Algorithmes parallèles à grain adaptatif et applications

El Mostafa Daoudi[*], — Thierry Gautier[**], — Aicha Kerfali[*], — Rémi Revire[**] — Jean-Louis Roch[**]

[*] *Université Mohamed 1e, Faculté des Sciences*
Dépt. de Mathématiques et d'Informatique
60 000, Oujda, MAROC

{kerfali,mdaoudi}@sciences.univ-oujda.ac.ma

[**] *Laboratoire ID-IMAG (UMR 5132)*
Projet APACHE (CNRS/INPG/INRIA/UJF),
51, av. Jean Kuntzmann, F-38330 Montbonnot Saint-Martin

{Jean-Louis.Roch,Thierry.Gautier}@imag.fr

RÉSUMÉ. *Nous proposons un schéma algorithmique générique original pour contrôler la granularité du parallélisme en cours d'exécution. Ce schéma est basé sur le couplage de deux algorithmes, l'un séquentiel, l'autre parallèle à grain fin. La génération de parallélisme n'est effectuée qu'en cas d'inactivité d'un processeur. Lors de l'exécution sur un nombre restreint ou variable de ressources, ce schéma permet de limiter le surcoût lié à la génération de parallélisme, sans limiter le degré de parallélisme potentiel. Il est adapté aux problèmes pour lesquels la parallélisation entraîne, malgré un gain de temps, une pénalité en nombre d'opérations ou en performance. Nous l'appliquons à la parallélisation de deux applications effectives :* `gzip` *[GAI 03] qui implémente la méthode de compression de Lempel-Ziv, problème $\mathcal{P}$-complet considéré difficilement parallélisable ; et* `PL` *[PRO] un moteur d'inférence probabiliste.*

ABSTRACT. *To tune granularity of tasks at runtime, we introduce an original algorithmic scheme. It is based on the coupling of two algorithms, one sequential, the other one parallel and fine grain. However, parallelism is generated only in case of idleness of some processor. Then, when executing the program on a limited number of resources, the overhead related to parallelism management is bounded with no restriction of potential parallelism. It is especially suited to applications for which parallelization drastically increases the number of operations or induces a loss of performance, despite a decreasing execution time. This scheme is applied to parallelisation of two effective applications:* `gzip` *[GAI 03] that implements Lempel-Ziv compression, a $\mathcal{P}$-complete problem; and* `PL` *[PRO] a probabilistic inference engine.*

MOTS-CLÉS : *algorithmique, parallélisme, ordonnancement glouton, vol de travail*

KEYWORDS: *algorithmic, parallelism, greedy schedule, work stealing*

1. Introduction

Pour être portable, un programme parallèle doit faire abstraction de l'architecture sous-jacente (*i.e.* le programme ne doit pas dépendre de l'architecture cible). Le programme parallèle décrit alors un parallélisme implicite, indépendant du nombre de processeurs. Lors de l'exécution, le système ou le noyau exécutif contrôle le repliement de ce parallélisme sur les ressources de l'architecture en ordonnançant plusieurs calculs sur un même processeur. Classiquement, le programme définit chacune des tâches à exécuter (concurrentes ou non) ainsi que les données nécessaires pour leurs exécutions ; le programme fixe ainsi la granularité des calculs et la granularité des données.

Deux points de vue antagonistes sont considérés pour chosir ce grain. D'un point de vue algorithmique pratique, le grain peut être fixé en fonction du nombre de ressources, généralement supposées identiques, disponibles lors de l'exécution. L'inconvénient d'une telle approche est que le nombre de ressources et leurs puissances peuvent être variables, comme c'est le cas dans la programmation sur grappe en contexte multi-utilisateur ou, plus encore, sur grille de machines. D'un point de vue algorithmique théorique [JAJ 92], le grain est fixé pour minimiser le temps d'exécution sur un nombre infini de processeurs tout en conservant un nombre total d'opérations proche du meilleur algorithme séquentiel (on parle d'algorithme parallèle *optimal*). On obtient ainsi un algorithme dit de *grain fin* avec un grand nombre de tâches. L'inconvénient d'une telle approche est que, lors d'une exécution effective sur un nombre toujours restreint de ressources, le placement et l'entrelacement de ces nombreuses tâches entraine un surcoût.

Ainsi, supposons que l'exécution du programme parallèle soit en fait effectuée sur un seul processeur, les autres processeurs étant mobilisés pour d'autres calculs. Dans ce cas, exécuter l'algorithme parallèle au grain prélablement choisi peut être pénalisant par rapport à l'exécuter séquentiellement en évitant tous les surcoûts incontournables liés au parallélisme : création de tâches, copie de données.

Pour limiter le surcoût d'ordonnancement d'un algorithme de grain fin, les techniques mises en œuvre dans les ordonnancements par vol de travail [RAY 90, BLU 98, BER 03] consistent à privilégier, dans le cas où tous les processeurs sont actifs, l'exécution séquentielle efficace de l'algorithme parallèle. Un nouveau processus n'est effectivement créé pour l'exécution d'une tâche prête que lorsqu'un processeur devient inactif et effectue une opération de vol : on parle de *dégénération séquentielle* [ROC 01]. De plus, ces opérations de vols étant peu nombreuses si le programme a un très grand degré de parallélisme [BLU 98], la plus grande partie du surcoût est alors reportée sur ces opérations peu fréquentes de vol plutôt que sur les opérations de création de tâches, fréquentes mais qui correspondent à la progression du *travail* du programme. Cependant, si elles diminuent le surcôut d'ordonnancement de l'algorithme parallèle, ces techniques ne s'attaquent pas au surcoût arithmétique lié à la parallélisation, qui peut s'avérer très coûteux en comparaison à un algorithme séquentiel optimisé.

En effet, un tel ordonnancement suppose que la sérialisation de l'algorithme parallèle est un algorithme séquentiel efficace. Pourtant, cette hypothèse est très restrictive. Par exemple, considérons le calcul des préfixes [JAJ 92, LAD 80], problème ayant un grand degré de parallélisme ; toute parallélisation qui divise par un facteur 2 le temps d'exécution requiert un nombre d'opérations d'un facteur 1.5 supérieur à un algorithme séquentiel [LAD 80].

Dans cet article, nous proposons un schéma algorithmique générique original basé sur le couplage de deux algorithmes distincts, l'un séquentiel et l'autre parallèle récursif à grain fin. Nous présentons d'abord le cadre des algorithmes visés et les problèmes de granularité (§2) illustrés par l'exemple du produit itéré. La technique classique consiste à regrouper les tâches de grain fin de l'algorithme parallèle pour les remplacer par un algorithme séquentiel plus performant. Dans [COS 89], cette notion de grain adaptatif apparait dans un contexte statique sur l'exemple de l'élimination de Gauss. Mais ici, le cadre dynamique rend l'adaptation algorithmique indépendante du nombre de processeurs de l'architecture. De manière duale, nous proposons un schéma original (§3) basé sur l'extraction de parallélisme à partir de l'exécution séquentielle en cours. Nous l'illustrons sur l'exemple du produit itéré (§3.3). Enfin, nous expérimentons ce schéma pour la parallélisation effective à grain adaptatif, donc indépendante du nombre de ressources disponibles et de leur puissance, de deux applications. Pour `gzip` (§4.1), la parallélisation originale ainsi obtenue est différente de l'algorithme séquentiel (problème P-complet [GRE 95]) ; ses performances sont évaluées en termes de temps parallèle et taux de compression par rapport au programme séquentiel [GAI 03]. Pour `PL` (§4.2), un moteur d'inférence probabiliste qui effectue le parcours récursif d'une arborescence de grande taille, l'extraction de parallélisme correspond ici à une version algorithmique du vol de travail. Les performances de la parallélisation sont comparées à celles du programme séquentiel initial. l'algorithme séquentiel initial gère implicitement une pile des sous-arbres restant à traiter ; l'extraction récursive de parallélisme est alors basée sur le retrait d'un sous-arbre de cette pile pour le dégénérer en séquentiel. Ces applications sont expérimentées sur différentes machines.

2. Algorithmes parallèles par découpe récursive avec seuil de granularité

Dans toute la suite, nous considérons un problème qui admet un algorithme séquentiel, *i.e.* sans instruction parallèle. On suppose que le problème peut être résolu par un algorithme parallèle à grain fin de type découpe récursive. En les illustrant sur l'exemple du produit itéré, nous rappelons tout d'abord les notations utilisées pour les coûts puis les techniques classiques de seuil de granularité et de dégénération séquentielle qui permettent de diminuer le surplus d'opérations effectuées par l'algorithme parallèle en comparaison avec le nombre d'opérations de l'algorithme séquentiel.

2.1. *Notations de coût – Exemple du produit itéré*

Dans toute la suite, on utilise les notations de coût proposées dans [BLU 98] : T_{seq} dénote le coût d'un algorithme séquentiel (*i.e.* sans instruction parallèle) ; T_∞ le temps d'exécution d'un algorithme parallèle sur un nombre infini de processeurs ; T_1 le temps total (*i.e.* nombre d'opérations) de cet algorithme parallèle ; T_p le temps d'exécution de cet algorithme ordonnancé sur p processeurs. Un ordonnancement glouton [GRA 69, BLU 98] garantit un temps $\max\left(\frac{T_1}{p}, T_\infty\right) \leq T_p \leq \frac{T_1}{p} + T_\infty$ (principe de Brent). De plus, nous noterons par τ_{op} le coût de l'opération séquentielle op ; τ_{op} est *a priori* inconnu et éventuellement variable mais supposé majoré par une constante.

Pour illustrer le problème du choix de la granularité, considérons l'exemple classique du produit itéré qui consiste à calculer $r = \star_{i=0}^{n} f(i)$ où $\star$ est une loi associative (f est une fonction qui retourne par exemple les éléments d'un tableau). Nous reprenons l'analyse de coût de [ROC 95] où le problème de choix de grain est mis en évidence en liaison avec la notion d'*irrégularité* d'un algorithme. Le calcul par boucle séquentielle a un coût $T_{seq}(n) = n.\tau_\star$.

```
1   Elt f( int i ) { ... } ;
2
3   struct Product { // produit binaire sequentiel
4     void operator()(Shared_r<Elt> r1, Shared_r<Elt> r2, Shared_w<Elt> r){
5       r.write ( r1.read() * r2.read() ) ;
6   } } ;
7
8   struct IteratedProduct { // algorithme parallele
9     void operator()( int i, int j, Shared_w<Elt> r ) {
10        if (i == j) { r.write( f(i) ) ; }
11        else {
12          Shared<Elt> r1, r2 ;
13          Fork< IteratedProduct > () (i, (i+j)/2, r1 ) ;
14          Fork< IteratedProduct > () ((i+j)/2+1, j, r2 ) ;
15          Fork< Product > () (r1, r2, r ) ;
16  } } } ;
17
18  Fork < IteratedProduct >() (0, n, prod ) // Appel principal
```

Figure 1. *Programmation du produit itéré en Athapascan*

La figure 1 montre une programmation dans le langage Athapascan [GAL 98, GAU 03] d'un algorithme basé sur une découpe récursive du problème en deux. Les lignes 8 et 9 déclarent un prototype de tâche Athapascan qui prend un intervalle $[i, j]$ et qui retourne, par écriture (Shared_**w**) dans une variable partagée, le résultat $\star_{i=0}^{n} f(i)$. La ligne 12 déclare deux variables partagées qui contiendront les résultats du produit

itéré sur les intervalles $[i,(i+j)/2]$ et $[(i+j)/2+1,j]$ calculés par les tâches créées (instruction Fork) aux lignes 13 et 14. Ces deux résultats sont lus par la tâche créée à la ligne 15 et dont le prototype est donné aux lignes 3 et 4 : celle-ci lit (Shared_**r**) les valeurs de deux variables partagées et en écrit le produit (Shared_**w**).

Cet algorithme permet une parallélisation à grain fin de temps parallèle $T_\infty(n) = \Theta(\log n)$. Le temps total $T_1(n)$ passé dans l'algorithme englobe, outre les n opérations $\star$, le surcoût introduit par les $2n$ opérations de création de parallélisme (`Fork`) et de synchronisation (`Shared`) ; ainsi $T_1(n) = T_{seq}(n)+2n(\tau_{Fork}+\tau_{Shared}) = T_{seq}(n)+O(n)$ où $2n(\tau_{Fork}+\tau_{Shared})$ est le surcoût lié au contrôle du parallélisme.

2.2. *Seuil de granularité*

Ce surcoût dû au parallélisme peut être réduit en augmentant la granularité : un seuil statique g, appelé *grain*, stoppe la récursivité. La ligne 10 est alors réécrite comme présenté dans la figure 2. La complexité parallèle devient $T_\infty(n) = g + \log \frac{n}{g}$

```
1    if ( (j - i) <= grain ) {
2        Elt tmp = f(i) ;
3        for (int k=i+1 ; k<=j ; k++) tmp = tmp * f(k) ;
4        r.write( tmp ) ;
5    }
```

Figure 2. *Modification de la ligne 10 du code de la figure 1 pour fixer le seuil de granularité*

et $T_1(n) = T_{seq}(n) + \frac{2n}{g}(\tau_{Fork}+\tau_{Shared})$. Le surcoût lié au contrôle du parallélisme est ainsi divisé d'un facteur g. Le choix de granularité $g = \log n$ permet de garder un temps $T_\infty(n) = \Theta(\log n)$ tout en obtenant $T_1(n) = T_{seq}(n) + O\left(\frac{n}{\log n}\right)$ soit asymptotiquement $T_1(n) \simeq T_{seq}(n)$.

On obtient ainsi un algorithme parallèle dont le temps sur une infinité de processeurs est équivalent au temps de l'algorithme parallèle de grain fin, mais dont le temps total d'exécution tend vers le temps séquentiel. Le surcoût lié au parallélisme apparaît alors asymptotiquement négligeable devant le temps séquentiel d'exécution.

Cependant, dans le cas non asymptotique, un inconvénient de l'algorithme précédent est que la création des tâches, décrite par l'exécution des instructions $Fork$, est effectuée même si le support d'exécution ne possède pas suffisamment de processeurs. Pour limiter ce surcoût de génération du parallélisme, nous allons considérer deux approches. La première consiste à réduire autant que possible le coût d'une création d'une tâche en utilisant la solution de *dégénération séquentielle* qui permet de reporter un grande partie de ce surcoût au cas où des processeurs sont inactifs. La

seconde approche, qui consiste à adapter dynamiquement le choix de la granularité en fonction de l'activité des processeurs, sera décrite dans la section 3.

2.3. *Seuil statique et dégénération séquentielle de l'algorithme parallèle*

Pour minimiser le surcoût de création de tâches parallèles, la technique appelée dégénération séquentielle [ROC 01] consiste à éviter la création d'une tâche immédiatement prête (cas des programmes série-parallèle, fork-join) en la réalisant directement comme un appel local de fonction séquentielle. Cette technique est notamment utilisée pour l'implémentation de l'ordonnancement par vol de travail (*work-stealing*) de Cilk [FRI 98].

Dans cet ordonnancement, les tâches ne sont créées localement par un processeur actif que lorsqu'il reçoit une requête de travail (interruption) de la part d'un processeur inactif. Dans ce cas, la tâche correspond à la *continuation* de la fonction en cours d'exécution sur le processeur inactif[1]. Dans les autres cas, toute déclaration de parallélisme (instruction `Fork`) est interprétée comme un appel de fonction séquentielle.

Dans le cas des programmes série-parallèles, le nombre de requêtes de travail émises par processeur est au plus T_∞ [BLU 98] ; ainsi, le surcoût dû au parallélisme est borné par $pT_\infty(\tau_{Fork} + \tau_{Shared})$. Le temps d'exécution T_p de l'algorithme parallèle exécuté sur p processeurs est majoré par $\frac{T_1}{p} + O(p.T_\infty)$. Le surcoût d'ordonnancement est diminué, mais le nombre total d'opérations arithmétiques T_1 effectuées est le même que dans l'algorithme à granularité statique, donc possiblement très supérieur au temps séquentiel T_{seq}. Dans la section suivante, nous présentons une parallélisation dite à grain adaptatif telle que le nombre d'opérations effectuées reste proche de T_{seq}.

3. Parallélisation à grain adaptatif de l'algorithme séquentiel

Dans le schéma précédent de dégénération séquentielle, c'est l'algorithme parallèle qui est exécuté mais, si tous les processeurs sont actifs, les créations de tâches se traduisent en appels de fonction séquentielle. Seule la granularité de l'ordonnancement est adaptée, pas celle de l'algorithme. Pour adapter non seulement la granularité de l'ordonnancement mais aussi celle de l'algorithme, notre méthode est basée sur une approche duale ; il s'agit d'exécuter un algorithme séquentiel par chaque processeur actif, mais en donnant la possibilité d'extraire du parallélisme sur le travail qui reste à effectuer. Cette extraction de parallélisme n'est effectuée que par un processeur devenu inactif (*voleur*), sans interruption de l'algorithme séquentiel en cours d'exécution sur le processeur actif (*volé*), mais en gérant la synchronisation entre voleur et volé à des points importants de l'algorithme adaptatif.

---

1. Cette technique préemptive est appelée *dégénération séquentielle passive* dans [ROC 01] où est présentée une alternative non-préemptive, appelée *dégénération séquentielle active*.

Ainsi, le recours à l'algorithme parallèle et le surcoût de génération de parallélisme associé n'est effectif qu'en cas d'inactivité d'un processeur. Ceci permet d'éviter les dégradations de performances liées au parallélisme en regard d'un algorithme séquentiel.

3.1. *Hypothèses*

Dans toute la suite, nous supposons que l'algorithme séquentiel est tel qu'à tout instant de l'exécution, la séquence des opérations qui termine l'algorithme peut être réalisée par un autre algorithme, parallèle et à grain fin (par exemple de type découpe récursive). L'opération qui consiste à extraire une partie du calcul séquentiel en cours pour la traiter en parallèle (par un autre algorithme séquentiel du même type) est appelée "extraction de parallélisme" (*ExtractPar*) ; après traitement, les résultats d'une part de la partie qui a subi l'extraction et d'autre part du travail extrait sont ensuite fusionnés (*Merge*).

Plus précisément, étant donné un algorithme séquentiel (resp. parallèle), le résultat r de son évaluation sur une entrée x est noté $r = f_{seq}(x)$ (resp. $f_{par}(x)$). Nous faisons l'hypothèse qu'une entrée x possède une structure de liste avec un opérateur de concaténation noté $\sharp$ et qu'il existe un opérateur (non nécessairement associatif) de fusion $\oplus$ des résultats.

À tout moment de l'évaluation de $f_{seq}(x)$, il est possible de découper x en $x_1 \sharp x_2$. Le résultat calculé par l'algorithme parallèle est alors $f_{par}(x) = f_{seq}(x_1) \oplus f_{seq}(x_2)$. Nous supposons que les deux résultats $f_{seq}(x)$ et $f_{par}(x)$ sont *équivalents* s'ils sont proches au sens d'une métrique $\| \ \|$. Cependant, $f_{seq}(x)$ et $f_{par}(x)$ ne sont pas nécessairement *identiques*, c'est-à-dire que les valeurs calculées ne sont pas égales.

Dans le cadre restreint des homomorphismes de liste [BIR 87], cette hypothèse s'écrit alors : $f_{seq}(x \sharp y) = f_{seq}(x) \oplus f_{seq}(y)$. Cependant, il existe des techniques qui permettent de construire un algorithme parallèle pour des problèmes qui ne sont initialement pas des homomorphismes de liste. Par exemple dans [COL 95], le problème du calcul du segment d'entiers de somme maximale est ainsi transformé en l'évaluation d'un homomorphisme suivi d'un filtrage, et ce au prix d'une augmentation du nombre d'opérations.

Nous présentons dans la section 4.1, une parallélisation du problème de la compression à partir de l'algorithme de GZIP, en supposant que le problème est un homomorphisme de liste alors qu'il n'en est pas un. Ainsi, les résultats $f_{seq}(x)$ et $f_{par}(x)$ ne sont pas identiques ; mais ils sont considérés équivalents dans le sens que $f_{seq}(x)$ et $f_{par}(x)$ fournissent tous deux des fichiers compressés, en pratique de taille similaire, sur lesquels l'application de `gunzip` donne le même fichier x.

Ainsi, parmi les problèmes qui admettent deux algorithmes (séquentiel et parallèle) vérifiant ces hypothèses figurent les algorithmes séquentiels classiques basés sur des opérations associatives tels le produit itéré, les préfixes ou la recherche arborescente

profondeur d'abord ; mais aussi de nombreux autres problèmes comme l'élimination de Gauss par exemple [COS 89], la compression ou en algèbre linéaire.

3.2. *Schéma général de calcul*

L'algorithme, dit à grain adaptatif, suppose que le système sous-jacent implémente un ordonnancement glouton des processeurs : lorsqu'un calcul est démarré sur un processus, son exécution est poursuivie jusqu'à une instruction de synchronisation ou la terminaison du calcul. Le modèle d'architecture considérée est une machine à mémoire distribuée composée de nœuds multiprocesseurs.

La solution que nous proposons dans l'algorithme 3 se déroule comme suit (cf figure 4) : tout d'abord le descripteur du travail à effectuer et l'emplacement pour stocker le résultat sont créés. Initialement, le processeur P_0 prépare une tâche pour un vol éventuel, en exécutant l'instruction *Fork<LastPartComputation>*. Puis, il démarre l'algorithme séquentiel *SeqCalcul*. En cas d'inactivité d'un processeur, la tâche prête à être volée sera exportée vers le processeur inactif, appelé *voleur*.

Si un processeur devient inactif, il viendra aider un processeur actif en exécutant la tâche *LastPartComputation* prête à être volée. Cette tâche, en s'exécutant :

– extrait une partie (par exemple la moitié) du travail qui est à la charge du processeur actif (appelé *victime* ou *volé*). Pour cela, elle modifie le travail du processeur victime et construit un nouveau travail à effectuer, en utilisant les informations minimales de la tâche *LastPartComputation.*

– Puis elle crée deux tâches de vol (prêtes à être volées) : une tâche concernant la partie du travail volé et une autre concernant la partie restante du travail pour le processeur victime.

– Après la création de ces deux tâches, la tâche applique l'algorithme séquentiel à la partie qui a été volée du processeur victime.

Le processus décrit ci-dessus et illustré dans la figure 3 se répète d'une façon récursive sur tous les travaux créés jusqu'à la terminaison. Sur une architecture distribuée, la synchronisation entre les deux processus voleur et volé s'implémente par un accès du voleur à la mémoire qui contient le contexte du volé pour extraire une partie du travail en cours. Pour maintenir la cohérence des données et du descripteur du travail lors des opérations d'accès par le volé et le voleur, des verrous ou des protocoles adaptés doivent être utilisés, éventuellement par l'utilisation de communication entre deux nœuds de l'architecture. Nous noterons dans la suite par τ_{coh} le coût d'une opération nécessaire au maintient de la cohérence du descripteur de travail.

3.3. *Application à la parallélisation du produit itéré*

Dans la section 2, nous avons montré les inconvénients posés par une parallélisation avec granularité statique pour le problème du produit itéré. Nous montrons ici

Côté volé : Exécution de l'algorithme séquentiel sur le descripteur de travail dw

```
1  // LastPartComputation implémente l'algorithme parallèle ;
2  // cette tâche permet l'extraction de parallélisme de l'algorithme séquentiel
4  Fork< LastPartComputation >( dw, LastRes ) ;
5  // SeqCalcul est l'algorithme séquentiel sur le travail dw
6  LocalRes = SeqCalcul ( dw ) ;
7  // MergeRes synchronise les deux algorithmes LastPartComputation et
8  // SeqCalcul pour la collection des résultats
9  Fork< MergeRes >( LocalRes, LastRes, FinalRes ) ;
```

Côté voleur : Extraction du parallélisme et exécution séquentielle

```
1  // ExtractPar modifie le travail dw de l'algorithme séquentiel volé et
2  // crée un nouveau travail new_dw
3  new_dw = ExtractPar( dw ) ;
4  // Création d'une nouvelle tâche LastPartComputation d'extraction
5  // de parallélisme sur l'ancien travail
6  Fork< LastPartComputation >()( dw, LastResVole ) ;
7  // Création d'une tâche d'extraction LastPartComputation d'extraction
8  // de parallélisme sur le nouveau travail
9  Fork< LastPartComputation >()( new_dw , LastResVoleur ) ;
10 // Exécution séquentielle sur le travail new_dw
11 LocalRes = SeqCalcul( new_dw ) ;
12 // Collecte des résultats (voir figure 4)
13 Fork< MergeRes >()( LocalRes, LastResVoleur, FinalResVoleur ) ;
14 Fork< MergeRes >()( LastResVole, LastResVoleur, LastRes ) ;
```

Figure 3. *Algorithme à grain adaptatif. La figure 4 illustre le comportement dynamique de cet algorithme.*

comment ces inconvénients peuvent être évités en appliquant le schéma algorithmique que nous avons proposé dans la section 3.2.

L'application de l'algorithme est illustré sur la figure 5 pour $n = 100$ et $p = 2$. Initialement, le processeur P_0 commence par créer une tâche prête à être volée ; ensuite il évalue en séquentiel le produit itéré décrit par les indices de début (*first*) et de fin (*last*). Chaque lecture de ces indices doit voir la dernière valeur modifiée par un éventuel voleur. Cette cohérence est implémentée soit en utilisant des outils de synchronisation de type verrou ou sémaphore, soit en utilisant un protocole adapté. Initialement, le travail total (*first*=0 et *last*=100) est affecté au processeur P_0 qui démarre l'exécution de l'algorithme séquentiel. Cette évaluation séquentielle produit un résultat local (LocalRes). Si à un instant donné P_1 devient inactif, il démarrera l'exécution de la tâche de vol préparée par P_0. Au moment du vol le processeur P_1 extrait la moitié du travail

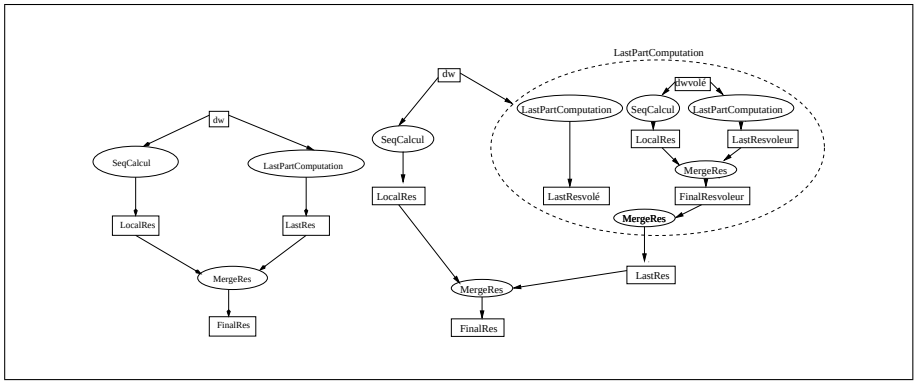

Figure 4. *Comportement dynamique de l'algorithme à grain adaptatif*

restant (ici *first*=52 et *last=100*) et laisse à P_0 l'autre moitié du travail (ici *first*=3 et *last*=51).

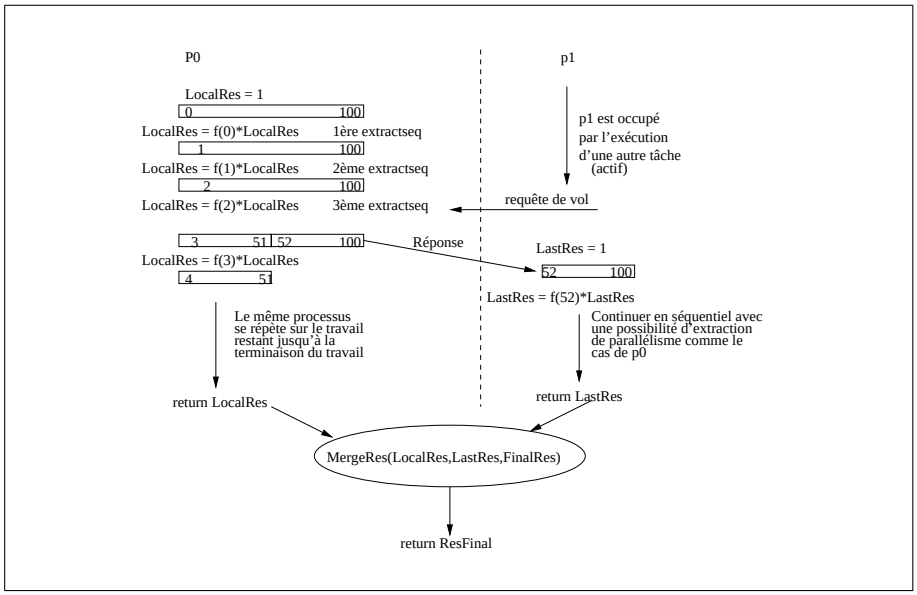

Figure 5. *Exécution du produit itéré à grain adaptatif pour* $n = 100$ *sur 2 processeurs.*

Sous l'hypothèse faite sur l'ordonnancement (glouton et non préemptif), le coût théorique de l'algorithme du produit itéré à grain adaptatif est le suivant :

– s'il y a 1 seul processeur, le temps d'exécution est $T_1(n) = T_{seq}(n) + 2(\tau_{Fork} + \tau_{shared}) + \Theta(n\tau_{coh})$: deux tâches sont créées, la deuxième étant vide (cette tâche est créée pour indiquer la possibilité de vol).

– s'il y a une infinité de processeurs inactifs, on a encore $T_\infty(n) = \Theta(\log_n)$ et $T_1(n) = T_{seq}(n) + 2n(\tau_{Fork} + \tau_{shared}) + \Theta(n\tau_{coh})$: les intructions de synchronisations induites par l'accès aux descripteurs de travaux sont parallèles, il n'y a pas de perte de parallélisme.

Basé sur l'algorithme séquentiel, l'algorithme adapte donc le degré de parallélisme aux ressources disponibles. Ainsi, avec p processeurs identiques à l'initialisation, il y aura au pire cas $p.T_\infty = O(p \log n)$ créations de tâches. On a alors $T_1(n) = T_{seq}(n) + 2p \log p(\tau_{Fork} + \tau_{shared}) + \Theta(n\tau_{coh})$ et le temps d'exécution $T_p(n)$ sur les p processeurs est majoré par $T_1(n)/p + O(\log n)$.

Sans perte de parallélisme, le surcoût de parallélisme est borné par $p.T_\infty$ comme pour la dégénération séquentielle ; ainsi basé sur l'inactivité de ressources, le schéma tire parti d'une architecture dynamique ou dans laquelle les vitesses des processeurs sont variables. Mais en plus de la dégénération séquentielle d'un algorithme parallèle, l'exécution permet de tirer parti d'un algorithme séquentiel spécialisé toujours privilégié à l'algorithme parallèle sur une ressource active. Cette caractéristique est particulièrement adaptée à la parallélisation de problèmes pour lesquels l'algorithme séquentiel est plus performant que l'algorithme parallèle, comme c'est le cas pour `gzip` étudié dans la section 4.1 suivante.

3.4. *Surcoût du maintien de cohérence*

Le surcoût dû au parallélisme de gestion des tâches se retrouve en partie dans la gestion de la synchronisation nécessaire lors des accès aux descripteurs de travaux créés sur inactivité des ressources. En effet, à la fois le processeur volé et le processeur voleur accèdent à un même descripteur de travail.

Le code de l'algorithme séquentiel doit donc être analysé afin d'identifier les sections critiques de code qui doivent être exécutées en exclusion mutuelle vis-à-vis des modifications du descripteur de travail. La réalisation effective de l'exclusion peut reposer sur l'utilisation de primitives générales de synchronisation (verrou, sémaphore). Dans des cas relativement simples, il est possible d'utiliser des protocoles adaptés, tel que T.H.E. [FRI 98], qui exploitent les caractéristiques des modèles de consistance mémoire, ou bien certaines instructions atomiques de lecture/écriture ou d'incrémentation des architectures matérielles spécifiques.

Ce coût de maintien de cohérence peut s'avérer important vis-à-vis des opérations de calcul et masquer le gain en nombre d'opérations arithmétiques grâce à l'utilisation d'un algorithme séquentiel optimisé à la place d'un algorithme parallèle. Dans ce cas, il est alors nécessaire de changer l'algorithme séquentiel afin d'augmenter sa granularité de maintien de cohérence, c'est-à-dire en augmentant le nombre d'opérations effectuées par accès aux descripteurs de travaux. Les techniques d'augmentation de

la granularité des algorithmes [ROC 95, ROC 01] d'algorithmes parallèles à grain fin sont candidates à la conception d'un tel algorithme.

4. Applications

Nous avons appliqué ce schéma d'algorithme à grain adaptatif à deux applications réelles : `gzip` qui est un utilitaire fréquemment utilisé pour le traitement de données informatiques et demandant un temps de traitement relativement important (jusqu'à 3s pour un fichier de 5 Mo) ; et `PL` qui est un moteur d'inférence probabiliste pour du calcul bayésien développé par le projet INRIA Sharp[2] et actuellement vendu par ProBayes[3].

4.1. *GZIP*

L'algorithme implanté par `gzip` est une variation de Lempel-Ziv LZ77 [NEL 95]. L'algorithme est basé sur une analyse du fichier utilisant une fenêtre de 32 kilo-octets ; la compression est réalisée à la volée en utilisant deux arbres de Huffman. La taille de ces arbres croît dynamiquement au fur et à mesure de la compression séquentielle, en fonction des symboles déjà rencontrés et des distances entre les occurrences d'un même symbole. Ainsi, le temps de recherche et d'insertion dans l'arbre peut devenir important. Pour limiter ce temps, lorsque la taille de ces arbres devient supérieure à un seuil (paramétrable par l'utilisateur de `gzip`), la compression en cours est stoppée. Une nouvelle compression recommence alors à partir du caractère courant avec des arbres de Huffman réinitialisés.

La compression avec ce nouvel arbre est donc indépendante de la précédente ; par contre, la position, dans le fichier, du début de cette compression n'est donnée par `gzip` que lorsque la taille de l'arbre dépasse le seuil, donc uniquement à la fin de la compression. Le problème de compression équivalent à cet algorithme de coût linéaire par rapport à la taille du fichier, est P-complet [GRE 95] et aucune parallélisation qui délivrerait la même compression que la compression séquentielle n'a été proposée à ce jour, même sur 2 processeurs. L'objectif d'une parallélisation n'est donc pas de fournir le même fichier que la compression séquentielle, mais d'avoir un taux de compression le plus proche possible de la compression séquentielle.

Dans la suite, nous présentons deux parallélisations de `gzip`, l'une classique à granularité statique, l'autre basée sur l'algorithme à grain adaptatif. Notons que les fichiers compressés délivrés par ces deux parallélisations de gzip peuvent être décompressés par l'utilitaire séquentiel `gunzip` classique.

---

2. Le site du projet est accessible à l'adresse `http ://www.inrialpes.fr/sharp`.
3. Startup de l'INRIA, Grenoble.

4.1.1. *Parallélisation à granularité statique*

Il est possible de paralléliser la compression d'un fichier en le découpant en plusieurs blocs (fichiers temporaires), chaque bloc étant compressé indépendamment. L'inconvénient est que cette découpe entraîne une diminution du taux de compression en comparaison avec la compression séquentielle sans découpe. L'idée de cette parallélisation est relativement simple ; elle ne nécessite que peu de modifications du code source de `gzip` pour le rendre réentrant et pouvoir l'encapsuler dans une tâche Athapascan-1. Le choix de la granularité (*i.e.* ici taille de chaque bloc sur lequel sera invoqué `gzip`) est problématique :

1) au niveau du temps d'exécution : le grain dépend non seulement du nombre de processeurs mais aussi du contenu des blocs. En effet, deux blocs de même taille peuvent requérir, selon leur contenu, des temps de compression très différents et difficilement prédictibles sans surcoût important. Cette non-prédictibilité motive le choix d'un grain fin de parallélisation.

2) Au niveau du taux de compression : la taille des blocs doit être la plus grande possible pour maximiser le taux de compression.

La dégénération séquentielle appliquée à cet algorithme statique permet d'éviter l'inconvénient des surcoûts liés à la création des tâches en les transformant en appel de fonctions ; cependant, l'algorithme exécuté restant l'algorithme parallèle au grain fixé, la dégradation du taux de compression subsiste. Dans un tel cas, il est donc primordial que la granularité de la parallélisation soit adapatable dynamiquement pour tirer au mieux parti, par exemple d'une grappe dont le nombre de processeurs est variable.

4.1.2. *Granularité adaptative*

En appliquant l'algorithme adaptatif que nous avons proposé, l'extraction du parallélisme (ici correspondant à la découpe du fichier à compresser en blocs) sera adaptée en fonction de l'inactivité des processeurs. L'algorithme suit le schéma présenté dans §3. Un processeur commence la compression séquentielle du fichier initial tout entier, page par page en faisant progresser la fenêtre de compression. Si un autre processeur est inactif, il viendra aider le premier processeur en lui volant (par exemple) la moitié du travail qui lui reste. Dans ce cas le reste du fichier sera divisé en deux blocs. Le processeur victime continue l'exécution de `gzip` sur la partie du fichier qui lui reste, tandis que le processeur voleur applique `gzip` sur la partie volée. Ce processus sera répété sur chaque processeur actif en cas d'inactivité d'autres processeurs.

4.1.3. *Expérimentation*

Lors d'une exécution distribuée, l'implémentation de cette parallélisation peut conduire à des mauvaises performances à cause du surcoût de copie des fichiers temporaires. Pour limiter ce surcoût, le fichier a été projeté en mémoire (*mmap*). Cela a nécessité la modification de l'initialisation dans le code source de *gzip* ; ainsi, une tâche est définie par une région à compresser dans le fichier. L'implémentation a été réalisée avec le code source de gzip et l'interface applicative de programmation paral-

lèle Athapascan-1 [GAU 03].

Taux de compression. Un facteur important à analyser dans cette étude est le taux de compression. Le nombre d'octets d'un fichier compressé par les versions parallèles doit être proche de celui obtenu par une compression séquentielle. Le tableau 1 donne les résultats des compressions de fichiers de différentes tailles. Les résultats expérimentaux montrent que sur les fichiers testés, les taux de compression des versions parallèles et sequentielles sont très proches.

Taille des fichiers en nombre d'octets	gzip	gzip adaptatif en fonction du nombre de processus		
		2	8	16
867268	272573	275692	280660	280660
5238292	1023558	1027057	1053338	1084160
9389308	6599839	6622849	6737923	6793216
10015140	1125516	1130042	1143827	1166000

Tableau 1. *Taux de compression*

L'algorithme adaptatif génère un nombre de blocs qui augmente avec le nombre de processeurs. Comme les compressions des différents blocs sont indépendantes, le taux de compression diminue lorsque le nombre processeurs augmente.

Comparaison des temps d'exécution. Un autre facteur à étudier est le temps de compression. Les résultats présentés dans le tableau 2 sont obtenus en comprimant avec p processeurs des fichiers de différentes tailles. Les temps d'exécution obtenus montrent que le travail est bien partagé entre les processeurs et que l'accélération avec la taille des fichiers car le grain de calcul augmente. Ceci est dû au comportement dynamique de notre algorithme qui est basé sur la charge des processeurs et non sur la taille des blocs à compresser, contrairement à une exécution avec découpe statique qui est basée sur la taille des blocs à compresser et non sur le contenu des blocs. En effet, deux blocs de même taille peuvent requérir, selon leur contenu, des temps de compression très différents et difficilement prédictibles.

Taille des fichiers en nombre d'octets	temps(s) gzip adaptatif en fonction du nombre de processus et (accélération / efficacité) par rapport à deux processus					
	2		8		16	
5238292	3,35	(1/1)	0,96	(3.50/0,87)	0,55	(6,09/0,76)
9389308	7,67	(1/1)	2,11	(3,63/0,91)	1,15	(6,67/0,83)
10015140	6,79	(1/1)	1,71	(3,97/0,99)	0,88	(7,72/0,96)

Tableau 2. *Temps d'exécution en secondes sur architecture à base de PII 800Mhz*

4.2. *PL*

PL est une bibliothèque composée d'un moteur d'inférence probabiliste pour du calcul bayésien. L'algorithme de calcul consiste en l'évaluation d'une fonction réelle sur chaque nœud d'un arbre. Pour un nœud $node$, la fonction F_{node} est de la forme : $F_{node}(arg) = \sum_{i \in D} \prod_j F_{child_j}(arg_{i,j})$. Où D est un domaine entier et $child_j$ le j-ème nœud fils du nœud considéré. La fonction est donc appelée de manière récursive sur chacun des nœuds de l'arbre, puis la somme et le produit sont calculés. Les opérations arithmétiques sont des opérations sur des nombres réels à virgule flottante.

4.2.1. *Parallélisation à grain statique*

La parallélisation du calcul consiste à décomposer la somme sur le domaine D en k sommes sur des sous-domaines $D_1, ..., D_k$ qui seront évaluées en parallèle. Le résultat final sera calculé en sommant les différents résultats sur chaque sous-domaine. De même que pour `gzip`, le choix du grain (*i.e.* la taille de chacun des sous-domaines) est problématique au niveau du temps d'exécution :

1) la décomposition d'un domaine est une opération relativement coûteuse qui incite à utiliser peu de sous-domaines et donc un grain important de parallélisation.

2) Le temps de calcul dépend non seulement de la taille des sous-domaines mais aussi de leurs contenu, de la structure du sous-arbre associé au nœud considéré, et de l'instance du calcul effectué. Cette non-prédicabilité, due à la nature aléatoire des calculs, motive le choix d'un grain fin de parallélisation.

À la différence de `gzip`, le résultat final ne dépend pas du grain de parallélisation.

La dégénération séquentielle appliquée à cet algorithme permet d'éviter le surcoût dû à la création des tâches mais ne permet pas de supprimer le surcoût dû aux décompositions des domaines en sous-domaines car l'algorithme exécuté reste l'algorithme parallèle. Dans un tel cas, il est primordial que le choix de décomposition s'adapte dynamiquement en fonction de l'activité des processeurs.

4.2.2. *Granularité adaptative*

Dans PL, le maintien de la cohérence sur les descripteurs de travaux peut difficilement faire l'impasse sur l'utilisation de verrou pour garantir un accès en exclusion mutuelle. En effet, la relative complexité de l'opération de décomposition d'un domaine rend difficile l'utilisation d'un protocole adapté et optimisé. De plus, la taille des sections critiques dans le code est petite : le surcoût d'une opération sur un verrou n'est pas négligeable vis-à-vis du coût arithmétique. Nous avons décidé d'amortir ce surcoût en augmentant le grain du maintien de cohérence par un parcours du domaine D de sommation par bloc. Cet algorithme est proche de l'utilisation de la technique de dégénération séquentielle sur l'algorithme parallèle mais avec un surcoût beaucoup plus faible car spécialisé pour l'itération par bloc.

4.2.3. *Expérimentation*

Le premier ensemble de mesures montre l'impact de cette taille de bloc pour le parcours du domaine D sur le temps d'exécution. Ensuite nous présentons les temps d'exécution obtenus sur plusieurs processeurs.

Influence du grain de maintien de cohérence. Nous avons vu dans la section 3.4 que le maintien de la cohérence des descripteurs de travaux entre le volé et le voleur induit un surcoût. Pour cela nous avons utilisé deux instances A et B du problème. La première (A) s'exécute séquentiellement en $1,55 \pm 0,002$ secondes ; la seconde (B) en $11,34 \pm 0,007$ secondes. Ces mesures ont été prises sur une machine à deux processeurs Athlon à $1,2Ghz$ fonctionnant sous Linux et elles sont données avec des intervalles de confiance fiable à 99%.

Taille du paramètre d'extraction	instance A temps	instance B temps
1	$2,43$	$18,87$
2	$2,04$	$15,27$
4	$1,82$	$13,33$
8	$1,72$	$12,32$
16	$1,66$	$11,82$
32	$1,63$	$11,57$
64	$1,62$	$11,44$
128	$1,61$	$11,39$
256	$1,61$	$11,34$

Tableau 3. *Influence de la taille du paramètre d'extraction du choix du grain de maintien de cohérence sur le temps d'exécution.*

Le tableau 3 montre le temps d'exécution de l'algorithme à grain adaptatif sur ces deux instances en fonction de la taille (du grain) du paramètre d'extraction pour le maintien de cohérence. Nous constatons, pour ce problème, que le surcoût dû aux instructions ajoutées est important vis-à-vis des calculs effectués : il existe des valeurs α petites pour lesquelles il n'est pas possible d'obtenir un temps d'exécution t inférieur à $(1+\alpha)t_{seq}$ sans trop limiter le parallélisme. Typiquement, sur l'instance B une taille de 256 bride entièrement le parallélisme : toutes les exécutions sont séquentielles. Pour une valeur de $\alpha = 0.05$, il est possible de calculer (de manière empirique) une taille de grain pour lequel $t < (1+\alpha)t_{seq}$: par exemple, cette taille vaut 40 pour l'instance A et environ 15 pour l'instance B.

Cette taille de bloc du parcours du domaine D dépend de l'instance et ne doit pas être trop importante afin de laisser un degré de parallélisme suffisant pour une exécution sur une architecture parallèle.

Comparaison des temps d'exécution sur machine à mémoire partagée. Nous avons pu utiliser 24 processeurs d'une Origin 3800 du CINES à Montpellier pour mesurer les temps d'exécution en fonction du nombre de processeurs. Après avoir mené à

bien les expériences décrites précédemment sur cette architecture, la taille du grain pour le maintien de la cohérence a été fixée à 64. La figure 6 présente l'efficacité,

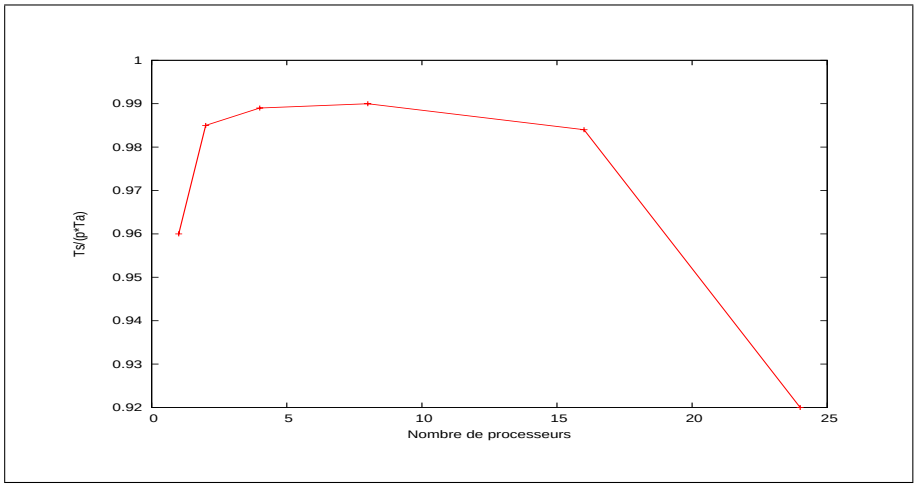

Figure 6. *Mesure d'efficacité par rapport au temps de l'algorithme séquentiel en fonction du nombre de processeurs utilisés sur une Origin 3800.*

$e = t_s/(pt_a)$, de l'algorithme à grain adaptatif (t_a) en comparaison avec l'algorithme séquentiel (t_s), sur l'instance B du problème, en faisant varier le nombre de processeurs utilisés. Ainsi pour un processeur, nous obtenons une efficacité de $0,96$ par rapport à l'algorithme séquentiel : la différence est dû au surcoût de maintien de la cohérence. Nous constatons que l'algorithme à grain adaptatif, ainsi que le moteur d'exécution d'Athapascan, sont performants jusqu'à 24 processeurs : l'efficacité est supérieure à $0,92$ (*i.e.* un facteur d'accélération de 22 sur 24) sur cet exemple. Notons que l'efficacité sur 2 processeurs est plus importante que sur 1 processeur : nous supposons qu'il s'agit d'un effet dû aux caches des processeurs. Nous expliquons la baisse relative de performances lorsque le nombre de processeurs est supérieur à 16 par le non-contrôle de l'ordonnancement système ; 24 processeurs au maximum ont été alloués sur cette machine qui en comporte 256 répartis par le système entre plusieurs applications concurrentes. Le degré potentiel important de parallélisme de l'algorithme à grain adaptatif permet donc d'assurer une très bonne efficacité dans ce contexte multi-utilisateur.

Comparaison des temps d'exécution sur machine à mémoire distribuée. La figure 7 présente l'efficacité, $e = t_s/(pt_a)$, de l'algorithme à grain adaptatif (t_a) en comparaison à l'algorithme séquentiel (t_s), sur l'instance B du problème sur sur le iCluster de l'IMAG (grappe de PC, 733Mhz, réseau ethernet à 100Mbit/s). Les mesures correspondent à des moyennes des temps d'exécution de 10 expériences. Nous constatons que l'algorithme à grain adaptatif, ainsi que le moteur d'exécution d'Athapascan, sont performants jusqu'à 100 processeurs : l'efficacité est supérieure à $0,77$

sur cet exemple. L'efficacité sur cette machine est moins bonne que sur l'Origin 3800 : en effet, chaque processeur inactif réalise une communication vers un processeur distant, cette communication est beaucoup plus coûteuse sur cette machine que l'accès à une mémoire distante sur l'Origin 3800. L'efficacité décroît donc plus rapidement mais reste appréciables sur une telle instance de petite taille ($28,4$ secondes en séquentiel sur cette architecture).

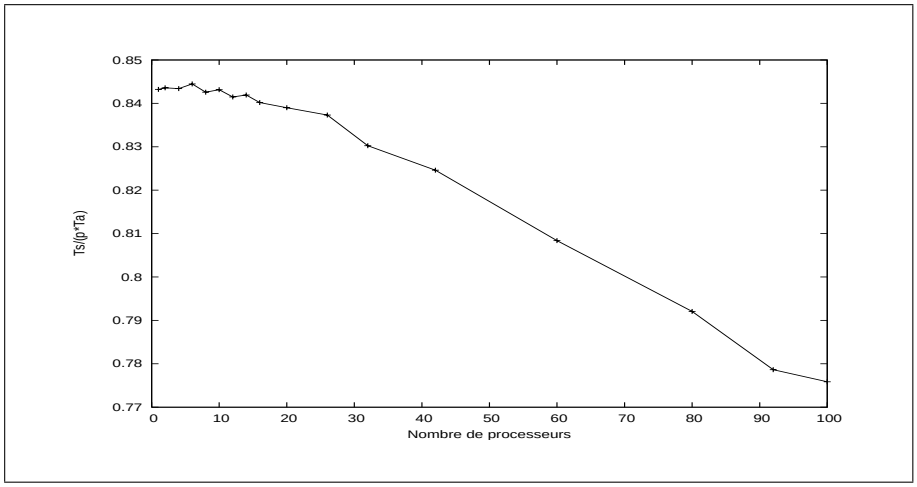

Figure 7. *Mesure d'efficacité par rapport au temps de l'algorithme séquentiel en fonction du nombre de processeurs utilisés sur une grappe de PC.*

5. Conclusion

Dans ce travail, nous avons proposé un schéma algorithmique générique original qui exploite un ordonnancement glouton des processus réalisé par le système sous-jacent pour contrôler la granularité du parallélisme en cours d'exécution. La génération de parallélisme n'est effectuée qu'en cas d'inactivité d'un processeur. Lors de l'exécution sur un nombre restreint voire variable de ressources, ce schéma permet ainsi de limiter le surcoût lié à la génération de parallélisme, sans limiter le degré de parallélisme potentiel. Nous avons appliqué ce schéma à la parallélisation du code séquentiel `gzip` qui implémente la méthode de compression de Lempel-Ziv, problème $\mathcal{P}$-complet considéré difficilement parallélisable, ainsi qu'à la parallélisation d'un moteur d'inférence probabiliste. Les expériences présentées montrent l'intérêt de cette approche algorithmique.

Les résultats expérimentaux obtenus sur grappe de PCs et sur machine à mémoire partagée corroborent les résultats théoriques en terme de performance. Dans le cas distribué, l'extraction d'une partie du travail est réalisée par l'exécution, sur interrup-

tion, d'un service à distance chez le processeur volé ; la réalisation de ce mécanisme se base sur l'utilisation d'une bibliothèque de multiprogrammation légère et de communication qui implémente le concept de « message actif ».

Nous envisageons d'étendre l'utilisation de ce schéma algorithmique à grain adaptatif pour l'exploitation d'une grille de grappes dans lesquelles les ressources sont hétérogènes ; le domaine d'application visé est celui de la réalité virtuelle et de la simulation interactive. Dans ce contexte, il est en effet important de pouvoir réagir à l'inactivité des ressources afin de pouvoir répondre à des critères de qualité tant numérique (précision des résultats) que visuel (fluidité) désirés par l'utilisateur final.

El Mostafa Daoudi *est professeur à la Faculté des Sciences d'Oujda, et responsable du laboratoire de recherche en informatique. Ses travaux portent sur l'étude et l'implémentation d'algorithmes parallèles.*

Thierry Gautier *est docteur ingénieur de l'Institut National Polytechnique de Grenoble en mathématiques appliqués. Il est actuellement chargé de recherches à l'INRIA dans le projet MOAIS. Ses activités de recherche concernent le couplage de code et les intergiciels pour le calcul sur grille.*

Aicha Kerfali *est doctorante sous la direction de Pr. El Mostafa Daoudi au laboratoire de recherche en informatique de la faculté des sciences d'Oujda au Maroc*

Rémi Revire *est docteur de l'Institut National Polytechnique de Grenoble en informatique. Il est actuellement ingénieur à Bull. Ses activités concernent l'évalutation des performances de machines parallèles de type grappes de SMP.*

Jean-Louis Roch *est docteur ingénieur de l'Institut National Polytechnique de Grenoble en mathématiques appliqués. Il est actuellement maître de conférences et chef du projet MOAIS. Ses activités de recherche concernent l'algorithmique parallèles ainsi que la certification de calcul sur grille.*

Remerciements

Ce travail est soutenu par le "Comité Mixte Franco-Marocain", AI N° MA/01/19.

6. Bibliographie

[BER 03] BERENBRINK P., FRIEDETZKY T., GOLDBERG L. A., « The Natural Work-Stealing Algorithm is Stable », *SIAM Journal on Computing*, vol. 32, n° 5, 2003, p. 1260-1279.

[BIR 87] BIRD R., « *Logic of Programming and Calculi of Discrete Design* », chapitre Introduction to the Theory of Lists, Springer-Verlag, 1987.

[BLU 98] BLUMOFE R. D., LEISERSON C. E., « Space-efficient scheduling of multithreaded computations », *SIAM Journal on Computing*, vol. 27, n° 1, 1998, p. 202–229.

[COL 95] COLE M., « Parallel programming with list homomorphisms », *Parallel Processing Letters*, vol. 5, n° 2, 1995, p. 191–204.

[COS 89] COSNARD M., TRYSTRAM D., « Communication Complexity of Gaussian Elimination on MIMD Shared Memory Computers », *Revista de Matemáticas Aplicadas*, vol. 10, n° 1, 1989, p. 27–48.

[FRI 98] FRIGO M., LEISERSON C. E., RANDALL K. H., « The Implementation of the Cilk-5 Multithreaded Language », *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, juin1998.

[GAI 03] GAILLY J.-L., « Gzip », rapport n° `http ://www.gzip.org`, 2003, GNU Project.

[GAL 98] GALILÉE F., ROCH J.-L., CAVALHEIRO G., DOREILLE M., « Athapascan-1 : Online Building Data Flow Graph in a Parallel Language », IEEE, Ed., *International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, Paris, France, octobre1998, p. 88–95.

[GAU 03] GAUTIER T., REVIRE R., ROCH J.-L., « Athapascan : API for Asynchronous Parallel Programming », rapport n° RR-0276, February 2003, INRIA Rhône-Alpes, projet APACHE.

[GRA 69] GRAHAM R., « Bounds on multiprocessing timing anomalies », *SIAM J. Appl. Math.*, vol. 17, n° 2, 1969, p. 416–426.

[GRE 95] GREENLAW R., HOOVER J., RUZZO W., *Limits to Parallel Computation : P-Completeness Theory*, Oxford University Press, 1995.

[JAJ 92] JAJÁ J., *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, Massachussets, 1992.

[LAD 80] LADNER R. E., FISCHER M. J., « Parallel prefix computation », *Journal of the ACM*, vol. 27, n° 4, 1980, p. 831–838.

[NEL 95] NELSON M., LOUP GAILLY J., *The data compression book*, M&T Books, New York, NY, 1995.

[PRO] PROBAYES, « `http ://www.probayes.com` ».

[RAY 90] RAYWARD-SMITH V., BURTON F. W., FUJIMOTO R., JANACEK G., « Load balancing strategies for the implementation of parallel programs », rapport, 1990, Simon Fraser University Center for System Science, Vancouver, Canada.

[ROC 95] ROCH J.-L., VILLARD G., ROUCAIROL C., « Algorithmes irréguliers et ordonnancement », AUTHIÉ G., AL., Eds., *Parallèlisme et applications irrégulières*, chapitre 4, p. 71-84, Hermès, 1995.

[ROC 01] ROCH J.-L., « Ordonnancement de programmes parallèles sur grappes : théorie versus pratique. », *Actes du Congrès International ALA 2001, Université Mohamm V*, Rabat, Maroc, mai2001, p. 131–144.

Annexe pour le service de fabrication

Article pour la revue :
Technique et science informatiques

Auteurs :
El Mostafa Daoudi[*], — *Thierry Gautier*[**], — *Aicha Kerfali*[*], — *Rémi Revire*[**] — *Jean-Louis Roch*[**]

Titre de l'article :
Algorithmes parallèles à grain adaptatif et applications

Titre abrégé :
Algorithmes à grain adaptatif

Traduction du titre :

Date de cette version :
10 novembre 2004

Coordonnées des auteurs :

– téléphone :
– télécopie :
– Email :

Logiciel utilisé pour la préparation de cet article :
LaTeX, avec le fichier de style `article-hermes.cls`,
version 1.10 du 17/09/2001.

Formulaire de copyright :
Joindre le formulaire de copyright signé, récupéré sur le web à l'adresse
`http://www.hermes-science.com`