Le chemin algébrique sur réseaux linéaires

Sanjay Rajopadhye[*] — Tanguy Risset[*] — Claude Tadonki[**]

** IRISA, Campus de Beaulieu*
35 042 Rennes

{Sanjay.Rajopadhye,Tanguy.Risset}@irisa.fr

*** Université de Yaoundé,*
B.P. 812, Yaoundé, Cameroun

cmtado@uycdc.uninet.cm

RÉSUMÉ. *Nous décrivons une architecture linéaire de type* SIMD *pour la résolution du problème du chemin algébrique. Pour un graphe ayant n sommets, notre réseau dans sa version de base est composé de n processeurs ayant chacun une mémoire locale de taille $2n$, et calcule le résultat en $3n^2 - 2n$ étapes. Du fait de la simplicité du contrôle et de la gestion de la mémoire, notre réseau offre une issue convenable pour une implémentation du type* VLSI. *Le réseau étant linéaire, on a un mécanisme d'entrées/sorties assez réaliste et facile à gérer. Dans le cas où on dispose d'un nombre réduit de processeur ($p < n$), on a une adaptation triviale du réseau par une exécution en plusieurs passes, et de plus, cette version améliore l'efficacité globale du travail. Formellement, pour une constante positive $\alpha < n$, le temps d'exécution de l'algorithme sur $\frac{n}{\alpha}$ processeurs est borné par $(2 + \alpha)n^2$, ce qui implique que le travail total est borné par $(1 + \frac{2}{\alpha})n^3$, quantité qui peut être rendue aussi proche que désirée de n^3 en augmentant α. Enfin, notons qu'avec la possibilité d'une version bloc de l'algorithme, on a une issue naturelle vers une implémentation acceptable sur des machines parallèles standards. Les résultats expérimentaux obtenus sur la machine* INTEL PARAGON *confirment nos analyses d'optimisation sur la taille des blocs d'une part, et nos prédictions de performances d'autre part.*

ABSTRACT. *The paper provides an efficient linear* SIMD *architecture for the algebraic path problem. The algorithm is for* VLSI *and general purpose processors. For a graph with n nodes, it runs on $\frac{n}{\alpha}$ processors ($1 \le \alpha \le n$) and computes the result in no more that $(2 + \alpha)n^2$ steps. Hence a work no more that $(1 + \frac{2}{\alpha})n^3$, which can be made as close to n^3 as desired by increasing α (i.e. decreasing p). We also look the case for standard parallel machine since there is a block version issue of the algorithm. Our expriments on the* INTEL PARAGON *agree with our optimal tile size analysis and performance predictions.*

MOTS-CLÉS : *fermeture transitive, plus court chemin, inversion de matrice, synthèse systolique, ordonnancement spatio-temporelle, équations récurrentes, machine à mémoire distribuée.*

KEYWORDS: *transitive closure, scheduling, pipeline process, space-time complexity, linear array, parallel distributed memory machine.*

1. Introduction

Le *problème du chemin algébrique*, en anglais *the algebraic path problem* (APP), unifie un nombre de problèmes classiques en un schéma formel unique. Il est défini de la manière suivante. Etant donné un graphe valué $G = \langle V, E, w\rangle$ avec des sommets $V = \{1, 2, \ldots n\}$, des arcs $E \subseteq V \times V$ et une fonction de poids $w : E \rightarrow \mathcal{S}$, où S est un semi anneau fermé $\langle \mathcal{S}, \oplus, \otimes, *, \mathbf{0}, \mathbf{1}\rangle$ (fermé dans le sens que $*$ est un opérateur unaire de "fermeture", définie par la somme infinie $x* = x \oplus (x \otimes x) \oplus (x \otimes x \otimes x) \oplus \ldots$). Un chemin dans G est une séquence (finie ou non) de sommets $p = v_1 \ldots v_k$, et le poids d'un chemin est défini comme étant le produit $w(p) = w(v_1, v_2) \otimes w(v_2, v_3) \otimes \ldots \otimes w(v_{k-1}, v_k)$. Soit $P(i, j)$ l'ensemble (fini ou non) de tous les chemins de i à j. L' APP est le problème du calcul, pour toutes les paires i, j tels que $0 < i, j \leq n$, de la valeur $d(i, j)$ définie comme suit ($\bigoplus$ est l'opérateur de "sommation" dans $\mathcal{S}$)

$$d(i, j) = \bigoplus_{p \in P(i,j)} w(p).$$

L'algorithme de Warshall pour la fermeture transitive, l'algorithme de Floyd pour les plus courts chemins et l'algorithme de Gauss-Jordan pour l'inversion matricielle sont tous des instances de l'algorithme générique de l'APP (c'est ainsi qu'il est appelé l'algorithme WFGJ), la seule différence étant le semi anneau impliqué. Sa complexité séquentielle (et donc le travail) est n^3 opérations de semi anneau.

Une importante activité s'est développée autour de l'implémentation de l'APP (ou d'une instance particulière) sur des réseaux systoliques (voir table 1). La plupart des premières solutions étaient obtenues de manière intuitive, tandis que les plus récentes reposent sur des méthodes systématiques de conception. Toutes ces implémentations requièrent un temps en $\Theta(n)$ ($3n$ au mieux et $5n$ en général) sur $\Theta(n^2)$ processeurs. Au début des années 90, une nouvelle localisation réduisant le temps d'exécution de $5n$ à $4n$ fut proposée indépendamment par un certain nombre d'auteurs [CHA 94, RAJ 93, RIS 91, TAK 92]. La technique de Scheiman-Cappello et Benaini-Robert [BEN 90b, SCH 90] peut être utilisée pour réduire le nombre de processeurs dans ces réseaux rapides à $n^2/3$ [TAY 00]. Toutefois, l'efficacité de ces solutions diminue lorsqu'elles sont mises en œuvre sur un nombre réduit de processeurs. Ceci est sans doute dû au fait que dans la version originale, les calculs sont effectués beaucoup plus tôt que dans la version partitionnée, laquelle version pourrait par ailleurs impliquer une importante modification de la logique des processeurs.

En pratique, cette limitation est importante. Une implémentation directe de l'une quelconque de ces architectures, par un partitionnement ayant pour but une exécution sur un réseau VLSI dédié, n'améliorerait pas l'efficacité globale du travail.

Dans cet article, nous proposons une architecture SIMD VLSI qui ne souffre pas de ce problème. Nous dérivons formellement une architecture ayant n processeurs et un temps d'exécution égal à $\Theta(n^2)$, et dont l'efficacité augmente avec le partitionnement. Plus précisément, le réseau est composé de n processeurs et calcule la solution en $3n^2 - 2n$ étapes. De plus, il s'adapte trivialement pour une exécution sur $p < n$

Auteurs	Application	Surface	Temps
Guibas et al. [GUI 79]	TC	n^2	$6n$
Nash-Hansen [NAS 84]	MI	$3n^2/2$	$5n$
Robert-Tchuenté [ROB 85]	MI	n^2	$5n$
Kung-Lo [KUN 85]	TC	n^2	$7n$
Rote [ROT 85]	APP	n^2	$7n$
Robert-Trystam [ROB 87]	APP	n^2	$5n$
Kung-Lo-Lewis [KUN 87]	TC & SP	n^2	$5n$
Benaini et al. [BEN 90a]	APP	$n^2/2$	$5n$
Benaini-Robert [BEN 90b]	APP	$n^2/3$	$5n$
Scheiman-Cappello [SCH 90]	APP	$n^2/3$	$5n$
Clauss-Mongenet-Perrin [CLA 92]	APP	$n^2/3$	$5n$
Takaoka-Umehara [TAK 92]	SP	n^2	$4n$
Rajopadhye [RAJ 93]	APP	n^2	$4n$
Risset-Robert [RIS 91]	APP	n^2	$4n$
Chang-Tsay [CHA 94]	APP	n^2	$4n$
Djamegni et al. [TAY 00]	APP	$n^2/3$	$4n$
Réseaux linéaires (systoliques and autres)			
Kumar-Tsai [PRA 89]	APP	n^2	$7n^2$
Myoupo-Fabret [MYO 96]	APP	n^2/α	$(3+\frac{4}{\alpha})n^2$

Tableau 1. *Implémentations systoliques de l'algorithme* WFGJ *et de ses instances (*SP, TC *et* MI *désignant respectivement les plus courts chemins, la fermeture transitive et l'inversion matricielle). La table n'est pas exhaustive mais suit l'ordre chronologique des contributions.*

processeurs en effectuant des passes multiples, et avec un gain d'efficacité. Avec α passes (sur $\frac{n}{\alpha}$ processeurs), le temps d'exécution est borné par $(\alpha+2)n^2$, et par suite, le travail total est borné par $(1+\frac{2}{\alpha})n^3$. De ce fait, en augmentant le nombre de passes, nous approchons autant que souhaité la performance optimale. Finalement, sur un nombre *fixe* de processeurs (donc $\alpha = n/p$ n'est pas *constant* mais proportionnel à n), notre implémentation est de travail optimal.

Notre architecture n'est pas «purement» systolique car elle requiert deux registres à décalage de taille n sur chaque processeur. Toutefois, du fait qu'un registre à décalage correspond tout simplement à un réseau linéaire de registres, on pourrait arguer que notre architecture est un vrai réseau systolique bidimensionel $n \times n$, dans lequel seuls les processeurs extrêmes effectuent des calculs, tandis que les «processeurs» internes contiennent uniquement des registres. Globalement, le réseau est assez modulaire et idéal pour une implémentation de type VLSI.

Toutefois, une telle architecture est à grain trop fin pour une machine parallèle standard. En effet, le temps total d'exécution reste égal à $\Theta(n^3/p)$ sur p processeurs, mais la constante impliquée devient $\alpha + \beta$, où α est le temps d'une opération, et β le coût de synchronisation de la machine parallèle (ou les frais d'un appel à une routine de communication, généralement appelé *latence*). Dans le cas général de machines parallèles, $\beta \gg \alpha$, et les performances deviennent inacceptables.

Une méthode bien connue pour venir à bout de ce problème est le partitionnement en blocs de l'espace de calcul. Nous dérivons formellement une version bloc de notre algorithme, et par suite un réseau linéaire similaire dans lequel les opérations sont effectuées sur des blocs au lieu des opérandes élémentaires. Le temps total d'exécution reste $\Theta(n^3/p)$ sur p processeurs, mais la constante est maintenant une fonction de b pour un partitionnement en blocs de taille $b \times b$. Nous formulons et résolvons le problème d'optimisation discrète visant à déterminer la valeur de b qui minimise le temps total d'exécution. Nos prédictions sont ensuite vérifiées sur la machine INTEL PARAGON.

Le reste de l'article est organisé comme suit. La section 2 introduit le formalisme des équations récurrentes. La section 3 formule le problème du chemin algébrique et donne un apercu de son état de l'art. Une description intuitive de notre solution est faite dans la section 4. La section 5 présente une analyse formelle et quantitative de l'algorithme, suivie par l'étude d'une version bloc équivalente en section 6. La section 7 présente les résultats de nos expérimentations. Nous concluons en section 8.

2. Formalisme des équations récurrentes

2.1. *Equations récurrentes*

Définition 1 *Une **équation récurrente** définissant une fonction (ou une variable) X en tous les points z d'un domaine D, est une équation de la forme*

$$X(z) = D^X \quad : \quad g(\ldots Y(f(z)) \ldots) \tag{1}$$

où

*– z est une **variable d'index** de dimension n.*

*– X et Y sont des **variables de donnée**. X correspond à une fonction de n arguments entiers ; on parle de variable de dimension n.*

*– $f(z)$ est une **fonction de dépendance**, $f : \mathcal{Z}^n \rightarrow \mathcal{Z}^q$;*

– g est une fonction scalaire ; elle apparaît souvent de manière implicite sous la forme d'une expression impliquant des opérandes de la forme $Y(f(z))$, combinée avec des opérateurs de base et des parenthèses.

*– D^X est une ensemble de points de $\mathcal{Z}^n$ et est appelé le **domaine** de l'équation. Les domaines sont très souvent des espaces d'index polyhédriques avec un ou plusieurs (notons ℓ) paramètres, $p \in \mathcal{Z}^\ell$.*

Une variable pourrait être définie par plusieurs équations. Dans ce cas, on utilise la syntaxe ci-dessous :

$$X(z) = \begin{cases} & \vdots \\ D_i^X & : \quad g_i(\ldots X(f(z))\ldots) \\ & \vdots \end{cases} \tag{2}$$

Chaque ligne correspond à un **cas**, et le domaine de X est une union des domaines disjoints de tous les cas, $D^X = \bigcup_i D_i^X$. On dire par ailleurs que la fonction de dépendance f est valable dans le (sous) domaine D_i^X.

Définition 2 *Une équation récurrente telle que définie par (1) est dite **affine** si chaque fonction de dépendance est de la forme $f(z) = Az + Bp + a$, où A (respectivement B) est une matrice constante $q \times n$ (respectivement $q \times l$) et a est une vecteur constant de dimension q. L'équation récurrente est dite **uniforme** si chaque fonction de dépendance est de la forme $f(z) = z + a$, où a est un vecteur constant de dimension n appelé vecteur de dépendance.*

Définition 3 *Un **système** d'équations récurrentes* (SER) *est un ensemble de m équations récurrentes, définissant des variables de données $X_1, \cdots, X_m$, où chaque variable X_i est de dimension n_i. Puisque les équations du système sont mutuellement récursives, la fonction de dépendance f sera de type approprié.*

Domaine.

Un aspect important du formalisme des équations récurrentes est la notion de *domaine*, ensemble d'indices sur lesquels des calculs particuliers sont définis. Ce domaine est d'habitude bien spécifié dans l'écriture d'un SER. Le domaine le plus souvent utilisé est un *polyhèdre* (ensemble de points entiers satisfaisant une nombre fini de contraintes linéaires d'(in)égalités), ou une réunion finie de polyhèdres.

Transformations de SER.

Une des plus importantes manipulations que l'on peut effectuer sur une SER est l'opération de *réindexation* (aussi appelée *changement de base* ou *réarrangement spatio-temporelle*) de ses variables. La transformation, notée $\mathcal{T}$, devrait admettre une *réciproque à gauche* pour tous les points du domaine de la variable. Lorsqu'elle appliquée à la variable X d'un SER défini comme suit :

$$X(z) \quad = \quad \begin{cases} & \vdots \\ D_i^X & : \quad g_i(\ldots Y(f(z))\ldots), \\ & \vdots \end{cases}$$

le SER obtenu en appliquant les règles suivantes est équivalent au SER original :

– Remplacer chaque D_i^X par $\mathcal{T}(D_i^X)$, c'est-à-dire son image par $\mathcal{T}$.

– Dans la partie droite de l'équation définissant X, remplacer chaque dépendance f par $f \circ \mathcal{T}^{-1}$, c'est-à-dire la composition de f et $\mathcal{T}^{-1}$.

– Dans toutes les occurrences $X(f'(z))$ de la partie droite de *chaque* équation définissant une variable différente de X, remplacer la dépendance f' par $\mathcal{T} \circ f'$.

– Pour le cas spécial des occurrences de X apparaissant dans la partie droite de l'équation définissant X, remplacer la dépendance f par $\mathcal{T} \circ f \circ \mathcal{T}^{-1}$.

3. Formulation du problème et état de l'art

Le formalisme de base largement utilisé pour étudier l'APP est celui des *équations récurrentes*. L'algorithme classique de l'APP est basé sur le système d'équations récurrentes suivant, où $D_0 = \{i, j, k \mid 0 < i, j \leq n; 0 \leq k \leq n\}$ est le domaine de F.

$$d(i,j) \quad = \quad \{i,j \mid 0 < i,j \leq n\} : F(i,j,n) \tag{3}$$

$$F(i,j,k) \quad = \quad \begin{cases} D_0 \cap \{i,j,k \mid k = 0\} & : a_{i,j} \\ D_0 \cap \{i,j,k \mid i = j = k\} & : F(i,j,k-1)* \\ D_0 \cap \{i,j,k \mid i = k \neq j\} & : F(k,k,k) \otimes F(i,j,k-1) \\ D_0 \cap \{i,j,k \mid j = k \neq i\} & : F(i,j,k-1) \otimes F(k,k,k) \\ D_0 \cap \{i,j,k \mid i \neq k; j \neq k\} & : F(i,j,k-1) \oplus \\ & \quad (F(i,k,k) \otimes F(k,j,k-1)) \end{cases} \tag{4}$$

Mis à part le plan d'initialisation $k = 0$, le domaine de F est un $n \times n \times n$ cube. Pour une valeur donnée de k, les points $[k, k, k]$ de la diagonale $i = j = k$ sont appelés *pivots*, tandis que les lignes de chacun des plans $i = k \neq j$ et $j = k \neq i$ (i.e., les lignes et colonnes contenant les points de la forme $[k, j, k]$ et $[i, k, k]$, respectivement) sont appelées *ligne pivot* et *colonne pivot*. Les points restants sont appelés les points *intérieurs*.

Observons que pour tout plan $k = cste$, nous avons l'ordre de calcul suivant (en supposant un nombre suffisant de processeurs). Premièrement, le pivot $[k, k, k]$ est calculé puisqu'il ne dépend que de la valeur correspondante du plan «précédent». Ensuite, le reste de la colonne pivot, c'est-à-dire les points $[i, k, k]$ pour $i \neq k$ (respectivement les points de la ligne pivot $[k, j, k]$ pour $j \neq k$) sont calculés, puisqu'ils ne dépendent que du pivot et éventuellement des valeurs correspondantes du plan «précédent». Finalement, les points intérieurs sont calculés puisqu'ils dépendent directement de la ligne pivot et de la colonne pivot. Notons aussi que le pivot du plan $k + 1$ dépend du premier point intérieur, de sorte qu'aucun calcul du plan suivant ne peut commencer avant que ce premier point intérieur ne soit mis à jour. En d'autres termes, il y a un chemin critique de longueur trois entre deux pivots successifs, qui est le suivant $[k, k, k] \rightarrow [k + 1, k, k] \rightarrow [k + 1, k + 1, k] \rightarrow [k + 1, k + 1, k + 1]$. Ceci implique que le temps d'exécution optimal est borné par $3n$, et cette borne est atteinte. En effet, il est bien connu [RAJ 93] que l'ordonnancement parallèle optimal pour le système d'équations récurrentes (3-4) est le suivant :

$$t_f(i,j,k) = \begin{cases} D_0 \cap \{i,j,k \mid k=0\} & : 0 \\ D_0 \cap \{i,j,k \mid i=j=k\} & : 3k-2 \\ D_0 \cap \{i,j,k \mid i=k\neq j\} & : 3k-1 \\ D_0 \cap \{i,j,k \mid j=k\neq i\} & : 3k-1 \\ D_0 \cap \{i,j,k \mid i\neq k; j\neq k\} & : 3k \end{cases} \quad (5)$$

Cet ordonnancement suppose un nombre non borné de processeurs, $\Theta(n^2)$ pour être précis, avec des communications globales : en particulier la diffusion du pivot et de la colonne pivot sont nécessaires. Les précédents résultats résumés dans la table 1 illustrent le fait que la localité requise dans une implémentation systolique impose une augmentation du temps total d'exécution qui passe de $3n$ à $4n$ ou plus.

Avant de continuer, nous allons présenter une version modifiée du système d'équations (3-4) qui sera la base de notre solution. Le but est de maintenir les lignes et colonnes pivots en premières positions et obtenir une formulation dans laquelle les opérations effectuées dans chaque plan $k = cte$ sont totalement identiques. A cette fin, on définit l'opérateur $\dot{+}$ par $i\dot{+}j = (i+j) \bmod n$. En appliquant la transformation $(i,j,k \to (i-k) \bmod n, (j-k) \bmod n, k)$ à tous les points du sous-domaine $\{i,j,k \mid 0 < i,j,k \leq n\}$ de D_0, on obtient le système d'équations récurrentes suivant, où $D_1 = \{i,j,k \mid 0 \leq i,j < n; 0 < k \leq n\} \cup \{i,j,k \mid k = 0 < i,j \leq n\}$ est le nouveau domaine de F.

$$d(i,j) \quad = \quad \begin{cases} \{i,j \mid 0<i,j<n\} & : F(i,j,n) \\ \{i,j \mid 0<i<n; j=n\} & : F(i,0,n) \\ \{i,j \mid i=n; 0<j<n\} & : F(0,j,n) \\ \{i,j \mid i=j=n\} & : F(0,0,n) \end{cases} \quad (6)$$

$$F(i,j,k) \quad = \quad \begin{cases} D_1 \cap \{i,j,k \mid k=-1\} & : a_{i,j} \\ D_1 \cap \{i,j,k \mid i=j=0\} & : F(i\dot{+}1,j\dot{+}1,k-1)* \\ D_1 \cap \{i,j,k \mid i=0<j\} & : F(i\dot{+}1,j\dot{+}1,k-1) \otimes F(0,0,k) \\ D_1 \cap \{i,j,k \mid i>0=j\} & : F(0,0,k) \otimes F(i\dot{+}1,j\dot{+}1,k-1) \\ D_1 \cap \{i,j,k \mid i,j>0\} & : F(i\dot{+}1,j\dot{+}1,k-1) \oplus \\ & \quad (F(i,0,k) \otimes F(1,j,k-1)) \end{cases} \quad (7)$$

Nous avons juste appliqué la transformation de réindexation bien connue de Kung Lo and Lewis [KUN 87]. Cette transformation ramène les lignes et colonnes pivots sur les «bords» ($i = 0$ et $j = 0$) du domaine D_1. Ceci peut être vu comme un décalage toroïdal de chaque plan k par rapport au précédent (ceci aussi explique pourquoi l'argument $F(i,j,k-1)$ est systématiquement remplacé par $F(i\dot{+}1,j\dot{+}1,k-1)$).

C'est avec le schéma de récurrence (7) que nous allons bâtir notre algorithme. Un avantage immédiat est la forte régularité dans les dépendances, car celles-ci sont actuellement identiques sur tous les plans tels que $k = cste$. L'idée de base de notre ordonnancement va être de trouver un ordre relatif des calculs dans chaque plan $k = cste$ tel que le flot de données en sortie des calculs d'un plan correspond au flot de données en entrée pour la mise à jour du plan suivant. D'un point de vue quantitatif, il est clair que la complexité du schéma global va fortement dépendre de la qualité des enchaînements.

4. Réseau systolique linéaire : description intuitive

Dans la solution que nous proposons, chaque plan k est exécuté par le processeur k. De ce fait, chaque processeur calcule une matrice $n \times n$ définie par le système (6-7). Toutefois, l'ordre dans lequel un processeur calcule ses valeurs est assez spécial. En effet, les éléments sont calculés «sous-matrice principale» par «sous-matrice principale» comme suit.

– L'élément pivot $F(0, 0, k)$ est calculé en premier.

– Après le calcul de la $(i - 1)^{\text{ème}}$ sous-matrice principale, les éléments des $i^{\text{ème}}$ ligne et colonne de la $i^{\text{ème}}$ sous-matrice principale sont calculés de manière alternative : $F(i, 0, k)$, $F(0, i, k)$, $F(i, 1, k)$, $F(1, i, k)$,...,$F(i, i-1, k)$, $F(i-1, i, k)$. A la fin, le dernier élément diagonal $F(i, i, k)$ est calculé. Ceci achève le calcul de la $i^{\text{ème}}$ sous matrice principale.

Nous insistons sur le fait que ceci nous donne l'ordre relatif dans lequel les calculs sont effectués sur un processeur donné, et que des cycles d'inactivité pourraient intervenir. La figure 1 illustre cet ordonnancement pour un graphe de taille $n = 8$.

Notons aussi que ceci est l'ordre dans lequel les éléments du plan $k^{\text{ème}}$ plan sont calculés. Ils sont envoyés au processeur suivant dans cet ordre mais avec des décalages qui justifient par ailleurs l'existence des cycles d'inactivité. En fait, le processeur met à jour le plan et effectue un décalage toroïdal avant d'envoyer les éléments du plan décalé au processeur suivant, dans l'ordre des sous matrices principales tel que nous l'avons décrit précédemment. La raison de ce décalage vient du fait de la dépendance $(i{+}1, j{+}1, k - 1) \leftarrow (i, j, k)$ qui signifie que la mise à jour d'un point dépend du point qui serait situé «juste en dessous» si le plan avait préalablement subi un décalage toroïdal. Ainsi, le premier point intérieur est la première valeur envoyée tandis que le pivot est la dernière. Après $2n - 2$ derniers cycles, les ligne et colonne pivots sont envoyées. Regardons attentivement l'implication de cet ordonnancement sur le processeur 2 (figure 1.b). Puisque le premier élément qu'il doit calculer est son propre pivot, et que celui-ci dépend de $F(1, 1, 1)$ qui est calculé à $t = 4$ par le processeur 1, le processeur 2 ne commence ses calculs qu'à $t = 5$. Ensuite, il est inactif durant les deux prochains cycles car d'une part les deux valeurs calculées pendant ces cycles par le processeur 1 ne sont pas envoyées, d'autre part le prochain calcul que doit effectuer ce processor 2 est celui de $F(1, 0, 2)$, qui dépend de $F(2, 1, 1)$, valeur produite à $t = 7$. A la suite de ceci, la mise à jour de la sous-matrice principale 2×2 est achevée. Au début des calculs de la troisième sous-matrice principale, on observe encore deux cycles d'inactivité pour des raisons analogues. Ceci va se répéter au début de la mise

1	3	6	11	18	27	38	51
2	4	8	13	20	29	40	53
5	7	9	15	22	31	42	55
10	12	14	16	24	33	44	57
17	19	21	23	25	35	46	59
26	28	30	32	34	36	48	61
37	39	41	43	45	47	49	63
50	52	54	56	58	60	62	64

(a) $k = 1$

	5	9	14	21	30	41	54	67
2 ■	8	10	16	23	32	43	56	69
2 ■	13	15	17	25	34	45	58	71
2 ■	20	22	24	26	36	47	60	73
2 ■	29	31	33	35	37	49	62	75
2 ■	40	42	44	46	48	50	64	77
2 ■	53	55	57	59	61	63	65	79
	66	68	70	72	74	76	78	80

(b) $k = 2$

	11	17	24	33	44	57	70	83
4 ■	16	18	26	35	46	59	72	85
4 ■	23	25	27	37	48	61	74	87
4 ■	32	34	36	38	50	63	76	89
4 ■	43	45	47	49	51	65	78	91
4 ■	56	58	60	62	64	66	80	93
2 ■	69	71	73	75	77	79	81	95
	82	84	86	88	90	92	94	96

(c) $k = 3$

(d) $k = 4$

	71	85	98	111	124	137	150	163
12 ■	84	86	100	113	126	139	152	165
10 ■	97	99	101	115	128	141	154	167
8 ■	110	112	114	116	130	143	156	169
6 ■	123	125	127	129	131	145	158	171
4 ■	136	138	140	142	144	146	160	173
2 ■	149	151	153	155	157	159	161	175
	162	164	166	168	170	172	174	176

(d) $k = 8$

Figure 1. *Illustration de l'ordonnancement pour $n = 8$. Chaque table présente les instants de temps du calcul des éléments $F(i, j, k)$ par le processeur k (pour $k = 1 \ldots 4$, et $k = 8$ la dernière étape). Le symbole x ■'s désigne x cycle d'inactivité. La table pour $k = 4$ est laissée au lecteur à titre d'exercice.*

à jour de chaque ligne (comme indiqué par le 2 ■ dans la figure 1.b). Toutefois, il n'y a pas de cycle d'inactivité au commencement de la dernière ligne parce que celle-ci correspond aux ligne et colonne pivots du processeur précédent, lesquelles étaient déjà calculées à l'avance et stockées dans les registres prévus à cet effet.

Notre compréhension du processus va actuellement s'éclaircir par l'étude de ce qui se passe au niveau du processeur 3 (figure 1.c). On voit qu'actuellement, on a deux cycles d'inactivité en plus au commencement de chaque ligne (le processeur 2 impose une latence additionnelle de 2 cycles sur chaque ligne). Toutefois, on voit aussi que dans l'avant-dernière ligne, on a 2 cycles d'inactivité au lieu de 4. Ceci est dû au fait que dans la dernière ligne du processeur précédent, il n'y avait pas de cycle d'inactivité. Comme toujours, l'ordonnancement suit l'ordre des sous-matrices principales et chaque point $F(i,j,k)$ est calculé immédiatement après que le point $F(i{+}1, j{+}1, k-1)$ ait été envoyé par le processeur précédent.

Nous laissons au lecteur le soin de completer la table du processeur 4 (figure 1.d).
REMARQUE. — L'expression de la fonction de temps décrite dans la figure 1 est donnée comme suit.

$$t(i,j,k) = \begin{cases} \text{if } k = 0 & \text{then } \; 0 \\ \text{if } k \neq 0 \wedge i \geq j & \text{then } \; t_d(i,k) - 2(i-j) \\ \text{if } k \neq 0 \wedge i < j & \text{then } \; t_d(j,k) - 2(j-i) + 1 \end{cases} \tag{8}$$

où

$$t_d(i,k) = \begin{cases} \text{if } i + k \geq n & \text{then } \; n^2 + 2n(i+k-n) + n - i - 1 \\ \text{if } i + k \leq n & \text{then } \; (i+k)^2 + k - 1 \end{cases} \tag{9}$$

Observons que $t_d(i,k)$ est la valeur de $t(i,j,k)$ sur la diagonale (ligne des points (i,j,k) tels que $i = j$).

5. Dérivation formelle du réseau

Nous avons précédemment décrit le mécanisme d'allocation des tâches aux processeurs et donné une explication intuitive du fonctionnement du réseau. Toutefois, nous n'avons pas donné d'éléments de preuve formelle de sa validité. Nous attaquons cette question.

Théorème 1 *La fonction $t(i,j,k)$ définie par (8-9) est un ordonnancement valide pour la variable F dans le système d'équations récurrentes (6-7).*

Preuve. Nous devons montrer que chaque fois qu'un point $(i,j,k) \in D_1$ dépend d'un autre point $(i',j',k') \in D_1$, alors $t(i,j,k) > t(i',j',k')$. Du fait que les vecteurs de dépendance (et même le nombre de dépendances) sont différents dans les différents sous domaines de D_1, la preuve doit suivre la structure du système (6-7). En effet, l'ordonnancement considéré sera valide si et seulement si on montre que le prédicat booléen défini par (10), qui vérifie la cohérence de l'instant de calcul de chaque point, peut être réduit à une tautologie. Ceci s'établit sans une difficulté de fond particulière, même si une rédaction complète implique la prise en compte de plusieurs détails. On pourrait envisager l'usage d'un démonstrateur automatique de théorèmes.

$$X(i,j,k) = \begin{cases} D_1 \cap \{i,j,k \mid i=j=0\} : t(i,j,k) > t(i\dotplus 1, j\dotplus 1, k-1) \\ D_1 \cap \{i,j,k \mid i=0<j\} : t(i,j,k) > \max(\\ \qquad t(i\dotplus 1, j\dotplus 1, k-1), t(0,0,k)) \\ D_1 \cap \{i,j,k \mid i>0=j\} : t(i,j,k) > \max(\\ \qquad t(0,0,k), t(i\dotplus 1, j\dotplus 1, k-1)) \\ D_1 \cap \{i,j,k \mid i,j>0\} \quad : t(i,j,k) > \max(t(i\dotplus 1, j\dotplus 1, k-1), \\ \qquad t(i,0,k), t(1,j,k-1)) \end{cases} \tag{10}$$

□

Rappelons que la fonction d'allocation au processeur est définie comme suit.

$$\forall (i,j,k) \in D_1,\ \ p(i,j,k) = k \tag{11}$$

Maintenant, nous devons montrer que les fonctions de temps et d'allocation ne sont pas en conflit, c'est-à-dire que deux points distincts de D_1 ne sont pas alloués à un même processeur au même instant. En effet, on fait la remarque suivante dont la vérification est directe.
REMARQUE.— $t(i,j,k) = t(i',j',k) \Rightarrow i = i' \wedge j = j'$.

Proposition 1 *Le temps total d'exécution de notre algorithme sur n processeurs est donné par $t(n-1, n-1, n) = 3n^2 - 2n$, ce qui implique une efficacité égale à $\frac{1}{3}$.*

Dans la situation où l'on dispose de $p < n$ processeurs, on peut profiter du fait que les calculs de chaque plan $k = cste$ sont alloués à un seul processeur et que ces calculs sont complètement identiques d'un plan à l'autre, pour réaliser la chaine de récurrence complète en effectuant des passes multiples dans le réseau. Rappelons que cette souplesse constitue un des points forts de notre algorithme, car elle lui confère une modularité naturelle très utile en pratique.

Nous allons maintenant faire une analyse du temps total requis par l'algorithme dans ce cas où il est exécuté sur $p < n$ processeurs avec la stratégie précédemment décrite. Dans la première passe, le k^{eme} processeur commence ses calculs à $t_s(k) = t(0,0,k) = k^2 + k - 1$ et est actif jusqu'à $t_f(k) = t(n-1, n-1, k) = n^2 + 2n(k-1)$. Il est donc actif pendant précisément $t_a(k) = t_f(k) - t_s(k) + 1 = n^2 + 2n(k-1) - (k^2 + k) + 2$ cycles, et puisque $n \geq k$, cette quantité *croit* avec k. Ceci se conçoit aisément dans la mesure où chaque processeur effectue exactement n^2 calculs, mais k augmentant, le processeur a de plus en plus de cycles d'inactivité. Donc, si on commence la prochaine passe dès que le premier processeur est libre (c'est-à-dire à $t = n^2 + 1$), il y aura des conflits.

En revanche, dans le cas d'un réseau partitionné (p processeurs, $p < n$), les cycles d'inactivité sont *écrasés* du fait que le partitionnement force à une inactivité plus importante encore. Puisqu'il y a exactement $t_s(p)$ cycles depuis le début d'une passe (sur le premier processeur) jusqu'au moment où le dernier processeur p entame ses calculs correspondants à cette passe, le *premier* processeur peut commencer sa seconde passe à $t_f(p) + 1 - t_s(p)$, c'est-à-dire $t_a(p) + 1$. Il n'y a donc pas de temps perdu lors du démarrage d'une nouvelle passe. Ce phénomène est illustré par la figure 2. Comme il y a $\frac{n}{p}$ passe, le temps total d'exécution est de $T_p = (\frac{n}{p} - 1)t_a(p) + t_f(p)$, soit :

$$\begin{aligned} T_p &= (\tfrac{n}{p} - 1)(n^2 + 2n(p-1) - (p^2 + p) + 2) + (n^2 + 2n(p-1)) \\ &= \tfrac{n^3}{p} + 2n\tfrac{p-1}{p} - (n-p)(p+1) + 2\tfrac{n}{p} - 2 \end{aligned}$$

D'où le théorème suivant :

Théorème 2 *Le temps total d'exécution sur p processeurs est donné par*

$$T_p = \frac{n^3}{p} + 2n^2\frac{p-1}{p} - (n-p)(p+1) + \frac{2n}{p} - 2 \qquad (12)$$

Corollaire 1 *Pour toute constante (entier positif) α, le temps total d'exécution sur $p = \frac{n}{\alpha}$ processeurs est donné ci-dessous, en supposant que $n \approx n + \alpha \approx n - \alpha$, et en ignorant les termes constants.*

$$T_p \approx n^2 \left(\alpha + 2 - \frac{\alpha - 1}{\alpha^2}\right) \leq n^2(\alpha + 2) \qquad (13)$$

Le travail total de l'algorithme avec des passes multiples sur $\frac{n}{\alpha}$ processeurs est $n^3(1 + \frac{2}{\alpha} - \frac{\alpha-1}{\alpha^3}) \approx (1 + \frac{2}{\alpha})n^3$. Donc, nous pouvons améliorer l'efficacité tout simplement en augmentant α, ce qui correspond à sacrifier le temps d'exécution par un facteur constant, avec une diminution correspondante du nombre de processeurs. Il est important de noter que cette récupération relative de cycles d'inactivité est due au fait que le premier processeur n'est jamais inactif dans aucune passe (voir figure 2), et qu'il émule l'activité d'un processeur qui en aurait eu des cycles d'inactivité dans une version à nombre réduit de passes. Ce mécanisme est illustré par la figure 2.

L'analyse qui vient d'être faite suppose que α est constant, c'est-à-dire qu'on utilise toujours $\Theta(n)$ processeurs, en maintenant proportionnelles la taille de l'architecture et celle du problème. D'un autre côté, le théorème 2 montre que avec un nombre fixe de processeurs, nous avons une implémentation optimale avec un temps d'exécution dominé par le terme $\frac{n^3}{p}$.

Nous allons maintenant nous pencher sur la version bloc de l'algorithme, ceci afin d'établir un certain équilibre entre les opérations et les communications (dont on sait qu'elles ont un coût significatif sur les machines parallèles usuelles).

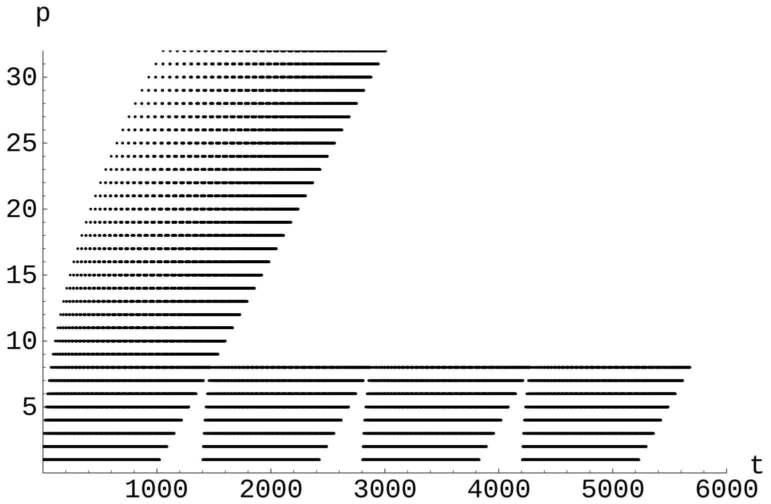

Figure 2. *Illustration du comportement spatio-temporel de deux réseaux (pour $n = 32$), l'un avec 32 processeurs qui s'exécute en une seule passe et termine à $t = 3008$, et l'autre avec 8 processeurs qui effectue 4 passes et termine à $T = 5678$. Un processeur est actif durant les instants représentés par une ligne solide, et inactif durant ceux représentés par des lignes en pointillés. Le lecteur est invité à bien observer la raison pour laquelle le premier processeur n'a pas de cycles d'inactivité.*

6. Version bloc de l'algorithme

6.1. *Formulation par blocs*

Nous commençons par montrer que la spécification de l'APP reste valable si on considère les blocs de points au lieu des entités élémentaires. Ceci semble plus naturel dans les instances de la fermeture transitive et des plus courts chemins. De ce fait, nous allons utiliser l'instance de la fermeture transitive pour illustrer nos propos.
Considérons un graph $G = (S, \Gamma)$ d'ordre N. Pour un diviseur donné p de n, considérons une partition de S en $q = \frac{N}{p}$ sous-ensembles S_i tels que $\mid S_i \mid = p$, $1 \leq i \leq q$. Pour $k \in \{1, ..., q\}$, notre but est de déterminer pour chaque pair $x, y \in S$, si oui ou non il existe un chemin entre x et y qui n'emprunte que les sommets dans S_k. Supposons que le problème de la fermeture transitive est déjà résolu dans S_k (sous-ensemble pivot). Si $\ell^{(k)}$ est un prédicat définie par $\ell^{(k)}(x, y) = 1$ s'il y a un chemin entre x entre y empruntant uniquement les sommets de S_k et $\ell^{(k)}(x, y) = 0$ sinon, alors on a les relations suivantes.

$$\ell^{(k)}(x, y) = \Gamma(x, y) \vee [\bigvee_{z \in S_k} (\Gamma(x, z) \wedge \ell^{(k)}(z, y))] \text{ si } x \notin S_k \text{ et } y \in S_k \tag{14}$$

$$\ell^{(k)}(x, y) = \Gamma(x, y) \vee [\bigvee_{z \in S_k} (\ell^{(k)}(x, z) \wedge \ell^{(k)}(z, y))] \text{ si } x \notin S_k \text{ et } y \notin S_k \tag{15}$$

Considérons les matrices carrées booléennes $M_{ij}^{(k)}$, $1 \leq i, j \leq q$, d'ordre p, définies par

$$M_{ij}^{(k)}(s,t) = \begin{cases} 1 & \text{si } \ell^{(k)}(x_s, y_t) = 1 \text{ pour } x_s \in S_i \text{ et } y_t \in S_j \\ 0 & \text{sinon} \end{cases} \tag{16}$$

$$M_{ij}(s,t) = \begin{cases} 1 & \text{si } \Gamma(x_s, y_t) = 1 \text{ pour } x_s \in S_i \text{ et } y_t \in S_j \\ 0 & \text{sinon} \end{cases} \tag{17}$$

Partant de $S = \{x_1, ..., x_N\}$, on considère la partition $S_k = \{x_{(k-1)p+1}, ..., x_{kp}\}$, $1 \leq k \leq q$. En appliquant la renumérotation $\{1, ..., p\}$ des sommets de chaque sous-ensembles S_k, il vient que les matrices M_{ij} sont les blocs correspondants de la matrice d'adjacence globale du graphe entier. Ainsi, si on traduit en termes d'opérations matricielles le calcul de la fermeture transitive de S_k et les opérations (14) et (15), on obtient une formulation compacte qui équivaut à la formulation considérant des entités élémentaires. Ceci est surtout valable dans les cas où c'est la ligne pivot en cours qui doit être prise en compte au lieu de sa valeur précédente. Cette hypothèse semble nécessaire, car de manière intuitive, les traitements sont effectuées par anticipation (calculs complets dans les blocs) , ce qui justifie le fait qu'il faut d'abord traiter les blocs situés sur la ligne ou colonne pivots avant leur usage les calculs restants. A première vue, seuls les cas de la fermeture transitive et des plus courts chemins (instances les plus couramment concernées) respectent ce prérequis.

6.2. *Recherche de la taille optimale des blocks*

On suppose maintenant que l'algorithme est appliqué à des blocs de taille $b \times b$. Le temps global d'exécution sur une machine parallèle à mémoire distribuée peut être directement modelisé en considérant la formule (12) multipliée par le facteur $\tau b^3 + \beta + \alpha b^2$ (représentant le temps d'une opération sur un bloc suivi de son transfert). Toutefois, la recherche d'une taille optimale de bloc à partir de l'expression obtenue conduit à la résolution d'une équation n'admettant pas de solution directe. Aussi, dans le but de fournir une expression analytique donnant la meilleure taille des blocs, nous allons procéder à quelques suppositions simplificatrices. En effet, si on considère la cas de p processeurs et une matrice de taille $N \times N$ partitionnée en $(\frac{N}{b})^2$ blocks de taille $b \times b$, on a les considérations suivantes :

– Il n'y a pas de latence entre deux passes consécutives sur le premier processeur.

– Le coût de chaque communication, qui est de la forme $\tau b^2 + \beta$, peut être approximé par β du fait des petites valeurs de b et d'un éventuel recouvrement entre la transmission d'un bloc et le traitement du bloc suivant.

– Chaque processeur effectue trois actions sur chaque bloc qui sont : la *réception*, le *calcul* et l'*envoi*.

– Chaque opération sur un bloc implique b^3 opérations scalaires.

Dans ces conditions, et du fait qu'on réalise $\frac{(\frac{N}{b})}{p}$ passes dans le réseau, on obtient que le temps total d'exécution peut être estimé par

$$T(p,b) = \frac{N}{bp}[(\frac{N}{b})^2(\tau b^3 + 2\beta)] + 2(\frac{N}{p})(\tau b^3 + 2\beta), \tag{18}$$

qui peut aussi s'exprimer par

$$T(p,b) = (\frac{N^3}{p}\tau)[(1 + 2\frac{\beta}{\tau b^3}))] + 2(\frac{N}{p})(\tau b^3 + 2\beta). \tag{19}$$

Toujours en supposant des valeurs de b suffisamment petites, ceci nous donne une efficacité de l'ordre de

$$\boxed{e = \frac{1}{1+2\frac{\beta}{\tau b^3}}} \tag{20}$$

La détermination de la valeur de b qui fournit un temps d'exécution minimal s'obtient en résolvant l'équation

$$\frac{\partial T}{\partial b} = -6N(\frac{\beta}{b^4}N^2 - b^2\tau) = 0 \tag{21}$$

qui donne

$$\boxed{b_{\text{opt}} = \sqrt[6]{\frac{\beta}{\tau}N^2}} \tag{22}$$

Nous insistons sur le fait que ceci est une analyse approximative qui vise à situer la taille qui donnerait de bonnes performances en pratique. A la base, le problème est le suivant. Plus les blocs sont grands, plus la latence entre le calcul d'un bloc et son envoi est importante, ce qui pénalise la performance globale. De même, si les blocs sont de très petite taille, le nombre d'appels aux routines de communication sera important et par conséquent, le surcoût de temps imputé aux communications sera prohibitif. Il faut donc trouver une taille qui assure au mieux le compromis entre ces deux aspects. Il apparait donc que trois paramètres sont importants dans cette analyse, il s'agit de la *vitesse des processeurs*, de la *latence des communications*, et de la *taille* du problème. A ce niveau, le lecteur va sans doute être perplexe de constater que le nombre de processeurs semble ne pas intervenir. En effet, l'impact du nombre de processeurs est négligeable devant celui des autres paramètres, de sorte que la performance du programme est moins sensible à variation de la taille du réseau.

7. Résultats expérimentaux

7.1. *Environnement matériel*

La machine INTEL PARAGON est une machine parallèle à mémoire distribuée, composée de 64 nœuds (8 pour le système et 56 pour les programmes utilisateurs) organisés en une grille bidimensionnelle. Chaque nœud contient deux processeurs $i860^{\mathrm{TM}}$ dont l'un est dédié aux calculs et est cadencé à 75 MHZ avec une mémoire locale de 16 MB, tandis que l'autre est spécialisé pour les communications (figure 3).

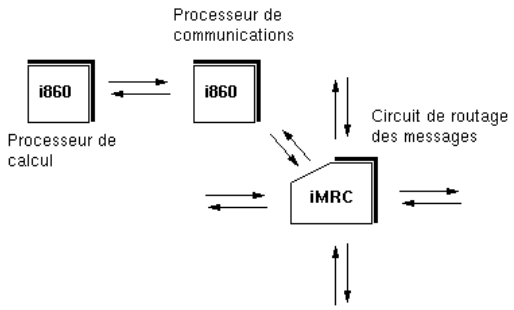

Figure 3. *Principe architectural de la* PARAGON

7.2. *Implémentation*

Le programme a été écrit en FORTRAN 77 avec la librairie de communication NX. C'est un programme de type SPMD dans lequel chaque processeur exécute le code nécessaire à la mise à jour d'un plan. Ici, la synchronisation se fait automatiquement au niveau des communications. Etant donné que chaque bloc envoyé est immédiatement consommé par le processeur qui le reçoit, il en ressort que les seuls cas de bufférisation pouvant apparaitre proviendront uniquement du léger déséquilibre de charge entre les temps de calculs sur les blocs. Par ailleurs, pour ce qui est des passes multiples, nous avons implémenté un mécanisme permettant d'anticiper la réception des blocs d'une passe ultérieure sur le premier processeur, afin d'éviter leur stockage temporaire éventuellement sur disque, ce qui aurait sans doute entrainé une sérieuse perte d'efficacité. Toutefois, ce dernier aspect est inévitable si le graphe est de très grande taille, mais, étant donné que seuls les processeurs extrêmes auraient à faire des accès disque, un certain équilibre pourrait être atteint dans la phase de plein régime.

7.3. *Performances mesurées*

Nous avons considéré le cas d'un graphe de taille $N = 512$. Commençons par évaluer la meilleure taille de bloc conformément à la formule (22). La mesure des paramètres τ et β nous a donné les valeurs suivantes sur la machine INTEL PARAGON : $\tau = 0.05 \times 10^{-6}$ and $\beta = 30 \times 10^{-6}$. L'évaluation par la formule (22) nous donne alors la valeur $b = \sqrt[6]{0.05 \times 30 \times 512} = 8.2 \approx 8$. Ceci nous a amenés à considérer les valeurs de l'ensemble $\{4, 8, 16\}$ comme tailles de blocs pour nos expérimentations. Les résultats obtenus sont résumés dans la table de la figure 4 et la courbe correspondante en figure 5.

Nombre de processeurs	1	2	4	8	16	32
$b = 4$	68	81.5	41	21	11.75	7
$b = 8$	68	39	18	9	5	3
$b = 16$	68	38.5	21	12	8.75	9

Figure 4. *Tableau Performances sur l'*INTEL PARAGON *avec les blocs de taille 4, 8, et 16 (Temps exprimés en secondes)*

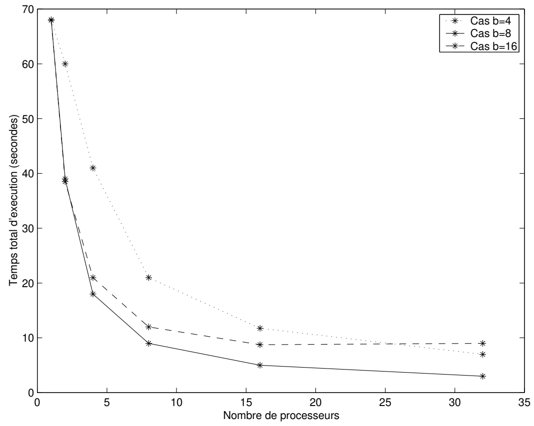

Figure 5. *Courbe de performances sur la* PARAGON *avec les blocs de taille 4,8, et 16*

La table de la figure 4 montre que la valeur $b = 8$ donne les meilleures performances comme prévu. En considérant cette valeur de b, on obtient la table de la figure 6 et la courbe des accélérations optimales en figure 7.

Nombre de processeurs	1	2	4	8	16	32
Temps d'exécution (s)	68	39	18	9	5	3
Accélération	1.00	1.74	3.78	7.56	13.60	22.67

Figure 6. *Table des performances optimales sur la* PARAGON

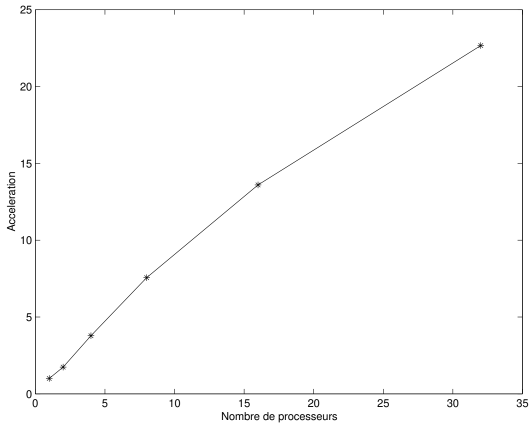

Figure 7. *Courbe des accélérations optimales sur la* PARAGON

7.4. *Commentaires*

De nos expérimentations, il apparaît (comme prévu d'ailleurs) que la taille des blocs a une importance capitale dans les performances globales de l'algorithme. Celle-ci ne doit être ni trop petite (trop d'appels aux routines de communication), ni trop grande (déséquilibre important entre les calculs et les transferts entraînant des inerties momentanées globalement coûteuses). On peut observer que les écarts de performances sont de plus en plus importants lorsque le nombre de processeurs croît. Par ailleurs, la contrainte d'avoir un nombre de processeurs p qui est un diviseur de la taille $\frac{N}{b}$ de la matrice bloc peut être contournée, soit en exécutant l'algorithme sur un graphe augmenté, avec des valeurs correspondant à l'élément neutre pour l'opération scalaire impliquée, soit en prenant en compte le fait que la dernière passe ne devrait pas être exécutée entièrement (même si on peut le faire sans altérer le résultat final, car au delà de l'étape de la récurrence finale, les opérations deviennent sans effet sur les valeurs). Cette deuxième alternative s'implémente sans difficulté particulière sur les machines parallèles standards dans la mesure où le réseau est virtuel. Dans tous les cas, notre estimation de la meilleure taille de bloc semble concorder avec les mesures effectives et de plus, les performances globales sont appréciables.

8. Conclusion

Nous avons proposé et étudié une architecture SIMD pour le problème du chemin algébrique. Nous sommes partis de la forme séquentielle réindexée, afin de bien comprendre ce qui se passe dans un plan et le reprendre dans les autres plans devenus identiques en terme de sous-graphe. Notre algorithme s'exécute en temps $3n^2$ sur n processeurs ayant chacun une mémoire FIFO de taille $2n$. Le réseau s'adapte trivialement à une exécution sur $p < n$ processeurs avec une augmentation d'efficacité. Cet aspect des choses montre qu'on a un réseau dont le fonctionnement ne dépend pas de la taille du problème. Toutefois, même si la taille de la mémoire locale par processeur est variable, on peut s'appuyer sur le fait que la mémoire nécessaire à l'exécution de l'algorithme pour une instance donnée du problème, n'est en fait qu'une partie (réservation d'espace mémoire) d'une mémoire physique qui sera généralement plus grande (la mémoire physique disponible est une borne de l'espace nécessaire). Nous avons aussi étudié l'issue d'une version bloc de notre algorithme, dans le but de réaliser pour une implémentation efficace sur machine parallèle à mémoire distribuée. Les résultats expérimentaux obtenus sur la machine INTEL PARAGON confirment nos analyses d'optimisation sur les tailles des blocs et illustrent l'efficacité de notre algorithme.

Remerciements

Claude Tadonki a été pris en charge par l'agence francaise *Aire dévéloppement* à travers le projet *calcul parallèle*, et par les projets Aladin et Cosi de l'Irisa.

9. Bibliographie

[BEN 90a] BENAINI A., QUINTON P., ROBERT Y., SAOUTER Y., TOURANCHEAU B., «Synthesis of a New Systolic Architecture for the Algebraic Path Problem», *Science of Computer Programming*, vol. 15, 1990, p. 135-158.

[BEN 90b] BENAINI A., ROBERT Y., «Space-time Minimal Systolic Arrays for Gaussian Elimination and the Algebraic Path Problem», *ASAP 90: International Conference on Application Specific Array Processors*, Princeton, NJ, September 1990, IEEE Press, p. 746-757.

[CHA 94] CHANG P. Y., TSAY J. C., «A Family of Efficient Regular Arrays for the Algebraic Path Problem», *IEEE Transactions on Computers*, vol. 43, n° 7, 1994, p. 769-777.

[CLA 92] CLAUSS P., MONGENET C., PERRIN G.-R., «Synthesis of Size-Optimal Toroidal Arrays for the Algebraic Path Problem: A New Contribution», *Parallel Computing*, vol. 18, 1992, p. 185-194.

[GUI 79] GUIBAS L., KUNG H. T., THOMPSON C. D., «Direct VLSI Implementation of Combinatorial Algorithms», *Proc. Conference on Very Large Scale Integration: Architecture, Design and Fabrication*, January 1979, p. 509-525.

[KUN 85] KUNG S. Y., LO S. C., «A Spiral Systolic Algorithm/Architecture for Transitive Closure Problems», *ICCD 85: International Conference on Circuit Design*, Rye Town, NY, 1985, IEEE, p. 622-626.

[KUN 87] KUNG S. Y., LO S. C., LEWIS P. S., «An Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems», *IEEE Transactions on Computers*, vol. C-36, n° 5, 1987, p. 603-614.

[MYO 96] MYOUPO J. F., FABRET C., «A Modular Systolic Linearization of the Warshall-Floyd Algorithm», *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, n° 5, 1996, p. 449-455.

[NAS 84] NASH J. G., HANSEN S., «Modified Faddeew Algorithm for Matrix Multiplication», *Proc., SPIE Workshop on Real-Time Signal Processing*, SPIE, 1984, p. 39-46.

[PRA 89] PRASANNA KUMAR V. K., TSAI Y.-C., «Designing Linear Systolic Arrays», *Journal of Parallel and Distributed Computing*, , n° 7, 1989, p. 441-463.

[RAJ 93] RAJOPADHYE S. V., «An Improved Systolic Algorithm for the Algebraic Path Problem», *INTEGRATION: The VLSI Journal*, vol. 14, n° 3, 1993, p. 279-296.

[RIS 91] RISSET T., ROBERT Y., «Synthesis of Processors Arrays for the Algebraic Path Problem: Unifying Old Results and Deriving New Architectures», *Parallel Processing Letters*, vol. 1, 1991, p. 19-28.

[ROB 85] ROBERT Y., TCHUENTÉ M., «Résolution Systolique de Systèmes Linéaires Denses», *RAIRO Modélisation et Analyse Numérique, Technique et Sciences Informatiques*, vol. 19, n° 2, 1985, p. 315-326.

[ROB 87] ROBERT Y., TRYSTAM D., «Systolic Solution of the Algebraic Path Problem», MOORE W., MCCABE A., URQUHART R., Eds., *Systolic Arrays*, 1, Oxford, UK, 1987, Adam Hilger, p. 171-180.

[ROT 85] ROTE G., «A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion)», *Computing*, vol. 34, n° 3, 1985, p. 191-219.

[SCH 90] SCHIEMAN C. J., CAPPELLO P. R., «A Processor-Time Minimal Systolic Array for Transitive Closure», *International Conference on Application Specific Array Processors*, Princeton, NJ, September 1990, IEEE Computer Society, IEEE Computer Society Press, p. 19-30.

[TAK 92] TAKAOKA T., UMEHARA K., «An Efficient VLSI Circuit for the All Pairs Shortest Path Problem», *Journal of Parallel and Distributed Computing*, vol. 16, 1992, p. 265-270.

[TAY 00] TAYOU DJAMEGNI C., QUINTON P., RAJOPADHYE S., RISSET T., «Derivation of Systolic Algorithms for the Algebraic Path Problem by Recurrence Transformations», *Parallel Computing*, vol. In Press, 2000.

Article reçu le 9 novembre 2000.
Version révisée le 11 janvier 2000.
Rédacteur responsable : HERVÉ GUYENNET

Sanjay Rajopadhye *est chargé de recherche* CNRS *dans le projet Inria Cosi et actuellement responsable de ce projet. Ses contributions scientifiques portent sur les fondements du modèle polyédrique. Il s'intéresse à la synthèse de circuits VLSI de très haut niveau, la parallélisation des programmes, l'algorithmique, l'optimisation ainsi qu'aux circuits asynchrones et aux langages fonctionnels de programmation.*

Tanguy Risset *est chargé de recherche Inria dans le projet Cosi de l'Irisa. Ses activités de recherche portent sur la compilation automatique de circuits spécialisés, en particulier dans le cadre du modèle polyédrique.*

Claude Tadonki *est un chercheur en informatique fondamentale et mathématique combinatoire. Ses domaines d'intérêt sont : le calcul parallèle, l'algorithmique et la complexité, l'algèbre linéaire, la modélisation et l'optimisation. Sa thèse, portant sur l'algorithmique parallèle, a été menée en co-tutelle entre les universites de Rennes I et de Yaounde I (Cameroun).*