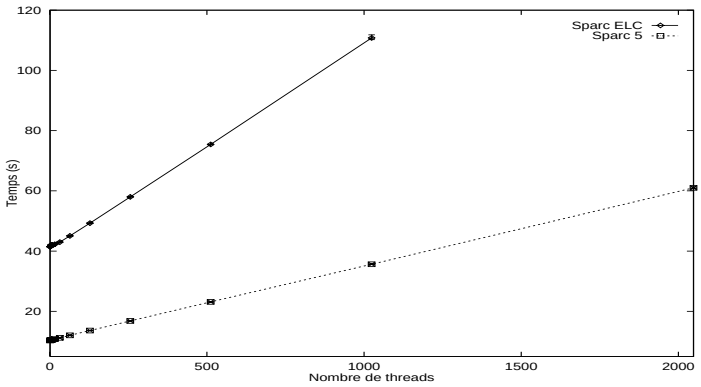


FIG. 3 - *Influence du nombre de threads sur la synchronisation globale.*

Analyse des résultats

La figure 3 nous apprend que le coût d'une synchronisation semble être proportionnel au nombre de threads. Essayons d'analyser plus finement ce qui se passe. Notre modèle d'exécution pour un thread est qu'il passe un certain temps à calculer puis qu'il se synchronise avec les autres threads. Nous supposons que le temps de calcul est inférieur au temps de préemption. Nous ne considérons donc pas de changements de contexte pendant les calculs. En fait, nous pouvons négliger ce coût car il représente moins de 1 μs toutes les 20 ms soit moins d'un millième du temps d'exécution (voir section 1).

Soit le coût d'une synchronisation en $\Omega(T)$ où T est le nombre de threads. En effet, une synchronisation entre T threads impliquent que $T-1$ threads se bloquent et qu'un thread, le dernier à atteindre la synchronisation, libère les $T-1$ threads bloqués. La libération de $T-1$ threads dans PM^2 consiste en une boucle sur les $T-1$ threads qui englobe un appel à la fonction de déblocage d'un thread. Son coût est donc en $\Omega(T)$. Par conséquent, le coût d'une synchronisation entre T threads est de $(T-1) * ($coût d'un changement de contexte $+$ coût pour débloquer un thread$)$

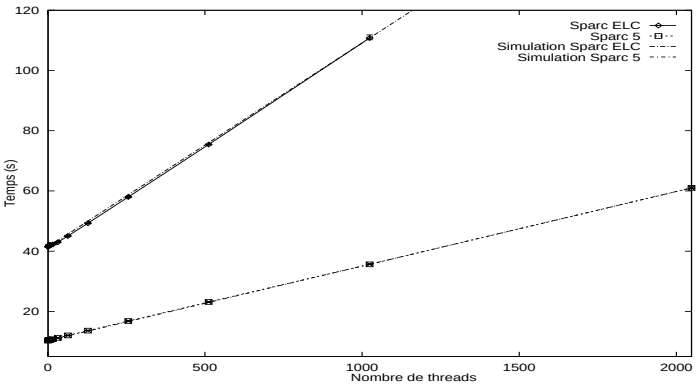


FIG. 4 - *Courbe expérimentale et de la modélisation de l'expérience.*

Les courbes expérimentales de la figure 4 constituent des droites affines de pente respective 0.025 et 0.068. La différence entre les courbes expérimentales et les courbes issues de notre modélisation diffère de moins de 1.5% en tout point. Elle est même fréquemment inférieure à 1%.

Pour la Sparc 5, le surcoût par thread comprenant les changements de contexte et les synchronisations est donc d'environ 25 ms par thread pour 1024 itérations soit environ 24 μs par thread et par itération. Nous avons donc là notre troisième contrainte pour décider de la taille des threads. En effet, si nous désirons maintenir le surcoût lié aux synchronisations en deçà d'un certain pourcentage, nous obtenons par une règle de trois le nombre de threads par processus et donc la taille des données d'un thread.

3.3 Discussion : synchronisation ou création de threads

Les threads ont la réputation d'avoir un coût très faible. C'est probablement pourquoi il a été pensé que les threads étaient un bon moyen pour gérer les différences de parallélisme d'un même code. Durant les parties séquentielles, il n'existe qu'un seul thread qui peut créer d'autres threads pour les parties parallèles du code. Ainsi, les problèmes de synchronisations des threads pour le code séquentiel sont résolus.

Environnement de threads	Temps moyen de création d'un thread
Sans préallocation de mémoire	
Threads Marcel (PM^2)	900 μs
Threads Solaris 2.4	800 μs
Avec préallocation de mémoire	
Threads Marcel (PM^2)	60 μs
Threads Solaris 2.4	500 μs

TAB. 4 - *Temps moyen de création d'un thread ayant une pile de 15 Ko sur une station Sparc 5 avec Solaris 2.4. Marcel est le noyau de gestion de threads sur lequel s'appuie PM^2. Nous avons utilisé le fait que Marcel garde la mémoire allouée à un thread à la fin de celui-ci pour simuler la préallocation de mémoire.*

Nous avons donc mesuré le temps moyen de création d'un nombre élevé de threads sans destruction de threads. Nous avons vérifié que les threads créés ne s'exécutaient pas durant les mesures.

Malheureusement, comme le montre le tableau 4, le temps de création d'un thread est loin d'être négligeable. Il est intéressant de noter que le fait de créer un thread en ayant préalloué la mémoire permet de gagner 300 μs soit 37% avec les threads Solaris. Malheureusement, PM^2 ne permet pas de préallocation mémoire. Nous nous contenterons de signaler que l'allocation de mémoire est une opération très coûteuse.

Cependant à la mort d'un thread, PM^2 garde la mémoire allouée pour la création de ce thread lorsque la taille de la mémoire est standard. Dans ce cas, la création d'un nouveau thread n'est plus que de 60 μs. La simulation de cette fonctionnalité avec des threads Solaris 2.4 permet de voir que la recréation d'un thread PM2 est beaucoup moins coûteuse que la création d'un thread Solaris, même avec préallocation de mémoire.

Le temps de création d'un thread est donc environ $900/24 \approx 40$ fois plus lent qu'une synchronisation (ou $60/24 \approx 3$ si nous considérons une recréation de thread). Cependant, il est probable que le coût de synchronisation peut être diminué en changeant de méthode de gestion des threads. En effet, le terme dominant de la synchronisation globale en $\Omega(T)$ est la fonction de réveil d'un thread. Cependant, si lorsqu'un thread s'endort, il se place dans une file spéciale tout en préparant son état prêt à être exécuté, le dernier thread n'a plus qu'à concaténer cette file à la file des threads actifs. Cette opération peut être réalisée à un coût très faible. Le coût de la synchronisation resterait en $\Omega(T)$ car chaque thread doit quand même s'endormir, mais la constante serait sensiblement plus petite (nous pensons au moins de moitié). Et donc, le coût d'une synchronisation par thread diminuerait presque d'autant. Bref, le ratio du temps de création d'un thread sur le temps de synchronisation a plus de chance d'augmenter que de diminuer.

Il n'est pas donc évident de savoir s'il vaut mieux créer des threads au fur et à mesure des besoins ou synchroniser un ensemble de threads. Dans le premier cas, nous avons la liberté de créer exactement le nombre de threads nécessaires mais pour un coût plus élevé. Dans le deuxième cas, nous disposons d'un ensemble de threads nécessitant des synchronisations ainsi qu'une programmation plus compliquée due notamment aux tests pour savoir qui doit travailler. Mais la deuxième solution a certainement un coût (très) inférieur à la première.

3.4 Conclusion

Le coût d'une synchronisation est donc proportionnel au nombre de threads participants. Le coût d'une synchronisation par thread sur une Sparc 5 est de l'ordre de 24 μs (à comparer avec le temps de création d'un thread et le temps de latence des réseaux qui s'expriment en centaine de microsecondes). Ce coût est raisonnable et permet donc d'utiliser un parallélisme de threads important.

Nous avons vu qu'une synchronisation est beaucoup moins coûteuse qu'une création de threads. Sans préallocation de mémoire, le rapport est d'environ 40.

4 Gestion des communications

Maintenant que nous avons une idée assez précise sur le comportement des threads, nous allons nous intéresser aux communications. Nous avons trois niveaux de mémoire et donc trois types d'accès mémoire à gérer :

1. les accès d'un thread à sa mémoire propre;
2. les accès d'un thread à la mémoire d'un autre thread du même processus;

3. et les accès d'un thread à la mémoire d'un autre thread d'un autre processus.

Le premier point est bien connu et l'introduction de threads n'apporte rien de nouveau.

Le deuxième point est un problème de mémoire partagée. En effet, nous voulons qu'un thread puisse directement aller lire dans la mémoire d'un autre thread qui est dans le même processus, afin de ne pas avoir de recopie mémoire inutile et coûteuse.

Le troisième point est le seul à générer des communications à travers le réseau. Nous pouvons soit considérer des communications entre threads ou bien agréger les communications et ne considérer que des communications entre processus.

Nous allons détailler les deux derniers points dans les sous-sections suivantes. Nous commencerons d'abord par les communications intra-processus puis par les communications inter-processus.

4.1 Description de notre expérience

Notre programme de test est très simple. Il s'agit de 800 itérations sur un tableau de flottants en double précision de taille $400 * 400$. En chaque point où cela a un sens, le programme calcule `A(i,j)` en fonction de `A(i,j)` et de `A(i,j+1)`.

Le calcul d'une ligne nécessite la connaissance de la ligne du dessous ce qui nous génère des communications.

4.2 Les communications intra-processus

Afin de ne pas avoir de buffer entre threads sur le même processus et donc de ne pas rajouter une surcharge, nous permettons à un thread d'aller directement lire dans la mémoire d'un autre thread. Nous utilisons un mécanisme de sémaphores pour garantir les dépendances. Comme nous considérons des programmes à parallélisme de données style HPF, nous devons introduire une synchronisation globale par itération dans un nid de boucles parallèles classique. Elle permet à ce qu'un thread n'aille pas lire trop tôt dans la mémoire d'un autre thread. C'est la synchronisation de boucle qui empêche un thread d'écraser ses données trop tôt.

Nombre de threads	Nb de comm.	Vol. d'1 comm. (en double)	Vol. total des comm. (en double)	Temps (en s)	Eff. (en %)
1	0	—	—	101.5	97.8
2	1	1	L	101.4	97.9
4	2	$L/2$	L	99.7	99.6
8	6	$L/2$	$3 * L$	100.1	99.2
16	12	$L/4$	$3 * L$	99.6	99.7
32	28	$L/4$	$7 * L$	100.5	98.8
64	56	$L/8$	$7 * L$	101.7	97.6

TAB. 5 - *Threads et communications intra-processus sur Sparc 5. Le temps séquentiel sans thread est de 99.3 secondes. Une ligne a une taille de* $L = 400$ *flottants.*

En définitive, les communications intra-processus ont comme coût celui d'une synchronisation interne. Nous avons vu précédemment que ce coût est proportionnel au nombre de threads. Le tableau 5 nous confirme que les synchronisations ne sont pas chères pour 64 threads (avec deux synchronisations par itération et 800 itérations). En effet, nous obtenons quasiment le même temps pour 64 threads que le temps séquentiel (+ 2.5%). Les variations de temps observées au tableau 5 doivent être dues à la forme du domaine à parcourir et aux effets de cache associés.

4.3 Les communications inter-processus

Mesure des communications

Les communications sous PM^2 se font via des appels de procédures distantes légères (*Light Remote Procedure Call, LRPC*). Les LRPC sont des messages PVM. C'est le thread de communication de PM^2 qui se charge de recevoir les messages PVM et d'appeler la fonction de traitement adéquate. Il existe sous PM^2 toute une variété de types de LRPC [20]: synchrone ou asynchrone, résultat différé ou non, avec création de thread ou non. Il y a donc un surcoût généré par le thread de gestion des LRPC.

	Ethernet		ATM	
	PVM	LRPC	PVM	LRPC
Latence	1.7 ms	2.0 ms	2.2 ms	2.5 ms
Débit	8.2 Mbits/s	6.9 Mbits/s	20 Mbits/s	13.6 Mbits/s

TAB. 6 - *Latence et débit sur réseau Ethernet entre deux stations Sparc 5.*

Nous avons mesuré la latence et le débit d'un LRPC sans création de thread entre 2 stations Sparc 5 sous Solaris 2.4 sur un réseau Ethernet quasiment inoccupé. Le tableau 6 résume nos mesures alors que la figure 5 montre que les courbes des débits des LRPC et des messages PVM sont semblables.

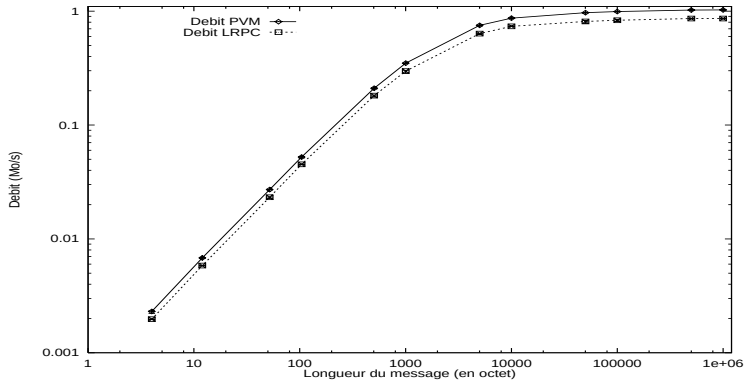


FIG. 5 - *Courbe du débit des LRPC et de PVM sur réseau Ethernet avec les options* `PvmDataRaw` *et* `PvmRouteDirect`.

Il y a plusieurs facteurs qui expliquent le surcoût des communications. Premièrement, il y a un surcoût logiciel dû à la gestion des LRPC. Deuxièmement, en plus de la préemption du processus par l'ordonnanceur Unix, nous avons aussi la préemption du thread de communication par PM^2. Cependant, le coût reste acceptable avec une latence de 18% plus longue et un débit de 16% plus faible.

Mesures des performances de notre programme

Nous n'avons pas introduit de système d'agrégation pour deux raisons. D'une part, pour vérifier que le comportement sans agrégation est bien celui attendu. D'autre part, agréger les communications revient à faire des synchronisations internes globales dont le coût est minime comparé aux communications. Le comportement devrait donc être celui avec un seul thread par communication.

Les résultats sont présentés dans le tableau 7. L est la taille de notre tableau (ici $L = 400$).

Nombre de processus	1	2	4			
Nombre de threads par processus	1	1	1	1	2	4
Nombre de communications	—	1	—	3	6	12
Volume d'une communication	—	L	—	L	$L/2$	$L/4$
Temps total (s)	101.5	54.2	27.8	29.2	35.3	42.0
Efficacité (%)	98	91	89	85	70	59
Temps de comm. par itération (ms)	—	4.31	3.0	4.8	12.4	20.8
Recouvrement des comm. (%)	—	43	50	78	56	49

TAB. 7 - *Threads et communications. Le temps séquentiel sans thread est de 99.3 secondes. La latence est de 2 ms et le débit de 6.9 Mbits/s.* $L = 400$ *flottants en double précision. L'option* `PvmRouteDirect` *de PVM est utilisée.*

Analyse des résultats

Nous constatons que le nombre de threads augmentant, le nombre de messages augmente et, avec lui, le temps de communication, bien que le volume total de communication reste constant. La raison en est que la latence du réseau Ethernet est importante. Cependant, dans la pire de nos expériences, les calculs arrivent à recouvrir encore environ 50% du surcoût engendré par les communications. Les threads, en fait, ne modifient pas le problème des communications. Ils permettent seulement de mieux recouvrir les communications par les calculs mais au prix d'un plus grand nombre de messages. Cette augmentation de message conduit pour notre expérience à des performances dégradées à cause du peu de calcul, d'un unique médium de communication et de la latence importante de ce médium.

4.4 Conclusion

Nous avons vu un moyen efficace pour gérer les communications inter-threads dans le même processus. Leur coût est au plus celui de 2 synchronisations entre les threads participant aux communications.

Le surcoût des communications entre processus est lié à la gestion des différents modes de LRPC possibles. C'est la latence qui subit la plus forte dégradation avec +18% alors que le débit n'est réduit que de 16%.

Notre expérience a montré que les threads recouvraient les communications mais que le prix à payer est une hausse du nombre de messages. Vu la taille des messages mis en jeu (moins de 4 ko) et la faible quantité de calcul par itération, le temps de latence s'avère trop important pour cette approche pour notre programme de test.

5 Migration et processus légers

5.1 Description de la migration de threads

Nous avons choisi l'environnement PM^2 parce qu'il offre la possibilité de migrer les threads. La migration d'un thread se fait par l'intermédiaire d'une fonction spéciale. Cette fonction se

charge d'empaqueter la partie utilisée de la pile du thread ainsi que les structures de contrôle du thread, d'envoyer les données sur le processus destination et de relancer l'exécution du thread sur ce processus.

PM^2 ne gère pas la cohérence des accès aux variables globales. C'est à l'utilisateur de la gérer. Pour faciliter cette gestion, PM^2 propose deux fonctions utilisateurs. L'une est appelée juste avant l'envoi du message et l'autre est appelée avant de relancer le thread. Ces fonctions permettent d'une part de gérer l'accès aux éventuelles variables locales du processus (comme la table des threads locaux) et d'autre part d'empaqueter avec le thread des éventuelles données non allouées sur la pile. En effet, dans notre implémentation, chaque thread alloue la mémoire dont il a besoin pour stocker son bloc de tableau sur le tas. PM^2 ne migrant pas ces données avec le thread, il nous est facile de les empaqueter dans le message de migration grâce à ces deux fonctions.

PM^2 fournit aussi la possibilité de compresser la pile avant de la migrer. L'algorithme par défaut ne fait que la compression de zéros. Il peut s'avérer intéressant d'utiliser un algorithme plus performant pour des réseaux lents.

5.2 Coût de la migration de threads

Nous voulons faire de la migration de threads. Il est par conséquent primordial d'avoir une idée du coût de la migration d'un thread.

Expérience : description et mesure

L'expérience consiste à faire effectuer un grand nombre d'allers-retours à un thread afin d'avoir le temps moyen d'une migration. Nous faisons varier la taille du thread à migrer ainsi que les types de réseaux. Nous ne couplons que des machines de même type, Sparc ELC ou Sparc 5. Nous avons d'abord placé les deux processus sur la même machine, puis sur des machines connectées par un réseau Ethernet puis par un réseau ATM. Nous n'utilisons pas l'option de compression ni l'option `PvmRouteDirect` de PVM. La migration utilise quand même l'option `PvmDataRaw` car la migration de threads ne peut actuellement se faire qu'entre des machines ayant, entre autres, la même représentation des nombres.

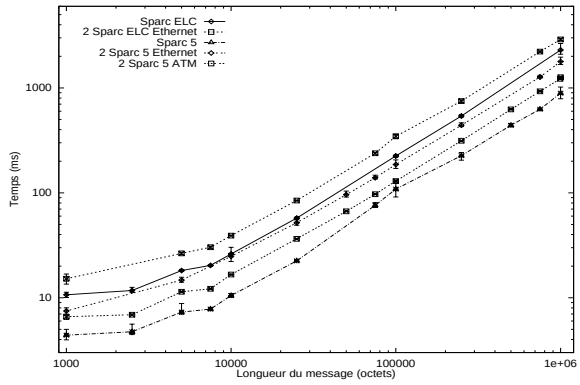


FIG. 6 - *Temps de migration de threads pour différents machines et réseau.*

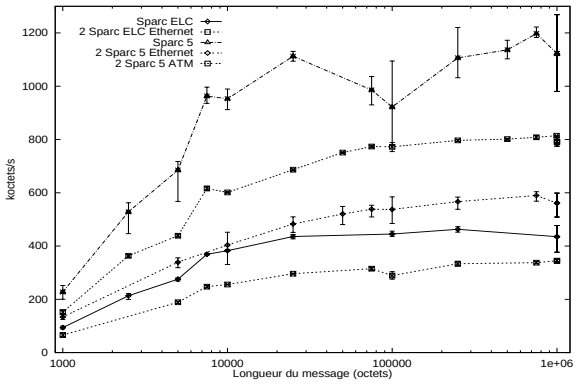


FIG. 7 - *Débit de la migration de threads pour différents machines et réseaux.*

Analyse des résultats

Les figure 6 et 7 nous montre que le coût est essentiellement dû aux communications et donc que le modèle en $\beta + L * \tau$ est encore valable en première approximation. Cependant, la latence et le débit observés ne sont pas ceux des réseaux. Il y a en effet un surcoût dû à la migration. Ce surcoût consiste essentiellement en un surcoût logiciel dû à la préparation/réception du thread à migrer ainsi qu'en une longueur minimale du message à cause des structures de contrôle du thread qui doivent être migrées.

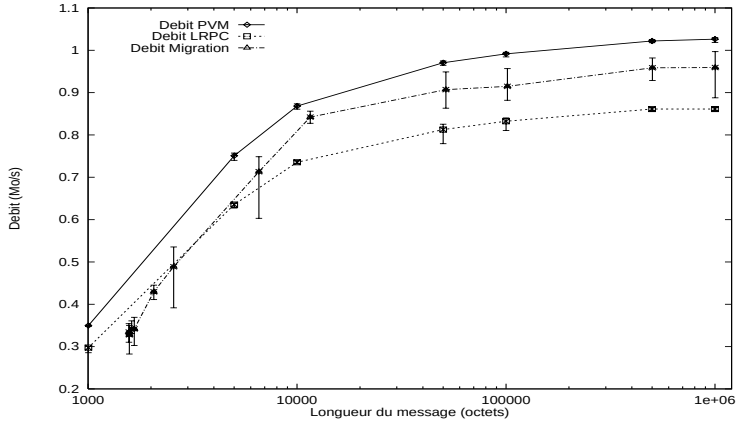


FIG. 8 - *Débit entre deux Sparc 5 sur réseau Ethernet de PVM, des LRPC et de la migration. L'option* `PvmRouteDirect` *de PVM est utilisée.*

	PVM	LRPC	Migration
Latence	1.7 ms	2.0 ms	4.7
Débit	8.3 Mbits/s	6.9 Mbits/s	7.7 Mbits/s
Temps d'un messages de 1.5 ko	3.5 ms	3.8 ms	—

TAB. 8 - *Latence et débit sur réseau Ethernet.*

Pour les performances, reportons-nous à la figure 8 et au tableau 8. Cette figure et ce tableau ont été obtenus en utilisant l'option `PvmRouteDirect`. Nous observons qu'entre deux Sparc 5, nous obtenons une latence de 4.7 ms. La latence peut paraître élevée mais c'est principalement parce que la longueur minimale d'un message contenant un thread à migrer est d'environ 1500 octets sous Solaris. C'est la taille que représente les structures de contrôle du thread plus la taille de la pile utilisée par notre thread de test. Ainsi, la latence de la migration n'est que de 24% (respectivement 36%) plus grande qu'un LRPC (respectivement un message PVM) de 1500 octets. Nous obtenons 93% du débit de PVM, soit presque 8 Mbits/s. Le débit d'une migration est supérieur au débit d'un LRPC. Cette différence doit être imputée au système de codage des nombres de PVM. En effet, la migration transfère la pile comme une suite d'octets, la compatibilité des représentations des nombres étant une conséquence de la compatibilité des piles. La migration utilise donc la fonction `pvm_pkbyte` qui se trouve être plus performante que la fonction `pvm_pkint` que nous avons utilisée pour empaqueter les données dans les LRPC. Bref, comme [12] le remarque, le codage des nombres baisse sensiblement les performances.

5.3 Conclusion

Le coût de la migration de processus légers dépend fortement du coût de communication. Par conséquent, sa modélisation se fait comme celle d'une communication classique.

La surcharge minimale introduite sous Solaris 2 par les threads se traduit au niveau des communications par l'envoi de 1.5 ko de données pour un thread qui ne consomme quasiment pas de pile. Ces données comprennent les structures de contrôle du thread ainsi que la pile.

Les performances observées sont proche de celles de PVM, soit 93% de son débit mais une latence de 130%. La latence est de 24% plus grande que celle d'un LRPC mais le débit est meilleur de 10%.

6 Conclusion

Lors de nos expériences, nous avons trouvé que l'introduction de environnement *multithread* PM^2 avec une préemption toutes les 20 ms rajoutait un surcoût de l'ordre de 2% sur une Sparc 5 et 4% sur une ELC. Les synchronisation de threads représentent un coût de 24 μs par thread sur une Sparc 5 à 70 MHz alors qu'une création de thread est 3 à 40 fois plus coûteuse.

Les communications inter-thread intra-processus ont comme coût au plus deux synchronisations lorsque nous permettons à un thread d'aller directement lire la mémoire d'un autre thread. Ces synchronisations sont là pour la garantir les dépendances entre les lectures et les écritures. Les communications inter-thread inter-processus au travers d'un réseau Ethernet montrent bien un recouvrement des communications par le calcul mais le coût de la latence d'Ethernet est un sérieux handicap vu que le nombre de messages augmente.

La migration des threads dans PM^2 est efficace. Nous obtenons une latence de 23% plus grande que celle de l'envoi d'un LRPC et le débit représente plus de 90% celui d'un message PVM. PM^2 rajoute en fait moins de 1.5 ko de données pour la gestion d'un thread.

Nous avons aussi donné trois critères pour le calcul de la taille d'un thread. Ces critères sont le surcoût maximal que nous désirons, une borne supérieure qui est la taille du cache données et le coût que nous désirons pour les synchronisations. Cependant, une étude plus poussée sur l'influence de ces critères serait nécessaire.

En conclusion, les threads sont de bons candidats pour encapsuler des processeurs virtuels de HPF. Leur coût reste acceptable alors qu'ils ont le grand avantage de ne quasiment pas remettre en cause la compilation de programme HPF. De nouvelles fonctionnalités de l'exécutif ainsi que quelques structures de contrôle en plus devraient suffire.

Remerciements

Je voudrais d'abord remercier Luc Bougé pour ses conseils et ses encouragements. Je tiens aussi à remercier Thomas Brandes pour ses explications relatifs à Adaptor et pour sa disponibilité. Je remercie également J.-M. Geib, J.-F. Mehaut, R. Namyst et Y. Denneulin du LIFL pour leur accueil et leur patience à m'expliquer les dessous de PM^2. Je remercie aussi Loïc Prylli du LIP pour son aide à propos d'ATM.

Références

[1] André (F.). – *A Multi-Threads Runtime For The Pandore Data-Parallel Compiler.* – Rapport technique n° 986, IRISA, février 1996. Available at URL http://www.irisa.fr/EXTERNE/bibli/pi/pi96.html.

[2] Bach (M. J.). – *The Design of the Unix Operating System.* – Prentice-Hall, 1986, *Software Series.*

[3] Blumofe (R.), Joerg (C.), Kuszmaul (B.), Leiserson (C.), Randall (K.) et Zhou (Y.). – Cilk : An efficient multithreaded runtime system. *In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, pp. 207–216. – Santa Barbara, California, juillet 1995.

[4] Brandes (T.). – Adaptor (HPF compilation system). – Available at URL http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html.

[5] Brandes (T.) et Zimmermann (F.). – Adaptor - a transformation tool for HPF programs. *In: Programming Environments for Massively Parallel Distributed Systems*, éd. par Decker (K. M.) et Rehmann (R. M.). pp. 91–96. – Birkhäuser Verlag.

[6] Casas (J.), Konuru (R.), Otto (S. W.), Prouty (R.) et Walpole (J.). – Adaptive load migration systems for PVM. *In: Proceedings of Supercomputing'94*, pp. 390–399. – Washington, D.C., novembre 1994.

[7] Christaller (M.), Castaneda-Retiz (MR.) et Gautier (T.). – Control parallelism on top of PVM : The Athapascan Environment. *In: EuroPVM'95*, éd. par Dongarra (J.), Gengler (M.), Tourancheau (B.) et Vigouroux (X.). LIP, ENS Lyon, pp. 71–76. – Lyon, France, septembre 1995.

[8] Cockcroft (A.). – *Sun Performance and Tuning: SPARC & Solaris.* – Sunsoft Press, Prentice-Hall, 1995.

[9] Collard (J.-F.). – *Parallélisation automatique des programmes à contrôle dynamique.* – Thèse de doctorat, Univ. Paris 6, janvier 1995.

[10] Cuvillier (G.) et Utard (G.). – *Compilation de programmes data-parallèles pour un réseau de stations de travail.* – Rapport technique n° 94-31, LIP, novembre 1994.

[11] Darte (A.). – *Techniques de parallélisation automatique de nids de boucles.* – France, Thèse de doctorat, LIP, ENS Lyon, 1993.

[12] Dillon (E.), Santos (C. Gamboa Dos) et Guyard (J.). – *Homogeneous and Heteregeneous Networks of Workstations: Message Passing Overhead.* – Rapport technique n° 238, France, Inria/Crin, juin 1995.

[13] Forum (High Performance Fortran). – *High Performance Fortran Language Specification.* – Rice University, Houston, Texas, novembre 1994. Version 1.1.

[14] Foster (I.), Kesselman (C.) et Tuecke (S.). – The Nexus Task-parallel Runtime System. *In: 1st Int. Workshop on Parallel Processing.*

[15] Hinz (D. Y.). – A run-time load balancing strategy for highly parallel systems. *Acta Informatica*, vol. 29, 1992, pp. 63–94.

[16] IEEE. – Threads extension for portable operating systems (draft 6), février 1992.

[17] Krone (O.), Aguilar (M.) et Hirsbrunner (B.). – PT-PVM: Using PVM in a multi-threaded environment. *In: EuroPVM'95*, éd. par Dongarra (J.), Gengler (M.), Tourancheau (B.) et Vigouroux (X.). LIP, ENS Lyon, pp. 83–88. – Lyon, France, septembre 1995.

[18] Namyst (R.) et Mehaut (J.-F.). – *Marcel: Une bibliothèque de processus légers.* – LIFL, Lille, France, 1995. Available at URL http://www.lifl.fr/~namyst/pm2.html.

[19] Namyst (R.) et Mehaut (J.-F.). – *PM2: Parallel Multithreaded Machine, Guide d'utilisation.* – LIFL, Lille, France, 1995. Version 1.5, available at URL http://www.lifl.fr/~namyst/pm2.html.

[20] Namyst (R.) et Mehaut (J.-F.). – PM2 parallel multithreaded machine : A multithreaded envronment on top of PVM. *In: EuroPVM'95*, éd. par Dongarra (J.), Gengler (M.), Tourancheau (B.) et Vigouroux (X.). LIP, ENS Lyon, pp. 179–184. – Lyon, France, septembre 1995.

[21] Nicol (D. M.). – Rectilinear partitioning of irregular data parallel computation. *Journal of Parallel and Distributed Computing*, vol. 23, 1994, pp. 119–134.

[22] Portland Group, Inc. (The). – The Portland Group: Welcome Page. – Available at URL http://www.pgroup.com.

[23] Sohn (A.), Sato (M.), Yoo (N.) et Gaudiot (J.-L.). – Effects of multithreading on data and workload distribution for distributed memory multiprocessors. *In: Proceeding of the 10th IEEE International Parallel Processing Symposium.* – Honolulu, Hawaii, avril 1996.

[24] Stellner (G.) et Pruyne (J.). – Resource management and checkpointing for PVM. *In: EuroPVM'95*, éd. par Dongarra (J.), Gengler (M.), Tourancheau (B.) et Vigouroux (X.). LIP, ENS Lyon, pp. 131–136. – Lyon, France, septembre 1995.

[25] Tanenbaum (A.). – *Systèmes d'exploitation: systèmes centralisés, systèmes distribués.* – InterEditions (Paris), Prentice Hall (London), 1994.

[26] Thinking Machines Corp., 245 First St., Cambridge, MA 02142. – *C* programming guide*, novembre 1990. Version Number 6.0.

[27] Utard (G.). – Compilation of data-parallel programs into PVM code for local area network of workstations. *In: EuroPVM'95*, éd. par Dongarra (J.), Gengler (M.), Tourancheau (B.) et Vigouroux (X.). LIP, ENS Lyon, pp. 125–130. – Lyon, France, septembre 1995.