ISSN:0975-9646

Performance Analysis of Matrix Multiplication Algorithms Using MPI

Javed Ali ,Rafiqul Zaman Khan

Department of Computer Science, Aligarh Muslim University, Aligarh.

Abstract :The practical analysis of parallel computing algorithms is discussed in this paper. The cluster is used to analyze the performance of the algorithms by using the various nodes of the cluster. Parallel computing by the MPI has made a tremendous impact on a variety of areas ranging from computational simulation for scientific and engineering applications to commercial application. We propose the performance analysis of the matrix multiplication algorithms through MPI.

Keywords: Imperative, Declarative, Speedup, Efficiency, correlation, researcher etc.

INTRODUCTION:

Parallel algorithms play an important role in the computation of the high performance computing environment. Dividing a task into the smaller tasks and assigning them to different processor for parallel execution are the two key concepts in the performance of parallel algorithms. Multiprocessor machines allow different application program to execute at the same time at different processor. They also allow a single application program to execute faster if it can be rewritten to use multiple processors. There are the two types of programs

(a) Imperative (b) Declarative

In imperative program, the programmer has to specify the action of each process and how they communicate and synchronized. This contrast with declarative program e.g. functional or logic programs in which the concurrency is implicit and there is no state information of a program [2]. In declarative programs, independent part of the program may execute in parallel; they communicate and synchronize implicitly when one part depends upon the results produced by another. The implementation of the declarative programs on the traditional machine is possible only when the imperative program is written on the machine.

The most common way to write a parallel program to use a sequential language and a subroutine library. In particular, the bodies of process are written in the sequential language such as C or FORTRAN. Process creation, communication and synchronization are then programmed by calling library function. For message passing environment we use the MPI.A parallel programming library contains subroutines for process creation, process management, communication and synchronization. The nature of routines and particularly there implementation depend upon weather the library supports shared variable programming or message passing. The MPI and PVM libraries are two common standards for message passing; both have widely used, public domain implementation that supports both C and FORTRAN [11].

The complete MPI [12] specification consists of nearly 129 calls. However, a beginner MPI programmer can get by with very few of them (six to twenty-four). All that is really required is a way for processes to exchange data, that is, to be able to send and receive messages.

The following outline can be used to structure most MPI programs:

- All MPI programs must include a header file (in C, mpi.h; in FORTRAN, mpif.h).
- All MPI programs must call MPI_INIT as the first MPI call, to initialize themselves.
- Most MPI programs call MPI_COMM_SIZE to determine the size of the current virtual machine, that is, how many processes are running.
- Most MPI programs call MPI_COMM_RANK to determine their rank, which is a number between 0 and size-1.
- Conditional process and general message passing can take place, for example, using the calls MPI_SEND and MPI_RECV.
- All MPI programs must call MPI_FINALIZE as the last call to an MPI library routine.

Hence, by using just the following six calls, that is, MPI_INIT, MPI_COMM_SIZE, MPI_COMM_RANK,MPI_SEND, MPI_RECV, and MPI_FINALIZE, a number of useful MPI programs can be written.

Architecture independent parallel algorithms may used to write a parallel code that is scalable portable and reusable. In a distributed architecture, processors have their own private memory and they interact using a communication network rather than a shared memory.So , processes can't communicate directly by sharing variables instead, they have to exchange message with each other. A message passing is best for programming producer consumer and interacting peers, whereas RPC (Remote Procedure Call) and rendezvous are best for client-server programming. The Parallel Universal Matrix Multiplication Algorithm(PUMMA) that include matrix multiplication routines and their performance depend weakly on processor configuration and block size[3].The PUMMA package may be implemented for single precision real and complex, and double precision real and complex.

Parallel matrix multiplication has been investigated extensively in the last two decades [4-5]. There are different approaches for matrix-matrix multiplication: 1Dsystolic [6], 2D-systolic, Cannon's algorithm [4], Fox's algorithm [7],Berntsen's algorithm [8], the

transpose algorithm [8] and DNS algorithm [7, 9,10]. Fox's algorithm was extended in PUMMA (Parallel Universal Matrix Multiplication Algorithm) using different data distribution formats. DIMMA (Distribution Independent Matrix Multiplication Algorithm) [11] is related to SUMMA(Super Scalar Matrix Multiplication Algorithm) but uses a different pipelined communication scheme for overlapping communication and computation. Digital image processing encompasses broad spectrum of mathematical methods. They are transform techniques, convolution, correlation techniques in filtering processes and set of linear algebraic methods like matrix multiplication, pseudo inverse calculation, linear system solver, different decomposition methods, geometric rotation and annihilation. Generally we can classify all image processing algorithms into two groups: basic matrix operations and special image processing algorithms. Fortunately, most of the algorithms fall in the classes of the matrix calculations, convolution, or transform type algorithms. These algorithms possess common properties such as regularity, locality and recursive. In this paper, the speedup of a parallel algorithm is defined where it can be defined as a ratio of the corresponding sequential and parallel times.

1.1 Interacting Peers: There are three useful communications pattern; centralized, symmetric and ring. The processes are the nodes in the graph and edges are the pairs of communication channels. According to the literature analysis the symmetric solution is the shortest and easy to program because every process does exactly the same thing .It also uses the largest no of message (unless broadcast is available).The message could be transmitted in parallel if the underlying communication network support concurrent transmission. Communication overhead greatly diminishes performance improvement (speedup) that might be gained from parallel execution. In the centralized solution, the message sent to the co-coordinators are all sent at about the same time; hence only the first receive statement executed by the coordinator is likely delay for very large problem. Similarly, the results are sent one after the other from the coordinator to the other process, so the other process should be to awaken rapidly. The ring solution is inherently the linear with no possibility of overlapping message transmissions. Hence, the ring based solution will perform poorly.

1.2 Iterative Parallelism: Matrix Multiplication

An iterative sequential program is one that uses for and while loops to examine data and computer result. An iterative parallel program contains two or more iterative processes. Each process compute results for a subset of data, then the result are combined.

e.g. Given matrix a, and b, assume that each matrix has n rows and columns, and that each has been initialized. This require computing n^2 inner products, one for each pair of rows and columns.

The shared variable declared as follows:

Double a[n,n], b[n,n], c[n,n];

The computation of the matrix multiplication follows the sequential program; The inner loop (with index k) computes the inner product of row I of matrix a and column j of matrix b, and then stores the result in c[i,j][1].Matrix multiplication is an example of embarrassingly parallel application, because there are a multitude of operations that can be executed in parallel. Two operations can be executed in parallel if they are independent. Since the write sets for pair of inner product are disjoint, we could compute all of them in parallel. Alternatively, compute row of result in parallel, column of results is parallel, or blocks of rows or columns in parallel.

The concurrent programming can be achieved by using **co**(concurrent) statement:

```
co[i=0 to n-1]{      # compute rows in parallel

      for [j=0 to n-1]{
          c[i,j]=0.0;
          for[k=0 to n-1]
c[i,j]=c[i,j] +a[i,k]*b[k,j];
}

}
```

The co statement specifies that its body should be executed concurrently, depending upon the number of processors, for each value of index i. The matrix multiplication may be achieved by using the column of c on parallel:

```
co[j=0 to n-1]{ # compute column in parallel

    for [i=0 to n-1]{
            c[i,j]=0.0;
            for[k=0 to n-1]
   c[i,j]=c[i,j] +a[i,k]*b[k,j];
   }

   }
```

It's safe to interchange two loops as long as the bodies are independent and hence compute the same result, as we do there. We can also compute all inner products in parallel.This can be programmed by using a single co statement with two indices:

```
co[i=0 to n-1 , j=0 to n-1]{# all rows all column in parallel

          c[i,j]=0.0;
          for[k=0 to n-1]
  c[i,j]=c[i,j] +a[i,k]*b[k,j];

  }}
```

The body of the above co statement is executed concurrently for each combination of values of i and j.The speedup and the efficiency analysis is performed by the formula given below:

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

Efficiency E(%) = Speedup (S) /No of processers (P)

1.3 Cluster Hardware Requirements: Hardware configuration for the cluster formation is the basic requirement for the computation of the parallel program for matrix multiplication. We used various nodes of the configuration, Pentium 4,2 GB RAM, Speed 2.80 GHz, Dell Intel ™ Core ™ 2 Duo CPU E-7400.The Fedora version of Linux operating system make the computation easy by using the MPI standard message passing library.

1.4 Conclusion: The inefficient partitioning of the tasks amongst the various nodes of the cluster, participating in the parallel computing give the poor results due to the scalability problem. Table 1, 2, 3 and the corresponding figures 1,2,3 shows the problem of scalability due to the inefficient partitioning of the matrix. The distributions of the tasks play the most important role in the parallel computing. While the distribution of the tasks take place in the efficient manner the results shows that the speed up and efficiency factor of the matrix multiplication algorithms increased at the significant level. Table 4 and corresponding results shows the speedup factor and efficiency enhancement of the matrix parallel computing. The efficiency increased by this method of computation is 81.55%.

NP(Number of Processor)	Running Time (S)
1	0.000016
2	12.510740
3	22.856150
4	21.223457
5	33.79125
6	33.8693

Table 1: Running Time Measurement

NP(Number of Processor)	Speedup
2	1.2789
3	0.7000
4	0.75388
5	0.47349

Table 2: Speedup Measurement

NP(Number of Processor)	Efficiency
2	0.6394
3	0.2333
4	0.1884
5	0.0946
6	0.0787

Table 3: Efficiency Measurement

No of the Processor for 3000*3000	Parallel Connectivity		
	Time (S)	Speedup (S)	Efficiency (E) (%)
2	325.23	1.05	52.0
3	187.25	1.83	61.0
4	110.56	3.10	77.50
5	84.26	4.07	81.46
6	70.14	4.89	81.55

Table 4: Computational Analysis if Serial Execution Time :343.23

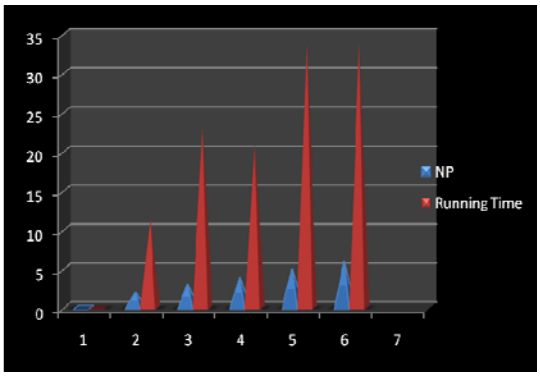

Figure 1:Running time Analysis

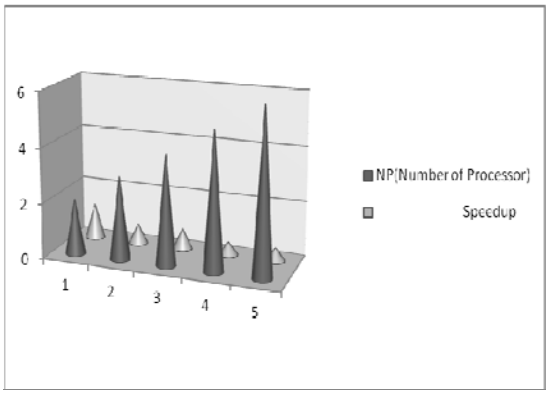

Figure 2:Speedup Analysis

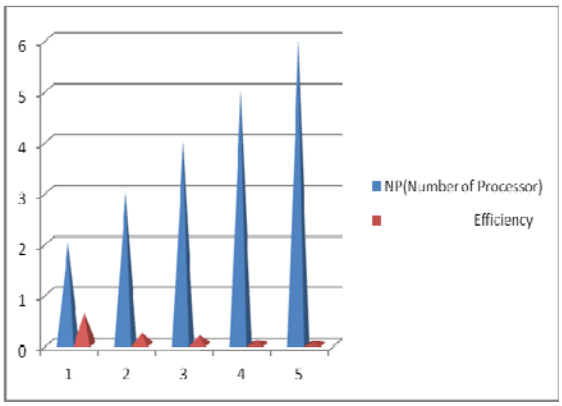

Figure 3:Efficiency Analysis

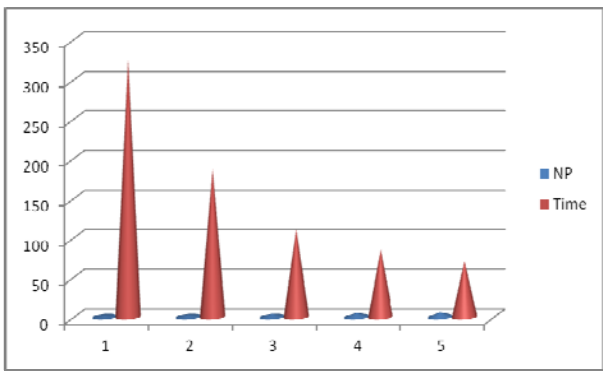

Figure 4:Running time Analysis

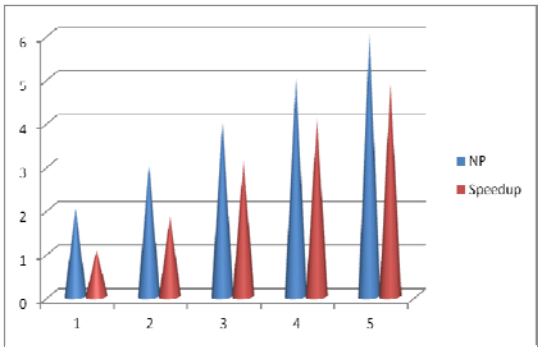

Figure 5:Speedup Analysis

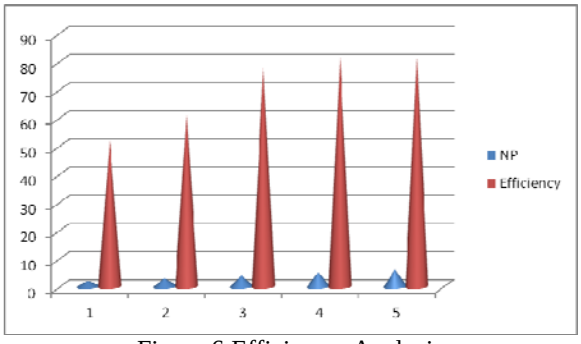

Figure 6:Efficiency Analysis

1.5 Future Work: We have presented the class of parallel matrix multiplication algorithms.Theoratical and experimental result shows that, by choosing the appropriate method of the task partitioning amongst the various nodes of the cluster give the better results in the comparison of the scalability problem. The efficiency increased in this paper is up to 81.55%.The researcher can show the hybrid algorithms that can solve the problem of choosing size automatically according to the need of the program and scalability problem.

REFERENCES:

1. Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarlings., Mckenney A., Sorensen D., "LAPACK Users' Guide, third ed. Society for Industrial and Applied Mathematics", Philadelphia, PA, 1999.
2. Whaley R. C., Petitet A., Dongarra J. J., "Automated empirical optimizations of software and the ATLAS project" Parallel Computing 27, 1.2 (2001), 3.35.
3. Choi,J., Dongarra J.J. and Walker D.W., "PUMMA:Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers", Concurrency Practice and experience,Vol6(7),54-570,1994.
4. Cannon L. E, "A cellular computer to implement the Kalman Filter Algorithm", Ph.D. dissertation, Montana State University, 1969.
5. Choi J., "A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers", in Proc. IPPS '97, 1997.
6. Golub G.H and Van C.H L., "Matrix Computations. ",Johns Hopkins University Press, 1989.
7. Fox G. C., Otto S. W., and Hey A. J. G., "Matrix algorithms on a hypercube I: Matrix multiplication",Parallel Computing, vol. 4, pp. 17-31. 1987.
8. Berntsen J., "Communication efficient matrix multiplication on hypercubes, Parallel Computing", vol. 12, pp. 335-342, 1989.
9. Dekel E., D. Nassimi, and S. Sahni, "Parallel matrix and graph algorithms", SIAM Journal on Computing", vol.10, pp. 657-673, 1981.
10. Ranka S. and Sahni S., "Hypercube Algorithms for Image Processing and Pattern Recognition", Springer- Verlag, New York, NY, 1990.
11. William G., Ewing Lusk, Nathan Doss, and Anthony Skjellum"High performance, portable implementation of the MPI message passing interface standard", Parallel Computing, 22(6):789-828,September 1996.
12. Fagg G. E., Bukovsky A., and Dongarra J. J., "Harness and fault tolerant MPI", Parallel Computing, 27(11):1479-1495, October 2001.