Development an Analytical Performance Models for Matrix Multiplication on Distributing Systems

Arabi Keshk
Computer Science Dept., Faculty of Computers and information, Menoufiya University
arabikeshk@yahoo.com

Abstract *In this paper, we suggest a mechanism for implementing a distributed application using RMI based on JAVA threads. The application is parallel matrices multiplication depending on distributed the products block of rows and columns on different machines. One server and seven clients are run to find the product of matrix multiplication. The server distributes the determine blocks of rows and columns on the registered clients. The clients return their product blocks to a server, which calculate the final product of matrix multiplication. Applications of this type will allow loaded servers to transfer part of the load to clients to exploit the computing power available at client side. Experimental result shows that the speed up ratio is equals 9 or the computation time of matrix multiplication with size of 2048 X 2048 is reduced by 89 % by using 7-client.*

Keywords: *Distributed systems, Performance prediction model, Matrix multiplication*

1. Introduction and related works

Matrix multiplication (MM) is an important linear algebra operation. A number of scientific and engineering applications include this operation as a building block. Due to its fundamental importance, much effort has been devoted to studying and implementing MM. MM has been included in several libraries. Many MM algorithms have been developed for parallel systems [1-4].

Traditional methods for distributed application are decomposed the entire task, which introduces various overheads. The most important are the communication and load balancing overheads. For example, partitioning MM into sub-matrix blocks and decomposing the MM operation are often technology dependent. Applying special implementations for sub-matrix blocks may improve performance since the workload of sub-matrix operations may vary. The information about the workload of a task for matrix operation may not be available at compile time or even at the time of initiating subroutines. It may be available only after these routines have been executed. Since this increases the complexity of load balancing, it is often ignored.

Most parallel algorithms are optimized based on the characteristics of the targeting platform. The PC cluster computing platform has recently emerged as a viable alternative for high-performance and low-cost computing [2]. Generally, the PCs in a cluster have a lot of resources that can be used simultaneously. They have relatively weak communication capabilities. They lack high performance implementation support for data communications compared to supercomputers. They only support some communication channels implemented by software that capitalizes on Ethernet connections. MM operations are embedded in many host programs.

The main reason for using parallel processing is to reduce the computation time required for what would otherwise be very long-running programs. Because poorly parallelized code tends to offer little performance benefit, there is great incentive to ensure that parallel programs are highly optimized. Unfortunately, a lack of sufficiently accurate and easy-to-use performance prediction methods for parallel programs has necessitated resort to a very time-consuming, which modifies design cycle to achieve this [5].

The biggest price we had to pay for the use of a PC cluster was the conversion of an existing serial code to a parallel code based on the message-passing philosophy. The main difficulty with the message passing philosophy is that one needs to ensure that a control node (or master node) is distributing the workload evenly between all the other nodes (the compute nodes). Because all the nodes have to synchronize at each time step, each PC should finish its calculations in about the same amount of time. If the load is uneven (or if the load balancing is poor), the PCs are going to synchronize on the slowest node, leading to a worst-case scenario. Another obstacle is the possibility of communication patterns that can deadlock [5]. A typical example is if PC A is waiting to receive information from PC B, while B is also waiting to receive information from A.

JAVA provides Remote Method Invocation (RMI) to allow one JAVA Virtual Machine (VM) to invoke methods running on another VM. RMI applications are often comprised of two separate programs, which are server and client. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application [6-8].

The related work [1– 4] deals with moving application from one machine to another in a set of machines. In the pervious distributed modes it is not very easy for a programmer to write an application a part of which can be executed on remote side and result of computation can be combined in original program to compute the final result. i.e. there is no method level distributed available. We have applied object mobility to LAN architecture. This allows development of applications that can be easily load balanced.

In this paper we study implementation for matrix multiplication on distributed systems using RMI based on JAVA threads, which distribute the load between the server and the clients. This system provides the user a level of control over the distribution of the program. The rest of this paper is organized as follow. Section 2 presents the mathematical model of distributed tasks. Section 3 introduces the techniques of matrix multiplication. Section 4 proposes matrix implementation performance models. Section 5 presents our implementation architecture. Section 6 shows the experiment results. Finally the conclusion is provided in Section 7.

2. Distributed tasks assignment mathematical modeling

In [9], generally a mathematical model to the distributed task assignment problem involves the following two steps:

(1) Formulate a cost function to represent the main purpose of the task assignment process,

(2) Formulate a set of constraints/inequalities in terms of both the application requirements and the availability of the system resources.

This section presents the main components of the cost function in general, and it describes the most important constraints that may be considered with the assignment problem. Finally, it presents an example for modeling the assignment problem mathematically.

2.1. Components of the cost function

As mentioned previously, the assignment problem is usually handled based on the optimization of a cost/objective function. Depending on the context, many components of the cost function may be defined. To do so, let X be an M x N binary matrix corresponding to an assignment of M tasks onto N processors (PCs) such that

$$X_{ip} = \begin{cases} 1 & \textit{if a task i is assigned to a processor p,} \\ 0 & \textit{otherwise} \end{cases}$$

Where, Xip is binary variables such that i is ranged over the set of tasks and p ranged over the set of processors. The main components of the cost function may be described as in the following.

2.1.1. Accumulative execution cost/time:

The cost of processing tasks assigned to a processor (EXECp) is the total execution time incurred by tasks running on the processor p. This cost depends on the size of the tasks residing at the processor p and on the speed of the processor p. Define TCp as the set of tasks that are assigned to the processor p under a task assignment X, such that TCp={i | Xip=1, 1≤i≤M, 1≤p≤N}, and let Cip denotes the cost of processing a task i on a processor p, then the actual execution cost EXECp may be formulated as:

$$EXEC_p = \sum_{i \in TC_p} C_{ip} = \sum_{i \in TC_p} d_i * e_p$$

Where, di is the size of the task i expressed in size of execution code, i.e., number of instructions to be processed, or in execution time on some normalized processor, and ep is the average processing time of one instruction on the processor p.

For an assignment X, the Accumulative Execution Time (AET) defines the total execution/processing time incurred for running all tasks at all processors in the distributed system. Hence, the AET may be formulated as:

$$AET = \sum_{p} EXEC_p = \sum_{p} \sum_{i} C_{ip}\, X_{ip}$$

2.1.2. Accumulative Communication Time

The actual communication cost/time at a processor COMMp is the total time of communicating data between tasks resident at the processor p with other tasks resident at other processors in the system. This cost depends on the quantity of information to be exchanged between tasks, the interconnection topology between computers of the distributed system, the propagation delay through the transmission media and the speed/bandwidth of the communication media. Let TCq be the set of tasks that are assigned to a processor q, where TCq = {j | Xjq=1, 1≤j≤M, 1≤q≤N}, and define Cijpq as the cost of sending a data between a task i residing at the processor p and a task j residing at other processor q through a communication path/link pq, then the actual communication cost/load COMMp may be formulated as:

$$COMM_p = \sum_{q \neq p} \sum_{i \in TC_p} \sum_{j \neq i, j \in TC_q} C_{ijpq} = \sum_{q \neq p} \sum_{i \in TC_p} \sum_{j \neq i, j \in TC_q} (S_{pq} + d_{ij}\, c_{pq})$$

Where, Spq is the time necessary for a processor p to set up the communication with other processor q, dij is the average quantity of information/data to be transferred between the processors p and q, and cpq is the average time of transferring a data unit from the processor p to the processor q through the path/route pq after the set up is completed.

For an assignment X, the Accumulative Communication Time (ACT) is the total time incurred for communicating/exchanging data between tasks residing at separate computers of the distributed system. Hence, the ACT may be formulated as:

$$ACT = \sum_{p} COMM_p = \sum_{p} \sum_{q \neq p} \sum_{i} \sum_{j \neq i} C_{ijpq}\, X_{ip}\, X_{jq}$$

The ACT is often considered to be one of the most important factors which need to be minimized by the task assignment.

It is worth noting that if two communicating tasks are assigned to different processors, the communication cost contributes to the load at the two processors, i.e., bilateral communication is assumed. Indeed, if two communicating tasks are assigned to the same processor, the communication cost is assumed to be zero as they use the local system memory for data exchange.

2.2. Allocation/Assignment Constraints

To meet the application requirements and not violate the availability of the system resources, the assignment should be done taking into account various kinds of constraints. These constraints depend on the characteristics of the involved application tasks, such as processing load requirements, memory requirements, amount of inter-task communication capacity requirements and precedence relation between tasks, and on the availability of the system resources including the available computation speed of processors, the available memory capacity and the available communication capacity of the communication network resources.

These constraints may be classified into two broad categories, namely, locality and devices constraints. The locality constraints concern the location of tasks on different processors. For instance, a task might require to be allocated onto a specific processor or two exclusive tasks must not be allocated on the same processor. On the other hand, the devices constraints define the relation between the tasks requirements from a physical device, used by the tasks during execution such as memory, CPU, and transmission media. Devices constraints might be the case that only a limited number of tasks can access a certain device at the same time to ensure that the amount of the resources used, by concurrent tasks, never exceeds a given limit at any instant. Hence, the devices constraints may be classified as: memory constraints, processing load constraints, and communication media capacity constraints. To describe these constraints, define the following parameters:

m_i	memory requirements of task i,
p_i	computation load requirements of task i,
d_{ij}	data to be transferred between tasks i and j.
b_{ij}	communication capacity requirements of edge (i,j).
M_p	available memory capacity of computer p,
P_p	available computation capacity of computer p,
A_s	available communication capacity of resource s,
C_{ip}	cost of processing task i on computer p.
C_{ijpq}	cost of transferring data d_{ij} between two tasks i and j if they are assigned to different computers p and q respectively. We assume that $C_{ijpq} = 0$ if the two tasks i and j are assigned to the same computer.

2.2.1. Location constraints:

The location constraints guarantee that each task is assigned to one and only one processor on which it is entirely executed without preemption, i.e., no software/task redundancy is considered. To do so, the following equality must hold at each task:

$$\sum_p X_{ip} = 1$$

2.2.2. Memory Constraints

For an assignment X, the total memory required by all tasks assigned to a processor p must be less than or equal to the available memory capacity of the processor p. Let mi denotes the amount of memory required for processing a task i and Mp defines the available memory capacity at the processor p, then the following inequality must hold at each processor p in the system:

$$\sum_{i \in TC_p} m_i \leq M_p$$

2.2.3. Processing Load Constraints

For an assignment X, the total processing load required by all tasks assigned to a processor p must be less than or equal to the available computational load of the processor p. Let pi denotes the processing load requirements of a task i and Pp denotes the available processing load of the processor p, then the following inequality must hold at each processor p in the system:

$$\sum_{i \in TC_p} p_i \leq P_p$$

2.2.4. Communication Capacity Constraints

For an assignment X, the total communication capacity required by all edges mapped to a communication path/link pq must be less than or equal to the available communication capacity of the path pq. Let bij denotes the amount of communication capacity required to communicate data between tasks i and j residing at different processors p and q respectively, and Apq denotes the available communication capacity of the path/link pq. Then, the following inequality must hold at each communication path/link pq.

$$\sum_{i \in TC_p} \sum_{j \neq i,\ j \in TC_q} b_{ij} \leq A_{pq}$$

Note that, the available communication capacity Apq of the path/link pq is the minimum available communication capacity over all the communication resources of the route from p to q. Let As be the available communication capacity of a communication resource s, then the available communication capacity of the path/link pq may be defined as Apq = min {As | s in resources (path pq)}.

In general, the above constraints are very important to be considered with the assignment problem. If such constraints are not considered, a task may be allocated to a machine that cannot process the task. Indeed, the communication capacity constraints influence the performance of the distributed system and should be considered with the task assignment.

2.3. Mathematical Modeling

Based on the different components of the cost function and the different types of constraints, several versions of the task assignment problem may be formulated. For example, for a given application of M tasks and a distributed system of N computers, if the objective is to find an assignment that minimizes the total sum of execution and communication costs such that each task i must be assigned to exactly one processor, then the problem may be formulated as follows:

$$min \sum_i \sum_p C_{ip}\, X_{ip} + \sum_{(i,j) \in E} \sum_p \sum_{q \neq p} C_{ijpq}\, X_{ip}\, X_{jq}$$

$$subject\ to$$

$$\begin{array}{ll} \sum_p X_{ip} = 1 & \forall\ tasks\ i \\ \sum_i p_i\, X_{ip} \leq P_p & \forall\ computers\ p \\ \sum_i m_i\, X_{ip} \leq M_p & \forall\ computers\ p \\ \sum_i \sum_{j \neq i} b_{ij}\, X_{ip}\, X_{jq} \leq A_{pq} & \forall\ paths\ pq \end{array}$$

3. Matrix Multiplication Techniques

Consider the matrix multiplication product C = A×B where A, B, C are matrices of size n X n as shown in Fig. 1 where n = 4. Next subsections present the various methods that used to find the matrix multiplication.

3.1. Sequential Method

The matrix operation derives a resultant matrix by multiplying two input matrices a & b, where matrix a is a matrix of N rows by P columns and matrix b is of P rows by M columns. The resultant matrix c is of N rows by M columns. The serial realization of this operation is quite straightforward as listed in the following:

```
for(k=0; k<M; k++)
  for(i=0; i<N; i++){
    c[i][k]=0.0;
    for(j=0; j<P; j++)
       c[i][k]+=a[i][j]*b[j][k];
  }
```

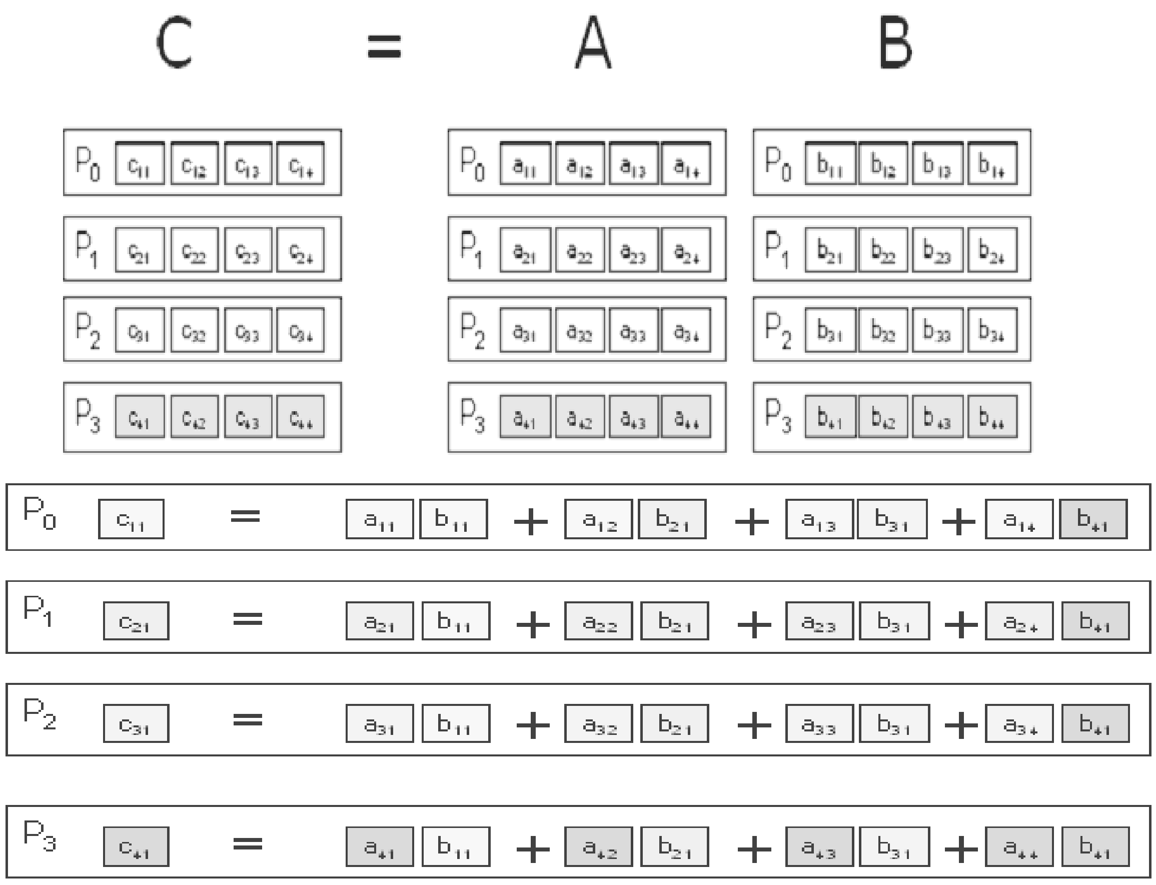

Fig. 1 Matrix multiplication

The above algorithm requires n^3 multiplications and n^3 additions, leading to a sequential time complexity of O (n^3).

3.2. Circular pipeline method

The slaves are peer processes that interact by means of circular pipeline as shown in Fig. 2. Assume that a, b, and c are n × n matrices. Each slave has one row and the first slave have b matrix. Thus each slave execute a series of rounds, where in each round it sends its column of b to the next slave and receives a different column of b from the previous slave.

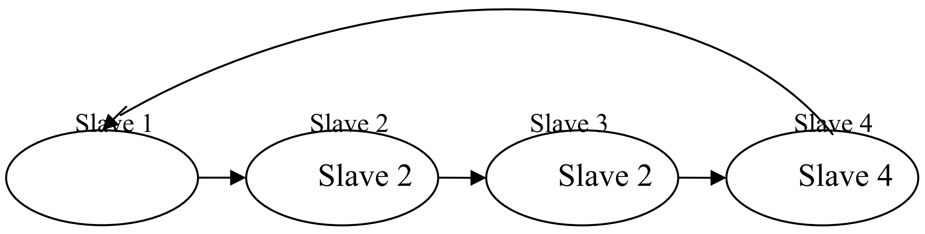

Fig.2 A Circular Pipeline

In Fig. 2, the next slave is the one with the next higher index, and the previous slave is the one with the next lower index (for the slave n, the next slave is 1, for the slave 1, the previous worker is n). The columns of matrix b are passed circularly among the slaves so that each slave sees every column. It is assumed that master process sends rows of A and columns of B to the slaves and then receives rows of C from the slaves.

3.3 Server Client Model

The matrix multiplication algorithm is implemented in MPI using the straight forward algorithm based on the Server Client paradigm [8]. Server Client computing paradigm, which also called replicated slave computing is consists of broken many computational problems into smaller pieces that can be computed by one or more processes in parallel. The computations are fairly simple, which usually compute-intensive, region of code. The size of loop is quite long.

Fig. 3 shows a Server Client computing paradigm, a Server process takes the work performed in the computationally intensive loop and divides it up into a number of tasks that it deposits into a task bag. One or more processes, known as Client grab these tasks, compute them and place the results back into a result bag. The Server process collects the results as they are computed and combines them into something meaningful such as a vector product.

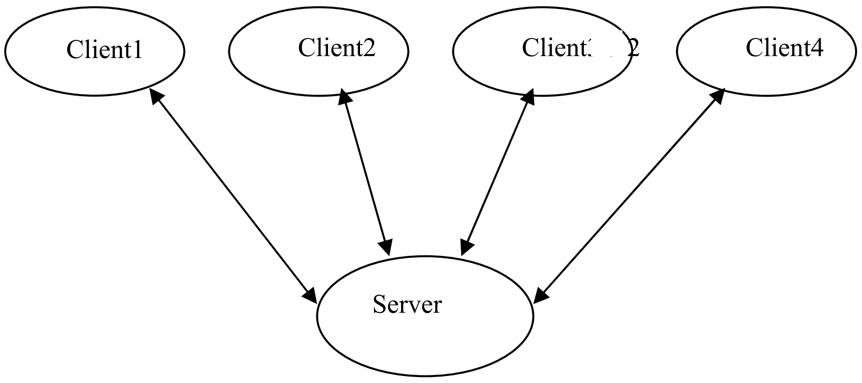

Fig. 3 Server Client computing paradigm

Message Passing Interface is a widely used standard for writing message-passing programs to establish a practical, portable, efficient, and flexible standard for message passing. The Server creates a set of random matrices. Each matrix multiplication job consists of pair of matrices to be multiplied. For each job the Server sends one entire matrix to each slave and distributes the rows of the matrix among the clients. In this way matrix multiplication jobs are computed in a parallel fashion as follow;

(1) The server process for each job, which sends the first matrix from the pair of matrices multiplication joined with a certain number of rows of the other matrix depending on the number of clients.
(2) Each client process receives one entire matrix and a certain number of rows of the other matrix based on the number of clients. Thus it computes the rows of the resulting matrix and sends it back to the server.
(3) The Server process collects the rows of resulting matrix from the clients.

4. Matrix implementation performance models

We develop an analytical performance model to describe the computational behavior of 3- matrix multiplication implementations. We consider the matrix multiplication product C = A×B where the size of matrices A, B, and C are n*n. The system is consists of one server machine and N clients, and the performance modeling of the three implementations is presented in next subsections.

4.1. Allocation block distribution of the matrix B columns

For the analysis, we assume that the entire matrix B stored in the local disk of the server. The basic idea of this implementation is as follows: The server broadcasts the matrix A to all clients. It partitions the matrix B into blocks of columns b and each block is distributed dynamically to a client. Each client executes a parallel matrix – vector multiplication algorithm between the matrix A and the corresponding block of b columns. Finally, each client sends back a block of size b columns of the matrix C.

The execution time of the dynamic implementation that is called matrix multiplication for distributed B columns block (MM-DBCB) can be broken up into five terms:

T1: It includes the communication time for broadcasting of the matrix A to all clients involved in processing of the matrix multiplication by using RMI. The size of each row of the matrix A is n*sizeof(int) bytes. Therefore, the total time T1 is given by:

$$T1= b* n *sizeof(int) / S$$

Where S is the communication speed

T2: It is the total time to read the columns of the matrix B into several blocks of size b*n*sizeof(int) bytes from the local disk of the server. The b is the number of columns. Therefore, the server reads n2*sizeof(int) bytes totally of the matrix. Then, the time T2 is given by:

$$T2= n2 *sizeof(int) / R$$

Where R is the I/O read time of the server

T3: It is the total communication time to send all blocks of the matrix B to all clients. The size of each block is b*n*sizeof(int) bytes. Therefore, the server sends n*sizeof(int) bytes totally. Then, the time T3 is given by:

$$T3 = n *sizeof(int) / S$$

T4: It is the average computation time across the systems. Each client performs a matrix – vector multiplication between the matrix A and the block of the matrix B with size n*(n/b)*sizeof(int) bytes. Then, the time T4 is given by:

$$T4 = n *(n/b)*sizeof(int)$$

T5: It includes the communication time to receive n / b results from all clients. Each client sends back b*n*sizeof(int) bytes. Therefore, the server will receive n 2*sizeof(int) bytes totally. Therefore, the time T5 is given by:

$$T5 = n 2*sizeof(int) / S$$

Therefore, the total execution time of our dynamic implementation, TN, using N clients, is given by:

$$TN = T1+T2+T3+T4+T5$$

4.2. Allocation blocks distribution of the matrix A rows and B columns

For the analysis, we assume that the entire matrices A and B stored in the local disk of the server. The basic idea of this implementation is as follows: The server broadcasts the blocks of matrix A by rows to all clients and. Also, it sends the corresponding blocks of columns from matrix B to the same clients. The server reads a block of size b rows and columns of the matrix A and B from the local disk of the server. Also, each client executes a parallel matrix – vector multiplication algorithm between the rows block of matrix A and the corresponding columns block of matrix B. Finally, each client sends back a block of size b columns of the matrix C. The execution time of the dynamic implementation that is called Matrix multiplication by distributed rows block of matrix A and columns block of matrix B (MM-DAR&BCB), can be broken up into five terms:

T1: It includes the communication time for broadcasting of the matrix A by rows and matrix B by columns to all clients involved. The amount of this time is equal to T3 of the previous dynamic implementation.

T2: It is the same time of T2 of the previous implementation.

T4: It includes the average computation time across the client. The amount of this time is similar to the time T4 of the previous implementation.

T5: It includes the communication time to receive the results of the matrix - vector multiplication from all clients. The amount of this time is same with the time T4 of the previous implementation.

Therefore, the total execution time of our dynamic implementation is reduced from the previous model by the value of T1. TN, using N clients, is given by:

$$TN = T1+T2+T4+T5$$

4.3. Allocation block distribution of the matrix A rows

This algorithm will give the same amount of TN of the first model as in section of 4.1 and is called matrix multiplication for distributed A rows block (MM-DARB).

5. Implementation architecture

In our work we develop server client model to calculate MM by using RMI Java threads for the above models in section 4 especially with detailed discussion on the 4.2 model. In our proposed model, the server determines the distributed numbers of rows from the first matrix and the corresponding columns of the second matrix depending on the balance of workload on registered clients. In the heterogeneous matrix, which mean the number of rows is not the same number of columns, the multiplication will done because the load distribution depend on the number of rows in the first matrix with its corresponding columns of the second matrix. One server and 7-clients are used to implement MM as shown in Fig. 4. RMI implementation algorithm is shown as follow;

Step 1 Client discovery;
Client will register itself with the server to take a task from it

Step 2 Generate Matrices;
Server generates two matrices randomly or getting them as inputs

Step 3 Data distribution
Server will distribute number of rows from first matrix and its corresponding columns of the second matrix on clients that it has been registered using Java threads.

Step 4 Sever waits for result;
Server will waits results from clients and append it in result matrix

Step 5 Results collecting;
Server will collect the results that sent by each client and compute the time that taken by each client and compute all time taken in this process

Step 6 Shutdown;
Finally, server will send shutdown to all clients.

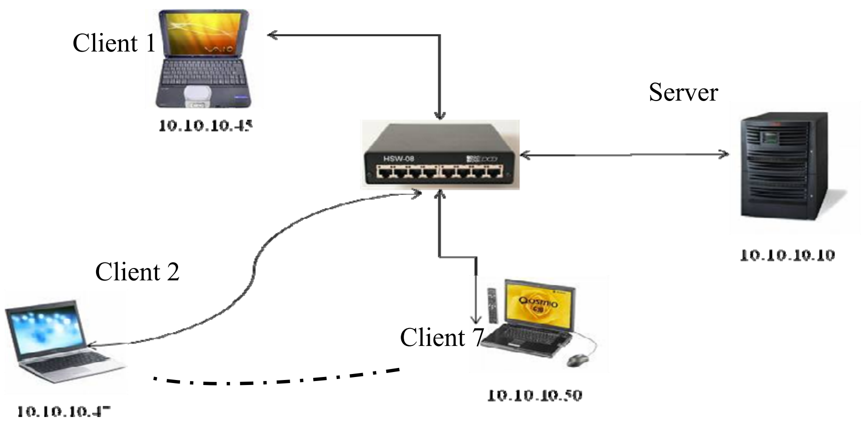

Fig. 4 RMI Server-Client architecture

Fig. 5 shows the flow diagram of the above algorithm. In our work we addressed the following issues:

(4) Performance: As the number of clients increases the time for computation will decreases.

(5) Load Distribution: the work will be distributed among the free clients. Rows and columns of MM are distributed uniformly on all registered clients. In RMI the load of MM was distributed by allowing each process (Threads) to compute a certain number of rows in the resulting matrix.

(6) Scalability: Performance of our model increases as the number of registered clients increases.

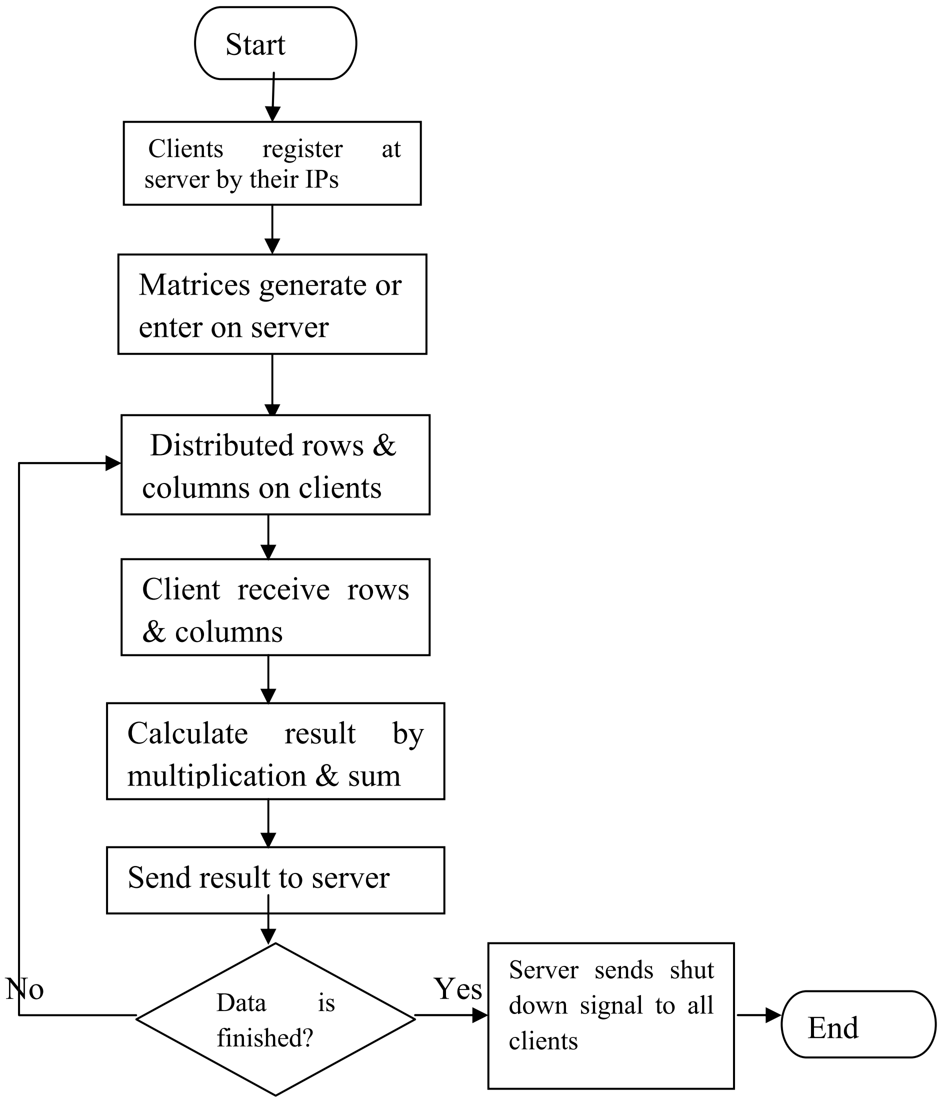
Start
Clients register at server by their IPs
Matrices generate or enter on server
Distributed rows & columns on clients
Client receive rows & columns
Calculate result by multiplication & sum
Send result to server
Data is finished?
No
Yes
Server sends shut down signal to all clients
End

Fig. 5 flow diagram of MM algorithm

6. Experiment results

We develop an analytical performance model to describe the computational behavior of matrix multiplication implementations. We consider the matrix multiplication product C = A×B where the size of matrices A, B, and C are n X n. We implement MM by using JDK 1.6 on LAN of 8-PC 2.66 GHz with 512 MB Ram. Also, the matrix size does not exceed the memory bound of any machine in the system.

Measuring the performance of a parallel program is a way to assess how well and efficient our development, which have been divided the big application into small modules cooperating with each other in parallel.

The most visible and easily recorded metric of performance is the execution time. By measuring the time consumed in execution of parallel program, we can directly measure its effectiveness. To find out how much better for our proposed does on the parallel machine, which it compared with running an application on only one processor. The ratio of execution time is taken into the account, which is called the speedup.

Speedup = (Serial Execution Time)/(Parallel Execution Time)

Speedup = T(1)/T(N)

Where T(N) represents the execution time taken by the program running on N processors, and T(1) represents the time taken by the best serial implementation of the application measured on one processor. Figure 6 shows speedup is equal to 9 for MM with size 2048 X 2048 or reduced the computation time by 89 %. Also, Figure 7 shows speedup is reduced and equal to 6 for MM with size 2048 X 2048 or reduced the computation time by 84 %.

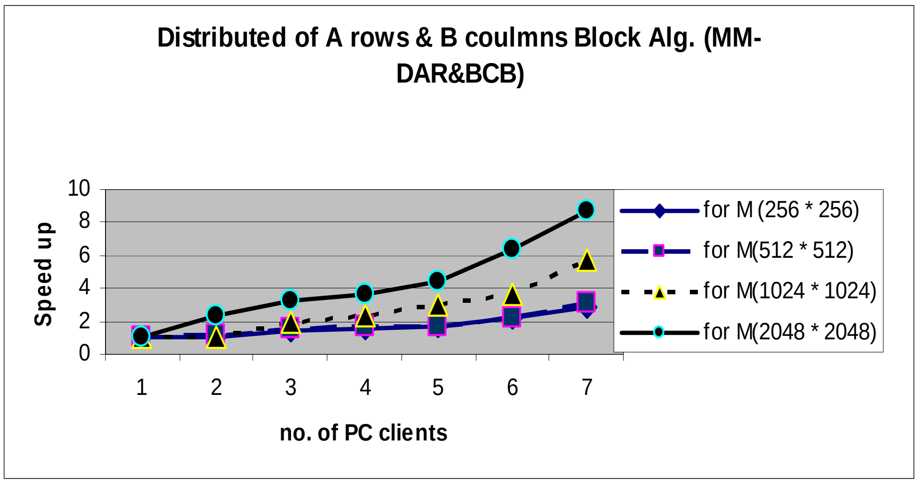

Fig. 6 Experiment results of (MM-DAR&BCB) model of section 4.2

7. Conclusion and Future Work

In this paper we implemented and analyzed the parallel matrix multiplication on distributed systems. Our mechanism will make it easier to automatic migrate the computation load to client. It has been shown that the execution time decreases by 89% for MM with size 2048 X 2048. Future work will apply this implementation on any practical application like weather prediction, databases systems, data compression and others with increasing the numbers of clients.

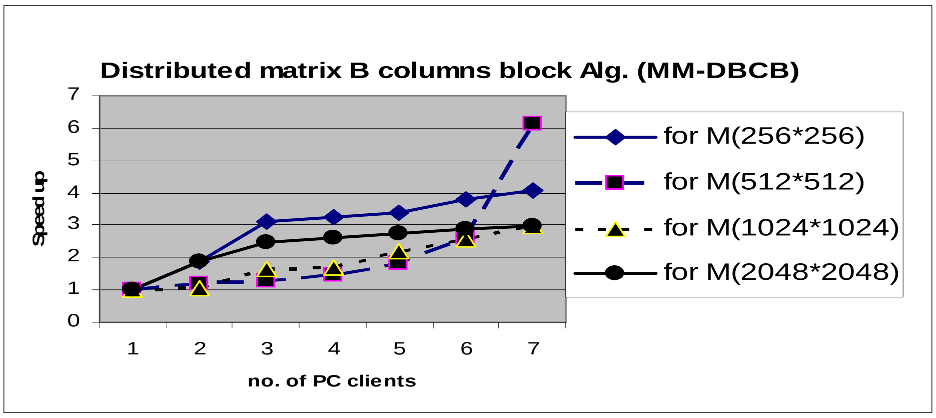

Fig. 7 Experiment results of (MM-DBCB) model of section 4.1

8. Refferences

[1] Tinetti, F., Quijano, A., Giusti, A., Luque, E. “Heterogeneous networks of workstations and the parallel matrix multiplication”, Proceedings of the Euro PVM/MPI 2001, Springer-Verlag, Berlin, pp. 296-303. 2001.

[2] T. Typou, Vasilis stefanids, P. Michailidis, and K. Margaritis,“Implementing Matrix Multiplication on An Cluster of Workstation”, 1st IC-SCCE, Athens, 8-10 Sep., 2004.

[3] Ju-wook Jang, Seonil Choi and Viktor K. Prasanna, “Energy-Efficient Matrix Multiplication on FPGAs”, IEEE Transactions on VLSI (TVLSI), Vol. 13, No. 11, pp. 1305-1319, 2005.

[4] Carrio and Geleertner “How to write Parallel Programs, A Guide to the Perplexed” ACM Computing Serveys, Vol. 21, No. 3, Sep.1999.

[5] Coulouris, et al., “Distributed Systems Concepts and Design”, 3rd Edition, Addision Wesely, Person Education 2001.

[6] Maassen, J., Nieuwpoort, R. V., Veldema, R., Bal, H., and Kielmann, T., “Wide-Area Parallel Computing in Java”, Proceedings of the ACM Conference on Java Grande, San Francisco, CA,(1999), pp. 8-14, 1999.

[7] Grama, A., Gupta, A., Karypis, G., and Kumar, V., “Introduction to Parallel Computing” (Second Edition), Pearson Education Limited, Harlow, England, (2003), pp. 345-349, 2003.

[8] Wilkinson, B., Allen, “Parallel Programming: Techniques and Applications Using Networking Workstations”, Prentice-Hall, Inc. 1999.

[9] G. Attiya and Y. Hamam, “Task allocation for maximizing reliability of distributed systems”, Elsevier journal of parallel and distributed computing, 66, pp. 1259-1266, 2006