Un algorithme adaptatif optimal pour le calcul parallèle des préfixes

Jean Louis Roch, Daouda Traoré†

Equipe MOAIS (CNRS-INRIA-INPG-UJF)
Laboratoire d'Informatique de Grenoble (LIG)
51, Av. Jean Kuntzman, 38330 Montbonnot, France
Jean-Louis.Roch@imag.fr, daouda.traore@imag.fr
† *Ce travail est co-financé par une bourse France-Mali.*

RÉSUMÉ. Dans cet article, nous proposons un nouvel algorithme parallèle de calcul des préfixes pour des processeurs dont la vitesse ou le nombre peut varier en cours d'exécution. Basé sur le couplage récursif d'un algorithme séquentiel optimal et d'un algorithme parallèle non optimal mais récursif à grain fin, il exploite un ordonnancement dynamique de type vol de travail (Cilk, Kaapi). Sa performance théorique est analysée sur p processeurs à vitesses variables, de vitesse moyenne Π_{ave}. Bien que cet algorithme adaptatif est indépendant du nombre de processeurs, son temps d'exécution est équivalent à $\frac{2n}{\Pi_{ave}\cdot(p+1)}$, ce qui est optimal si les processeurs sont identiques (i.e. $\Pi_{ave} = 1$). Expérimentalement, cet algorithme adaptatif est comparé à un algorithme optimal pour un nombre fixé p de processeurs identiques avec ordonnancement statique optimal sur une machine SMP octo-processeurs. Même pour des petites valeurs de n (100) et de p (de 1 à 8), ses performances sont analogues dans le cas où les processeurs sont dédiées au calcul, et il est beaucoup plus rapide dans le cas général où la machine exécute concurremment d'autres processus (cas multi-utilisateurs).

ABSTRACT. In this paper, we propose a new parallel prefix computes algorithm for processors which number or speed may vary during execution. Based on the coupling of two algorithms,– one sequential, the other one parallel and fine grain –, it involves an on-line workstealing scheduling. Its theoretical perfomance is analyzed in both on p processors with variable speed, the average speed being Π_{ave}. While this adaptive algorithm is independent from the number p of processors, its execution time is equivalent to $\frac{2n}{\Pi_{ave}\cdot(p+1)}$, which is optimal when processors are identical (i.e. $\Pi_{ave} = 1$). It has been experimented on an octo-processor SMP machine and compared to an optimal algorithm for a fixed number p of processors with an optimal off-line scheduling. Even for small values of n (100) and of p (1 to 8), its performances are analoguous when the machine is dedicated to the computation, and it is faster when the machine concurrently executes other processes (multiuser case).

MOTS-CLÉS : algorithme parallèle, parallélisme, ordonnancement dynamique, vol de travail

KEYWORDS : parallel algorithm, parallelism, online scheduling, workstealing

1. Introduction

Etant donnés $x_0, x_1, \ldots x_n$, les n préfixes π_k, pour $1 \leq k \leq n$, sont définis par $\pi_k = x_0 \star x_1 \star \ldots \star x_k$ où $\star$ est une loi associative.

Le calcul des préfixes est une opération qui apparaît dans de nombreux algorithmes notamment l'évaluation de polynômes et les additions modulaires [3], le packing, la parallélisation des boucles [8].

Le calcul séquentiel itératif des préfixes requiert n opérations $\star$. Mais tout circuit parallèle de profondeur d requiert au moins $2n-d$ opérations $\star$ [7]. Le temps parallèle minimal est $\Omega(\log n)$ sur une machine sans écriture concurrente. Ladner et Fischer proposent un algorithme parallèle [7] de temps $2\log n$ avec $2n$ opérations ; Fich [4] montre que cet algorithme est asymptotiquement optimal car tout algorithme de temps $\log n$ nécessite $4n$ opérations. L'algorithme de Ladner-Fischer est à grain fin, indépendant du nombre de processeurs effectivement disponibles ; il peut être émulé sur p processeurs identiques en temps asymptotique $\frac{2n}{p} + 0(\log n)$ pour $p < \frac{n}{\log n}$ mais effectue $2n$ opérations donc n'est pas optimal.

Nicolau et Wang [9] montrent qu'une borne inférieure pour le temps parallèle sur p processeurs est supérieur à $\left\lceil \frac{2n}{p+1} \right\rceil$ pour $n \geq p(p+1)/2$; ils donnent un algorithme, basé une découpe en $(p+1)$ blocs et un pipeline entre blocs, qui atteint cette borne. Pour $n \gg p$, cette borne est aussi atteinte par l'algorithme rappelé dans la section 3 de temps $\frac{2n}{p+1} + O(p)$ et strictement optimal en nombre d'opérations. Les différentes implantations proposées, sur architectures distribuées [8] ou circuits dédiés [3], dépendent toutes du nombre p de processeurs et d'un ordonnancement statique. L'inconvénient d'un algorithme optimal pour p processeurs fixé est que le nombre d'opérations est au moins $2\frac{p}{p+1}$ fois supérieur au nombre n d'opérations de l'algorithme séquentiel.

Ces algorithmes, bien qu'optimaux sur p proceseurs, ne sont pas performants si l'on dispose d'une machine avec des processeurs différents, ou d'une machine utilisée par plusieurs utilisateurs car la charge des processeurs varie au cours de l'exécution. Pour traiter ce problème, nous proposons dans cet article un nouvel algorithme de calcul de préfixes, dit à grain adaptatif. Il repose sur l'ordonnancement par vol de travail (§2) implémenté dans Kaapi [6] et étendu au cas de processeurs à vitesses variables par Bender et Rabin [1]. La section 4 présente le nouvel algorithme à grain adaptatif et analyse sa complexité dans le cas de processeurs identiques et de vitesses variables. Dans la section 6 nous présentons des comparaisons expérimentales entre cet algorithme et un algorithme optimal sur une machine octo-processeurs, dans les deux cas processeurs dédiés et processeurs perturbés par des processus additionnels ; conformément à l'analyse théorique, l'algorithme adaptif est le plus rapide.

Le couplage proposé ici entre deux algorithmes complémentaires concurrents, l'un séquentiel et l'autre parallèle, est inspiré de [2] où il est appliqué à des algorithmes de même nombre d'opérations sur des processeurs identiques. Mais son utilisation pour des algorithmes de coûts différents (cas des préfixes) et des vitesses variables, ainsi que la technique utilisée pour analyser sa complexité sont originales. Très générales, nous pensons qu'elles peuvent s'appliquer à d'autres problèmes.

2. Notations et ordonnancement par vol de travail

Nous désignons par W_1 le nombre total d'opérations effectuées par un algorithme parallèle ; W_∞ le chemin critique en nombre d'opérations de cet algorithme ; T_p le temps d'exécution de cet algorithme ordonnancé sur p processeurs. Pour ordonnancer un programme à parallélisme récursif sur un nombre fixé de processeurs, Cilk[5] et Kaapi [6] implémentent un algorithme distribué de type vol de travail, selon le principe "Travail d'abord" (work-first principle). Chaque processeur exécute localement, selon l'ordre séquentiel, les tâches qu'il a créées. Lorsqu'un processeur P_v devient inactif (il n'a plus de tâches prêtes), il choisit (uniformément au hasard) un autre processeur P_w qui a au moins une tâche prête. La tâche prête que P_w exécuterait en dernier est alors migrée sur P_v ; on parle de vol réussi. Kaapi [6] implémente cette migration par un algorithme distribué qui minimise la synchronisation entre processeur volé et processeur voleur.
Le théorème suivant permet de donner un encadrement de T_p en fonction de W_1 et W_∞.

Théorème 2.1 *[5] Avec une grande probabilité*[1] *, le nombre de vols réussis est majoré par $O(pW_\infty)$ et le temps d'exécution T_p est majoré par*

$$T_p \leq \frac{W_1}{p} + O\left(W_\infty\right)$$

Ainsi, si W_∞ est négligeable devant W_1, cet ordonnancement à une performance asymptotique $\frac{W_1}{p}$ optimale.

Ce théorème est étendu dans [1] au cas de processeurs hétérogènes ou de vitesses différentes, non connues ou variables. La vitesse instantanée d'un processeur est mesurée en nombre d'opérations par top ; on note Π_{ave} la vitesse moyenne de l'ensemble des processeurs au cours de l'exécution. Au niveau du vol de travail, la seule modification est lorsque p_v vole du travail à un processeur actif p_w : si p_w est en cours d'exécution mais n'a pas de travail prêt à être volé et si p_v est beaucoup plus plus rapide que p_w (par exemple au moins deux fois plus rapide) alors la tâche en cours d'exécution sur p_w est préemptée et migrée sur p_v. Le temps d'exécution T_p est alors majoré par $T_p \leq \frac{W_1}{p.\Pi_{ave}} + O\left(\frac{\beta.W_\infty}{\Pi_{ave}}\right)$.

3. Algorithme parallèle sur p processeurs identiques

Dans ce paragraphe, nous rappelons un algorithme parallèle pour le calcul des n préfixes $(\pi_i)_{i=1,\ldots,n}$ qui est optimal asymptotiquement sur p processeurs identiques. Il est basé sur une découpe en $p+1$ blocs $B_0, \ldots B_p$ de même taille (à un élément près) des $n+1$ éléments en entrée $(x_i)_{i=0,\ldots,n}$. Pour simplifier, on suppose que chaque bloc B_i contient $K = \frac{n}{p+1}$ éléments consécutifs.

1) **Etape 1** : En parallèle pour $i = 0, \ldots, p-1$, on calcule sur le processeur i les préfixes séquentiels du bloc B_i. Soient α_i le dernier préfixe du bloc B_i. On remarque que les préfixes $(\pi_j)_{j=1,\ldots,K}$ du bloc B_0 sont ainsi calculés.

2) **Etape 2** : On calcule les $p-1$ préfixes $\beta_0 = \alpha_0$, $\alpha_1 = \alpha_0 \star \alpha_1, \ldots, \beta_{p-1} = \beta_{p-2} \star \alpha_{p-1}$ des valeurs $\alpha_0, \ldots, \alpha_{p-1}$.

3) **Etape 3** : Sur le processeur 0, on calcule les préfixes des éléments β_{p-1}, B_p pour obtenir les préfixes $\pi_{pK}, \ldots, \pi_n$. Et, en parallèle pour $i = 1, \ldots, p-1$, on calcule

1. i.e. pour tout $c > 0$ et n assez grand, la probabilité est supérieure à $1 - n^{-c}$.

sur le processeur i le produit par β_{i-1} de chaque élément du bloc B_i ; on remarque que ces produits sont tous indépendants, même si ils sont effectués ici en séquentiel. On obtient ainsi tous les préfixes π_i.
Le temps d'exécution de cet algorithme est $2K + p - 1 \equiv \frac{2n}{p+1}$ donc asymptotiquement optimal. Son nombre d'opérations $2n-(2k+p-1)$ est strictement optimal car il atteint la borne inférieure de Fich[4]. De plus, on remarque qu'en prenant $K = 2$ et en exécutant de manière récursive l'étape 2, on retrouve l'algorithme de Ladner et Fisher [7] qui effectue $W_1 = 2n$ opérations $\star$ avec un chemin critique $W_\infty = 2\log_2 n$.

Cependant, cet algorithme n'est pas adapté si les vitesses des processeurs ne sont pas identiques ou si le temps d'une opération varie. Le temps d'exécution de l'algorithme sera alors celui du processeur le plus lent. Dans la section suivante nous présentons un algorithme adaptatif qui atteint aussi un temps et un nombre d'opérations asymptotiquement optimal sur p processeurs mais qui s'adapte automatiquement à la vitesse des processeurs par vol de travail.

4. Algorithme parallèle à grain adaptatif

Notre algorithme parallèle à grain adaptatif est basé sur le couplage d'une part d'un processus séquentiel P_s qui calcule séquentiellement des préfixes et d'autre part d'une variante de l'algorithme parallèle précédent, mais à grain fin, qui est ordonnancé par vol de travail sur $p-1$ autres processus P_v. Chacun de ces p processus est placé sur un processeur.

Initialement, le processus P_s démarre le calcul des préfixes de 1 à n. Soient $a = \frac{n}{p+1}$ et $b = \frac{p}{p+1}n$; les préfixes de 1 à a et de b à n seront calculés par ce processus P_s. Mais l'intervalle d'indices $[a, b]$ peut être volé et découpé récursivement par les processus P_v devenus inactifs. Les algorithmes pour P_s et les processus P_v sont les suivants :

• **Algorithme séquentiel sur un processus** P_s :

1) P_s calcule séquentiellement les préfixes à partir de l'indice 1 (i.e. π_1), jusqu'à atteindre un indice u_1 tel que l'intervalle $[u_1, u_2]$ d'indices a été volé par un processeur P_v.

2) P_s préempte alors P_v et récupère le dernier indice $k \leq u_2$ calculé par P_v, qui a donc déjà calculé $r_{u_1} = x_{u_1}, r_{u+1} = r_{u_1} \star x_{u_1+1}, \ldots, r_k = r_{k-1} \star x_k$. P_s envoie la valeur π_{u_1-1} à P_v et redémarre P_v (voir ci-dessous).

3) P_s calcule $\pi_k = \pi_{u_1-1} \star r_k$; puis il reprend le calcul séquentiel des préfixes de $k+1$ à n à partir de $k+1$ en revenant à l'étape 1. On parle d'opération de saut ; pour chaque opération de saut, P_s effectue une opération $\star$.

4) P_s s'arrête lorsqu'il a calculé π_n (les préfixes d'indices de b à n ne peuvent pas être volés). Après avoir calculé π_n, il devient un processus voleur et exécute l'algorithme P_v.

• **Algorithme parallèle sur** $p-1$ **processus** P_v :

– Lorqu'il est préempté par P_s (voir algorithme P_s), P_v a déjà calculé localement des préfixes partiels $r_{u_1}, \ldots, r_{u_k}$ de l'intervalle $[u_1, u_k]$. Il recçoit alors la valeur du dernier préfixe $\beta = \pi_{u_1-1}$ calculée par P_s. Il finalise alors l'intervalle $[u_1, u_k]$ en calculant les produits $\pi_i = \beta \star r_i$ pour $u_1 \leq i \leq u_k$. Ces produits sont parallèles ; sur inactivité d'un autre processus voleur, une moitié de ces calculs restant à faire sur P_v dans cet intervalle peut alors être volée.

– Lorsqu'il est inactif, le processus P_v choisit au hasard un processeur jusqu'à trouver un processeur actif P_w. Il peut s'agir soit de P_s, soit d'un autre processus voleur. Si la victime est P_s, le vol n'est possible que si P_s possède un intervalle d'indices compris entre a et b restant à calculer.

1) P_v découpe l'intervalle restant à calculer et volable sur P_w en deux parties ; il extrait la partie droite $[u_1, u_2]$ de l'intervalle et la vole. La partie gauche reste en cours de calcul sur P_w.

2) P_v démarre le calcul sur l'intervalle volé $[u_1, u_2]$. Il peut s'agir : soit d'un calcul de préfixes locaux (i.e. $r_{u_1} = x_{u_1}, r_{u_1+1} = r_{u_1} \star x_{u_1+1}, \ldots$) ; soit de la finalisation de calculs de préfixes à partir de valeurs r_k déja calculées (i.e. $\pi_{u_1+1} = \pi_{u_1} \star r_{u_1+1}, \pi_{u_1+2} = \pi_{u_1} \star r_{u_1+2}, \ldots$).
Le programme s'arrête lorsque tous les processeurs sont inactifs. L'intérêt de cet algorithme est qu'un processeur devenu lent sera soit préempté par le processus séquentiel, soit volé par un processus parallèle. La section suivante analyse la complexité de cet algorithme adaptatif.

5. Optimalité asymptotique de l'algorithme adaptatif

Le théorème suivant montre que l'algorithme de préfixe à grain adaptatif précédent est asymptotiquement optimal sur p processeurs identiques.

Théorème 5.1 *Soit T_p le temps du calcul de n préfixes par l'algorithme adaptatif sur p processeurs identiques. Alors, avec une grande probabilité :*

$$T_p \leq \frac{2n}{p+1} + O(\log n).$$

Preuve. On analyse l'exécution en la découpant en 2 phases.

– **phase 1** : Jusqu'à ce que le processeur P_s qui exécute la partie séquentielle de l'algorithme adaptatif termine avec le calcul du dernier préfixe π_n.

– **phase 2** : Après la phase 1 et jusqu'à la fin du calcul.
Soit n_{seq} (resp. j) le nombre de préfixes (resp. sauts) calculés de manière séquentielle par P_s dans la phase 1. Soit x (resp. y) le nombre de préfixes finaux calculés par les processus P_v (algorithme parallèle), dans la phase 1 (resp. phase 2). On a $n = n_{seq} + j + x + y$. On a $W_1 = n_{seq} + 2x + 2y + 2j$. Soit I_1 (resp. I_2) le nombre de tops d'inactivité dans la phase 1 (resp. 2). On a $(n_{seq} + j)(p-1) = 2x + y + j + I_1$ pour la phase 1. Soient $T_p^{(1)}$ (resp $T_p^{(2)}$) le temps parallèle de la phase 1 (resp 2) ; on a
$$\begin{cases} T_p^{(1)} = n_{seq} + j = \frac{2x+y+j+I_1}{p-2} \ ; T_p^{(2)} = \frac{y+I_2}{p} \\ T_p = T_p^{(1)} + T_p^{(2)} \\ (p+1)T_p = 2n + j + \frac{y}{p} + (I_1 + I_2) + \frac{I2}{p} \end{cases}$$
Le nombre de sauts j est inférieur au nombre vols effectués par les $(p-1)$ processeurs. Or les processus P_v exécutent un algorithme parallèle, avec découpe récursive sur vol de travail, donc de profondeur $W_\infty \leq \log_2(n - n_{seq}) + 2 \leq \log_2 n$. D'après le théorème 2.1, on en déduit qu'avec une grande probabilité, $j \leq (p-1)\log_2(n)$. De plus, l'ordonnancement étant glouton durant les deux phases, $I_1 + I_2 \leq pW_\infty \leq p\log_2(n)$. On obtient donc $j + (I_1 + I_2) + \frac{I_2}{p} \leq 2p\log_2(n)$. Par ailleurs, dans la phase 1 on a $(p-1)(n_{seq} + j) = 2x + y + j + I_1$ et $x + y + j = n - n_{seq}$.

Puisque $n_{seq} \geq \frac{2n}{p+1}$, on obtient $y \leq I_1 - pj \leq (p-1)\log_2 n$. Finalement, on a $T_p \leq \frac{2n}{p+1} + 2\log_2(n) + (\frac{p-1}{p(p+1)})\log_2(n) + O(1)$ ce qui termine la preuve. $\triangle$

Le cas de processeurs variables s'étudie similairement en décomposant le programme en deux phases et en dénombrant le nombre d'opérations dans chaque phase. Le théorème suivant suppose que la vitesse moyenne Π_{ave} du processus séquentiel est la même que celle $\Pi_{ave}^{(1)}$ (resp. $\Pi_{ave}^{(2)}$) des autres processeurs durant la phase 1 (resp.2). Cette hypothèse est réaliste si l'exécution est suffisamment longue et si les processeurs logiques sont ordonnancés de manière équitable sur les p processeurs physiques (cas de Linux par ex exemple). De plus, elle est cohérente pour une comparaison avec l'exécution séquentielle de l'algorithme, qui serait alors supposée prendre un temps $\frac{n}{\Pi_{ave}}$.

Théorème 5.2 *Soit T_p le temps du calcul de n préfixes par l'algorithme adaptatif sur p processeurs de vitesse globale moyenne $p.\Pi_{ave}$. Alors, avec une grande probabilité :*

$$T_p \leq \frac{2n}{(p+1)\Pi_{ave}} + \frac{O(\log_2 n)}{\Pi_{ave}} \sim_{n\to\infty} \frac{2n}{(p+1)\Pi_{ave}}.$$

La preuve de ce théorème, similaire à la preuve précédente, n'est pas détaillée. Elle utilise l'ordonnancement par vol de travail modifié décrit en section 2 [1].

En conclusion, en comparaison à l'algorithme séquentiel de préfixe supposé exécuté à la vitesse moyenne Π_{ave}, l'algorithme adaptatif de préfixe a une accélération asymptotique égale à $\frac{2}{p+1}$; il est optimal et ce sans faire aucune hypothèse sur les vitesses des différents processeurs.

Dans le paragraphe suivant, nous vérifions expérimentalement ces résultats théoriques dans les deux cas processeurs identiques et processeurs de vitesses variables.

6. Expérimentations

Nous avons fait des expérimentations sur une machine à mémoire partagée (SMP) à 8 processeurs (Intel Itanium-2 à 1.5GHz), avec 31 GB de mémoire et en contexte multi-utilisateurs sous le système GNU-Linux 2.6.7. Les algorithmes parallèle et adaptatif ont été implémentés sur Kaapi [6] qui intègre l'ordonnancement par vol de travail.

Les expérimentations consistent au calcul des préfixes de 100 éléments (double) avec un temps de 100ms par opération $\star$ en faisant varier p le nombre de processeurs utilisés (de 1 à 8). Le temps séquentiel optimal de référence est de 10s.

Les tableaux 1 et 2 donnent les temps d'exécution obtenus par les deux algorithmes parallèles (à grain fixé sur p processeurs et à grain adaptatif). Pour chaque expérience, 10 mesures ont été effectuées et seuls sont reportés les temps de l'exécution la plus rapide, la plus lente et le temps moyen des 10 exécutions.

Le tableau 1 compare les temps d'exécution lorsqu'il n'y a pas d'autres calculs en cours sur les processeurs. On remarque que les mesures de temps sont stables (écart entre temps minimum et maximum inférieur à 5% pour l'algorithme à grain fixé et inférieur à 4% pour l'algorithme à grain adaptatif). On vérifie l'optimalité de l'algorithme à grain fixé dont le temps est à moins de 5% de la borne inférieure. Mais surtout, on vérifie l'optimalité de l'algorithme adaptatif (théorème 5.1) qui est aussi à moins de 5% de la borne inférieure.

Dans, le tableau 2, des processus de charge additionnels sont injectés pour perturber l'occupation de la machine et simuler le comportement d'une machine réelle, perturbée

par d'autres utilisateurs. Par souci de reproductibilité, chaque expérience sur $p \leq 8$ processeurs est perturbée par $8 - p + 1$ processus artificiels de durée supérieure à 10s. On peut vérifier dans le tableau 2 que l'algorithme adaptatif est au moins 27% plus rapide.

	Sequentiel	Statique				Adaptatif			
		p=2	p=4	p=6	p=7	p=2	p=4	p=6	p=7
Minimum	10,00	6,70	4,00	3,00	2,60	6,70	4,00	2,80	2,60
Maximum	10,00	6,72	4,04	3,00	2,61	6,75	4,01	2,90	2,65
Moyenne	10,00	6,71	4,01	3,00	2,60	6,73	4,01	2,90	2,62
Borne inférieure	10,00	6,66	4,00	2,85	2,5	6,66	4,00	2,85	2,5

Tableau 1. *Compraison des temps des trois algorithmes sur p processeurs identiques.*

	Statique				Adaptatif			
	p=2	p=4	p=6	p=8	p=2	p=4	p=6	p=8
Minimum	9,94	5,05	4,24	3,19	8,07	4,49	3,21	2,39
Maximum	13,38	6,93	5,03	3,92	13,18	5,32	3,97	2,77
Moyenne	12,80	6,43	4,89	3,52	11,93	4,79	3,36	2,54

Tableau 2. *Compraison des temps des algorithmes sur p processeurs perturbés. Sur les 10 exécutions de chacun des tests, l'algorithme adaptatif est le plus rapide.*

En conclusion, l'algorithme à grain adaptatif apporte une performance garantie lorsque la machine est partagée entre plusieurs utilisateurs, en s'adaptant automatiquement aux ressources disponibles au cours de l'exécution. De plus sa performance reste proche de l'optimale même dans le cas idéal où les processeurs sont tous dédiés à l'application. Il apparaît donc être plus performant que l'algorithme séquentiel ou qu'un algorithme parallèle à grain fixé.

Ceci est confirmé par une autre expérimentation où chaque test élémentaire correspond au lancement simultané en concurrence des neuf programmes : algorithme adaptatif sur huit processeurs, algorithme séquentiel et l'algorithme parallèle à grain fixé pour les sept valeurs $p = 2, \ldots, 8$ processeurs. Le tableau 3 résume les résultats sur une campagne de 10 tests. Pour les 10 exécutions menées, l'algorithme à grain adaptatif est toujours le plus rapide. Son temps moyen d'exécution est en moyenne 19% fois plus court que celui de l'algorithme optimal à grain fixé sur 8 processeurs, avec des écarts pouvant aller jusqu'à 40% sur l'un des tests.

	Sequentiel	Statique					Adaptatif
		p=2	p=4	p=6	p=7	p=8	p=8
Minimum	21,83	18,16	15,89	14,99	13,92	12,51	8,76
Maximum	23,34	20,73	17,66	16,51	15,73	14,43	12,70
Moyenne	22,57	19,50	17,10	15,58	14,84	13,17	11,14
Mediane	22,58	19,64	17,38	15,57	14,63	13,11	11,01

Tableau 3. *Compraison des temps des neuf algorithmes lancés simultanément. Sur les 10 exécutions de chacun des tests, l'algorithme adaptatif est le plus rapide.*

7. Conclusion

Motivés par l'utilisation de machines multi-processeurs partagées entre plusieurs utilisateurs, nous avons introduit un nouvel algorithme parallèle pour le calcul des préfixes qui s'adapte automatiquement et dynamiquement aux processeurs effectivement disponibles. Nous avons montré son travail était asymptotiquement optimal. Il est équivalent à celui de l'algorithme séquentiel lorsqu'un seul processeur est disponible et à celui d'un algorithme parallèle optimal lorsque p processeurs identiques sont disponibles. Dans le cas de p processeurs de vitesses variables, son temps est équivalent à celui d'un algorithme optimal sur p processeurs identiques de vitesse égale à la moyenne des vitesses. Ces résultats théoriques sont validés par les expérimentations menées sur une machine SMP à 8 processeurs.

Dans le cadre du projet français ANR BGPR, une première perspective est de le valider sur la grille hétérogène nationale GRID'5000. Plus généralement, notre algorithme adaptatif est basé sur le couplage récursif et dynamique de deux algorithmes un algorithme séquentiel, optimal en nombre d'opérations, et l'autre parallèle avec un degré maximal de parallélisme. Ce shéma a été proposé dans [2] avec deux algorithmes réalisant le même nombre d'opérations. Pour les préfixes, nous avons étendu ici ce schéma au cas où l'algorithme parallèle requiert plus d'opérations que l'algorithme séquentiel. Nous pensons que ce schéma, analysé ici pour les préfixes, est plus général et s'applique à un grand nombre de problèmes, en particulier pour la résolution de systèmes linéaires dans le cadre du projet IMAG-INRIA AHA.

8. Bibliographie

[1] Michael A. Bender and Michael O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory Comput. Syst.*, 35(3) :289–304, 2002.

[2] El-Mostafa Daoudi, Thierry Gautier, Aicha Kerfali, Rémi Revire, and Jean-Louis Roch. Algorithmes parallèles à grain adaptatif et applications. *TSI*, 24 :1—20, 2005.

[3] Giorgos Dimitrakopoulos and Dimitris Nikolos. High-speed parallel-prefix vlsi ling adders. *IEEE Trans. Computers*, 54(2) :225–231, 2005.

[4] Faith E. Fich. New bounds for parallel prefix circuits. In *STOC '83 : Proceedings of the fifteenth annual ACM symp. Theory of computing*, pages 100–109, New York, NY, USA, 1983. ACM Press.

[5] 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.

[6] Samir Jafar, Thierry Gautier, Axel W. Krings, and Jean-Louis Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In LNCS Springer-Verlag, editor, *EUROPAR'2005*, Lisboa, Portogal, August 2005.

[7] Richard Ladner and Michael Fischer. Parallel prefix computation. *J. ACM*, 27(4) :831–838, 1980.

[8] Jigang Liu, Fenglien Lee, and Kai Qian. A parallel prefix convex hill algorithm using maspar. In *Proceedings PDPTA'02, Las Vegas, vol. 3*, pages 1089–1095, 2002.

[9] Alexandru Nicolau and Haigeng Wang. Optimal schedules for parallel prefix computation with bounded resources. In *PPOPP '91 : Proceedings of the third ACM SIGPLAN symp. Principles and practice of parallel programming*, pages 1–10, New York, NY, USA, 1991. ACM Press.

Annexe.

Dans cette annexe, nous donnons la preuve du théorème 5.2 qui n'a pu être incluse par manque de place et souci de lisibilité.

On considère ici des processeurs hétérogènes ou de vitesses différentes, éventuellement non connues et variables ; la vitesse instantanée d'un processeur est notée Π_k et mesurée en nombre d'opérations par top. Nous utilisons l'ordonnancement par vol de travail proposé dans [1]. Cet ordonnancement est dit à haute utilisation de facteur β et vérifie la propriété suivante : si à un top, il y a $i < p$ processeurs actifs, alors le processeur inactif le plus rapide est au plus β fois plus rapide que le processeur actif le plus lent à ce top. Le paramètre β est choisi arbitrairement ; si $\beta = 1$, on obtient un ordonnancement à utilisation maximale. Dans [1], β est introduit pour diminuer le nombre de préemptions et faciliter l'implémentation de l'ordonnancement par vol de travail et est appliqué à Cilk. Au niveau du vol de travail, la seule modification est lorsqu'un processeur inactif p_i vole du travail à un processeur actif p_a : si p_a n'a pas de travail prêt à être volé et si p_i est plus de β fois plus rapide que p_a, alors la tâche en cours d'exécution sur p_a est migrée sur p_i. Le paramètre β permet de limiter le nombre de migrations lorsque la vitesse des processeurs varie mais le nombre de changements de vitesse reste limité.

Soit T_p le temps de l'exécution sur p processeurs de vitesses variables d'un programme parallèle réalisant W_1 opérations et de chemin critique W_∞ en nombre d'opérations. On note Π_{ave} la vitesse moyenne de l'ensemble des processeurs au cours de l'exécution.

Théorème 8.1 *[1] Avec une grande probabilité, le nombre de vols réussis est majoré par $O(\beta W_\infty p)$ et le temps d'exécution T_p est majoré par*

$$T_p \leq \frac{W_1}{p.\Pi_{ave}} + O\left(\frac{\beta.W_\infty}{\Pi_{ave}}\right)$$

Ainsi, si W_∞ est négligeable devant W_1, cet ordonnancement à une performance asymptotique égale à celle d'un ordonnancement optimal sur p processeurs identiques de vitesse Π_{ave}.

Nous utilisons cet ordonnancement par vol de travail modifié de paramètre β pour exécuter l'algorithme à grain adaptatif de calcul de préfixe sur p processeurs de vitesses variables. Le théorème 5.2 est alors déduit du théorème suivant.

Théorème 8.2 *Avec une grande probabilité,*

$$T_p \leq \frac{2n}{(p+1)\Pi_{ave}} + \frac{(\beta+2)\log_2 n}{\Pi_{ave}} \sim_{n\to\infty} \frac{2n}{(p+1)\Pi_{ave}}.$$

Preuve.

Comme dans la preuve du théorème 5.1, l'exécution est découpée en deux phases successives, ϕ_1 puis ϕ_2 :

1) durant la phase ϕ_1 de durée $T_p(\phi_1)$, l'algorithme séquentiel de préfixe est en cours d'exécution à tout top sur un processeur et il effectue $n_{seq}+j$ opérations $\star$. On note Π_{seq} la vitesse moyenne à laquelle est exécuté cet algorithme : $\Pi_{seq} = \frac{n_{seq}+j}{T_p(\phi_1)}$.
A tout top, les $p-1$ autres processeurs effectuent la partie parallèle de l'algorithme adaptatif et effectuent au total $2x+y+j$ opérations $\star$ et I_1 opérations d'inactivité. Durant ϕ_1, la vitesse d'exécution moyenne par processeur de cette partie de l'algorithme (effectuée à tout top sur $p-1$ processeurs) est $\Pi_{ave}(\phi_1) = \frac{2x+y+j+I_1}{(p-1).T_p(\phi_1)}$.
Le nombre total d'opérations $\star$ dans la phase ϕ_1 est $W(\phi_1) = n_{seq} + 2j + 2x + y$.

2) durant la phase ϕ_2, la partie séquentielle de l'algorithme adaptatif est terminée ; les p processeurs finalisent les y calculs de préfixes anticipés en parallèle et non terminés dans la phase ϕ_1. Durant la phase ϕ_2, les p processeurs effectuent donc y opérations $\star$ et I_2 opérations d'inactivité. La vitesse moyenne par processeur durant ϕ_2 est $\Pi_{ave}(\phi_2) = \frac{y+I_2}{p.T_p(\phi_2)}$.
Le théorème est donné sous l'hypothèse $\Pi_{seq} = \Pi_{ave}(\phi_1) = \Pi_{ave}(\phi_2) = \Pi_{ave}$.

Pendant ϕ_1, les processeurs qui n'exécutent pas l'algorithme séquentiel (i.e. $p-1$ à chaque top) effectuent $2x+y+j$ opérations $\star$ avec un chemin critique $W_\infty(\phi_1) \leq 2.\log_2 n$ lié à la découpe récursive. En appliquant le théorème 8.1, on obtient $T_p(\phi_1) \leq \frac{2x+y+j}{(p-1)\Pi_{ave}} + O\left(\frac{\beta.\log n}{\Pi_{ave}}\right)$. On a aussi $T_p(\phi_1) = \frac{n_{seq}+j}{\Pi_{seq}} = \frac{n_{seq}+j}{\Pi_{ave}}$. D'où : $(p+1)T_p(\phi_1) = (p-1)T_p(\phi_1) + 2T_p(\phi_1) \leq \frac{2n_{seq}+2x+y+3j}{\Pi_{ave}} + O\left((p-1)\frac{\beta.\log n}{\Pi_{ave}}\right)$. Comme $n = n_{seq} + x + y + j$, on obtient :

$$(p+1)\Pi_{ave}T_p(\phi_1) \leq 2n - y + j + O\left((p-1)\beta.\log n\right).$$

Par ailleurs, en appliquant le théorème 8.1 à ϕ_2, on obtient $T_p(\phi_2) \leq \frac{y}{p\Pi_{ave}} + O\left(\frac{\beta.\log n}{\Pi_{ave}}\right)$. D'où

$$(p+1)\Pi_{ave}T_p = (p+1)\Pi_{ave}T_p(\phi_1) + (p+1)\Pi_{ave}T_p(\phi_2) \leq 2n + j + \frac{y}{p} + O\left(2p\beta.\log n\right).$$

Le nombre j de sauts est inférieur au nombre de vols réussis, soit $O(\beta \log n)$ d'après le théorème 8.1 puisque $W_\infty(\phi_1) \leq 2\log_2 n$. De plus, en utilisant $n_{seq} \geq \frac{2n}{p+1}$ et similairement à la preuve du théorème 5.1, $y \leq I_1 \leq (p-2)\log_2 n$. Finalement, on obtient $(p+1)\Pi_{ave}T_p \leq 2n + O\left((\beta+2)\log n\right)$. △

Remarque. On peut remarquer que la preuve et le théorème reste valide dans le cas plus général et réaliste où $\Pi_{seq} \geq \Pi_{ave}(\phi_1)$ (l'algorithme séquentiel est toujours exécuté par un processeur plus rapide que la moyenne des processeurs) et $\Pi_{ave}(\phi_2) \geq \Pi_{ave}(\phi_1)$ (le processeur séquentiel ajouté dans la phase 2 est plus rapide que la moyenne des autres processeurs).