MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

SEMINAR
REPORT ON

Parallel Algorithm for Fast Matrix Exponentiation using MPI

SUBMITTED
TO

Department of Computer Science & Engineering

by

Neel Dani
160905396
Roll number: 57
Semester: 7
Section: A

Name and Signature of Evaluator 1 Name and Signature of Evaluator 2

Contents

1 Introduction

Linear algebra is the branch of mathematics which is concerned with linear equations, linear functions and their representations through matrices and vector spaces. Linear algebra deals with matrix manipulations and matrix algorithms. One such manipulation is calculating the power of a matrix. Power of a matrix can be considered as a self-product of the given matrix, calculated over and over again. Matrix multiplication is a computationally expensive algorithm having a time complexity of $\mathcal{O}(m^3)$.

To improve this algorithm, the authors have proposed a parallel algorithm for the Matrix Fast Exponentiation Algorithm (MFEA), which is discussed later. Matrix power computation is an important numerical calculation, its performance directly influences other numerical calculation. With the advent of the big data age, the computational efficiency of existing algorithms can not meet the needs of data processing capacity of various applications. Matrix Fast Exponentiation Algorithm (MFEA) is an algorithm for fast solving matrix power based on the divide and conquer approach.

It has important applications in many problems such as PageRank Algorithm, linear recurrences, graph theory and so on. In order to improve the computational efficiency of MFEA, the authors of this paper analyze the feasibility of parallelizing MFEA with the multi-core processor, propose a Parallel Matrix Fast Exponentiation Algorithm (PMFEA) and implement PMFEA based on message passing interface (MPI). The experimental results show the superiority of PMFEA: PMFEA achieves about $8.41x$ speedup compared to MFEA on 32-core server.

2 Matrix Fast Exponentiation Algorithm (MFEA)

MFEA is an efficient algorithm for computing matrix powers based on the divide and conquer approach and the associative law of matrix multiplication. The main calculation method of MFEA is as follows: It continually cuts the power of matrix in half, adding an extra matrix multiplication if the power of matrix is odd, until the power of matrix is 1, and then returns the result to compute the matrix of higher powers, and finally gets the answer. The function expression of the algorithm is shown as: (A is the matrix whose power q, is to be calculated)

$$A^q = \begin{cases} A^{\lfloor q/2 \rfloor} \times A^{\lfloor q/2 \rfloor} \times A & \text{if } q \text{ is odd} \\ A^{\lfloor q/2 \rfloor} \times A^{\lfloor q/2 \rfloor} & \text{if } q \text{ is even} \end{cases}$$

The MFEA algorithm is based on the divide and conquer approach. The pseudo code for matrix fast exponentiation algorithm (MFEA) is as shown:

Algorithm 1 Recursive Implementation of MFEA

input: matrix A, power q
output: A^q

```
1: function fastExp(q)
2:     if q = 1 then
3:         return A
4:     end if
5:     result ← fastExp(q / 2)
6:     result ← result × result
7:     if q is odd then
8:         result ← result × A
9:     end if
10:    return result
11: end function
```

Algorithm I: MFEA pseudo-code

3 Parallelism Analysis

In Algorithm I, the time complexity of decomposing the matrix power is $\mathcal{O}(\log q)$ via the approach of divide and conquer, where q denotes the power to be calculated. Traditional matrix multiplication has a time complexity order of $\mathcal{O}(m^3)$, assuming the matrix is of dimension $m \times m$. The MFEA algorithm thus has a time complexity of $\mathcal{O}(m^3 \cdot \log q)$. Without loss of generality, we assume the matrix to be a square matrix. From the run-time complexity of MFEA, we can find that the algorithm takes less time to decompose the power and spends more time in the matrix multiplication which confines the capacity of MFEA in the use of big data. So, the optimization of matrix multiplication becomes the most important factor when it comes to improving the efficiency of MFEA.

The authors propose to parallelize the matrix multiplication by dividing the original matrix into smaller matrices and calculating these matrices at different nodes/slaves. By this method, matrix multiplication can make full use of parallel computing resources and improve observably, its computing efficiency as the number of nodes increases. Through a comparative study, the authors were also successful in establishing that the decomposition of power takes negligible time as compared to the actual process of computing each element of the resultant matrix. If the decomposing of power was parallelized, then computing efficiency of the algorithm would decrease due to the communication cost between nodes. As a result, the step of decomposing the power was carried out via sequential programming in PMFEA on all nodes.

3.1 Amdahl's Law

In computer architecture, Amdahl's law is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors.
According to Amdahl's law the maximum of speedup of a parallel algorithm, can be calculated as:

$$\psi \leq \frac{1}{f+(1-f)/p}$$

where f is the proportion of the part performed by sequential code and p is the number of processes.

The paper states that the proportion of the sequential part of the PMFEA algorithm is 11%. Using this value, the maximum theoretical speedup according to Amdahls law is:

$$\lim_{p \to +\infty} \frac{1}{0.11 + (1 - 0.11)/p} \approx 9.1$$

In conclusion, the parallelization of MFEA is feasible and will improve significantly the performance of MFEA.

4 Design and Implementation

In the paper, the authors state that the implementation of PMFEA was done in C++ using Message Passing Interface (MPI). The main steps involved in PMFEA are as stated:

a) The master node reads the data and distributes it to other nodes.

b) All nodes decompose the matrix power.

c) Determine whether matrix multiplication is required, and if necessary, it is performed.

d) Repeat b) and c) until the matrix power is 1, in which case the result is obtained.

4.1 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and FORTRAN. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

4.2 Parallel MFEA (PMFEA)

The formal algorithm starts with the process of decomposing power which is executed in the following way. Firstly, the algorithm launches the calculation on all nodes. When a matrix multiplication is required, the master node divides the matrix into smaller matrices and respectively sends them to computing nodes. Then, computing nodes perform matrix multiplication and send the result to the master node. Finally, the master node merges results and all nodes continue executing the operation of decomposing. In the implementation of the process decomposing, the paper optimized the strategy of divide and conquer to avoid the external cost caused by recursion.

In the design of parallel matrix multiplication, the algorithm decomposes the matrix by row. The authors assume that all matrices are square matrices of dimension $m \times m$ and the number of computing nodes is less than the number of rows ($p \leq m$).

Firstly, the master node divides matrix into a number of smaller $\lfloor m/p \rfloor \times m$ matrices (each called A_i) that are adjacent in the original matrix and sends A_i to the i^{th} computing node. Then, each computing node calculates $\lfloor m/p \rfloor \times m$ final matrix and sends it to the master node. Finally, master node merges all into a single matrix. So, the runtime complexity of calculating the matrix on each computing node is $\mathcal{O}(m^2 \cdot m/p)$ In the ideal condition, the run-time complexity of parallel matrix multiplication equals the computing node's time complexity which is much better than the traditional algorithm of multiplication.

According to our design and implementation, we can obtain the run time complexity of PMFEA, in theory, is $\mathcal{O}(m^3/p)$, which is $1/p$ faster than MFEAs. So, the authors predict that the parallelization of MFEA will improve its performance and this improvement will increase with the increase in the number of computing nodes. Shown below is the pseudo code of the PMFEA.

Algorithm 2 Decomposing power

input: matrix A, power q
output: A^q

```
1: if taskid = master then \*master*\
2:     while q > 0 do
3:         if q is odd then
4:             result = Mastermatmul(result, A)
5:         end if
6:         A = Mastermatmul(A, A)
7:         q = q ≫ 1
8:     end while
9: else\*slave*\
10:    while q > 0 do
11:        if q is odd then
12:            Slavematmul()
13:        end if
14:        Slavematmul()
15:        q = q ≫ 1
16:    end while
17: end if
```

Algorithm II: PMFEA pseudo code

5 Experimental Results and Analysis

In these experiments, all the matrices are $m \times m$ square matrices in which all elements are 1. The authors obtained the experimental results by repeating 20 times for every experiment and taking the mean of experimental results.

5.1 Experiments with matrix size

To experimentally determine the impact of the matrix size to the performance of PMFEA, q, the power of matrix, was fixed at 64 and PMFEA was experimented for different $m(200, 400, 600, 800, 1000, 1200)$, size of matrices, and $p(4, 8, 16, 32)$, number of cores. The authors also compare these results to the performance of MFEA. Experimental results are shown in Table I and Fig. I.

matrix size	MFEA	4-core	8-core	16-core	32-core
200	0.457	0.223	0.208	0.117	0.075
400	3.02	1.74	0.745	0.577	0.422
600	7.496	2.844	1.557	1.081	0.891
800	23.383	8.747	4.584	2.81	2.825
1000	32.4	11.746	6.102	5.147	4.145
1200	53.36	19.774	10.287	7.15	6.809

Table I: Measured performance for different cores and matrix powers

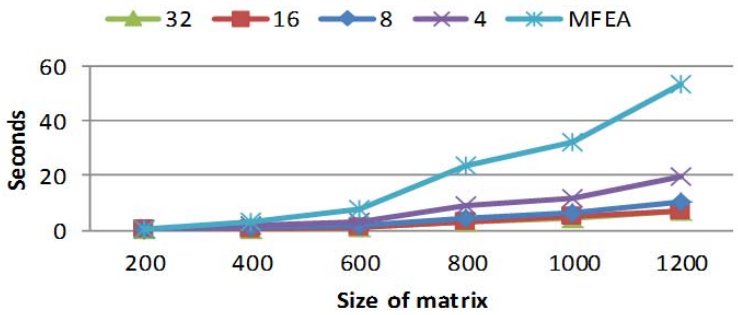

Fig. I: Measured performance for execution time vs size of the matrix for different number of cores

From Table I and Fig. I, the authors observe a substantial difference in the run-time between PMFEA and MFEA and the gap between them increases with the size of matrices as well as number of cores. For $m = 1200$, the run-time of MFEA is 53.36s, but the run-time of

PMFEA is 6.8s on 32 cores machines, which performs 7 times efficiently than MFEA. At the same time, the authors find that PMFEA showed good performance in a certain number of cores but it also showed almost the same results when the number of cores are increased.

In order to analyze the reason of this phenomenon, the authors calculated the speedup of PMFEA on different $p(8, 16, 32)$, the number of cores. It was seen that the speedup of PMFEA increases with the size of matrices on 8 cores and 16 cores. It reached 8.41 with the matrices of size 600 on 32 cores machines, which is close to 9.1 (the maximum theoretical speedup of PMFEA, proposed in this paper). This verifies the feasibility of PMFEA. The external communication cost that will increase with the size of matrices leads to the actual speedup being lower than the ideal speedup. When the reduction in the calculation cost is lower than the increase in communication cost, the run-time of PMFEA will increase and the speedup of PMFEA will decrease.

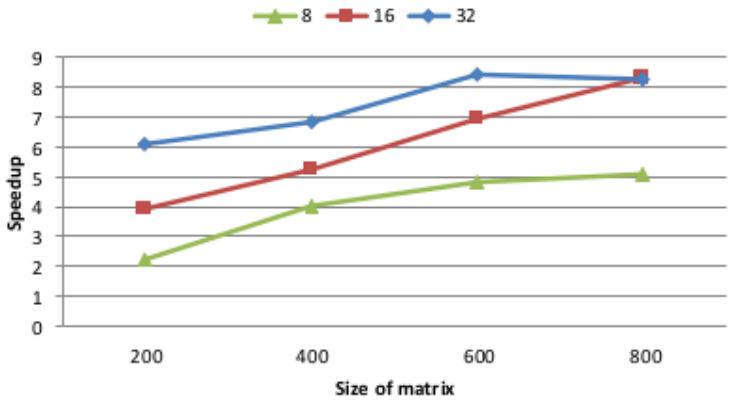

Fig. II: Measured performance for speedup vs size of matrix for different number of cores

5.2 Experiments with matrix power

To experimentally determine the impact of the power of matrix to the performance of PMFEA, the authors fixed m, the size of matrix, at 600 and experimented PMFEA for different $q(8, 16, 32, 64)$, the power of matrices, and $p(4, 8, 16, 32)$, the number of cores. The authors also compared these results with the performance of MFEA. Experimental results are shown in Fig. 3. In Fig. 3, the run-time of PMFEA decreases rapidly as the increase in the number of cores, which is similar to the change in PMFEA's performance with different sizes of matrices. These results demonstrate the performance of PMFEA is superior to the performance of MFEA in both different size of matrices and different power of matrices. The authors also find that when PMFEA runs on the ascertain number of cores, the change of the run-time is

little in different power of matrices, which verifies the author's analysis that the key factor impacting the performance of MFEA is the size of matrix instead of the power of matrix.

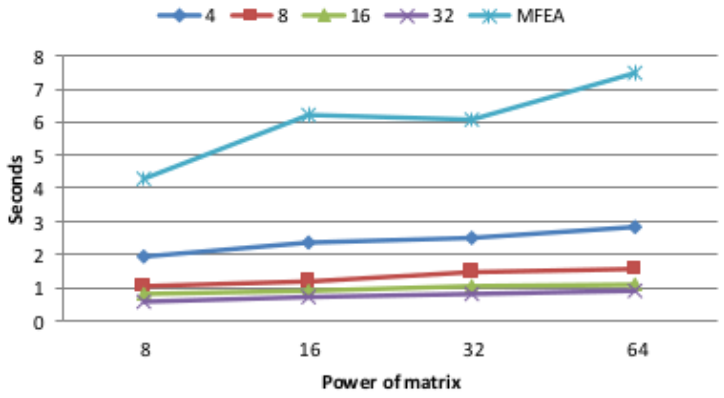

Fig. III: Measured performance against number of cores

6 Conclusion

In this paper, based on multi-core architecture and message passing interface (MPI), a Parallel Matrix Fast Exponentiation Algorithm (PMFEA) is designed and implemented. Firstly, Matrix Fast Exponentiation Algorithm (MFEA) is introduced and the problems in the era of big data and the feasibility of parallel MFEA is deeply analyzed. Then, for the efficiency bottlenecks of MFEA, PMFEA based on MPI is designed and the strategy of divide and conquer of PMFEA is optimized in implementation. Finally, the experimental results clearly indicate that the performance of PMFEA is higher than the performance of MFEA and the speedup of PMFEA has reached 8.41 on 32-cores server. In further research, the authors plan to focus on improving algorithm design to reduce communication cost and memory overhead.

7 References

1. MFEA algorithm, https://medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27 [Accessed on 14/08/2019]

2. Parallel Algorithms, https://www.tutorialspoint.com/parallel_algorithm/parallel/_algorithm_introdu [Accessed on 14/08/2019]

3. Recursive call for MFEA, https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec20-master/lec20.html [Accessed on 14/08/2019]

4. Amdahl's Law, https://en.wikipedia.org/wiki/Amdahl/%27s_law [Accessed on 14/08/2019]

5. Message Passing Interface, https://en.wikipedia.org/wiki/Message_Passing_Interface [Accessed on 14/08/2019]