Performance of Matrix Multiplication by Using MPI Parallel Programming Approach

[*1]Berna Seref and [2]M. Akcay
[*1] Faculty of Engineering, Department of Computer Engineering Dumlupinar University, Turkey
[2]Faculty of Engineering, Department of Computer Engineering Dumlupinar University, Turkey

Abstract

With the increasing requirements of high performance computing, parallel programming has become popular and important. One of the reason is parallel programming gives an opportunity to use computer resources more efficient. Thus, running time decreases and software quality increases. In this study, matrix multiplication is done by using message passing interface (MPI) parallel programming approach and it is tested on various core machines with different size of matrixes such as 500X500, 1000X1000 and 2000X2000. It is tested with the process number 1, 2, 4, and 8. Performance of matrix multiplication with MPI is studied. Relationship between computation time –matrix size– number of cores and processes is examined, experimental results are collected and analyzed.

Key words: High performance computing, parallel programming, message passing interface, matrix multiplication.

1. Introduction

Parallel programming means that programming more than one computer or computers with more than one core to solve a problem in a short time with greater computational speed [1]. In here, problem is divided into smaller parts and executed concurrently. After execution is completed, results from all parts are combined and problem is solved.

With the increasing requirements of high performance computing, parallel programming has become very popular. The reason of that it is easy to solve larger and complex problems in a short time thanks to the computing resources that do multiple things at a time concurrently [2]. In order to write efficient parallel programs, hardware knowledge is as important as software knowledge for the reason that using all cores and memory in an efficient way.

Parallel programming models can be classified as POSIX (Portable Operating System Interface) Threads, Shared Memory OpenMP (Open Multi-Processing), Message Passing, CUDA (Compute Unified Device Architecture), OpenCL (Open Computing Language), DirectCompute and Array Building Blocks [3]. In this study, matrix multiplication is done by using Message Passing Model

*Corresponding author: Address: Faculty of Engineering, Department of Computer Engineering Dumlupinar University, 43100, Kutahya TURKEY. E-mail address: berna.seref@dpu.edu.tr

on Visual Studio platform and coded with C++ programming language. It is tested on various core machines with different size of matrixes such as 500X500, 1000X1000, 2000X2000, and with the process number 1, 2, 4, and 8 for the aim of observing relationship between computation time -matrix size- number of cores and processes.

In the second part of this study, materials and methods are explained, information about Message Passing Model and Message Passing Interface is given and algorithm which is used to compute matrix multiplication is explained. In the third part, computation time of matrix multiplication with different size of matrixes such as 500X500, 1000X1000, 2000X2000, and with the process number 1, 2, 4, and 8 is given with tables and graphics. At the last part, relationship between computation time–matrix size–number of cores and processes is explained. Finally, conclusions are given and future work is explained.

2. Materials and Method

In this study, matrix multiplication is done by using message passing interface on Visual Studio platform and coded with C++ programming language.

In the Message Passing Model, each process has its own address space and communication is performed by sending and receiving messages [4].

Message Passing Interface (MPI) is a language-independent communications protocol and message passing system that is used with the aim of high performance, scalability, and portability [5]. Implementation of Message Passing Model is performed by defining library of routines such as point to point communication functions and collective operations [6]. Message passing is realized by sending and receiving messages. To send data to whole group of processes it is broadcasted [7].

```
MPI_Send(void* data, int count, MPI_Datatype datatype,int destination, int tag, MPI_Comm communicator)
```

MPI communication routine which is shown above is used to send data in size of "`count`", in data type of "`datatype`" to the process whose id is "`destination`" with the message tag "`tag`" to the communicator "`communicator`".

```
MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)
```

MPI communication routine which is shown above is used to receive data in expected size of "`count`", in data type of "`datatype`", from the process whose process id is "`source`" with the message tag "`tag`" to the communicator "`communicator`", in status "`status`".

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int
root, MPI_Comm comm)
```

MPI communication routine which is shown above is used to send the same data to all processes in the communicator . According to the above equation, data in size of "`count`", in data type of "`datatype`" is sent to all processes in communicator "`comm`" from "`root`". Other processes in communicator must call `MPI_Bcast` routine to receive data because there is no MPI call to receive `MPI_Bcast` routine.

Most commonly used MPI routines are `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Abort`, `MPI_Get_processor_name`, `MPI_Get_version`, `MPI_Initialized`, `MPI Wtime`, `MPI Wtick` and `MPI Finalize`. `MPI Init` is used to initialize MPI execution environment, so, it must be called by every MPI programs only for once before all MPI functions. `MPI_Comm_size` is used to learn total number of processes for specified communicator. `MPI_Comm_rank` routine returns rank number of MPI process. `MPI_Abort` is used to terminate all MPI processes for specified communicator where as `MPI_Finalize` is used to terminate MPI environment. To learn processor name `MPI_Get_processor_name`; to learn MPI version and subversion `MPI_Get_version`; to learn elapsed wall clock time in seconds for specified processor `MPI_Wtime` routines functions are used [8].

2.1. Theory/calculation

Matrix multiplication is done by using asynchronous and synchronous message passing. Firstly A and B matrixes are created randomly on mode 10 and on size N which can be 500, 1000 and 2000.

```
for(i=0;i<N;i++)
          for(j=0;j<N;j++) {
              A[i][j] = rand() % 10;
              B[i][j] = rand() % 10;
    }
```

Then, interval which is equal to division of `N` by `Total Number Of Processes` and remainder which is equal to mode of `N` on `Total Number Of Processes` is calculated. While matrix `A` is sent to processes interval by interval, matrix `B` is broadcasted to all processes. To send matrix `A` interval by interval to the processes, synchronous message passing is used. As a result, sender and receiver wait for each other to transfer intervals of matrix `A`.

```
for(s=1;s< TotalNumberOfProcesses;s++){
    MPI_Isend(A+(s*interval),interval*N,MPI_DOUBLE,s,s,MPI_COMM_W
ORLD,ireq+s);
}
```

```
MPI_Bcast(B,N*N,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

Matrix `B` and intervals of matrix `A` are received by worker processes.

```
MPI_Bcast(B,N*N,MPI_DOUBLE,0,MPI_COMM_WORLD);

MPI_Recv(A+(rank*interval),interval*N,MPI_DOUBLE,0,rank,
MPI_COMM_WORLD,&status);
```

Multiplication of first interval and remainder part of matrix `A` with matrix `B` is calculated by root process. Multiplication of other intervals with matrix `B` are calculated by worker processes.

```
void compute(int startpoint, int interval)
{
  int i, j, k;
  for (i=startpoint;i<startpoint+interval;i++)
    for(j=0;j<N;j++)
      {
    AB[i][j]=0;
    for(k=0;k<N;k++)
      AB[i][j] += A[i][k] * B[k][j];
      }
}
```

All worker processes calculate its own part of matrix multiplication and send the result to root process.

```
MPI_Send(AB+(rank*interval),interval*N,MPI_DOUBLE,0,rank,MPI_COMM
_WORLD);
```

Results are received by root process by using synchronous message passing. As a result, sender will not continue to send result of matrix multiplication until root has received the result.

```
for(s=1;s< TotalNumberOfProcesses;s++)
MPI_Irecv(AB+(s*interval),interval*N,MPI_DOUBLE,s,s,MPI_COMM_WORL
D,ireq+s);
```

Calculation time is found by using `MPI_Wtime()` function. After randomly creating `A` and `B` matrixes, time is recorded.

```
time1 = MPI_Wtime();
```

After root gets the results from worker processes, time is recorded for the second time.

```
time2 = MPI_Wtime();
```

To get calculation time, these two time is subtracted from each other.

```
time = time2 - time1;
```

3. Results

Computation time is recorded for the matrix sizes 500X500, 1000X1000 and 2000X2000 with the number of processes 1, 2, 4 and 8 on cores 1, 2 and 4 for five times. Average time in second is observed. Results are shown below in Table 1, 2, and 3 and Figure 1, 2, and 3:

Table 1. Average Time in Second for Single, Dual, and Quad Core Machines for matrix in size 500X500

# of processes	Single Core Machine	Dual Core Machine	Quad Core Machine
1	0.818254	0.897007	1.754392
2	0.666878	0.423973	0.870584
4	0.206335	0.216844	0.414477
8	0.124731	0.128673	0.292490

Table 2. Average Time in Second for Single, Dual, and Quad Core Machines for matrix in size 1000X1000

# of processes	Single Core Machine	Dual Core Machine	Quad Core Machine
1	22.586789	9.648817	10.209541
2	10.209541	4.614202	5.527593
4	5.527593	2.307669	2.718064
8	2.718064	1.205743	1.374338

Table 3. Average Time in Second for Single, Double, and Quad Core Machine for matrix in size 2000X2000

# of processes	Single Core Machine	Dual Core Machine	Quad Core Machine
1	198.992849	74.525381	101.415833
2	94.303229	37.127187	50.750530
4	50.566620	18.702950	25.316927
8	26.695430	9.623290	13.316375

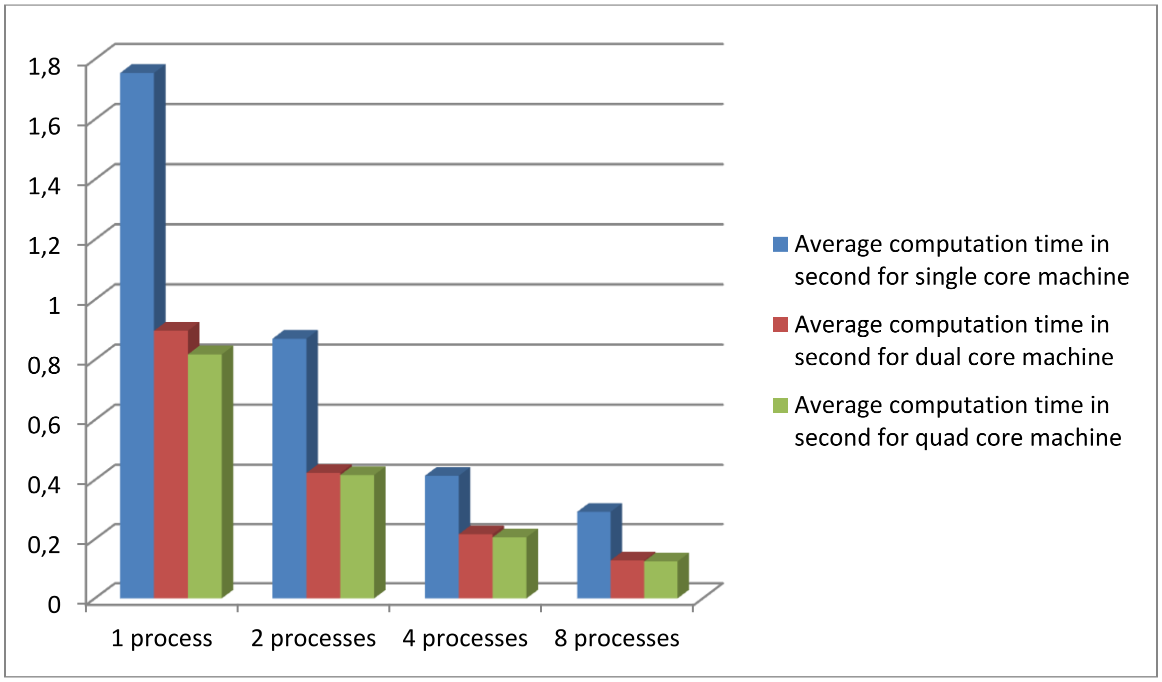

Figure 1. Average computation time in second for matrix size in 500X500 and for process number 1, 2, 4 and 8 on single core, dual core and quad core machine.

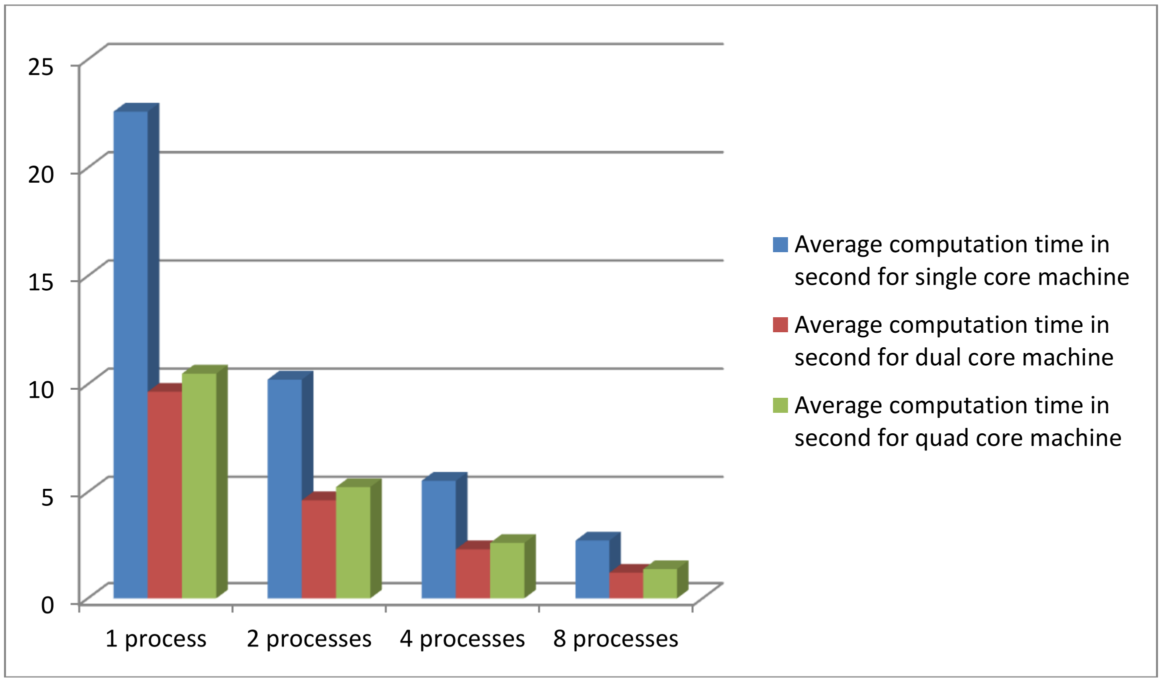

Figure 2. Average computation time in second for matrix size in 1000X1000 and for process number 1, 2, 4 and 8 on single core, dual core and quad core machine.

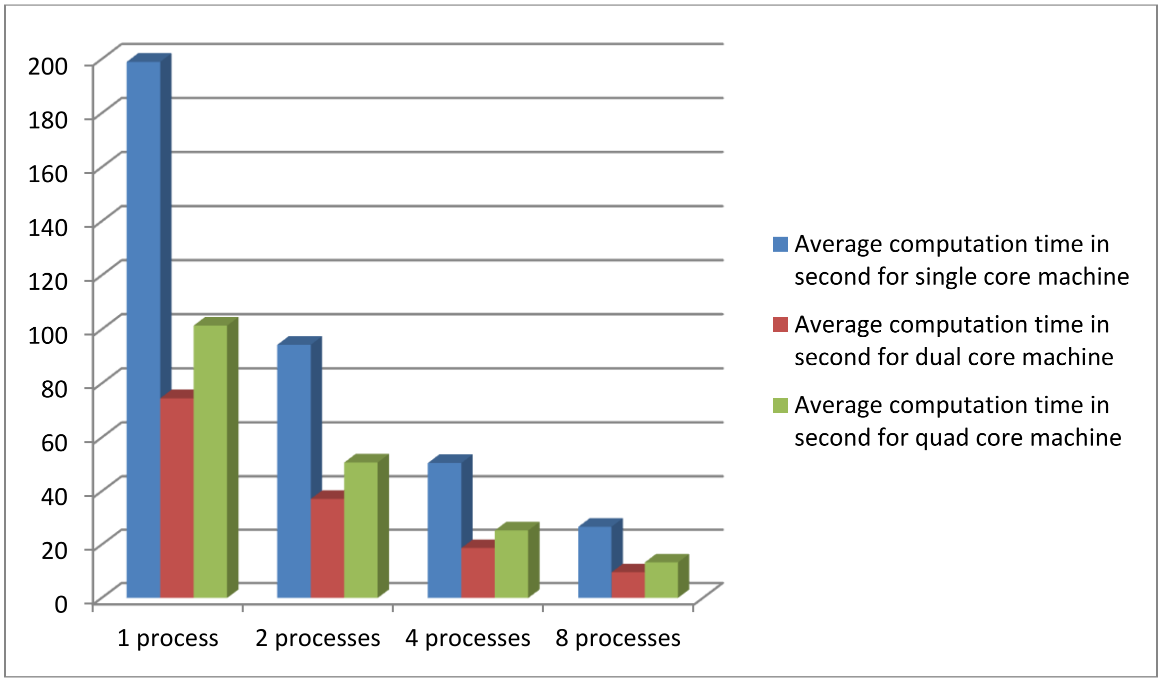

Figure 3. Average computation time in second for matrix size in 2000X2000 and for process number 1, 2, 4 and 8 on single core, dual core and quad core machine.

4. Discussion

When matrix size is larger, number of core and the number of process start to be more important. The reason for this; for larger matrix size, when more processes are created and a machine which has more cores is used, calculation time decreases greatly. For example, in this study, intervals of matrix `A` is sent to processes and matrix `B` is broadcasted. Then, multiplication of matrix `A` and `B` is calculated concurrently by each process. Then, results are sent to `root` process. As a result, performance of matrix multiplication is increased.

In this study, creating more processes on single core, dual core or quad core machine for all size of matrixes and using multi core machines decreased computation time. In addition, it is observed that using machine which has more cores compared to others, does not always give the best performance. For example; in this study, for matrix in size 2000X2000, dual core machine gives better performance when compared to quad core machine. The reason of that is communication between processes and cores cause time missing and this situation creates a negative impact on computation time.

Conclusions

In this study, matrix multiplication is calculated by using Message Passsing Interface (MPI) on Visual Studio Platform and coded with C++ programming language. Communication is done between processes by calling library routines. As a library; `stdio.h, stdlib.h, mpi.h,` and `time.h` are used. Algorithm is tested in size of matrixes 500X500, 1000X1000 and 2000X2000 with the process number 1, 2, 4 and 8 on various core machines. Test results are shown and explained in Part 3 and Part 4.

As a future work, it is planned to calculate matrix multiplication by using CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language), and compare efficiency of them.

References

[1] Zhang Junhua, Zhu Xiaodong, Huang Zhiqiu. Parallel Programming Patterns with Granular Computing. Computer Application and System Modeling (ICCASM), 2010; V7-300 - V7-302.
[2] https://computing.llnl.gov/tutorials/parallel_comp/#WhyUse, 06.02.2014.
[3] Diaz, J., Munoz-Caro, C. Nino, A. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. Parallel and Distributed Systems, IEEE Transactions on Volume: 23 , Issue: 8; 2012; pp: 1369 – 1386.
[4] Hongzhang Sha, Singh J.P., Oliker L., Biswas R. A Comparison of Three Programming Models for Adaptive Applications on the Origin 2000. Supercomputing, ACM/IEEE 2000 Conference. 2000 , Page(s): 11.
[5] http://en.wikipedia.org/wiki/Message_Passing_Interface 13.02.2014.
[6] Nicholas T. Karonis, Brian Toonen, Ian Foster. MPICH-G2: A Grid-enabled implementation of the Message Passing Interface. Journal of Parallel and Distributed Computing, Volume 63, Issue 5, May 2003, Pages 551-563.
[7] Rolf Hempel, David W Walker. The emergence of the MPI message passing standard for parallel computing. Computer Standards & Interfaces, Volume 21, Issue 1, 25 May 1999, Pages 51-62.
[8] https://computing.llnl.gov/tutorials/mpi/ 13.02.2014.