A Distributed Recursive Algorithm for Matrix Multiplication on Distributed Memory Computers

Duc Kien NGUYEN

Abstract - **Fast matrix multiplication (FMM) algorithms algorithm to multiply two $n \times n$ matrices reduce the asymptotic operation count from $O(n^3)$ of the traditional algorithm to $O(n^{2.xx})$. Thus for matrix multiplication on distributed memory computer, the application of FMM algorithms always gives considerable results. At present, all the parallel matrix multiplication algorithms must use the traditional algorithm on one or several modules. In this paper, we introduce a performance parallel algorithm which is completely based on Strassen algorithm to benefit its advantage in complexity and also a new method to design distributed recursive algorithms.**

Index Terms **– Distributed recursive algorithms, fast matrix multiplication, Strassen algorithm, recursive wave.**

I. INTRODUCTION

La multiplication de matrices est l'une des opérations les plus fondamentales dans l'algèbre linéaire et sert de module principal dans de nombreux algorithmes, dans la solution des systèmes des équations linéaires, de l'inversion de matrice, de l'évaluation du déterminant de la matrice et de la fermeture transitive d'un graphe. Dans plusieurs cas, l'asymptote des complexités de ces algorithmes dépend directement de la complexité de la multiplication de matrice, ce qui rend très motivant l'étude des possibilités d'accélérer la multiplication matricielle. En outre, l'inclusion de la multiplication matricielle dans beaucoup de problèmes montre son rôle comme facteur déterminant pour la performance d'exécution des calculs à grande vitesse.

C'est la raison pour laquelle beaucoup de scientifiques ont travaillé dur pour améliorer l'algorithme de la multiplication matricielle. Strassen semble être le premier à présenter un meilleur algorithme de multiplication matricielle avec $O(n^{\log_2 7})$ opérations (ci-après référé comme S-algo) alors que les algorithmes classiques ont besoin de $O(n^3)$ opérations (ci-après référé comme T-algo). Ensuite Winograd [6] a présenté une variante de l'algorithme de Strassen avec le même exposant mais le nombre des additions/soustractions est réduit de 18 à 15 (ci-après référé comme W-algo). Le record de la complexité appartient à Coppersmith et Winograd est $O(n^{2,276})$, résolu à partir de l'agrégation arithmétique [2].

Cependant, seulement W-algo et S-algo offrent une meilleure performance que celle de T-algo pour des matrices des tailles pratiques - moins de 10^{20} [11], par conséquent dans cet article, nous concentrons seulement sur l'implémentation de S-algo sur des machines à mémoire distribuée. En fait, notre méthode est applicable avec tous algorithmes de multiplication rapide de matrices, qui sont toujours avec la forme récursive [14].

Généralement, il y avait 3 approches pour paralléliser des algorithmes de FMM. La première approche est d'utiliser T-algo au niveau supérieur entre des processeurs, et des algorithmes de FMM au niveau plus bas dans chaque processeur. Les plus générales algorithmes qui utilisent T-algo entre des processeurs incluent 1D-systolic [7], 2D-systolic [7], Fox (BMR) [6], Cannon [1], PUMMA [3], BiMMeR [9], SUMMA [16], DIMMA [2]. Puisque des algorithmes de FMM sont plus efficaces pour des matrices de grande taille (grâce à la grande différence de la complexité entre l'opération de multiplication et l'opération d'addition/soustraction de la matrice), ils sont bien adaptés pour être utilisé au niveau supérieur. Pour cette raison, dans [12] Luo et Drake ont présenté un algorithme qui utilise S-algo au niveau supérieur et l'algorithme de Fox au niveau le plus bas. C'est le premier travail qui représente la deuxième approche : utiliser des algorithmes de FMM au niveau supérieur entre des processeurs, et des algorithmes parallèles de multiplication matricielle sont utilises aussi entre des processeurs au niveau plus bas. Pour continuer, une amélioration est présentée dans [8] : l'algorithme SUMMA est utilisé au lieu de l'algorithme de Fox. Pour la troisième approche, des algorithmes de FMM sont utilisés au niveau supérieur entre des processeurs, et des algorithmes séquentiels de multiplication matricielle sont utilises localement sur chaque processeur au niveau le plus bas. La première implémentation appliquant cette approche sur le parangon d'Intel (en employant S-algo au dessus et au niveau inférieur) [4] a atteint une meilleure performance que celle de T-algo. Cependant, S-algo dans [4] exige que le nombre de processeurs employés dans le calcul doit être une puissance de 7. C'est une restriction grave puisque la plupart des ordinateurs de MIMD est sous forme d'hypercube ou de grille donc la puissance de sept des nombres de processeurs

Duc Kien NGUYEN: LaISC – Ecole Pratique des Hautes Etudes et Université Paris 8, France (e-mail: Kien.Duc-Nguyen@univ-paris8.fr).

n'est pas un groupement normal. Par conséquent, l'algorithme présenté dans [4] n'est pas adaptable. D'ailleurs, il exige d'une large espace de travail, avec chaque produit des matrices est recalculé 3 ou 4 fois [17], de plus il n'est pas général à cause de l'exécution manuel la récursivité (niveau 2) de S-algo.

Dans cet article, nous présentons un algorithme parallèle qui est complètement basé sur l'algorithme de Strassen pour bénéficier son avantage à la complexité et également une nouvelle méthode pour concevoir des algorithmes récursifs distribués. La conception des algorithmes distribuées doit également inclure une étape de preuve toutes les caractéristiques de l'algorithme. Dans notre approche l'étape de conception est exécutée simultanément avec l'étape de preuve. Notre algorithme est principalement basée sur des schémas récursifs parallèles en utilisant deux existant et largement outils disponibles : l'Appel de Procédure à Distance (APD) et le primitive de l'exécution parallèle des processus PAR (une sémantique proche de celle d'OCCAM [18]). Cette méthodologie peut s'étendre à des problèmes d'optimisation combinatoire comme les algorithmes de Séparation et évaluation Progressive, des algorithmes du type A*, AO*, alpha/bêta ou encore SSS*…

II. STRASSEN ALGORITHM

Nous commençons par le produit $Q=XY$, où

$$Q \in \Re^{m\times n}, X \in \Re^{m\times k} \text{ and } Y \in \Re^{k\times n}.$$

Nous supposerons que m, n, et k sont tous entiers pairs. Par la division

$$X = \begin{pmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{pmatrix},\ Y = \begin{pmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{pmatrix},\ \text{and Q} = \begin{pmatrix} Q_{00} & Q_{01} \\ Q_{10} & Q_{11} \end{pmatrix}$$

où

$$Q_{ij} \in \Re^{\frac{m}{2}\times\frac{n}{2}}, X_{ij} \in \Re^{\frac{m}{2}\times\frac{k}{2}} \text{ and } Y_{ij} \in \Re^{\frac{k}{2}\times\frac{n}{2}},$$

Il peut montrer [6, 11] que les calculs suivants calculent le produit $Q = XY$:

$$\begin{aligned} M_0 &= (X_{00} + X_{11})(Y_{00} + Y_{11}) \\ M_1 &= (X_{10} + X_{11})Y_{00} \\ M_2 &= X_{00}(Y_{01} - Y_{11}) \\ M_3 &= X_{11}(-Y_{00} + Y_{10}) \\ M_4 &= (X_{00} + X_{01})Y_{11} \\ M_5 &= (-X_{00} + X_{10})(Y_{00} + Y_{01}) \\ M_6 &= (X_{01} - X_{11})(Y_{10} + Y_{11}) \\ Q_{00} &= M_0 + M_3 - M_4 + M_6 \\ Q_{01} &= M_1 + M_3 \\ Q_{10} &= M_2 + M_4 \\ Q_{11} &= M_0 + M_2 - M_1 + M_5 \end{aligned} \quad (1)$$

S-algo fait le calcul ci-dessus récursivement jusqu'à ce qu'une des dimensions des matrices soit 1.

III. DISTRIBUTED RECURSIVE STRASSEN ALGORITHM ON DISTRIBUTED MEMORY COMPUTER

A. Vague Récursive

1) Premier Algorithmes à Vagues

Chang est le premier à utiliser la technique de vague. Dans [19] il fait une description des algorithmes distribués d'*écho.* Bien que le nom de vague ne soit pas mentionne dans cet article, le principe des algorithmes présentés est le même. Les algorithmes d'*écho* suivent les principes suivants :

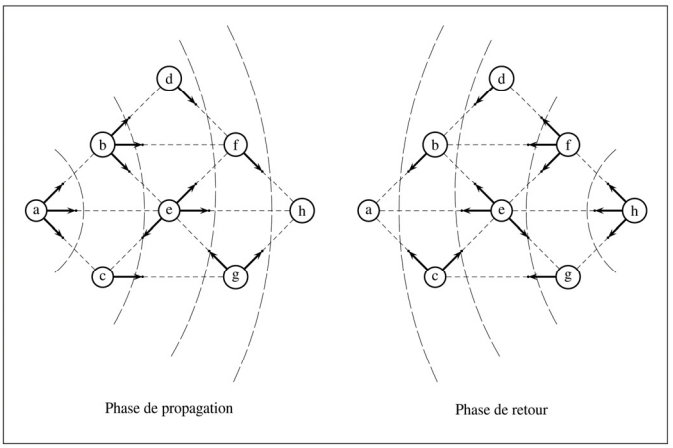

Figure 1: Principe d'évolution de la vague dans un graphe.

- L'opération fondamentale des algorithmes à écho est l'envoi d'un message. La traversée d'un graphe se fait en envoyant des messages d'un nœud vers un autre.
- Pour parcourir un graphe, deux phases sont alors nécessaires
  - une phase d'avance réalisée par des explorateurs,
  - une phase d'écho réalisée par des échos.
- Lorsqu'un explorateur visite pour la première fois un nœud, ce dernier en émet d'autres, en parallèle, vers tous ses voisins directs excepté celui qui lui avait envoyé cet explorateur (ce nœud est appelé le premier voisin). Un explorateur est systématiquement transformé en écho lorsqu'il visite un noeud déjà visité par un autre explorateur.
- Lors d'une visite concurrente, à un noeud, de plusieurs explorateurs, un seul est choisi en tant que premier explorateur. Tous les autres sont alors considérés comme étant des échos.
- Un nœud ne peut transmettre d'écho à son premier voisin que s'il reçoit les échos de tous les explorateurs qu'il a envoyés.
- L'ensemble des explorateurs et des échos transporte des informations à propos des noeuds qu'ils ont visitées.

Les algorithmes d'échos fonctionnent en parallèle et de manière asynchrone en utilisant les envois de messages. Ils peuvent démarrer ou se terminer dans un seul ou plusieurs noeuds. Dans ce dernier cas, les noeuds doivent collaborer, de

manière consistante, même si chacun d'entre eux a initialisé indépendamment sa participation dans l'algorithme. Avec ce type d'algorithmes, Chang apporte une nouvelle technique dans l'algorithmique distribuée qui est le calcul à vague. Plus tard, Raynal et Helary [20] ont présentés les premiers algorithmes à vague comme un outil de synchronisation en donnant une définition formelle de la vague.

2) Algorithmes à Vagues

Puisque dans un système distribué il n'existe pas d'horloge globale, il est difficile d'avoir parmi deux événements quelconques, lequel se produit d'abord. Pour cela nous allons adopter la relation «avant que» (voir [21]), qui sera notée « $\rightarrow$ ». Cette relation de causalité a été définie par Lamport comme étant un ordre partiel sur les événements d'un système distribué.

L'envoi ou la réception d'un message par un processus est considéré par Lamport comme l'un de ces événements. La relation $\rightarrow$ dans l'ensemble des événements doit satisfaire les conditions suivantes :

. si e_1 et e_2 sont des événements au sein du même processus p, et e_1 se produit avant e_2 alors on peut écrire : $e_1 \rightarrow e_2$.

. si e_1 au sein d'un processus p_1 émet un message, et $e2$ au sein d'un processus p_2 reçoit ce message alors on peut écrire : $e_1 \rightarrow e_2$.

. si $e_1 \rightarrow e_2$ et $e_2 \rightarrow e_3$ alors $e_1 \rightarrow e_3$.

Dans [22], Tel donne une définition formelle d'un algorithme à vagues. Soit P l'ensemble des processus et E l'ensemble des canaux de communication. Un calcul est un ensemble d'événements ordonnés par la relation causale «avant-que» $\rightarrow$. Le nombre d'événements d'un calcul C est noté $|C|$, et le sous-ensemble d'événements qui ont lieu dans un processus p est noté C_p. Ces notations permettent d'énoncer la définition de Tel d'un algorithme à vagues:

Définition 1 Un algorithme à vagues est un algorithme distribué qui respecte les trois conditions suivantes:

- *Terminaison*. Chaque calcul est fini : $\forall C : |C| \in \mathbb{N}$

- *Décision*. Dans chaque calcul, il y a un événement de décision.

$\forall C : \exists e \in C : e$ est un événement de décision

- *Dépendance*. Dans chaque calcul, chaque événement de décision est précédé par un autre événement dans chaque processus.

$\forall C : \exists e \in C : e$ est un événement de décision

$\Rightarrow \forall p \in P \exists f \in C_p : f \rightarrow e$

Tel définit une vague comme étant le calcul effectué par un algorithme à vagues [Tel94].

3) Définition de la Vague Récursive

Le concept de la vague récursive a été présenté pour la première fois en 1991 par Florin et Lavallée [23]. Puis il a été repris, avec la collaboration de Gomez, en 1993 ensuite 1995 [24, 25]. Il met en oeuvre à la fois le calcul diffusant, la récursivité et les APD. Un cocktail de concepts et d'outils qui permet d'avoir un outil idéal pour concevoir des algorithmes distribués. Mais avant de passer à la description du schéma de base de la structure de contrôle d'un algorithme distribué, nous allons définir ce qu'est la vague récursive.

Définition 2 Une vague récursive est une procédure qui, durant son exécution, va lancer plusieurs autres, éventuellement aucune, exécutions concurrentes d'elle même dans d'autres processus.

Une vague récursive engendre un arbre des exécutions de procédures. Cet arbre admet les caractéristiques suivantes :

. la racine de l'arbre est associée à la première exécution de la procédure,

. les noeuds non feuilles et la racine sont associés à des procédures dont l'exécution est en attente de la terminaison des appels récursifs qu'elles ont lancés,

. les feuilles sont associées aux exécutions actives de la procédure avant ou après l'exécution des appels récursifs, ou bien elles sont associées à une exécution terminale sans appel récursif.

De plus, une vague récursive définie par une procédure est le plus souvent intégrée dans un ensemble de procédures.

B. Schéma Général de la Vague Récursive

1) Instruction PAR

Puisque nous supposons possible l'exécution d'un APD sur n procédures, nous devons disposer d'une structure de contrôle d'exécution concurrente de n tâches sur un même processeur, au sein d'une même procédure. Nous considérerons donc par la suite, disposer de l'instruction PAR définie ci-après et ayant une sémantique analogue à celle qu'elle a en OCCAM : à chaque valeur prise par le paramètre sur le domaine est associée une tâche parallèle exécutée sur le processeur courant.

Par ⟨**paramètre**⟩ **In** ⟨**Domaine**⟩ **Do**
⟨ **Bloc d'instruction** ⟩ **;**
EndPar

Figure 2: Schéma général de l'instruction PAR.

L'instruction PAR est terminée lorsque toutes les tâches parallèles qu'elle a activé sont terminées. Le calcul se poursuit alors en séquence par l'exécution de l'instruction suivante.

2) Syntaxe de l'APD

Afin de rester cohérent avec l'esprit de l'appel de procédure distante, nous considérerons aussi la structure suivante permettant de définir à la fois une procédure et la localisation de son exécution (le noeud du réseau et le processus au sein duquel elle s'exécute) :

⟨proc-name⟩(⟨parameter_list⟩)ON⟨Proc_Id⟩;

La procédure identifiée par son nom proc-name et ses paramètres rangés dans parameter_list sera exécutée par le processus désigné par Proc_Id. D'autres règles peuvent être ajoutées précisant, suivant le contexte, l'exécution de cette

procédure. Dans ce qui suit un identificateur de processus est généralement associé à une variable qui prend ses valeurs dans un domaine déterminé (le groupe de processus).

3) Schéma Général de la Vague Récursive

Les schémas d'algorithmes que nous présenterons, seront décrits dans un langage très proche d'ADA. Comme nous utiliserons souvent la notion de groupe de processus, nous devons gérer cet ensemble. Nous considérerons donc un paquetage implantant un type ensemble (SETOF) et les opérations qui lui sont associées (union, intersection, ...). La spécification du schéma général de la vague récursive distribuée est donnée par la figure ci-dessous :

```
[A.1]  TYPE Process_Id Is ;              {Définition de type pour un nom de
       processus.}
[A.2]  PROCÉDURE vague_récursive (⟨paramètres⟩) Is
[A.3]     ⟨déclarations⟩ ;
[A.4]     i : Process_Id ;
[A.5]     groupe_de_processus : SETOF Process_Id ;
[A.6]     Begin
[A.7]        ⟨Bloc d'instructions A⟩ ;
[A.8]        If condition Then
[A.9]           ⟨Bloc d'instructions B⟩ ;
[A.10]          Par i In groupe_de_processus Do
[A.11]             ⟨Bloc d'instructions C⟩ ;
[A.12]             vague_récursive( f(⟨paramètres⟩)) ON i ;
[A.13]             ⟨Bloc d'instructions D⟩ ;
[A.14]          EndPar
[A.15]          ⟨Bloc d'instructions E⟩ ;
[A.16]       EndIf
[A.17]       ⟨Bloc d'instructions F⟩ ;
[A.18]    End
[A.19] End vague_récursive.
```

Figure 3: Schéma général de la vague récursive.

. le bloc d'instructions A de la ligne [A.7] représente tout ensemble séquentiel d'instructions exécutées localement, y compris en parallèle.

. la structure de contrôle IF...THEN...ELSE (ligne [A.8]) est celle relative à la _n de la récursivité.

. à la ligne [A.10], l'instruction PAR provoque l'exécution en parallèle, sur chaque processus du groupe, un appel récursif de la procédure *vague_récursive*.

C. Strassen Algorithm on Distributed Memory Computer

Dans cette section, nous allons décrire une version distribuée de l'algorithme de Strassen basé sur la notion de la vague récursive. Nous allons utiliser pour cela le schéma de vague tel qu'il a été spécifié au-dessus.

1) Modèle Distribué de la Multiplication de Strassen

L'étape principale de la méthode de Strassen est le calcul des sept termes m_1, m_2, ...,m_7 tels qu'ils ont été spécifiés par la relation (1). Nous allons considérer que chaque terme m_i est le produit de deux matrices α_i et β_i de taille arbitraire (p x p). Afinn d'obtenir un terme mi, qui est en fait une matrice, il faudra appliquer, de manière récursive, la méthode de Strassen pour effectuer le produit de α_i par β_i. L'application récursive de la méthode se poursuit jusqu'à atteindre un niveau tel que les matrices α_i et β_i correspondent à des scalaires (matrices de taille 1 x 1).

Les m_i représentent des tâches indépendantes [27] puisque leurs données en entrée, à savoir les coefficients a_{ij} et b_{ij}, sont des données non calculées. Ce qui nous permet de calculer les m_i d'une manière concurrente. Nous avons à notre disposition la structure de contrôle PAR qui peut se charger de lancer de tels traitements sur des processeurs différents. Comme la méthode est récursive, le schéma de la vague récursive s'y prête parfaitement. En effet, une procédure récursive qui fait appel à elle-même 7 fois, d'une manière concurrente, sur des processeurs différents, constituera notre approche pour la vague récursive de la multiplication matricielle de Strassen.

2) Algorithme de Strassen par Vague Récursive

Nous présentons ici un algorithme de Strassen basé sur le schéma général de la vague récursive. L'algorithme tient en une procédure (*Straßen_VR*) qui va générer une vague pour le calcul d'un produit matriciel. En effet, lorsque la procédure est exécutée sur un processeur, elle va lancer 7 exécutions d'elle même sur 7 autres processeurs. C'est la structure de contrôle PAR qui, selon le résultat du test, va lancer les 7 appels d'exécution de la même procédure sur des processeurs différents. Ainsi, l'exécution de cet algorithme va engendrer un graphe d'exécution ayant la structure d'un arbre (au sens de la théorie des graphes [26]). Cet arbre est aussi appelé arbre des appels récursifs [28, 24].

Lemme 1

L'arbre des appels récursifs a une hauteur au moins égale à k, avec k = log n.

Preuve :

Considérons la relation (1), nous dirons que les produits mi sont matriciels si leurs opérandes (a_{xy}) et (b_{zt}) sont des matrices de taille au moins (2 x 2), sinon nous dirons qu'ils sont des produits scalaires. La preuve est établie par récurrence :

. n = 1 : Condition de terminaison de la récursivité. L'initiateur calcule localement un simple produit scalaire. L'arbre des appels récursifs dans ce cas se réduit à un seul nœud, la racine, de hauteur zéro.

. n = 2 : L'initiateur aura à récupérer 7 produits scalaires avant de calculer les coefficients C_{ij} de sa matrice C. Il va alors soumettre ces produits scalaires à 7 processeurs. L'arbre des appels récursifs dans ce cas a une hauteur égale à 1 (log2).

. n = 2k : L'initiateur va générer 7 processus qui auront à calculer chacun un produit de deux matrices de taille (n_1 x n_1) avec $n_1 = 2^{k-1}$. Pour k = 1, on se retrouve dans la même situation du cas 2. Si k > 1, alors k niveaux seront générés à

partir de l'initiateur.

Chaque processeur du dernier niveau k doit effectuer un produit scalaire comme dans le cas 1. Donc la plus longue chaîne aura une longueur k, d'où la hauteur de l'arbre des appels récursifs.

□

Remarques :

. la fonction *Alpha(A,B,i)* (respectivement *Béta(A,B,i)*) représente α_i l'opérande gauche (respectivement β_i l'opérande droit) du $i^{ème}$ produit de Strassen m_i de la relation (1).

. la fonction *Reconstituer(C)* permet de regrouper les résultats reçus, suite aux sept APD lancées, pour une matrice *C* selon la relation (1).

```
PROCÉDURE Straßen_VR(A,B) Is
  TYPE id_processus IS ...
  i : id_processus ;
  Begin
    If (Taille(A) > 1) Then   {Condition de propagation de la vague.}
      Par i In [1···7] Do
        Straßen_VR(Alpha(A,B,i), Beta(A,B,i)) ON i ;          {APD sur i}
      EndPar
    EndIf
    Reconstituer(C) ;
  End
End Straßen_VR.
```

Figure 4: La vague récursive de la méthode de Straßen.

IV. COMPLEXITY

Puisque la procédure *Straßen_VR()* va générer un arbre de processus relatifs aux appels récursifs de la même procédure, et que seules les feuilles de cet arbre correspondent aux processus actifs (occupés à effectuer un calcul ou à transmettre un résultat), alors le nombre maximum de processus actifs en même temps est majoré par $7^{\log n}$. Faisons l'hypothèse qu'un cycle de calcul englobe l'achèvement du calcul d'un mi. Alors, selon ce point de vue, nous pouvons énoncer le lemme 2 qui donne le nombre minimum de cycles pour notre méthode de Strassen par vague récursive :

Lemme 2

La procédure Straßen_VR() donne le produit de deux matrices de taille (n x n) en (1 + log n) cycles de calcul au minimum.

Preuve :

La preuve sera établie en utilisant celle du lemme 1. En effet, les cycles de calculs correspondent en fait aux niveaux de l'arbre des appels récursifs. Il est par conséquent simple de voir que la plus longue chaîne a une longueur log n arêtes et passe nécessairement par (1 + log n) nœuds (niveaux).

□

Les extremums énoncés par les lemmes 1 et 2 sont atteint pour un certain nombre maximum de processeurs disponibles. Ce nombre est de $7^{\log n}$ et dénombre ainsi les feuilles (au niveau desquelles un calcul scalaire est effectué) de l'arbre des exécutions de la procédure *Straßen_VR()*. Le théorème 1 généralise le lemme 2 en précisant le nombre de cycles de calcul nécessaires à la méthode de Strassen par vague récursive pour multiplier deux matrices en utilisant *p* processeurs.

Théorème 1

Pour multiplier deux matrices carrées de tailles (n x n) par la méthode de Strassen à vague récursive en utilisant p processeurs, le nombre de cycles de calcul NC est donné par la relation suivante :

$$NC = 1 + r + \left\lceil \sum_{i=r+1}^{\log_2 n} \frac{7^i}{p} \right\rceil$$ où r est défini comme suit :

$$7^r \le p < 7^{r+1}$$

Preuve :

D'après le lemme 2, lorsque $7^{\log n}$ processeurs sont disponibles, le nombre de cycles de calcul est alors de (1 + log n). Cela correspond à $7^{\log n} \le p < 7^{1+\log n}$; avec r = log n. La quantité $\left\lceil \sum_{i=r+1}^{\log_2 n} \frac{7^i}{p} \right\rceil$ est dans ce cas nulle.

En revanche, lorsque $p < 7^{\log n}$, l'arbre des appels récursifs va avoir une largeur (nombre maximum de nœuds par niveau) maximale égale à p. Ce qui oblige le «report» de $\left\lceil \sum_{i=r+1}^{\log_2 n} 7^i \right\rceil$ nœuds (cycles de calcul) sur des niveaux supérieurs ayant une largeur p. Le nombre de ces niveaux supplémentaires est égal à $\left\lceil \sum_{i=r+1}^{\log_2 n} \frac{7^i}{p} \right\rceil$.

□

Par la même relation, nous pouvons facilement véri_er que l'exécution de la procédure d'une manière séquentielle sur un seul processeur nécessitera $\sum_{0}^{\log_2 n} 7^i$ cycles de calcul, puisque *r* vaut 0 et la constante 1 peut être écrite comme 7^0, ce qui permet de l'intégrer dans la somme. Or cette somme correspond au nombre de nœuds de l'arbre des appels récursifs de la procédure *Straßen_VR()*.

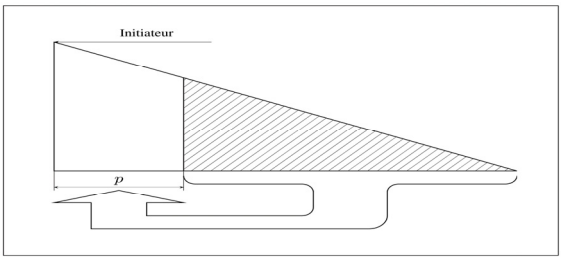

Figure 5: Répartition des cycles de calcul sur un arbre d'exécution de largeur *p*.

V. CONCLUSION

On vient de présenter un algorithme général et adaptable pour tous les algorithmes de multiplication de matrices sur des machines à mémoire distribuée de mémoire qui utilisent des algorithmes de FMM au niveau d'inter-processeur.

Avec une différence considérable aux précédents algorithmes de paralléliser des algorithmes de FMM qui doivent encore utiliser l'algorithme traditionnel dans un ou plusieurs modules, notre algorithme est complètement basé sur des algorithmes de FMM pour bénéficier 100% leurs avantages de complexité. Dire autrement, on a une nouvelle direction à paralléliser des algorithmes de FMM.

Cette méthodologie peut encore s'étendre à des problèmes d'optimisation combinatoire comme les algorithmes de Séparation et évaluation Progressive, des algorithmes du type A*, AO*, alpha/bêta ou encore SSS*…

REFERENCES

[1] L. E. Cannon. *A cellular computer to implement the kalman filter algorithm*. Ph.d. thesis, Montana State University, 1969.

[2] J. Choi. A fast scalable universal matrix multiplication algorithm on distributed-memory concurrent computers. In *11th International Parallel Processing Symposium*, pages 310–317, Geneva, SWITZERLAND, April 1997. IEEE CS.

[3] J. Choi, J. J. Dongarra, and D. W. Walker. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[4] C.-C. Chou, Y. Deng, G. Li, and Y. Wang. Parallelizing strassen's method for matrix multiplication on distributed memory MIMD architectures. *Computers and Math. with Applications*, 30(2):4–9, 1995.

[5] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[6] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.

[7] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins University Press, 2nd edition, 1989.

[8] B. Grayson, A. Shah, and R. van de Geijn. A high performance parallel strassen implementation. *Parallel Processing Letters*, 6(1):3–12, 1996.

[9] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhang. Matrix multiplication on the Intel touchstone delta. *Concurrency: Practice and Experience*, 6(7):571–594, 1994.

[10] B. Kumar, C.-H. Huang, R. W. Johnson, and P. Sadayappan. A tensor product formulation of strassen's matrix multiplication algorithm. *Applied Mathematics Letters*, 3(3):67–71, 1990.

[11] J. Laderman, V. Y. Pan, and H. X. Sha. On practical algorithms for accelerated matrix multiplication. *Linear Adgebra and Its Applications*, 162:557–588, 1992.

[12] Q. Luo and J. B. Drake. A scalable parallel strassen's matrix multiplication algorithm for distributed memory computers. In *Proceedings of the 1995 ACM symposium on Applied computing*, pages 221 – 226, Nashville, Tennessee, United States, 1995. ACM Press.

[13] W. Niklaus. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.

[14] V. Y. Pan. How can we speed up matrix multiplication ? *SIAM Review*, 26(3):393–416, 1984.

[15] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[16] R. van de Geijn and J.Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[17] S. Winograd. On multiplication of 2 x 2 matrices. *Linear Algebra and its Applications*, 4:381–388, 1971.

[18] W. Jones, M. Goldsmith. Programming in OCCAM. Prentice Hall, 1988.

[19] E. Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Trans. On Soft. Eng.*, 8(4), July 1982.

[20] M. Raynal and J. M. Helary. Synchronisation et contrôle des systèmes et des programmes répartis. Eyrolles, Paris, 1988.

[21] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communication of the ACM*, 21(7) :558.565, July 1978.

[22] G. Tel. Introduction to distributed algorithms. University Press, 1994.

[23] G. Florin and I. Lavallée. La récursivité mode de programmation distribuée. Rapport de Recherche 1536, INRIA, Octobre 1991.

[24] G. Florin, C. R. Gomez, and I. Lavallée. Recursive distributed programming schemes. In *Proceedings of ISADS'93*, Kawazaki.Japan, March.April 1993.

[25] C. R. Gomez. *Conception et spécification d'algorithmes distribués : la vague récursive*. Thèses de doctorat, Université Paris 8, Avril 1995.

[26] C. Berge. Graphes et hypergraphes. Dunod, Paris, 1970.

[27] H. Baala. Parallélisation d'algorithmes. Projet de maîtrise d'informatique, Université Paris 8, Département Informatique, Juin 1991.

[28] H. Baala. *Récursivité distribuée et alogorithme SSS**. Mémoire de dea intelligence artificielle, Universités Paris 8 et Paris 13, Octobre 1992.