A General Scalable Implementation of Fast Matrix Multiplication Algorithms on Distributed Memory Computers

Duc Kien NGUYEN, Ivan LAVALLÉE, Marc BUI
Laboratoire de Recherche en Informatique Avancée
University of Paris 8
No. 41, Gay Lussac street, 75005 Paris , France
{Kien.Duc-Nguyen, Ivan.Lavallee, Marc.Bui}@univ-paris8.fr

Quoc Trung HA
Faculty Information Technology
Hanoi University of Technology
No. 1, Dai Co Viet road, Hanoi, Vietnam
Trung.Ha-Quoc@it-hut.edu.vn

Abstract

Fast matrix multiplication (FMM) algorithms to multiply two $n \times n$ matrices reduce the asymptotic operation count from $O(n^3)$ of the traditional algorithm to $O(n^{2.38})$, thus on distributed memory computers, the association of FMM algorithms and the parallel matrix multiplication algorithms always gives remarkable results. Within this association, the application of FMM algorithms at inter-processor level requires us to solve more difficult problems in designing but it forms the most effective algorithms. In this paper, a general model of these algorithms will be presented and we also introduce a scalable method to implement this model on distributed memory computers.

1. Introduction

Matrix multiplication (MM) is one of the most fundamental operations in linear algebra and serves as the main building block in many different algorithms, including the solution of systems of linear equations, matrix inversion, evaluation of the matrix determinant and the transitive closure of a graph. In several cases the asymptotic complexities of these algorithms depend directly on the complexity of matrix multiplication - which motivates the study of possibilities to speed up matrix multiplication. Also, the inclusion of matrix multiplication in many benchmarks points at its role as a determining factor for the performance of high speed computations.

These are the reasons for the birth of the FMM algorithms. Strassen was the first to introduce a better algorithm (hereafter referred as S-algo) [15] for MM with $O(N^{\log_2 7})$than the traditional one (hereafter referred as T-algo) which needs $O(N^3)$ operations. Then Winograd variant [17] of Strassen's algorithm (hereafter referred as W-algo) has the same exponent but a slightly lower constant as the number of additions/subtractions is reduced from 18 down to 15. The record of complexity owed to Coppersmith and Winograd is $O(N^{2.376})$, resulted from arithmetic aggregation [5]. However, only W-algo and S-algo offer better performance than T-algo for matrices of practical sizes, say, less than 10^{20}[11], hence in this paper, we concentralize only on the implementation of W-algo and S-algo on distributed memory computers. In fact, our method is applicable with all the FMM algorithms, which are always with the recursive form [14].

W-algo is a variant of S-algo, hence in this paper we will use S-algo in all the expressions and W-algo is only mentioned when the differences between theses two algorithms appear.

There have been mainly three approaches to parallelize S-algo. The first approach is to use T-algo at the top level (between processors) and S-algo at the bottom level (within one processor). The most commonly algorithms used T-method between processors include 1D-systolic [7], 2D-systolic [7], Fox (BMR) [6], Cannon [1], PUMMA [3], BiMMeR [9], SUMMA [16], DIMMA [2]. Since S-algo is most efficient for large matrices (thanks to the great difference of complexity between the operation multiplication and the operation addition/subtraction of matrix), it is well

suited to use at the top level, not at the bottom level. The second approach is to use S-algo at both the top and the bottom level. The first implementation applying this approach [4] on Intel Paragon reached better performance than T-algo. However, S-algo in [4] requires that the number of processors used in the computation to be a power of seven. This is a severe restriction since many MIMD computers use hypercube or mesh architecture and powers of seven numbers of processors are not a natural grouping. Therefore, the algorithm presented in [4] is not scalable. Moreover, it requires a large working space, with each matrix to be multiplied being duplicated 3 or 4 times. For these reasons, in [12] Luo and Drake explored the possibility of other parallel algorithms with more practical potential: they introduced an algorithm which uses S-algo at the top level and Fox algorithm at the bottom level. This is the first work that represents the third approach: use FMM algorithms at the top level (between processors) and T-algo at the bottom level (also between processors). To continue, an improvement is introduced in [8]: algorithm SUMMA is used in the place of Fox algorithm at the bottom level. The third approach is more complicated than the others, but it gives the scalable and effective algorithms in multiplying large matrices [12].

In this article, we will generalize these algorithms by using Cannon algorithm at the bottom level and show that the total running time for the Strassen-Cannon algorithm decreases when the recursion level r increases. This result is also correct when we replace Cannon algorithm at the bottom level with the other parallel MM algorithms. To use S-algo at the top level, the most significant point is to determine the sub matrices after having recursively executed r time the Strassen formula (these sub matrices are corresponding to the nodes of level r in the execution tree of S-algo) and then to find the result matrix from these sub matrices (corresponding to the process of backtracking the execution tree). It is simple to solve this problem for a sequential machine, but it's much harder for a parallel machine. With a definite value of r, we can manually do it like [4], [12], and [8] made (r = 1, 2, 3) but the solution for the general case has not been found. In this paper, we present our method to determine all the nodes at the unspecified level r in the execution tree of Strassen algorithm, and to show the expression representing the relation between the result matrix and the sub matrices at the level recursion r; this expression allows us to calculate directly the result matrix from the sub matrices calculated by parallel matrix multiplication algorithms at the bottom level. By combining this result with a good storage map of sub matrices to processor, and with the parallel matrix multiplication algorithms based on T-algo (1D-systolic, 2D-systolic, Fox (BMR), Cannon, PUMMA, BiMMeR, SUMMA, DIMMA ...) we have a general scalable implementation of FMM algorithms on distributed memory computers.

2. Background

2.1. Strassen Algorithm

We start by considering the formation of the matrix product $Q = XY$, where

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

We will assume that m, n, and k are all even integers. By partitioning

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

$$Q = \begin{pmatrix} Q_{00} & Q_{01} \\ Q_{10} & Q_{11} \end{pmatrix}$$

where

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

and it can be shown [17, 7] that the following computations compute $Q = XY$:

$$\begin{array}{l} M_0 = (X_{00} + M_{11})(Y_{00} + Y11); \\ M_1 = (X_{10} + X_{11})Y_{00}; M_2 = X_{00}(Y_{01} - Y_{11}); \\ M_3 = X_{11}(-Y_{00} + Y_{10}); M_4 = (X_{00} + X_{01})Y_{11}; \\ M_5 = (X_{10} - X_{00})(Y_{00} + Y_{01}); \\ M_6 = (X_{01} - X_{11})(Y_{10} + Y_{11}); \\ Q_{00} = M_0 + M_3 - M_4 + M_6; \\ Q_{01} = M_1 + M_3; \\ Q_{10} = M_2 + M_4; \\ Q_{11} = M_0 + M_2 - M_1 + M_5; \end{array} \quad (1)$$

S-algo does the above computation recursively until one of the dimensions of the matrices is 1. With Winograd's formula, the number of additions/subtractions is reduced from 18 down to 15:

$$\begin{array}{lll} S_0 = X_{10} + X_{11} & S_1 = S_0 - X_{00} & S_2 = X_{00} - X_{10} \\ S_3 = X_{01} - S_1 & S_4 = Y_{01} - Y_{00} & S_5 = Y_{11} - S_4 \\ S_6 = Y_{11} - Y_{01} & S_7 = S_5 - Y_{10} & \\ M_0 = S_1 S_5 & M_1 + X_{00}Y_{00} & M_2 = X_{01}Y_{10} \\ M_3 = S_2 S_6 & M_4 = S_0 S_4 & M_5 = S_3 Y_{11} \\ & M_6 = X_{11} S_7 & \end{array}$$

$$\begin{array}{ll} T_0 = M_0 + M_1 & T_1 = T_0 + M_3 \\ Q_{00} = M_1 + M_2 & Q_{01} = T_0 + M_4 + M_5 \\ Q_{10} = T_1 - M_6 & Q_{11} = T_1 + M_4 \end{array}$$

2.2. Cannon Algorithm

Cannon algorithm [1] is a commonly used parallel matrix multiply algorithm based on the T-algo. It can be used

on any rectangular processor templates and on matrices of any dimensions [3]. For simplicity of discussion, we only consider square processor templates and square matrices. Suppose we have p^2 processors logically organized in a $p \times p$ mesh. The processor in i^{th} row and j^{th} column has coordinates (i, j), where $0 \leq i, j \leq p$-1. Let matrices X, Y, and Q be of size $m \times m$. For simplicity of discussion we assume m is divisible by p. Let $s = m/p$. All matrices are partitioned into $p \times p$ blocks of $s \times s$ sub matrices. The block with coordinates (i, j) is stored in the corresponding processor with the same coordinates. With the addition of a link between processors on opposite sides of the mesh (a torus interconnection), the mesh can be thought of as composed of rings of processors both in the horizontal and vertical directions. The Cannon method requires communication between the processors of each ring in the mesh. The blocks of the matrix X are passed in parallel to the left along the horizontal rings. The blocks of the matrix Y are passed to the top along the vertical rings. This communication pattern results in the shifting leftwards of matrix X and upwards of matrix Y. Let X_{ij}, Y_{ij}, Q_{ij} stand for the blocks of X, Y, Q respectively stored in the processor with coordinates (i, j). The following pseudo code describes the Cannon algorithm. The running time of the Cannon algorithm consists of two parts: the communication time T_{shift} and the computation time T_{comp}. On the distributed memory computer, the communication time for a single message is

---

The complete i^{th} row of X is shifted leftward i times
(i.e., $X_{ij} \leftarrow X_{i,j+i}$)
The complete j^{th} column of Y is shifted upward j times
(i.e., $Y_{ij} \leftarrow Y_{i+j,j}$)
$Q_{ij} = X_{ij}Y_{ij}$ for all processors $(i,\ j)$
DO $(p-1)$ times
Shift X leftwards and Y upwards
(i.e., $X_{ij} \leftarrow X_{i,j+1}$; $Y_{ij} \leftarrow Y_{i+1,j}$)
$Q_{ij} = Q_{ij} + X_{ij}Y_{ij}$ for all processors
ENDDO

$$T = \alpha + \beta n,$$

where α is the latency, β is the byte-transfer rate, and n is the number of bytes in the message. In the Cannon method, both matrices X and Y are shifted p times. There are a total of 2p shifts. The total latency is 2$p\alpha$. In each shift a sub matrix of order ($m/p \times m/p$) is passed from one processor to another, where m is the dimension of the matrices. Therefore the total byte transfer time is $2p\beta B(m/p)^2$, where B is the number of bytes used to store one entry of the matrices. The total communication time is

$$T_{shif} = 2p\alpha + \frac{2B\beta}{p}m^2. \tag{2}$$

The computation time is

$$T_{comp} = \frac{2t_{comp}}{p^2}m^3, \tag{3}$$

where t_{comp} is the execution time for one arithmetic operation. Here we assume that floating point addition and multiplication has the same speed. The total running time is

$$T(m) = \frac{2t_{comp}}{p^2}m^3 + \frac{2B\beta}{p}m^2 + 2p\alpha. \tag{4}$$

In order to make the Cannon algorithm work, an additional working space of size m^2 is needed to temporarily store the products of the sub matrices of X and Y.

3. General Scalable Implementation of Fast Matrix Multiplication Algorithms

3.1. Strassen-Canon algorithm and storage pattern of matrices

The motivation for the Strassen-Cannon algorithm comes from the observation that S-algo is most efficient for large matrices and therefore should be used at the top level (between processors) instead of the bottom level (within one processor). The 7 sub matrix multiplications of S-algo at each recursion seem at first to lead to a task parallelism. The difficulty in implementing a task parallelism of S-algo on a distributed memory computer results from the fact that the matrices must be distributed among the processors. Sub matrices in S-algo must be stored in different processors and if tasks are spawned these sub matrices must be copied or moved to the appropriate processors [4].

For a distributed memory parallel algorithm the storage map of sub matrices to processors is a primary concern. If the sub matrices used in the S-algo are stored among processors in the same pattern at each level of recursion, then they can be added or multiplied together just as if they are stored within one processor. Here we have a pattern to store the matrices which is based on the result of Luo and Drake in [12]. Figures 1 and 2 show the pattern of storing matrix X with 6 x 6 blocks when the recursion level is 1. Figure 1 is from a matrix point-of-view. Note that the four sub matrices with 3 x 3 blocks are stored among the 9 processors in the same pattern. Figure 2 is from a processor point-of-view. Each processor stores one block of the four sub matrices. Figures 3 and 4 show the pattern when the recursion level is 2. The four sub matrices with 6 x 6 blocks are stored in the same pattern, as well as the 16 sub matrices with 3 x

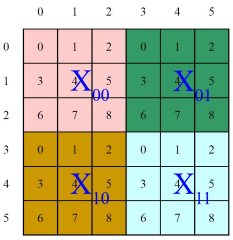

Figure 1. Matrix X with 6×6 blocks is distributed over a 3×3 processor template from a matrix point-of-view. The 9 processors are labeled from 0 to 8. This pattern is used in the Strassen-Cannon algorithm when the recursion level is 1.

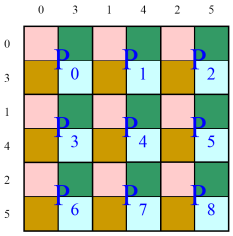

Figure 2. Same as Figure 1, but from a processor point-of-view.

3 blocks. This pattern can be easily replicated for higher recursion levels.

These patterns of storing matrices make it possible for all the processors to act as one processor. Each processor has a portion of each sub matrix at each recursion level. The addition (or subtraction) of sub matrices performed in S-algo at all recursion levels can thus be performed in parallel without any inter processor communication. From the processor point-of-view, each processor does its local sub matrix additions and subtractions. Sub matrix multiplications are calculated recursively using S-algo. At the last level of recursion the sub matrix multiplications are calculated using the Cannon algorithm. Therefore, the Strassen-Cannon algorithm uses S-algo at the top level and the T-algo at the bottom level.

Suppose the recursion level in S-algo is r. Let $n = m/p, m_0 = m/2$, and $n_0 = m_0/p$. Assume $n, m_0, n_0 \in \mathbb{N}$. Since there are 18 sub matrix additions and subtractions and 7 sub matrix multiplications in each recursion, the total running time for the Strassen-Cannon algorithm is:

$$T(m) = 18T_{add}\frac{m}{2} + 7T\frac{m}{2} \tag{5}$$

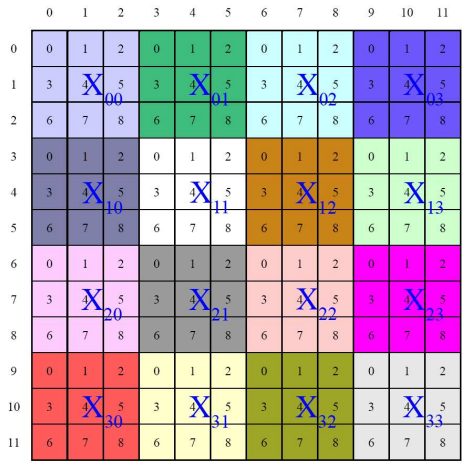

Figure 3. Matrix X with 12×12 blocks is distributed over a 3×3 processor template from a matrix point-of-view. The 9 processors are numbered from 0 to 8. This pattern is used in the Strassen-Cannon algorithm when the recursion level is 2.

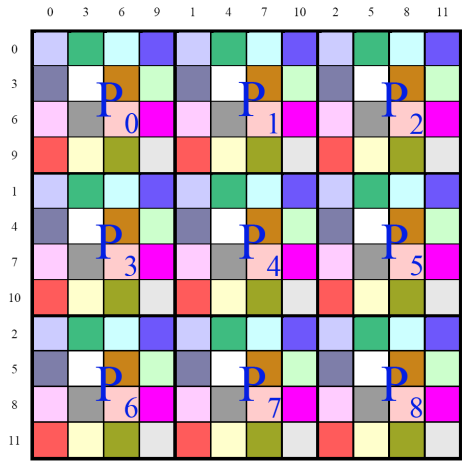

Figure 4. Same as Figure 3, but from a processor point-of-view.

where $T_{add}\frac{m}{2}$ is the running time to add or subtract sub matrices of order $m/2$. Note that there are p^2 processors running in parallel. Therefore:

$$T_{add}\frac{m}{2} = \frac{(\frac{m}{2})^2 t_{comp}}{p^2} \tag{6}$$

Substitute the above formula into equation (5) we have

$$T(m) = 18\left(\frac{18t_{comp}}{4p^2}\right)m^2 + 7T(\frac{m}{2}) = sm^2 + 7T(\frac{m}{2}) \tag{7}$$

where $s = \frac{18t_{comp}}{4p^2}$. With W-algo $s = \frac{15t_{comp}}{4p^2}$. Use the above formula recursively to obtain

$$\begin{aligned} T(m) &= sm^2 + 7T\tfrac{m}{2} \\ &= sm^2 + 7T\left(s\left(\tfrac{m}{2}\right)^2 + 7T\left(\tfrac{m}{4}\right)\right) \\ &= sm^2\left(1+\tfrac{7}{4}\right) + 7^2T\tfrac{m}{2^2} \\ &= sm^2\left(1+\tfrac{7}{4}\right) + 7^2\left(s\left(\tfrac{m}{4}\right)^2 + 7T\tfrac{m}{8}\right) \\ &= sm^2\left(1+\tfrac{7}{4}+\left(\tfrac{7}{4}\right)^2\right) + 7^3T\tfrac{m}{2^3} \qquad (8) \\ \cdots &\quad \cdots \\ &= sm^2\left(1+\tfrac{7}{4}+\cdots+\left(\tfrac{7}{4}\right)^{r-1}\right) + 7^rT\tfrac{m}{2^r} \\ &= sm^2\tfrac{1-\left(\frac{7}{4}\right)^r}{1-\frac{7}{4}} + 7^rT(m_0) \\ &\approx \tfrac{4}{3}s\left(\tfrac{7}{4}\right)^r + 7^rT(m_0) \end{aligned}$$

At the bottom level, the Strassen-Cannon algorithm uses the Cannon algorithm for sub matrix multiplications. Therefore we can use equation (4) to find $T(m_0)$. Substituting the value of $T(m_0)$ and s we have

$$\begin{aligned} T_m &\approx \tfrac{5\left(\frac{7}{4}\right)^r t_{comp}}{p^2}m^2 + 7^r\left(\tfrac{2t_{comp}}{p^2}m_0^3 + \tfrac{2B\beta}{p}m_0^2 + 2p\alpha\right) \\ &= \left(\tfrac{7}{8}\right)^r\tfrac{2t_{comp}}{p^2}m^3 + \tfrac{5\left(\frac{7}{4}\right)^r t_{comp}}{p^2} + \left(\tfrac{7}{4}\right)^r 2p\alpha \end{aligned} \qquad (9)$$

There are four terms in the above equation. The first term is a cubic term with respect to m. It is the computational dominant part and it decreases as the recursion level r increases. The second term is quadratic and it results from the additional sub matrix additions and subtractions in S-algo. It increases as r increases. The last two terms represent the communication time. They increase as r increases. Since the first term is the dominant cubic term, the Strassen-Cannon algorithm should be faster than the Cannon algorithm when m is large enough.

3.2. Recursion Removal in Fast Matrix Multiplication

In formula (9), we showed that the total running time for the Strassen-Cannon algorithm decreases when the recursion level r increases. This result is also correct when we change Cannon algorithm at the bottom level by the other parallel MM algorithms. To use the Strassen algorithm at the top level, the most significant point is to determine the sub matrices after having recursively executed r time the formula (1) (these sub matrices are corresponding to the nodes of level r in the execution tree of Strassen algorithm) and then to find the result matrix from these sub matrices (corresponding to the process of backtracking the execution tree). It is simple to solve this problem for a sequential machine, but it's much harder for a parallel machine. With a definite value of r, we can manually do it like [4], [12], and [8] made ($r = 1, 2, 3$) but the solution for the general case has not been found. The following part presents our method

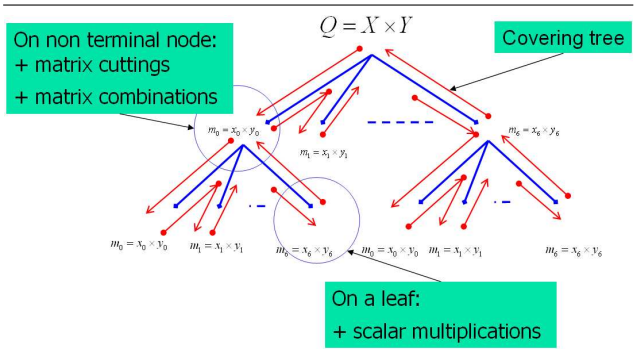

Figure 5. Behavior of Strassen algorithm.

to determine all the nodes at the unspecified level r in the execution tree of Strassen algorithm and to determine the direct relation between the result matrix and the sub matrices at the level recursion r.

Our idea is to create an algorithm that executes the same atomic computations as the recursive one but directly at the leaf of the tree, without the intermediate computing. It is possible because the number of the scalar multiplications is determinable (7^k), so we know exactly how many multiplications we must do. We only have to determine exactly which to do and with which parameters in each step. In each step, the algorithm must execute a multiplication between 2 factors, which are linear combinations of elements of X and Y, respectively. We can consider that each factor is the sum of all elements from each matrix, with coefficient 0, -1, or 1. The execution of the recursive algorithm can be described by an execution tree [13]. In such a representation each scalar multiplication is associated with a leaf of the execution tree (see figure 5). The path from the root to the leaf indicates the recursive calls leading to the corresponding multiplication. Thus, by the fact that all computations in each call are linear, we can composite them at only one computation at a leaf. At the leaf, the coefficient of each element is obtained by the combination of all computation in the path from the root. In each recursive call, the coefficient obtained for each element depends on:

- The index of the call.
- The fact that in which quarter one finds the element in the division of the matrix by 4 sub matrices.

In another way, the recursive algorithm's execution takes a path which covers twice the tree. Our idea is to replace the whole tree's covering path with the leaf's covering path. Because the tree is balanced with determined depth and degree, the leaf covering path can be determined. All computations are linear, so they can be combined in the leaf (see

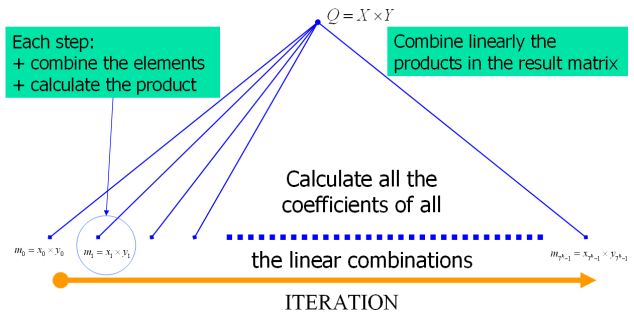

Figure 6. After recursion removal.

figure 6). Each of the 7^k multiplications is a product of 2 linear combinations of X's and Y's elements. Q is a linear combination of these 7^k multiplications.

We represent the Strassen's formula:

$$\begin{aligned} & m_l = \sum_{i,j=0,1} x_{ij} SX(l,i,j) \times \sum_{i,j=0,1} y_{ij} SY(l,i,j) \\ & l = 0 \cdots 6 \\ & \text{and } q_{ij} = \sum_{l=0}^{6} m_l SQ(l,i,j) \end{aligned} \tag{10}$$

with

$SX=$

lij	00	01	10	11
0	1	0	0	0
1	0	1	0	0
2	0	0	1	1
3	-1	0	1	1
4	1	0	-1	0
5	0	0	1	1
6	0	0	0	1

$SY=$

lij	00	01	10	11
0	1	0	0	0
1	0	0	1	0
2	-1	1	0	0
3	1	-1	0	1
4	0	-1	0	1
5	0	1	0	1
6	-1	1	1	-1

$SQ=$

lij	00	01	10	11
0	1	1	1	1
1	1	0	0	0
2	0	1	0	0
3	0	1	1	1
4	0	0	0	1
5	0	1	0	0
6	0	0	0	1

And for the Winograd's formula, we have:

$SX=$

lij	00	01	10	11
0	-1	0	1	1
1	1	0	0	0
2	0	1	0	0
3	1	0	-1	0
4	0	0	1	1
5	1	1	-1	-1
6	0	0	0	1

$SY=$

lij	00	01	10	11
0	1	-1	0	1
1	1	0	0	0
2	0	0	1	0
3	0	-1	0	1
4	0	1	0	-1
5	0	0	0	1
6	1	-1	-1	1

$SQ=$

lij	00	01	10	11
0	0	1	1	1
1	1	1	1	1
2	1	0	0	0
3	0	0	1	1
4	0	1	0	1
5	0	1	0	0
6	0	0	-1	0

Each of 7^k product can be represented as in the following:

$$\begin{aligned} & m_l = \sum_{i,j=0,n-1} x_{ij} SX_k(l,i,j) \times \sum_{i,j=0,n-1} y_{ij} SY_k(l,i,j) \\ & l = 0 \ldots 7^k - 1 \\ & \text{and } q_{ij} = \sum_{l=0}^{7^k-1} m_l SQ_k(l,i,j) \end{aligned} \tag{11}$$

In fact, $SX = SX_1, SY = SY_1, SQ = SQ_1$. Now we have to determine values of matrices SX_k, SY_k, and SQ_k from SX_1, SY_1, and SQ_1. In order to obtain this, we extend the definition of tensor product in [10] for arrays of arbitrary dimensions. Here we show directly our result. The mathematical demonstration of this result, which is a little long and complicated, will be soon presented to you in detail in another paper.

$$\begin{aligned} SX_k(l,i,j) &= \prod_{r=1}^{k} SX(l_r, i_r, j_r) \\ SY_k(l,i,j) &= \prod_{r=1}^{k} SY(l_r, i_r, j_r) \\ SQ_k(l,i,j) &= \prod_{r=1}^{k} SQ(l_r, i_r, j_r) \end{aligned} \tag{12}$$

Apply (12) in (11) we have nodes leafs m_l and all the elements of result matrix.

To implement a fast matrix multiplication algorithm on distributed memory computers, we stop at the recursion level r and thanks to (12) and (11), we have the entire corresponding sub matrices:

$$\begin{aligned} M_l = & \sum_{i,j=0,2^r-1} X_{ij} \left(\prod_{t=1}^{r} SX(l_t, i_t, j_t) \right) \\ & \times \\ & \sum_{i,j=0,2^r-1} Y_{ij} \left(\prod_{t=1}^{r} SY(l_t, i_t, j_t) \right) \\ & l = 0 \ldots 7^r - 1 \end{aligned} \tag{13}$$

with

$$X_{ij} = \begin{pmatrix} x_{i*2^{k-r}, j*2^{k-r}} & \cdots & x_{i*2^{k-r}, j*2^{k-r}+2^{k-r}-1} \\ \cdots & \cdots & \cdots \\ x_{i*2^{k-r}+2^{k-r}-1, j*2^{k-r}} & \cdots & x_{i*2^{k-r}+2^{k-r}-1, j*2^{k-r}+2^{k-r}-1} \end{pmatrix}$$

$$Y_{ij} = \begin{pmatrix} y_{i*2^{k-r}, j*2^{k-r}} & \cdots & y_{i*2^{k-r}, j*2^{k-r}+2^{k-r}-1} \\ \cdots & \cdots & \cdots \\ y_{i*2^{k-r}+2^{k-r}-1, j*2^{k-r}} & \cdots & y_{i*2^{k-r}+2^{k-r}-1, j*2^{k-r}+2^{k-r}-1} \end{pmatrix}$$

$$Q_{ij} = \begin{pmatrix} q_{i*2^{k-r}, j*2^{k-r}} & \cdots & q_{i*2^{k-r}, j*2^{k-r}+2^{k-r}-1} \\ \cdots & \cdots & \cdots \\ q_{i*2^{k-r}+2^{k-r}-1, j*2^{k-r}} & \cdots & q_{i*2^{k-r}+2^{k-r}-1, j*2^{k-r}+2^{k-r}-1} \end{pmatrix}$$

$i = 0, 2^r - 1, j = 0, 2^r - 1$

Thanks to the storage map of sub matrices to processors that we have just presented, the sub matrices

$$\left(\sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} X_{ij} \left(\prod_{t=1}^{r} SX\left(l_t, i_t, j_t\right) \right) \right)$$

and $\left(\sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} Y_{ij} \left(\prod_{t=1}^{r} SY\left(l_t, i_t, j_t\right) \right) \right)$ are locally determined within each processor. Their product M_l will be calculated by parallel algorithms based on T-algo like Fox algorithm, Cannon algorithm, SUMMA, PUMMA, DIMMA...

Finally, thanks to (12) & (11) we have directly sub matrix elements of result matrix by applying matrix additions instead of backtracking manually the recursive tree to calculate the root in [12] and [8]:

$$\begin{aligned} Q_{ij} &= \sum_{l=0}^{7^r-1} M_l SQ_r(l,i,j) \\ &= \sum_{l=0}^{7^r-1} M_l \left(\prod_{t=1}^{r} SQ\left(l_t, i_t, j_t\right) \right) \end{aligned} \tag{14}$$

4. Conclusion

We have just presented a general scalable implementation for all the matrix multiplication algorithms on distributed memory computers that use FMM algorithms at inter-processor level. The running time for these algorithms decreases when the recursion level increases hence this general solution enables us to find the optimal algorithms (which correspond with a definite value of the recursive level and a definite parallel matrix multiplication algorithm at the bottom level) for all the particular cases.

From a different view, we generalized the formulas of Strassen and Winograd for the case where the matrices are divided into 2^k parts (the case $k = 2$ gives us original formulas) thus we have a whole new direction to parallelize the FMM algorithms.

In addition, we are applying the ideas presented in this paper to generalize the algorithm in [4].

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.