RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Algorithmes parallèles à grain adaptatif – Application à la parallélisation de *gzip*

Aicha Kerfali*, Jean-Louis Roch**, El Mostafa Daoudi*

* 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
38330 Montbonnot Saint-Martin
Jean-Louis.Roch@imag.fr

Résumé

Nous proposons un schéma algorithmique générique original qui exploite un ordonnancement glouton des processus réalisé par le système sous-jascent pour contrôler la granularité du parallélisme en cours d'exécution. Le 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 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. Il est particulièrement adapté à la résolution de problèmes pour lesquels la parallélisation entraîne malgré un gain de temps une pénalité en nombre d'opérations ou en performance ; nous appliquons 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.

Mots-clés : algorithmique, parallélisme, ordonnancement glouton, compression

---

1. Introduction

Pour être portable, un programme parallèle doit faire abstraction de l'architecture sous-jascent (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 fixe la granularité des calculs (un grain de calcul est dans la suite appelé *tâche*) et la granularité des données. Deux points de vue antagonistes sont considérés pour 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 grappes en contexte multi-utilisateur ou, plus encore, sur grille de machines. D'un point de vue algorithmique théorique [6], un objectif est de 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*. L'inconvénient d'une telle approche est que, lors d'une exécution effective sur un nombre toujours restreint de ressources, le contrôle, par le noyau d'exécution, de cet ordonnancement de grain fin sur les ressources entraine un surcoût.

Pour limiter ce surcoût, les techniques dites de vol de travail [8, 1] consistent à privilégier l'exécution séquentielle efficace de l'algorithme parallèle dans le cas où tous les processeurs sont actifs ; on parle de *dégénération séquentielle* [9]. Cependant, si elle diminue le surcôut d'ordonnancement de l'algorithme parallèle, ces techniques ne s'attaquent pas au surcoût arithmétique liée à la parallélisation, qui peut s'avérer très coûteuse en comparaison à un algorithme séquentiel optimisé.

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. Cette notion de grain adaptatif apparait dans [2] 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 (§4). Enfin, nous étudions son application à une parallélisation effective de `gzip` (§5) indépendante du nombre de ressources disponibles et de leur puissance. Les premières expérimentations (§6) permettent d'évaluer les performances de la parallélisation originale ainsi obtenue, en termes de temps parallèle et taux de compression par rapport au programme séquentiel [4].

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, de coût séquentiel optimisé T_{seq}. On suppose que le problème peut être résolu par un algorithme parallèle à grain fin de type découpe récursive. Nous rappelons tout d'abord les notations utilisées pour les coûts illustrées sur l'exemple du produit itéré 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 T_{seq}.

Notations de coût – Exemple du produit itéré Dans toute la suite, on utilise les notations de coût proposées dans [1] : 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 [5, 1] 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, τ_{op} désigne 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 [10] 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 (fig. 1-l.5) a un coût $T_{seq}(n) = n.\tau_\star$. Une découpe récursive du problème en deux permet une parallélisation à grain fin (figure 1) 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.

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 16 est alors réécrite comme suit :

```
16      if ( (j - i) <= grain ) { tmp = f(i) ; for (int k=i+1 ; k<=j ; k++) tmp = star(tmp, f(k)) ; r.write( tmp ) ; }
```

On a alors $T_\infty(n) = g + \log \frac{n}{g}$ 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 d'obtenir asymptotiquement $T_\infty(n) = \Theta(\log n)$ et $T_1(n) = T_{seq}(n) + O\left(\frac{n}{\log n}\right)$ soit $T_1(n) \simeq T_{seq}(n)$.

On obtient ainsi un algorithme parallèle qui a asymptotiquement le même temps sur une infinité de

Algorithme 1 Programmation du produit itéré en Athapascan

```
Elt f( int i ) { ... } ;

void SeqIteratedProduct { // algorithme sequentiel
   void operator()( int i, int j, Shared_w< Elt > r ) {
      for (Elt res = f(i), int k=i ; k <=j ; k++) res = res * f(k) ;
      r.write ( res ) ;
} } ;

struct Product { // produit binaire sequentiel
   void operator()( Shared_w< Elt > r2, Shared_w< Elt > r2, Shared_w< Elt > r ) {
      r.write ( r1.read() * r2.read() ;
} } ;

 struct IteratedProduct { // algorithme parallele
   void operator()( int i, int j, Shared_w<Element> r ) {
      if (i == j) { r.write( f(i) ) ; }
      else {
         Shared<Element> r1, r2 ;
         Fork< IteratedProduct > () (i, (i+j)/2, r1 ) ;
         Fork< IteratedProduct > () ((i+j)/2+1, j, r2 ) ;
         Fork< Product > () (r1, r2, r ) ;
 } } } ;

 Fork < IteratedProduct >() (0, n, prod ) // Appel principal
```

processeurs que 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.
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. Dans le paragraphe suivant, nous présentons la solution de dégénération séquentielle pour limiter ce surcoût de génération du parallélisme.

Seuil statique et dégénération séquentielle de l'algorithme parallèle Pour minimiser le surcoût de création de tâches parallèles, la technique appelé dégénération séquentielle [9] ou *work first principle* [1] 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 [3].
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 temps d'exécution T_p de l'algorithme parallèle exécuté sur p processeurs est majoré par [1] $\frac{T_1}{p} + O(p.T_\infty)$. 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 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 algorithme parallèle à grain

[1] Cette technique préemptive est appelé *dégénération séquentielle passive* dans [9] où est présentée une alternative non-préemptive, appelée *dégénération séquentielle active*.

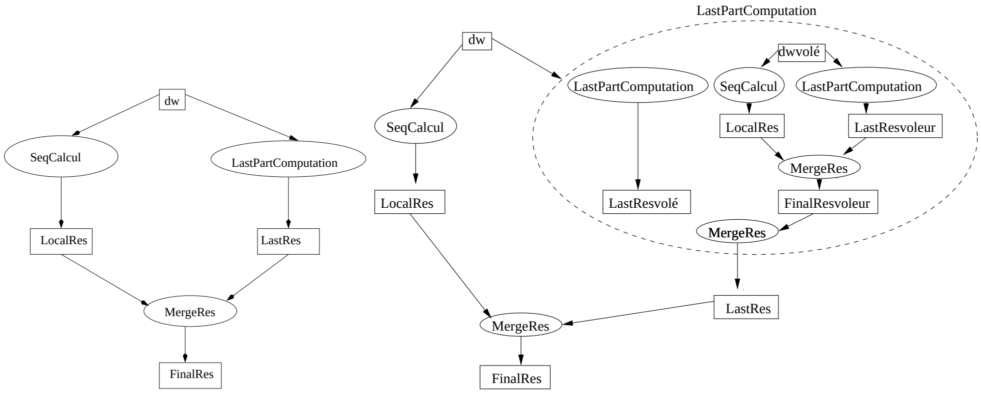

FIG. 1 – Comportement dynamique de l'algorithme à grain adaptatif

fin de type découpe récursive. Parmi les problèmes dans ce cadre figurent les algorithmes séquentiels classiques tels le produit itéré, les préfixes ou la recherche arborescente profondeur d'abord ; mais aussi des problèmes tels la compression.

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 chez un processeur actif un algorithme séquentiel, mais en donnant la possibilité d'extraire du parallélisme sur le travail restant 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*).

Algorithme 2 Algorithme à grain adaptatif

côté volé

// exécution par défaut de l'algorithme séquentiel sur le descripteur de travail dw
if (dw.is_not_completed()) { *// Il reste du travail à faire*
// LastPartComputation implémente l'algorithme parallèle ; elle permet l'extraction de parallélisme de l'algorithme séquentiel
Fork< LastPartComputation >(dw, LastRes) ;
// SeqCalcul est l'algorithme séquentiel sur le travail dw
LocalRes = SeqCalcul (dw) ;
// MergeRes synchronise les deux algorithmes LastPartComputation et SeqCalcul pour la collection des résultats
Fork< MergeRes >(LocalRes, LastRes, FinalRes) ;
}

côté voleur

// Extraction du parallélisme et lancement d'un nouvel algorithme à grain adaptatif
// ExtractPar modifie le travail dw de l'algorithme séquentiel volé et crée un nouveau travail ndw
ndw = ExtractPar(dw) ;
// création d'une nouvelle tâche LastPartComputation d'extraction de parallélisme sur l'ancien travail
Fork< LastPartComputation >()(dw, LastResVole) ;
// création d'une tâche d'extraction LastPartComputation d'extraction de parallélisme sur le nouveau travail
Fork< LastPartComputation >()(ndw, LastResVoleur) ;
// exécution séquentielle sur le travail ndw
LocalRes = SeqCalcul(ndw) ;
// Collecte des résultats (voir figure 1)
Fork< MergeRes >()(LocalRes, LastResVoleur, FinalResVoleur) ;
Fork< MergeRes >()(LastResVole, LastResVoleur, LastRes) ;

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.

L'algorithme, dit à grain adaptatif, suppose que le système sous-jascent 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. La solution que nous proposons dans l'algorithme 2 se déroule comme suit (cf figure 1) :

- 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 *LastPart-Computation* prête à être volée :
  - Il extrait une partie (par exemple la moitié) du travail qui est à la charge du processeur actif (appelé *victime* ou *volé*). Pour cela, il modifie le travail du processeur victime et construit un nouveau travail à effectuer, en utilisant les informations minimales de la tâche *LastPartComputation*.
  - Puis il crée deux tâches de vol (prêtes à être volées) : une tâche concernant la partie du travail qu'il a prise en charge et une autre concernant la partie restante du travail pour le processeur victime.
  - Après la création de ces deux tâches, le processeur voleur applique l'algorithme séquentiel à la partie qu'il a volé du processeur victime.

Ce processus se répète d'une façon récursive sur tous les travaux créés jusqu'à la terminaison. Dans le cas d'une architecture à mémoire partagée, le voleur accède directement à la mémoire partagée pour extraire une partie du travail en cours. Pour maintenir la cohérence des données, l'accès aux données partagées doit être protégé, par exemple par des verrous ou des protocoles adaptés à l'instar de THE [3].

4. 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 comment ces inconvénients peuvent être évités en appliquant l'algorithme que nous avons proposé dans la section précédente.

L'application de l'algorithme est illustré sur la figure 2 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*). Initialement, le travail total (*first*=0 et *last*=100) est assigné au processeur P_0 qui démarre l'exécution de l'algorithme séquentiel. Cette évaluation séquentielle produit un resultat 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 restant (ici *first*=52 et *last=100*) et laisse à P_0 l'autre moitié du travail (ici *first*=3 et *last*=51).

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 :

- si il y a 1 seul processeur, le temps d'exécution est $T_1(n) = T_{seq}(n) + 2(\tau_{Fork} + \tau_{shared})$: deux tâches sont créées, la deuxième étant vide (cette tâche est créée pour indiquer la possibilité de vol).
- si il y a une infinité de processeurs inactifs, on a encore $T_\infty(n) = \Theta(\log_n)$ et $T_1(n) = T_{seq}(n) + 2n(\tau_{Fork} + \tau_{shared})$: on n'a pas perdu 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 $p.T_\infty = O(p \log p)$ créations de tâches. On a alors $T_1(n) = T_{seq}(n) + 2p \log p(\tau_{Fork} + \tau_{shared})$ et le temps d'exécution $T_p(n)$ sur les p processeurs est majoré par $T_1(n)/p + O(\log p)$.

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

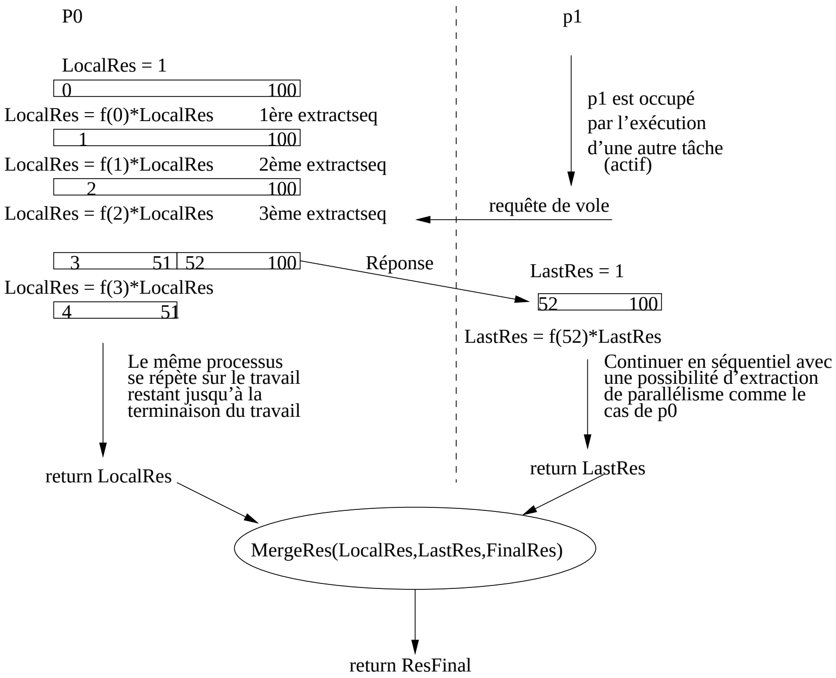

FIG. 2 – Exécution du produit itéré à grain adaptatif pour $n = 100$ sur 2 processeurs.

que l'algorithme parallèle, comme c'est le cas pour `gzip` étudié dans la section suivante.

5. Application à la parallélistion de GZIP

`gzip` [4] est un utilitaire fréquemment utilisé pour le traitement de données informatique et demandant un temps de traitement relativement important (jusqu'à 3s pour un fichier de 5 Mo). L'algorithme implanté par `gzip` est une variation de Lempel-Ziv LZ77 [7]. L'algorithme est basé sur une analyse du fichier utilisant une fenêtre de 32 Kbytes ; 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 dans `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. Cet algorithme, de coût linéaire de la taille du fichier, est P-complet 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 parallélistions de gzip peuvent être décompressés par l'utilitaire séquentiel `gunzip` classique.

5.1. 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 comprimé indépendamment. L'inconvénient est que cette découpe entraîne une diminution du taux de compression en comparaison à 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.

5.2. 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.

6. Expérimentation

L'implémentation de cette parallélisation peut conduire à des mauvaises performances à cause du coût de la copie des fichiers temporaires et des communications pour une exécution distribuée. Pour remédier à ces deux problèmes, 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 parallèle Athapascan-1.
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 à peu près le même que celui obtenu par une compression séquentielle. Nous avons vérifié ce point et nous pouvons noter que sur les fichiers testés, les taux de compression des versions parallèles et sequentielles sont très proches. Le tableaux ci dessous, donne les résultats des compressions de fichiers de différentes tailles.

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

Table1 : *Taux de compression*

Comparaison des temps d'exécution Un autre facteur à étudier est le temps de compression. Les résultats préliminaires présentés dans le tableau ci-dessous sont obtenus en comprimant avec p processus des fichiers de différentes tailles ; les temps d'exécutions obtenus montrent que le travail est bien partagé entre les processus. 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		
	2	8	16
5238292	3.35	0.96	0.55
9389308	7.67	2.11	1.15
10015140	6.79	1.71	0.88

Table 1 : *Temps d'exécution en seconds*

7. Conclusion

Dans ce travail, nous avons proposé un schéma algorithmique générique original qui exploite un ordonnancement glouton des processus réalisé par le système sous-jascent 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.

Les résultats expérimentaux préliminaires pour une exécution SMP corroborent les résultats théoriques en termes de taux de compression et de temps d'exécution parallèle. Une implémentation dans le cas distribué est en cours sur une grappe de PCs. Dans ce cas, l'extraction d'une partie du travail est réalisée par l'exécution d'un service à distance chez le processeur volé.

Bibliographie

1. Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1) :202–229, 1998.
2. Michel Cosnard and Denis Trystram. Communication Complexity of Gaussian Elimination on MIMD Shared Memory Computers. *Revista de Matemáticas Aplicadas* , 10(1) :27–48, 1989.
3. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, June 1998.
4. Jean-Loup Gailly. Gzip. Technical Report `http ://www.gzip.org`, GNU Project, 2003.
5. R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2) :416–426, 1969.
6. J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Massachussets, 1992.
7. Mark Nelson and Jean loup Gailly. *The data compression book*. M&T Books, New York, NY, 1995.
8. V.J. Rayward-Smith, F. W. Burton, R.M. Fujimoto, and G.J. Janacek. Load balancing strategies for the implementation of parallel programs. Technical report, Simon Fraser University Center for System Science, Vancouver, Canada, 1990.
9. Jean-Louis Roch. Ordonnancement de programmes parallèles sur grappes : théorie versus pratique. In *Actes du Congrès International ALA 2001, Université Mohamm V*, pages 131–144, Rabat, Maroc, 28–31 Mai 2001.
10. Jean-Louis Roch, Gilles Villard, and Catherine Roucairol. Algorithmes irréguliers et ordonnancement. In Gérard Authié and al., editors, *Parallèlisme et applications irrégulières*, chapter 4, pages 71–84. Hermès, 1995.