Parallelization of Fast Matrix Multiplication Algorithms on Distributed Memory Computers

Duc Kien NGUYEN
CHArt
École Pratique des Hautes Études & Université Paris VIII

Ph.D. thesis defense in Informatics, December 12th 2007.
Jury members :

Prof. Hacène FOUCHAL	Université Antilles Guyane	President
Prof. Alain BUI	Université de Reims	Reviewer
Prof. Gérard FLORIN	CNAM	Reviewer
Prof. Yakup PAKER	University of London	Examiner
Prof. Ivan LAVALLÉE	Université Paris VIII	Thesis advisor

Outline

- Problem
- State of the Art
- Contribution
- Implementation
- Conclusion

Outline

- Problem
  - Introduction
  - Background
    - Strassen's Algorithm & Winograd's Algorithm
    - Cannon Algorithm
- State of the Art
- Contribution
- Implementation
- Conclusion

Outline

- Problem
- State of the Art
  - Traditional Algorithm + FMM Algorithms
  - FMM Algorithms + Sequential MM Algorithms
  - FMM Algorithms + Parallel MM Algorithms
- Contribution
- Implementation
- Conclusion

Outline

- Problem
- State of the art
- Contribution
  - Traditional Algorithm + FMM Algorithms
  - FMM Algorithms + Sequential MM Algorithms
  - FMM Algorithms + Parallel MM Algorithms
- Implementation
- Conclusion

Outline

- Problem
- State of the Art
- Contribution
- Implementation
  - Recursion Removal in Fast Matrix Multiplication (FMM)
  - FMM Algorithms + Sequential Matrix Multiplication (MM) Algorithms
    - Complexity
    - Experimental Results
  - FMM-Cannon(Fox) Algorithm and Pattern to Store the Matrices
  - General Scalable Parallelization of FMM Algorithms
    - Complexity
    - Experimental Results
- Conclusion

Outline

- Problem
- State of the Art
- Contribution
- Implementation
- Conclusion
  - Conclusion
  - Future Works

Outline

- Problem
  - Introduction
  - Background
    - Strassen's Algorithm & Winograd's Algorithm
    - Cannon Algorithm
- State of the Art
- Contribution
- Implementation
- Conclusion

Introduction

- To multiply two $n \times n$ matrices :
  - $\boldsymbol{O}(n^3)$ with the traditional algorithm
  - $\boldsymbol{O}(n^{2.xx})$ with the FMM algorithms

    $\Rightarrow$ *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 presentation :
  - a general model of these algorithms
  - the scalable method to implement this model on distributed memory computers

Strassen's Algorithm & Winograd's Algorithm

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

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

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

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

where

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

Strassen's Algorithm

- It can be shown that the following computations compute $Q = XY$:

$$M_0 = (X_{00} + X_{11})(Y_{00} + Y_{11})$$
$$M_1 = (X_{10} + X_{11})Y_{00}$$
$$M_2 = X_{00}(Y_{01} - Y_{11})$$
$$M_3 = X_{11}(-Y_{00} + Y_{10})$$
$$M_4 = (X_{00} + X_{01})Y_{11}$$
$$M_5 = (-X_{00} + X_{10})(Y_{00} + Y_{01})$$
$$M_6 = (X_{01} - X_{11})(Y_{10} + Y_{11})$$

$$Q_{00} = M_0 + M_3 - M_4 + M_6$$
$$Q_{01} = M_1 + M_3$$
$$Q_{10} = M_2 + M_4$$
$$Q_{11} = M_0 + M_2 - M_1 + M_5$$

- Strassen's algorithm does above computation recursively until one of the dimensions of the matrices is 1.

Winograd's Algorithm

- It can be shown that the following computations compute $Q = XY$:

$$S_0 = X_{10} + X_{11} \qquad S_1 = S_0 - X_{00} \qquad S_2 = X_{00} - X_{10}$$
$$S_3 = X_{01} - S_1 \qquad S_4 = Y_{01} - Y_{00} \qquad S_5 = Y_{11} - S_4$$
$$S_6 = Y_{11} - Y_{01} \qquad S_7 = S_5 - Y_{10}$$
$$M_0 = S_1 S_5 \qquad M_1 = X_{00} Y_{00} \qquad M_2 = X_{01} Y_{10}$$
$$M_3 = S_2 S_6 \qquad M_4 = S_0 S_4 \qquad M_5 = S_3 Y_{11}$$
$$M_6 = X_{11} S_7$$
$$T_0 = M_0 + M_1 \qquad T_1 = T_0 + M_3$$
$$Q_{00} = M_1 + M_2$$
$$Q_{01} = T_0 + M_4 + M_5$$
$$Q_{10} = T_1 - M_6$$
$$Q_{11} = T_1 + M_4$$

- Winograd's algorithm does above computation recursively until one of the dimensions of the matrices is 1.

Cannon Algorithm

- Cannon algorithm is a commonly used parallel matrix multiply algorithm based on the traditional algorithm. It can be used on any rectangular processor templates and on matrices of any dimensions.
- 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.

Cannon Algorithm

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., $Y_{ij} \leftarrow X_{i+j,j}$)

$Q_{ij} = X_{ij}Y_{ij}$ 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

Outline

- Problem
- State of the Art
  - Traditional Algorithm + FMM Algorithms
  - FMM Algorithms + Sequential MM Algorithms
  - FMM Algorithms + Parallel MM Algorithms
- Contribution
- Implementation
- Conclusion

Traditional algorithm + FMM algorithms

There have been mainly three approaches to parallelize FMM algorithms

1. The first approach is to use the traditional algorithm (hereafter referred as T-algo) at the top level (between processors) and FMM algorithms at the bottom level (within one processor).
   - The most commonly algorithms used T-method between processors include 1D-systolic, 2D-systolic, Fox (BMR), Cannon, PUMMA, BiMMeR, SUMMA, DIMMA …
   - Since FMM algorithms is more 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 the bottom level.

FMM algorithms + sequential MM algorithms

2. The second approach is to use FMM algorithms at the top level (between processors) and FMM algorithms or T-algo at the bottom level (within one processor).
   - The first implementation applying this approach on Intel Paragon [*] reached performance better than T-algo.
   - However, S-algo in [*] requires that the number of processors used in the computation 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 [*]'s algorithm is not scalable.

---

[*] C.-C. Chou, Y.-F. Deng, G. Li, and Y. Wang, "Parallelizing Strassen's Method for Matrix Multiplication on Distributed Memory MIMD architectures", *Computers & Math. with Applications*, vol. 30, no. 2, p. 4-9, 1995

FMM algorithms + parallel MM algorithms

3. The third approach is to use FMM algorithms at the top level (between processors) and T-algo at the bottom level (also between processors).
   - Luo and Drake [1995] introduced an algorithm using S-algo at the top level and Fox algorithm at the bottom level.
   - To continue, an improvement is presented by Grayson, Shah and Geijn [1995]: 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.

FMM algorithms at the inter-processor level

3. The third approach is to use FMM algorithms at the top level (between processors) and T-algo at the bottom level (also between processors).
   - Luo and Drake [1995] introduced an algorithm using S-algo at the top level and Fox algorithm at the bottom level.
   - To continue, an improvement is presented by Grayson, Shah and Geijn [1995]: 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.

FMM algorithms at the inter-processor level

How did the precedent works do to parallelize the FMM algorithms at inter-processor level ?

- First, decompose the matrix X into 2 x 2 blocks of sub matrices Xij where $i, j = 0, 1$, then further decompose these four sub matrices into four 2 x 2 (i.e. 4 x 4) blocks of submatrices x_{ij} where $i, j = 0, 1, 2, 3$

$$X = \begin{pmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{pmatrix} = \begin{pmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{pmatrix}$$

- Similarly, perform the same decomposition on matrix Y and get

$$Y = \begin{pmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{pmatrix} = \begin{pmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{pmatrix}$$

FMM algorithms at the inter-processor level

How did the precedent works do to parallelize the FMM algorithms at inter-processor level ?

- Then, use the Strassen's formula to multiply the matrices X and Y, and get the following seven matrix multiplication expressions

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

FMM algorithms at the inter-processor level

How did the precedent works do to parallelize the FMM algorithms at inter-processor level ?

- Next, apply the Strassen's formula to these seven expressions to obtain 49 matrix multiplication expressions on sub matrices x and y. Taking *M0* as an example

$$M_{00} = (x_{00} + x_{22} + x_{11} + x_{33})(y_{00} + y_{22} + y_{11} + y_{33})$$
$$M_{01} = (x_{10} + x_{32} + x_{11} + x_{33})(y_{00} + y_{22})$$
$$M_{02} = (x_{00} + x_{22})(y_{01} + y_{23} - y_{11} - y_{33})$$
$$M_{03} = (x_{11} + x_{33})(y_{10} + y_{32} - y_{00} - y_{22})$$
$$M_{04} = (x_{00} + x_{22} + x_{01} + x_{23})(y_{11} + y_{33})$$
$$M_{05} = (x_{10} + x_{32} - x_{00} - x_{22})(y_{00} + y_{22} + y_{01} + y_{23})$$
$$M_{06} = (x_{01} + x_{23} - x_{11} - x_{33})(y_{10} + y_{32} + y_{11} + y_{33})$$

FMM algorithms at the inter-processor level

How did the precedent works do to parallelize the FMM algorithms at inter-processor level ?

- Similarly, each of the remaining six matrix multiplication expressions M_i for $i = 1, 2, \ldots, 6$ can also be expanded into six groups of matrix multiplications in terms of x and y

$$M_{10} = (x_{20} + x_{22} + x_{31} + x_{33})(y_{00} + y_{11})$$
$$M_{11} = (x_{30} + x_{32} + x_{31} + x_{33})\, y_{00}$$
$$M_{12} = (x_{20} + x_{22})(y_{01} - y_{11})$$
$$M_{13} = (x_{31} + x_{33})(y_{10} - y_{00})$$
$$M_{14} = (x_{20} + x_{22} + x_{21} + x_{23})\, y_{11}$$
$$M_{15} = (x_{30} + x_{32} - x_{20} - x_{22})(y_{00} + y_{01})$$
$$M_{16} = (x_{21} + x_{23} - x_{31} - x_{33})(y_{10} + y_{11})$$

$$M_{20} = (x_{00} + x_{11})(y_{02} - y_{22} + y_{13} - y_{33})$$
$$M_{21} = (x_{10} + x_{11})(y_{02} - y_{22})$$
$$M_{22} = x_{00}(y_{03} - y_{23} - y_{13} + y_{33})$$
$$M_{23} = x_{11}(y_{12} - y_{32} - y_{02} + y_{22})$$
$$M_{24} = (x_{00} + x_{01})(y_{13} - y_{33})$$
$$M_{25} = (x_{10} - x_{00})(y_{02} - y_{22} + y_{03} - y_{23})$$
$$M_{26} = (x_{01} - x_{11})(y_{12} - y_{32} + y_{13} - y_{33})$$

FMM algorithms at the inter-processor level

How did the precedent works do to parallelize the FMM algorithms at inter-processor level ?

$$M_{30} = (x_{22} + x_{33})(y_{20} - y_{00} + y_{31} - y_{11})$$
$$M_{31} = (x_{32} + x_{33})(y_{20} - y_{00})$$
$$M_{32} = x_{22}(y_{21} - y_{01} - y_{31} + y_{11})$$
$$M_{33} = x_{33}(y_{30} - y_{10} - y_{20} + y_{00})$$
$$M_{34} = (x_{22} + x_{23})(y_{31} - y_{11})$$
$$M_{35} = (x_{32} - x_{22})(y_{20} - y_{00} + y_{21} - y_{01})$$
$$M_{36} = (x_{22} + x_{33})(y_{30} - y_{10} + y_{31} - y_{11})$$

$$M_{40} = (x_{00} + x_{02} + x_{11} + x_{13})(y_{22} + y_{33})$$
$$M_{41} = (x_{10} + x_{12} + x_{11} + x_{13})y_{22}$$
$$M_{42} = (x_{00} + x_{02})(y_{23} - y_{33})$$
$$M_{43} = (x_{11} + x_{13})(y_{32} - y_{22})$$
$$M_{44} = (x_{00} + x_{02} + x_{01} + x_{03})y_{33}$$
$$M_{45} = (x_{10} + x_{13} - x_{00} - x_{02})(y_{22} + y_{23})$$
$$M_{46} = (x_{01} + x_{03} - x_{11} - x_{13})(y_{32} + y_{33})$$

FMM algorithms at the inter-processor level

How did the precedent works do to parallelize the FMM algorithms at inter-processor level ?

- Therefore, they have identified 7 x 7 = 49 matrix multiplications and naturally they will either use 7 or 49 processors to perform these matrix multiplications of *x* and *y*

$$M_{50} = (x_{20} - x_{00} + x_{31} - x_{11})(y_{00} + y_{02} + y_{11} + y_{13})$$
$$M_{51} = (x_{30} - x_{10} + x_{31} - x_{11})(y_{00} + y_{02})$$
$$M_{52} = (x_{20} - x_{00})(y_{01} + y_{03} - y_{11} - y_{13})$$
$$M_{53} = (x_{31} - x_{11})(y_{10} + y_{12} - y_{00} - y_{02})$$
$$M_{54} = (x_{20} - x_{00} + x_{21} - x_{01})(y_{11} + y_{13})$$
$$M_{55} = (x_{30} - x_{10} - x_{20} + x_{00})(y_{00} + y_{02} + y_{01} + y_{03})$$
$$M_{56} = (x_{21} - x_{01} - x_{31} + x_{11})(y_{10} + y_{12} + y_{11} + y_{13})$$
$$M_{60} = (x_{02} - x_{22} + x_{13} - x_{22})(y_{20} + y_{22} + y_{31} + y_{33})$$
$$M_{61} = (x_{12} - x_{32} + x_{13} - x_{33})(y_{20} + y_{22})$$
$$M_{62} = (x_{02} - x_{22})(y_{21} + y_{23} - y_{31} - y_{33})$$
$$M_{63} = (x_{13} - x_{33})(y_{30} + y_{32} - y_{20} - y_{22})$$
$$M_{64} = (x_{02} - x_{22} + x_{03} - x_{23})(y_{31} + y_{33})$$
$$M_{65} = (x_{12} - x_{32} - x_{02} + x_{22})(y_{20} + y_{22} + y_{21} + y_{23})$$
$$M_{66} = (x_{03} - x_{23} - x_{13} + x_{33})(y_{30} + y_{32} + y_{31} + y_{33})$$

FMM algorithms at the inter-processor level

How did the precedent works do to parallelize the FMM algorithms at inter-processor level ?

- After finishing these 49 matrix multiplications, they need to combine the resulting M_{ij} where $i, j = 0, 1, \ldots, 6$ to form the final product matrix

$$Q = \begin{pmatrix} Q_{00} & Q_{01} \\ Q_{10} & Q_{11} \end{pmatrix} = \begin{pmatrix} q_{00} & q_{01} & q_{02} & q_{03} \\ q_{10} & q_{11} & q_{12} & q_{13} \\ q_{20} & q_{21} & q_{22} & q_{23} \\ q_{30} & q_{31} & q_{32} & q_{33} \end{pmatrix}$$

- First, define some variables

$$\delta_i = \begin{cases} -1, & \text{if } i = 4 \\ 1 & \text{otherwise} \end{cases} \qquad \gamma_i = \begin{cases} -1, & \text{if } i = 1 \\ 1 & \text{otherwise} \end{cases}$$

$$S_1 = \{0,3,4,6\},\ S_2 = \{1,3\},\ S_3 = \{2,4\},\ \text{and } S_4 = \{0,1,2,5\}$$

FMM algorithms at the inter-processor level

How did the precedent works do to parallelize the FMM algorithms at inter-processor level ?

- The 4 x 4 blocks of sub matrices forming the product matrix Q can be written as :

$$q_{00} = \sum_{i \in S_1} \delta_i \left(M_{i0} + M_{i3} - M_{i4} + M_{i6} \right) \qquad q_{10} = \sum_{i \in S_1} \delta_i \left(M_{i1} + M_{i3} \right)$$

$$q_{01} = \sum_{i \in S_1} \delta_i \left(M_{i2} + M_{i4} \right) \qquad q_{11} = \sum_{i \in S_1} \delta_i \left(M_{i0} + M_{i2} - M_{i1} + M_{i5} \right)$$

$$q_{02} = \sum_{i \in S_3} M_{i0} + M_{i3} - M_{i4} + M_{i6} \qquad q_{12} = \sum_{i \in S_3} M_{i1} + M_{i3}$$

$$q_{03} = \sum_{i \in S_3} M_{i2} + M_{i4} \qquad q_{13} = \sum_{i \in S_3} M_{i0} + M_{i2} - M_{i1} + M_{i5}$$

$$q_{20} = \sum_{i \in S_2} M_{i0} + M_{i3} - M_{i4} + M_{i6} \qquad q_{30} = \sum_{i \in S_2} M_{i1} + M_{i3}$$

$$q_{21} = \sum_{i \in S_2} M_{i2} + M_{i4} \qquad q_{31} = \sum_{i \in S_2} M_{i0} + M_{i2} - M_{i1} + M_{i5}$$

$$q_{22} = \sum_{i \in S_4} \gamma_i \left(M_{i0} + M_{i3} - M_{i4} + M_{i6} \right) \qquad q_{32} = \sum_{i \in S_4} \gamma_i \left(M_{i1} + M_{i3} \right)$$

$$q_{23} = \sum_{i \in S_4} \gamma_i \left(M_{i2} + M_{i4} \right) \qquad q_{33} = \sum_{i \in S_4} \gamma_i \left(M_{i0} + M_{i2} - M_{i1} + M_{i5} \right)$$

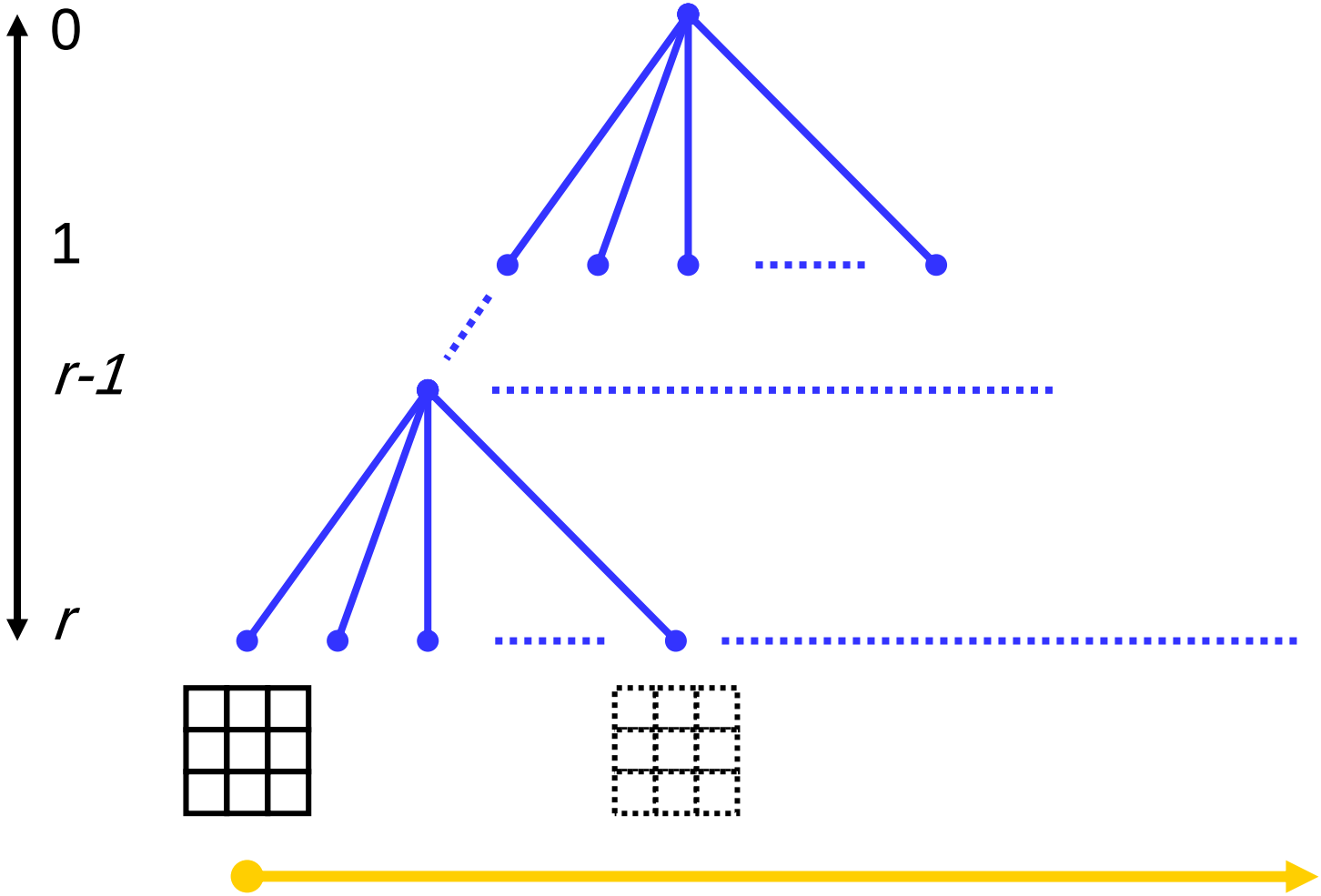
Outline

- Problem
- State of the art
- Contribution
  - Traditional Algorithm + FMM Algorithms
  - FMM Algorithms + Sequential MM Algorithms
  - FMM Algorithms + Parallel MM Algorithms
- Implementation
- Conclusion

Traditional algorithm + FMM algorithms

- Experiment these algorithms to highlight the advantages/disadvantages of the parallel MM algorithms which are based on the traditional algorithm.
  - These experimental results enable us to pertinently use these algorithms in the 3rd approach and to show the comparison of execution between our methods and the others.
- Performance tests for all the algorithms discussed here are carried out on a Fujitsu Siemens Computers/hpcLine at JAIST (Japan Advanced Institute of Science and Technology):
  - 16 nodes (32 CPUs) switched Ethernet 1Gbit
  - Intel Xeon 2,4GHz
  - 2GB RAM

FMM algorithms at the inter-processor level



FMM algorithms at the inter-processor level

- The method to determine all the nodes at the unspecified level *r* in the execution tree of FMM algorithms
- The expression representing the relation between the result matrix and the sub matrices at the recursion level *r*
- A good storage map of sub matrices to processor

Theses results + the parallel matrix multiplication algorithms based on T-algo (1D-systolic, 2D-systolic, Fox (BMR), Cannon, PUMMA, BiMMeR, SUMMA, DIMMA …) -> a general scalable parallelization of FMM algorithms on distributed memory computers.

Outline

- Problem
- State of the Art
- Contribution
- Implementation
  - Recursion Removal in Fast Matrix Multiplication (FMM)
  - FMM Algorithms + Sequential Matrix Multiplication (MM) Algorithms
    - Complexity
    - Experimental Results
  - FMM-Cannon(Fox) Algorithm and Pattern to Store the Matrices
  - General Scalable Parallelization of FMM Algorithms
    - Complexity
    - Experimental Results
- Conclusion

Recursion Removal in Fast Matrix Multiplication

- We represent the Strassen's formula:

$$M_0 = (X_{00}.1 + X_{01}.0 + X_{10}.0 + X_{11}.1) \times (Y_{00}.1 + Y_{01}.0 + Y_{10}.0 + Y_{11}.1)$$
$$M_1 = (X_{00}.0 + X_{01}.0 + X_{10}.1 + X_{11}.1) \times (Y_{00}.1 + Y_{01}.0 + Y_{10}.0 + Y_{11}.0)$$
$$M_2 = (X_{00}.1 + X_{01}.0 + X_{10}.0 + X_{11}.0) \times (Y_{00}.0 + Y_{01}.1 + Y_{10}.0 + Y_{11}.-1)$$
$$M_3 = (X_{00}.0 + X_{01}.0 + X_{10}.0 + X_{11}.1) \times (Y_{00}.-1 + Y_{01}.0 + Y_{10}.1 + Y_{11}.0)$$
$$M_4 = (X_{00}.1 + X_{01}.1 + X_{10}.0 + X_{11}.0) \times (Y_{00}.0 + Y_{01}.0 + Y_{10}.0 + Y_{11}.1)$$
$$M_5 = (X_{00}.-1 + X_{01}.0 + X_{10}.1 + X_{11}.0) \times (Y_{00}.1 + Y_{01}.1 + Y_{10}.0 + Y_{11}.0)$$
$$M_6 = (X_{00}.0 + X_{01}.1 + X_{10}.0 + X_{11}.-1) \times (Y_{00}.0 + Y_{01}.0 + Y_{10}.1 + Y_{11}.1)$$

$$Q_{00} = M_0.1 + M_1.0 + M_2.0 + M_3.1 + M_4.-1 + M_5.0 + M_6.1$$
$$Q_{01} = M_0.0 + M_1.1 + M_2.0 + M_3.1 + M_4.0 + M_5.0 + M_6.0$$
$$Q_{10} = M_0.0 + M_1.0 + M_2.1 + M_3.0 + M_4.1 + M_5.0 + M_6.0$$
$$Q_{11} = M_0.1 + M_1.-1 + M_2.1 + M_3.0 + M_4.0 + M_5.1 + M_6.0$$

$$\Longrightarrow \quad M_l = \sum_{i=0,1\, j=0,1} X_{ij} SX(l,i,j) \times \sum_{i=0,1\, j=0,1} Y_{ij} SY(l,i,j)$$

$$l = 0 \ldots 6$$

$$\text{and } Q_{ij} = \sum_{l=0}^{6} M_l SQ(l,i,j)$$

Recursion Removal in Fast Matrix Multiplication

$$M_0 = (X_{00}.1 + X_{01}.0 + X_{10}.0 + X_{11}.1) \times (Y_{00}.1 + Y_{01}.0 + Y_{10}.0 + Y_{11}.1)$$
$$M_1 = (X_{00}.0 + X_{01}.0 + X_{10}.1 + X_{11}.1) \times (Y_{00}.1 + Y_{01}.0 + Y_{10}.0 + Y_{11}.0)$$
$$M_2 = (X_{00}.1 + X_{01}.0 + X_{10}.0 + X_{11}.0) \times (Y_{00}.0 + Y_{01}.1 + Y_{10}.0 + Y_{11}.-1)$$
$$M_3 = (X_{00}.0 + X_{01}.0 + X_{10}.0 + X_{11}.1) \times (Y_{00}.-1 + Y_{01}.0 + Y_{10}.1 + Y_{11}.0)$$
$$M_4 = (X_{00}.1 + X_{01}.1 + X_{10}.0 + X_{11}.0) \times (Y_{00}.0 + Y_{01}.0 + Y_{10}.0 + Y_{11}.1)$$
$$M_5 = (X_{00}.-1 + X_{01}.0 + X_{10}.1 + X_{11}.0) \times (Y_{00}.1 + Y_{01}.1 + Y_{10}.0 + Y_{11}.0)$$
$$M_6 = (X_{00}.0 + X_{01}.1 + X_{10}.0 + X_{11}.-1) \times (Y_{00}.0 + Y_{01}.0 + Y_{10}.1 + Y_{11}.1)$$

$$Q_{00} = M_0.1 + M_1.0 + M_2.0 + M_3.1 + M_4.-1 + M_5.0 + M_6.1$$
$$Q_{01} = M_0.0 + M_1.1 + M_2.0 + M_3.1 + M_4.0 + M_5.0 + M_6.0$$
$$Q_{10} = M_0.0 + M_1.0 + M_2.1 + M_3.0 + M_4.1 + M_5.0 + M_6.0$$
$$Q_{11} = M_0.1 + M_1.-1 + M_2.1 + M_3.0 + M_4.0 + M_5.1 + M_6.0$$

$SX=$

l\ij	00	01	10	11
0	1	0	0	1
1	0	0	1	1
2	1	0	0	0
3	0	0	0	1
4	1	1	0	0
5	-1	0	1	0
6	0	1	0	-1

$SY=$

l\ij	00	01	10	11
0	1	0	0	1
1	1	0	0	0
2	0	1	0	-1
3	-1	0	1	0
4	0	0	0	1
5	1	1	0	0
6	0	0	1	1

$SQ=$

l\ij	00	01	10	11
0	1	0	0	1
1	0	1	0	-1
2	0	0	1	1
3	1	1	0	0
4	-1	0	1	0
5	0	0	0	1
6	1	0	0	0

Recursion Removal in Fast Matrix Multiplication

- For the Winograd's formula :

SX=

Ni j	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=

Ni j	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=

Ni j	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

Recursion Removal in Fast Matrix Multiplication

- Each of 7^r products can be represented in the following way:

$$M_l = \sum_{i=0,n-1\, j=0,n-1} X_{ij} SX_r(l,i,j) \times \sum_{i=0,n-1\, j=0,n-1} Y_{ij} SY_r(l,i,j)$$

$$l = 0 \ldots 7^r - 1 \qquad (1)$$

and

$$Q_{ij} = \sum_{l=0}^{7^r-1} M_l SQ_r(l,i,j)$$

- In fact, $SX = SX_1, SY = SY_1, SQ = SQ_1$.

Recursion Removal in Fast Matrix Multiplication

We have to determine values of matrices *SXk*, *SYk*, *SQk* from *SX1*, *SY1*, *SQ1*.

We extend the definition of tensor product in [*] for arrays of arbitrary dimensions:

- *Let A and B are arrays of same dimension l and of size m1 × m2 × . . . × ml, n1 × n2 × . . . × nl respectively. Then the tensor product (TP) is an array of same dimension and of size m1n1 × m2n2 × . . . × mlnl defined by replacing each element of A with the product of the element and B.*

$$P = A \otimes B \text{ where}$$

$$P[i_1, i_2, ..., i_l] = A[k_1, k_2, ..., k_l] B[h_1, h_2, ..., h_l],$$

$$i_j = k_j n_j + h_j \text{ with } \forall 1 \le j \le l;$$

[*] 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

Recursion Removal in Fast Matrix Multiplication

Let $P = \bigotimes_{i=1}^{n} A_i = (\ldots(A_1 \otimes A_2) \otimes A_3)\ldots \otimes A_n)$ with A_i is a array of dimension l and of size $m_{i1} \times m_{i2} \times \ldots \times m_{i1}$.

- **Theorem 1**

$$P[j_1, j_2, \ldots, j_l] = \prod_{i=1}^{n} A_i[h_{i1}, h_{i2}, \ldots, h_{il}]$$

where

$$j_k = \sum_{s=1}^{n} \left(h_{sk} \prod_{r=s+1}^{n} m_{rk} \right)$$

Recursion Removal in Fast Matrix Multiplication

- **In particular, if all *Ai* have the same size *m*1 × *m*2 × . . . × *ml*, we have**

$$P[j_1, j_2, \ldots, j_l] = \prod_{i=1}^{n} A_i[h_{i1}, h_{i2}, \ldots, h_{il}]$$

$$\text{where } j_k = \overline{h_{i1} h_{i2} \ldots h_{in\,(m_k)}}$$

- **Theorem 2**

$$SX_k = \bigotimes_{i=1}^{k} SX$$

$$SY_k = \bigotimes_{i=1}^{k} SY$$

$$SQ_k = \bigotimes_{i=1}^{k} SQ$$

Recursion Removal in Fast Matrix Multiplication

- Thanks to theorem 1 and theorem 2 we have

$$SX_k(l,i,j) = \prod_{r=1}^{k} SX(l_r, i_r, j_r)$$

$$SY_k(l,i,j) = \prod_{r=1}^{k} SY(l_r, i_r, j_r) \qquad (2)$$

$$SQ_k(l,i,j) = \prod_{r=1}^{k} SQ(l_r, i_r, j_r)$$

where

$$l = \overline{l_1 l_2 \ldots l_k}_{(7)}, i = \overline{i_1 i_2 \ldots i_k}_{(2)}, j = \overline{j_1 j_2 \ldots j_k}_{(2)}$$

FMM Algorithms + Sequential MM Algorithms

- Stop at the recursion level r and thanks to (1) & (2), we have the entire corresponding sub matrices:

$$M_l = \sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} X_{ij} SX_r(l,i,j) \times \sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} Y_{ij} SY_r(l,i,j)$$

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

$$l = 0 \ldots 7^r - 1$$

FMM Algorithms + Sequential MM Algorithms

with:

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

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

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

- Products M_l of these sub matrices are locally calculated by the processors

FMM Algorithms + Sequential MM Algorithms

- Finally, the result matrix

$$Q_{ij} = \sum_{l=0}^{7^r-1} M_l SQ_r(l,i,j)$$

$$= \sum_{l=0}^{7^r-1} M_l \left(\prod_{t=1}^{r} SQ(l_t, i_t, j_t) \right)$$

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

FMM Algorithms + Sequential MM Algorithms | Complexity

- Total running time when the Strassen's algorithm is used at the inter-processor level et the traditional matrix multiplication algorithm is used at the node level

$$T(n) \approx$$

$$\left(\frac{1}{8}\right)^r 2t_{comp}n^3 + \frac{6t_{comp}}{4^r}n^2 + b\beta\left(7^r + \left(\frac{7}{4}\right)^r - 1\right)n^2 + 7^r 2\alpha$$

where

t_{comp} is the execution time for one arithmetic operation,

α is the latency,

B is the number of bytes used to store one entry of the matrices,

β is the byte transfer rate.

FMM Algorithms + Sequential MM Algorithms | Complexity

- Total running time when the Winograd's algorithm is used at the inter-processor level et the traditional matrix multiplication algorithm is used at the node level

$$T(n) \approx$$

$$\left(\frac{1}{8}\right)^r 2t_{comp}n^3 + \frac{5t_{comp}}{4^r}n^2 + b\beta\left(7^r + \left(\frac{7}{4}\right)^r - 1\right)n^2 + 7^r 2\alpha$$

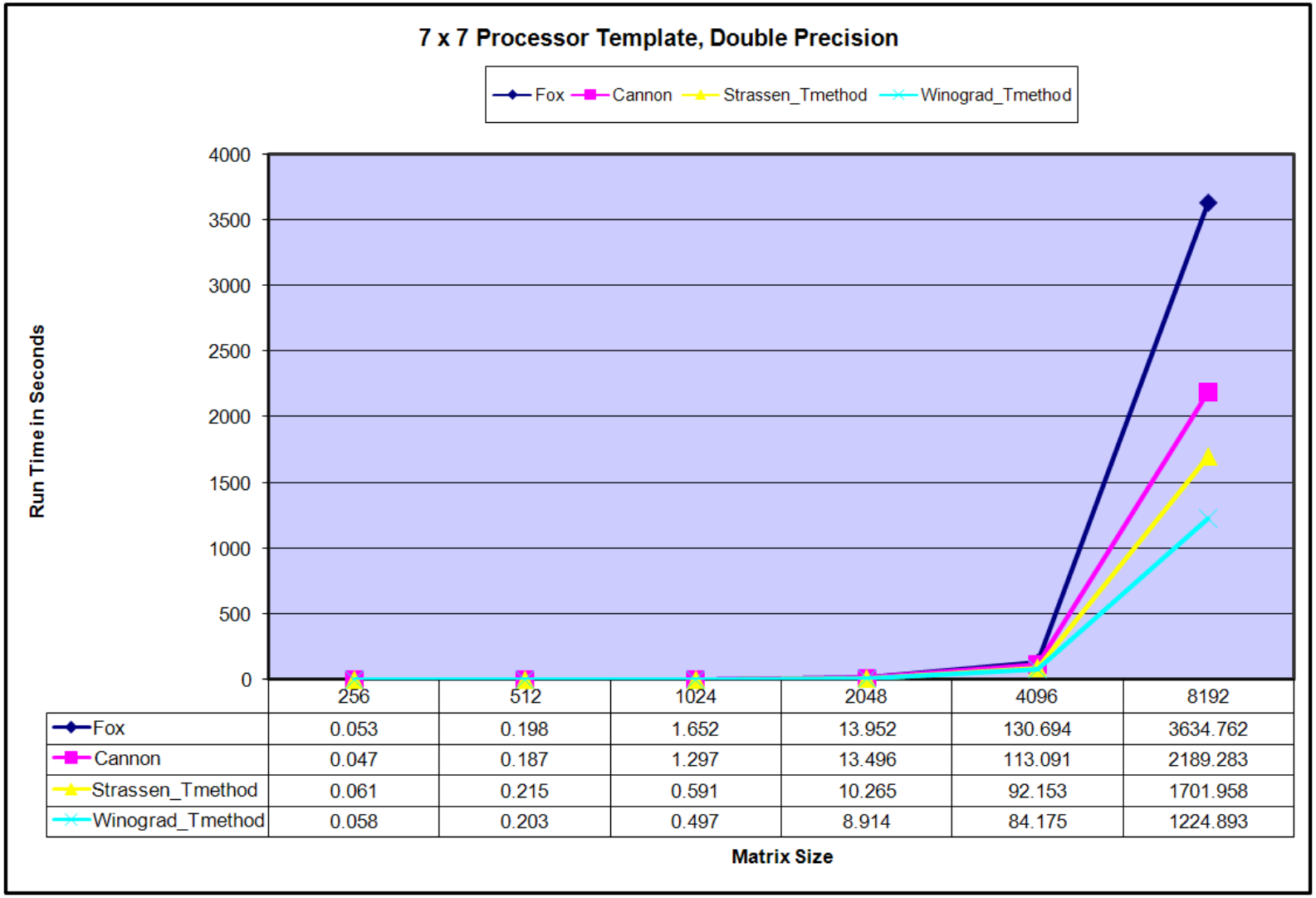
- Running time of the Cannon's algorithm

$$= \left(\frac{1}{7}\right)^r 2t_{comp}n^3 + b\beta 2\frac{1}{\sqrt{7}^r}n^2 + 2\sqrt{7}^r\alpha$$

FMM Algorithms + Sequential MM Algorithms | Experimental Results

- Performance tests are carried out on a Fujitsu Siemens Computers/hpcLine, 16 nodes (32 CPUs), switched Ethernet 1Gbit. Each processor is an Intel Xeon 2.4 GHz with 2GB memory
- Entries of all the matrices tested here are random double precision numbers uniformly distributed between -1 and 1

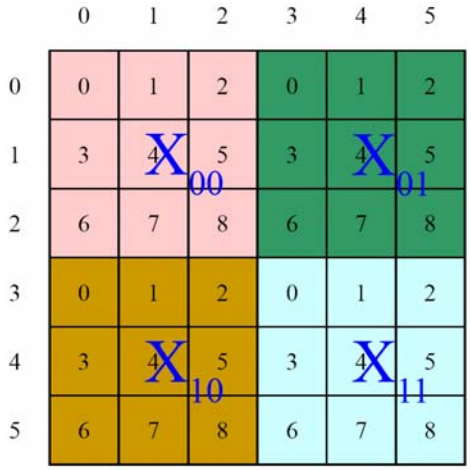
FMM Algorithms + Sequential MM Algorithms | Experimental Results



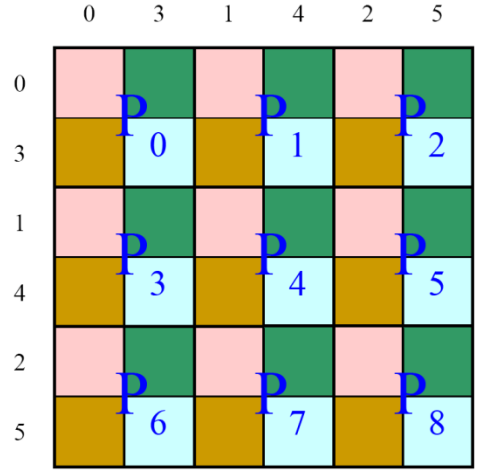
	256	512	1024	2048	4096	8192
Fox	0.053	0.198	1.652	13.952	130.694	3634.762
Cannon	0.047	0.187	1.297	13.496	113.091	2189.283
Strassen_Tmethod	0.061	0.215	0.591	10.265	92.153	1701.958
Winograd_Tmethod	0.058	0.203	0.497	8.914	84.175	1224.893

FMM-Cannon(Fox) Algorithm and Pattern to Store the Matrices

- If the sub matrices used in the FMM algorithms 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 introduce a pattern to store the matrices

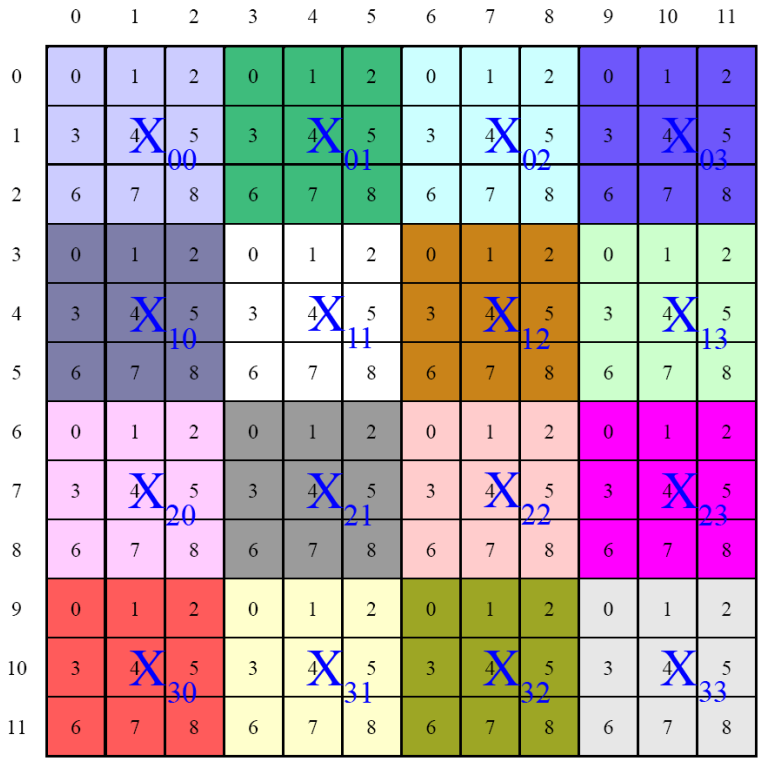


Matrix *X* with 6 × 6 blocks is distributed over a 3 × 3 processor template from a matrix point-of-view. The 9 processors are labeled from 0 to 8. This pattern is used in the Strassen-Cannon algorithm when the recursion level is 1.

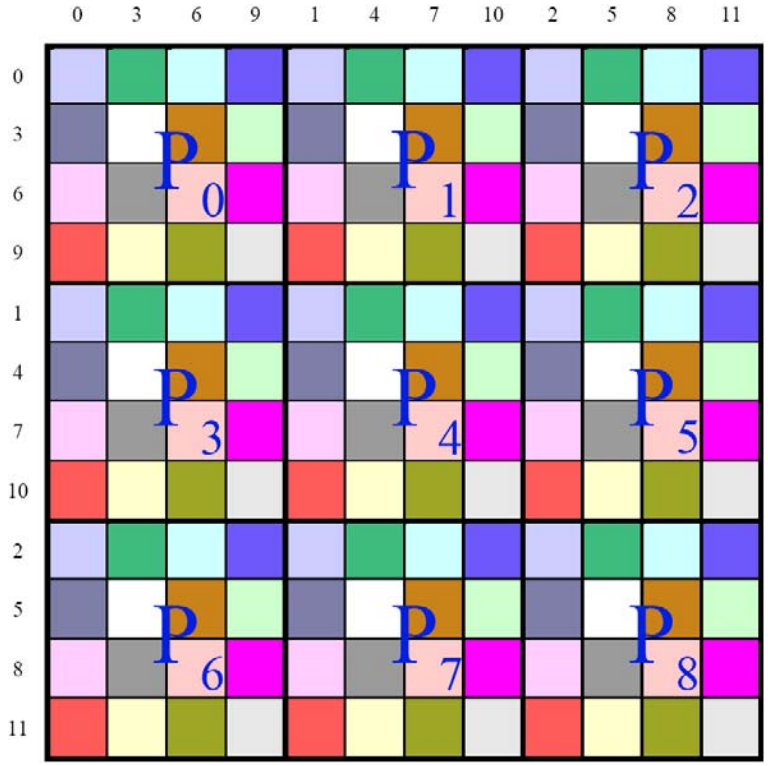


Same as Figure 1, but from a processor point-of-view.

FMM-Cannon(Fox) Algorithm and Pattern to Store the Matrices



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.



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

General Scalable Parallelization of FMM Algorithms

- To implement FMM algorithms on distributed memory computers, we stop at recursion level *r* and thanks to (1) & (2), we have the entire corresponding sub matrices:

$$M_l = \sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} X_{ij} SX_r(l,i,j) \times \sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} Y_{ij} SY_r(l,i,j)$$

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

$$l = 0 \ldots 7^r - 1$$

General Scalable Parallelization of FMM Algorithms

with:

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

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

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

General Scalable Parallelization of FMM Algorithms

- Make the storage map of sub matrices to processors.
- Calculate the product M_l of

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

by parallel algorithms based on T-algo (Fox, Cannon, SUMMA, PUMMA, DIMMA, ...).

- Have the result matrix from M_l

$$Q_{ij} = \sum_{l=0}^{7^r-1} M_l SQ_r(l, i, j)$$

$$= \sum_{l=0}^{7^r-1} M_l \left(\prod_{t=1}^{r} SQ(l_t, i_t, j_t) \right)$$

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

General Scalable Parallelization of FMM Algorithms | Complexity

- The total running time for the Strassen-Cannon algorithm

$$T(n) \approx$$

$$\left(\frac{7}{8}\right)^r \frac{2t_{comp}}{p^2} n^3 + \frac{6\left(\frac{7}{4}\right)^r t_{comp}}{p^2} n^2 + \left(\frac{7}{4}\right)^r \frac{2B\beta}{p} n^2 + 7^r (2p\alpha)$$

where

t_{comp} is the execution time for one arithmetic operation,

α is the latency,

β is the byte transfer rate.

General Scalable Parallelization of FMM Algorithms | Complexity

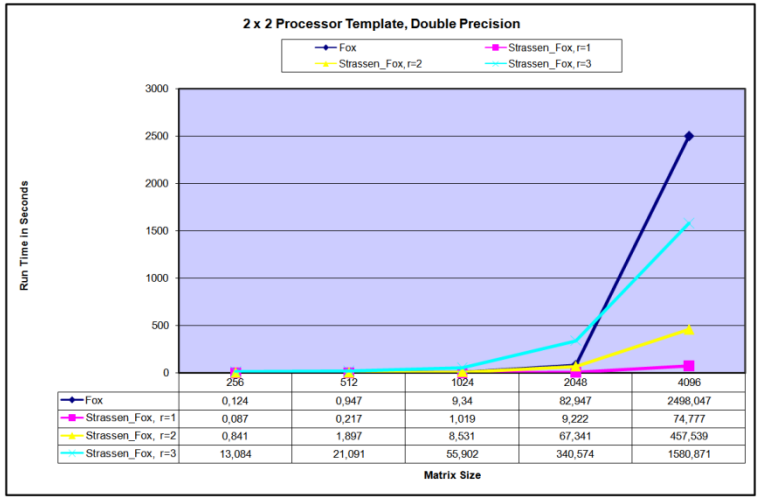
- The total running time for the Winograd-Cannon algorithm

$$T(n) \approx \left(\frac{7}{8}\right)^r \frac{2t_{comp}}{p^2} n^3 + \frac{5\left(\frac{7}{4}\right)^r t_{comp}}{p^2} n^2 + \left(\frac{7}{4}\right)^r \frac{2B\beta}{p} n^2 + 7^r (2p\alpha)$$

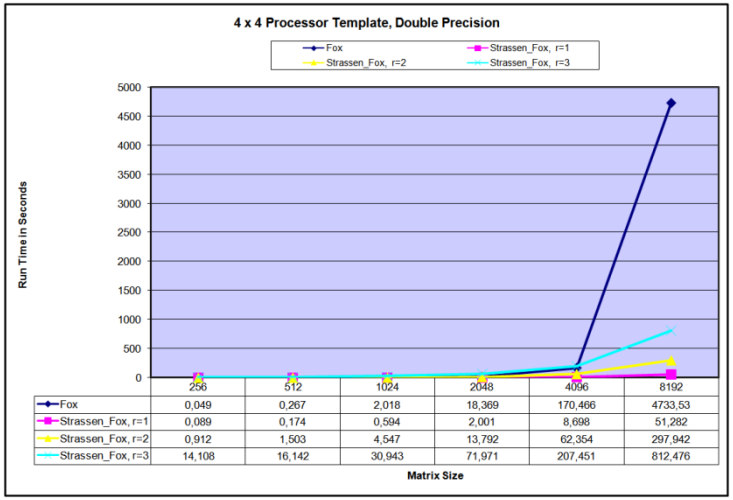
- Running time of the Cannon's algorithm

$$= \frac{2t_{comp}}{q^2} n^3 + \frac{2b\beta}{q} n^2 + 2q\alpha$$

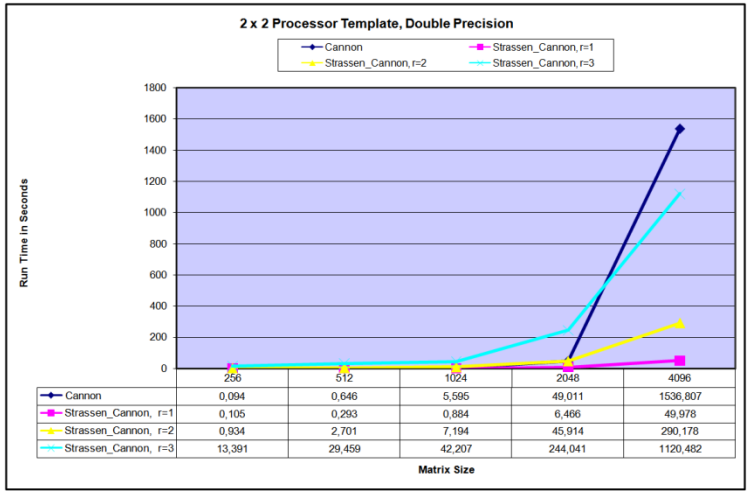
General Scalable Parallelization of FMM Algorithms | Experimental Results



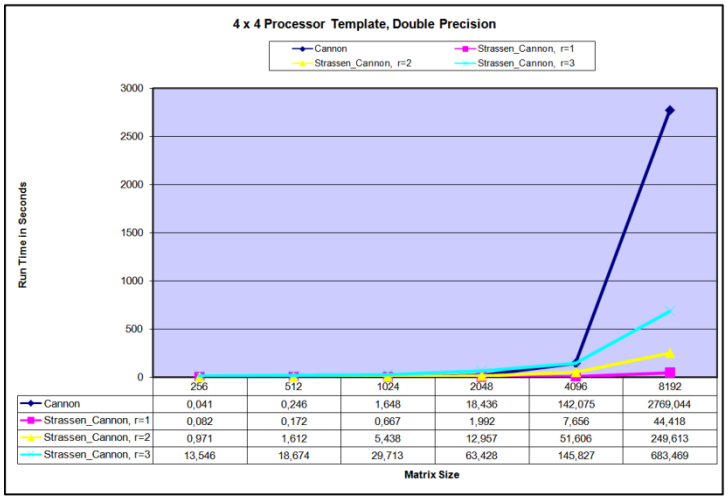
	256	512	1024	2048	4096
Fox	0,124	0,947	9,34	82,947	2498,047
Strassen_Fox, r=1	0,087	0,217	1,019	9,222	74,777
Strassen_Fox, r=2	0,841	1,897	8,531	67,341	457,539
Strassen_Fox, r=3	13,084	21,091	55,902	340,574	1580,871



	256	512	1024	2048	4096	8192
Fox	0,049	0,267	2,018	18,369	170,466	4733,53
Strassen_Fox, r=1	0,089	0,174	0,594	2,001	8,698	51,282
Strassen_Fox, r=2	0,912	1,503	4,547	13,792	62,354	297,942
Strassen_Fox, r=3	14,108	16,142	30,943	71,971	207,451	812,476

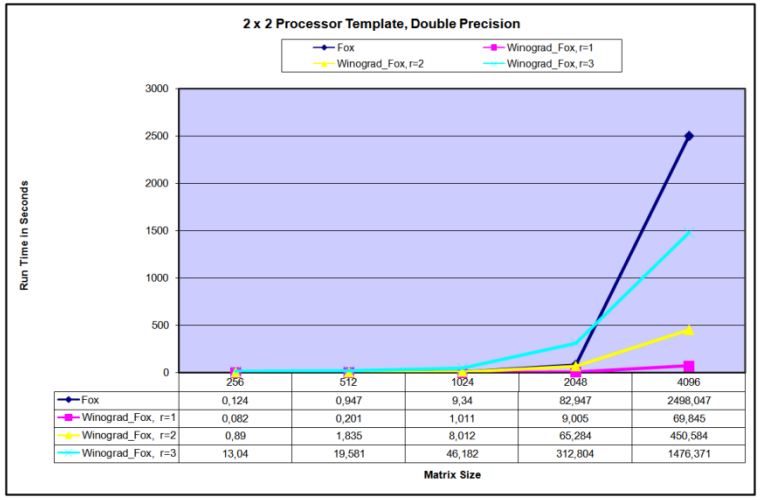


	256	512	1024	2048	4096
Cannon	0,094	0,646	5,595	49,011	1536,807
Strassen_Cannon, r=1	0,105	0,293	0,884	6,466	49,978
Strassen_Cannon, r=2	0,934	2,701	7,194	45,914	290,178
Strassen_Cannon, r=3	13,391	29,459	42,207	244,041	1120,482

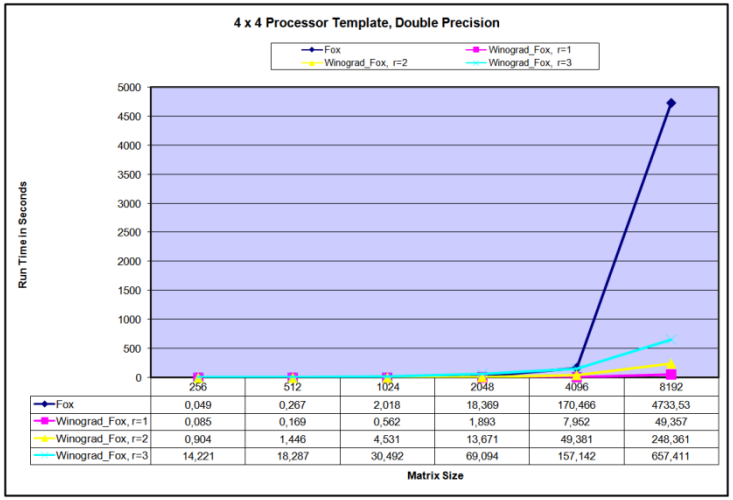


	256	512	1024	2048	4096	8192
Cannon	0,041	0,246	1,648	18,436	142,075	2769,044
Strassen_Cannon, r=1	0,082	0,172	0,667	1,992	7,656	44,418
Strassen_Cannon, r=2	0,971	1,612	5,438	12,957	51,606	249,613
Strassen_Cannon, r=3	13,546	18,674	29,713	63,428	145,827	683,469

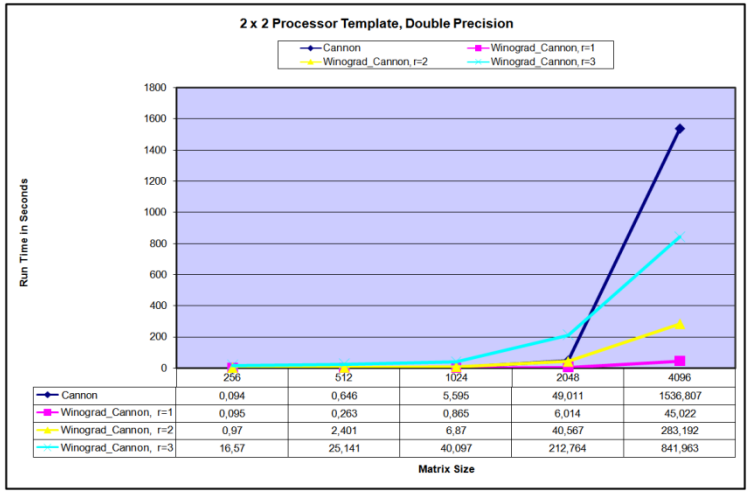
General Scalable Parallelization of FMM Algorithms | Experimental Results



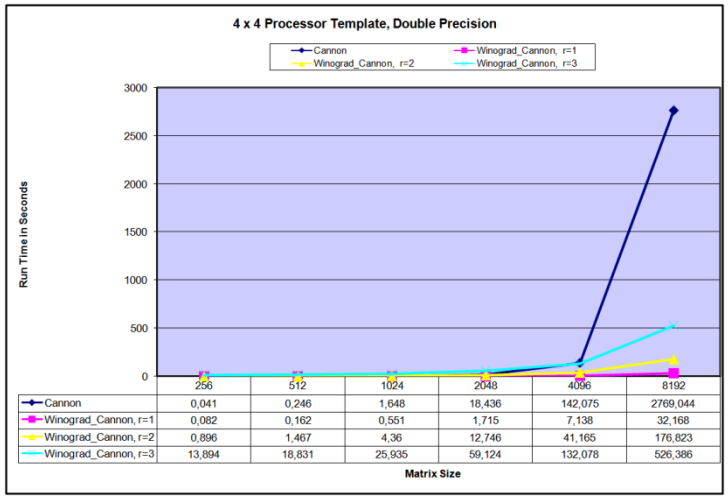
	256	512	1024	2048	4096
Fox	0,124	0,947	9,34	82,947	2498,047
Winograd_Fox, r=1	0,082	0,201	1,011	9,005	69,845
Winograd_Fox, r=2	0,89	1,835	8,012	65,284	450,584
Winograd_Fox, r=3	13,04	19,581	46,182	312,804	1476,371



	256	512	1024	2048	4096	8192
Fox	0,049	0,267	2,018	18,369	170,466	4733,53
Winograd_Fox, r=1	0,085	0,169	0,562	1,893	7,952	49,357
Winograd_Fox, r=2	0,904	1,446	4,531	13,671	49,381	248,361
Winograd_Fox, r=3	14,221	18,287	30,492	69,094	157,142	657,411



	256	512	1024	2048	4096
Cannon	0,094	0,646	5,595	49,011	1536,807
Winograd_Cannon, r=1	0,095	0,263	0,865	6,014	45,022
Winograd_Cannon, r=2	0,97	2,401	6,87	40,567	283,192
Winograd_Cannon, r=3	16,57	25,141	40,097	212,764	841,963



	256	512	1024	2048	4096	8192
Cannon	0,041	0,246	1,648	18,436	142,075	2769,044
Winograd_Cannon, r=1	0,082	0,162	0,551	1,715	7,138	32,168
Winograd_Cannon, r=2	0,896	1,467	4,36	12,746	41,165	176,823
Winograd_Cannon, r=3	13,894	18,831	25,935	59,124	132,078	526,386

General Scalable Parallelization of FMM Algorithms | Experimental Results

- Our implementation of the Strassen-Cannon(Fox) method is better than the Cannon(Fox)'s algorithm when the matrix size is large
  - For example : to calculate the product of 8196x8196 matrices on 4x4 processors, the difference in running time between these two methods is 50 times
- Why ?
  - Better in complexity
  - Pattern to store the matrices
  - Low-speed network of the test system

Outline

- Problem
- State of the Art
- Contribution
- Implementation
- Conclusion
  - Conclusion
  - Future Works

Conclusion

- We have just presented a general scalable parallelization for all the matrix multiplication algorithms on distributed memory computers that use fast matrix multiplication algorithms at inter-processor level.
- Complexity analysis shows that our algorithms should be better than the parallel algorithms based on traditional method when the matrix size is large and our work is relevant to exploit better algorithms when the recursion level is large enough. Experimental results on Fujitsu Siemens Computers/hpcLine show that our algorithms perform better than Cannon algorithm from 2 to 50 times for 8196x8196 matrices.
- 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 new direction to parallelize the FMM algorithms.

Future Works

- Analyze performance of the other parallel matric multiplication algorithms based on traditional method like 1D-systolic, 2D-systolic, PUMA, BiMMeR, SUMMA, DIMMA… at the bottom level to find a better combination of algorithms.
- Find the value optimal of the recursion level r according to the matrix size and to the specificities of the machines on which the algorithm is implemented.
- Apply the technique "trains" of matrices in matrix multiplication to approach the minimal bound given by V. Y. Pan for the complexity of $\boldsymbol{O}(n^{2.32})$.
- Reconsider this algorithm to handle the large matrices with huge numbers. For the moment, the value of the elements of matrices X and Y is limited.