International Journal of New Computer Architectures and their Applications (IJNCAA) 4(2): 108-116
The Society of Digital Information and Wireless Communications, 2014 (ISSN: 2220-9085)

Performance Evaluation of Message Passing vs. Multithreading Parallel Programming Paradigms on Multi-core Systems

Hadi Khalilieh[1], Nidal Kafri[2] and Rezek Mohammad[3]

[1,2]Department of Computer Science, Al-Quds University, Jerusalem, Palestine

[3]Palestinian Technical University/Khadoorie, Middle East Technical University/physics department

[1]hkhalilia1@science.alquds.edu,

[2]nkafri@science.alquds.edu

[3]esteteh@hotmail.com

ABSTRACT

Present and future multi-core computational system architecture attracts researchers to utilize this architecture as an adequate and inexpensive solution to achieve high performance computation for many problems. The multi-core architecture enables us to implement shared memory and/or message passing parallel processing paradigms. Therefore, we need appropriate standard libraries in order to utilize the resources of this architecture efficiently and effectively. In this work, we evaluate the performance of message passing using two versions of the well-known message passing interface (MPI) library: MPICH1 vs. MPICH2. Furthermore, we compared the performance of shared memory using OpenMP that supports multithreading with MPI. The results show that the performance when MPICH2 is used is better than MPICH1. The results indicate that multithreading performs better than message passing.

KEYWORDS

Parallel Processing, Performance Evaluation, Message Passing, MPICH1, MPICH2, Multithreading, Multi-core systems, WIEN2K.

1 INTRODUCTION

In order to achieve high performance computing (i.e. reducing computing elapsed time), parallel processing is widely used in multimedia computing, signal processing, scientific computing, engineering, general purpose application, industry, computer systems, statistical applications, and simulation. Usually, mainframes and super computers are used to implement shared memory parallel computing, while clusters and grid computing are utilized to speed up the computation using message passing. Thus, parallel processing was carried out on expensive supercomputers and mainframes. After that, the emerging high performance computer network and protocols attracted the researcher to use message passing on distributed memory to implement parallel processing on clusters of on shelf computers and grid computing.

Obviously, parallel processing is implemented on shared memory computer architectures using Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), Single Program Multiple Data (SPMD) Techniques, or multithreading. Whilst message passing paradigm can be used on distributed memory architectures by means of SPMD and MIMD, a hybrid approach

using both paradigms can also be implemented on both architectures.

However, the emerging and promising multi-core computer architecture attracts the researchers to utilize this architecture as an adequate and inexpensive solution to gain high performance computation for many problems. Therefore, this architecture shifted the interest of many researchers towered parallel computing on such multi-core systems. Thus, we can achieve relatively cheap high performance using message passing, share memory, or hybrid techniques on a single or cluster of multi-core computers[2][3]. This architecture enables us to implement both shared memory and/or message passing parallel processing paradigms. Therefore, we need to evaluate which paradigm can be used more efficiently and effectively on multi-core architectures. Furthermore, to carry out our computations, we need appropriate standard libraries in order to utilize the resources efficiently for a given computational problem. Hence, to facilitate realization of parallel programming on different platforms, there are several supporting libraries. For example, we can use PVM, JPVM and MPI for message passing on distributed memory. Posix and OpenMP are also used for multithreading on shared memory [3]. It should be noted that these libraries provide us with a well defined standard interface to achieve portability and flexibility of usage. However, the developers of these libraries intend to improve the implementation to cope with the emerging platforms to increase the utilization efficiency.

In this work, we focus on evaluation of the performance of parallel computing using message passing (multi-processes) and shared memory (multiprocessing) on multi-core systems. We used different versions of MPI library namely MPICH1 and MPICH2 for message passing and OpenMP for multithreading in our experiments.

Since, one of the important applications that is needed to speed up computation is the WIEN2K application, which is based on Density Functional Theory (DFT), we used it as a benchmark to evaluate the performance of MPICH1 vs. MPICH2. The WIEN2K application enables us to simulate physical and chemical systems which form new materials. This is necessary for laboratory researchers who can produce desired materials such as drugs and medicine [8]. The WIEN2K applied a parallel method to solve quantum mechanics equations based DFT to find the cohesive energy of any material. It should be noted that the current official version of this application uses MPICH1. In addition, we used a matrix multiplication benchmark to evaluate the performance of multi-processes (message passing) vs. multithreading parallel programming performance and efficiency on a multi-core system.

In this work we evaluated the performance of MPICH1 and MPICH2 by running WIEN2K that originally used MPICH1 and the new implementation of WIEN2K on MPICH2. Moreover, we implemented a matrix multiplication on both MPICH1 and MPICH2 message passing and OpenMP for testing multithreading technique.

The paper is organized as follows: section 2 introduces a background and literature review**.** Next, section 3 discusses the experiment and the results. Finally, section 4 concludes this work and introduces future work.

2 BACKGROUND & LITERATURE REVIEW

Multi-core systems and clusters become an interesting and affordable platform for running parallel processing to achieve high performance computing for many applications and experiments. Some examples include internet services, databases, scientific computing, and simulation. This is due to their scalability performance/cost ratio [1].

There are two main approaches that support parallel computing via multi-core processors: shared memory and distributed memory approaches. Thus, we will provide an overview of the evolution of the two main approaches.

2.1 Shared Memory Approach

Shared memory based parallel programming models communicate by sharing the data objects in the global address space. Shared memory models assume that all parallel activities can access all of memory. Consistency in the data need to be achieved when different processors communicate and share the same data item, this is done by using the cache coherence protocols used by the parallel computer. All operations such as load and store for data carried out by the automatically without direct intervention by the programmer. For shared memory based parallel programming models, communication between parallel activities is completed via a shared mutable state that must be carefully managed to ensure correctness. Various synchronization primitives such as locks or transactional memory are used to enforce this management [3]. In this approach a main memory is shared between all processing elements in a single address space.
The advantages with using shared memory based parallel programming models are presented below.

- Shared memory based parallel programming models facilitate easy development of the application more than distributed memory based multiprocessors.
- Shared memory based parallel programming models avoid the multiplicity of data items and allows the programmer to not be concerned about the programming model's responsibility.
- Shared memory based programming models offer better performance than the distributed memory based parallel programming models.

The disadvantages with using the shared memory based parallel programming models are described below.

- The hardware requirements for the shared memory based parallel programming models are very high, complex, and cost prohibitive.
- Shared memory parallel programming models often encounter data races and deadlocks during the development of the applications.

A diverse range of shared memory based parallel programming models are developed to this day. They can be classified into mainly three types as: threading, directive based, and tasking models [16, 17]. However, we will only focus on the threading model.

Threading models

These models are based on the thread library that provides low level library routines for parallelizing the application. These models use mutual exclusion locks and conditional variables for establishing communications and synchronizations between threads. Some of the well known libraies are OpenMP and Posix. The advantages with threading models are as follows:

- More suitable for applications based on the multiplicity of data.
- Flexibility provided to the programmer is very high.
- Threading libraries are widely used and threading model tools are readily available.
- Performance can still be improved by using conditional waits and try locks.
- Easy to develop parallel routines for threading models

The disadvantages associated with threading models include the following:

- Hard to write applications using threading models because establishing a communication or synchronization incurs code overhead which is hard to manage, thereby leaving more scope for errors.
- The developer should be more careful in using global data otherwise this leads to data races, deadlocks, and false sharing.
- Threading models stand at low level of abstraction, which isn‘t required for a better programming model.

2.2 Distributed Memory Approach

This type of parallel programming approach allows communication between processors by using the send/receive communication routines**. Message passing models** avoids communications

between processors based on shared/global data [16]. They are typically used to program clusters, wherein each processor in the architecture gets its own instance of data and instructions. The advantages of distributed memory based programming models as follows:

- The hardware requirement for the message passing models is low, less complex, and comes at very low cost.
- The message passing models avoids the data races and as a consequence the programmer is freed from using the locks.

The disadvantages with distributed memory based parallel programming model are listed below:

- Message passing models in contrast encounter deadlocks during the process of communications.
- Development of applications on message passing models is hard and takes more time.
- The developer is responsible for establishing communication between processors.
- Message passing models are less performance oriented and incur high communication overheads.

A comparison base characteristic using methods between shared vs. distributed is listed in Table 1 [17]. The message passing interface (MPI) is a set of API functions that facilitate parallel programming based on message passing paradigm. One of the well-known APIs is MPICH1 which is based on an MPI standard founded on April 29-30, 1992 at a work shop in Williamsburg, Virginia [4]. This library API supports FORTRAN and C programming languages. It has been issued with several modifications and extensions to support dynamic processes, one-sided communication, parallel I/O, etc [13][14]. MPICH2 standard is intended for use by all those who want to write portable message-passing programs in Fortran 77, FORTRAN 95, C and C++ [5]. The improvement of MPICH2 focused on many issues and functionalities such as dynamic processes, one-sided communication, parallel I/O, etc. [13][14].

Table 1: A Comparison between Shared vs. distributed

Architecture	Distribu-ted Memory MPI	Shared Memory Arch OpenMP	Hybrid Dist. & Shared Memory
Creation mathematical model	Easy	Slightly complic-ated	Difficult
Balancing	Change-able with difficulties	Change-able easily	Easily changeab-le
Simulation of parallel models	Advisab-le	Conveni-ent	Useful
Synchronization models	Simple	Complic-ated	Complica-ted
Transfer dates between models	Large	Little	Intermedi-ate
Power of large modules	Reasona-ble	Big	Big

Of course, a number of changes to dynamic spawning tasks, the nature of communication, and how one runs them will be different. By adding new features in MPICH2, it will be more robust, efficient, and convenient to use [4]. Consequently, we will focus on the improvements in MPICH2 that we believe they have an impact on the performance:

1. MPICH1 focused mainly on point-to-point communications, but MPICH2 included a number of collective communication routines and was thread-safe [4].
2. MPICH2 supports dynamic spawning of tasks. It provides primitives to spawn processes during the execution and enables them to communicate together [11].
3. MPICH2 supports one-sided communication. It provides three communication calls: MPI_PUT (remote write), MPI_GET (remote read), and MPI_ACCUMULATE (remote

update). These operations are non-blocking [12] [14].

4. MPICH2 used generalized requests that aren't used by MPICH1**.** These requests allow users to create new non-blocking operations with an interface [14].
5. In MPICH2, significant optimizations required for efficiency (e.g. asynchronous I/O, grouping, collective buffering, and disk-directed I/O) are achieved by the parallel I/O system [14].
6. MPICH-1 defined collective communication for intra-communicators and two routines for creating new intercommunicators. But **MPICH-2** introduces extensions of many of the MPICH-1 collective routines to intercommunicators, additional routines for creating intercommunicators, and two new collective routines: a generalized all-to-all and an exclusive scan [14].
7. **MPICH2** supports MPI THREAD MULTIPLE by using a simple communication device, known as "ch3 device" (the third version of the "channel" interface), but MPICH1 doesn't support MPI THREAD MULTIPLE [5].
8. **MPICH1** is not concerned with communication, but rather process management. But **MPICH2** is concerned with communication rather than process management. However, MPICH2 provides a separation of process management and communication. The default runtime environment consists of a set of daemons, called mpd's, that establish communication among the machines to be used before application process startup, thus providing a clearer picture of what is wrong when communication cannot be established. In addition, it provides a fast and scalable startup mechanism when parallel jobs are started. But MPICH1 doesn't separate them and mpd's are built in [15].
9. **MPICH1** required access to command line arguments in all application programs before startup, including FORTRAN ones. Thus, MPICH1's configuration devotes some effort to finding the libraries, such as libraries that contained the right versions of iargc and getarg. But **MPICH2** does not require access to command line arguments of applications before startup and MPICH2 does nothing special for configuration. If one needs them in their applications, they must ensure that they are available in the environment being used [15].

Various operating systems such as Linux, Solaris, and Windows can be used for scheduling computer resources such as memory, I/O, and CPU [6].

2.3 Cohesive Energy & WIEN2K

Condense matter physics looks different than 50 years ago. Scientist know that solids obey the laws of quantum mechanics, by solving these quantum equations all properties of solids including electrical, magnetic, optical and thermal can be found. The main scalable quantity for measuring the stability of any material is the cohesive energy; cohesive energy equals the difference between the total energy of the material in the combined form and the sum of the free atom's energy in their free state as shown in equation (1)

$$\boldsymbol{E}_{\text{cohesive energy}} = \boldsymbol{E}_{\text{compound}} - \sum \boldsymbol{E}_{\text{free atoms}} \quad (1)$$

Each stable form of these atoms can produce positive value for the cohesive energy. Furthermore, the material can normally take more than one stable state, and the state with the highest cohesive energy is the most stable one [10].

In order to study the previous characteristics of the materials we have to solve many second body order differential equation called equation of state. This equation obeys the laws of quantum mechanics. The equation of state is composed of the kinetic energy operators for both the nucleus and electrons, the potential energy resulting from interaction between electrons themselves, nucleis themselves, and nucleis and electrons; these operators are measured by solving many-body Hamiltonian for the system, which is illustrated in equation (2) [7][10].

This equation can be solved numerically after transforming it to a one body problem after some approximations. This method called Density Functional Theory (DFT) [8][9].

$$H\Psi = E\,\Psi$$

$$\hat{H} = -\frac{h^2}{2}\sum_i \frac{\nabla^2_{\vec{R}}}{M_i} - \frac{h^2}{2}\sum_i \frac{\nabla^2_{\vec{r}}}{m_e} - \frac{1}{4\pi\epsilon_0}\sum_{i,j}\frac{e^2 Z_i}{\left|\vec{R}_i - \vec{r}_j\right|} - \frac{1}{8\pi\epsilon_0}\sum_{i\neq j}\frac{e^2}{\left|\vec{r}_i - \vec{r}_j\right|} + \frac{1}{8\pi\epsilon_0}\sum_{i\neq j}\frac{e^2 Z_i Z_j}{\left|\vec{R}_i - \vec{R}_j\right|} \quad (2)$$

Program packages like WIEN2K [3], using Full potential Linear Augmented Plane Wave and Local Orbital's (FP-LAPW+Lo) technique allows such studies on the basis of quantum mechanics using density functional theory (DFT). In these studies, we have two main factors controlling the calculation. The first factor is the time of calculation and the second is the sample actuality; the sample actuality meaning the number of atoms constituting the sample, the bigger the number is the more actual case we have, and more complexity, which costs a lot of calculation time.

WIEN2K package is composed of these five modules: **LAPW0, LAPW1, LAPW2, LCORE** and **MIXER.** Each module solves one equation to get the highest cohesive energy. The state with the highest cohesive energy is the most stable one [10]. The calculation is repeated until it obtains the highest cohesive energy.

The authors in [8] compared two parallel approaches that run on MPICH1 channel. The two methods are: distributed k-point and data distribution. However, the first one runs each of the two modules (LAPW1, LAPW2) in parallel way. The other runs each of the first three modules in parallel. In addition, a comparison between serial and parallel approaches for running Matrix Multiplication on MPICH1 was in [1].

3 EXPERIMENT AND RESULTS DISCUSSION

In this work two cases of experiments were carried out. In the first case (Case 1), we focused on distributing tasks of WIEN2K program using MPICH1 and MPICH2 on multi-core machine. Whereas in [8] the experiments were carried out on a cluster using MPICH1 to distribute WIEN2K task. In the second case (Case 2) of experiments, we tested the performance of parallel matrix multiplication using multi-processing (message passing) using MPICH1 and MPICH2, and multithreading paradigms using OpenMP.

Our experiments were running on Linux (Fedora 14) installed on a multi-core (quad) machine (Intel Core i5 3GHz processor); the specification details of the experiments platform/machine are listed in Table 2.

Table 2: Machine Specifications

No	Specification	Multi-Core PC
1	CPU speed	Quad 3 GHz
2	RAM size	8 GB
3	Cache	8 Mbyte
4	HD speed	7200 RPM

To accomplish the calculations, a set of programs were installed on Fedora Linux version 14 and optimized with appropriate options together with WIEN2K. These programs are listed in Table 3.

Recall that we continue the work of [8], where they installed and used MPICH1 to run WIEN2K program. For this work we installed MPICH2 channel then installed WIEN2K MPICH2 version and run "LAPW0," which is a basic module of WIEN2K. This is done via determined parallel commands. These commands were written on the terminal of the operating system.

The experiments were carried out by running the programs LAPW0 as benchmarks using MPICH1 MPICH2 on one, two, three, and four processors

of the quad multi-core machine, where, each processor has a unique id from 0 to 3. Each experiment was repeated several times and the average of the elapsed time was recorded. The experiments were divided into two cases: the first one ran LAPW0 for one cycle. In the second experiment (Case 2), the matrix multiplication was implemented using MPICH1, MPICH2, and OpenMP.

Table 3: Software Requirements

Program name	Version	Source
WIEN2K	13.1	www.WIEN2K.at
MPI Channel	MPICH1.3 & MPICH2-1.0.5p3	www.mpich.org
Intel Fortran 90 Compiler	11.072	Intel
Intel C Compiler	10.074	Intel
Mathematical Kernel Library (MKL)	11.0	Intel
Fastest Fourier Transform in the west (FFTW)	FFTW-2.1.5	Intel

Case 1:

The experiments on MPICH1 used "mpirun" command and "mpiexec" for MPICH2. For example, the steps of the LAPW0 execution on MPICH2 are shown in Figure (1).

The results of the average running time for case 1 (LAPW0) are summarized in Table 4. This table shows the execution time on MPICH1 and MPICH2 and the improvement factor (*if*) by the number of processors. The improvement factor (*if*) is measured as the ratio of the difference between the execution time on MPICH1 and MPICH2 to the Execution time on MPICH1 i.e. $(T_{MPICH1}-T_{MPICH2})/T_{MPICH1}$.

$$if = \frac{T_{MPICH1} - T_{MPICH2}}{T_{MPICH1}}$$

```
[rezek@rezek-dell15~]$ cd/home/ rezek
/mpich2 /examples
[rezek@rezek-dell15 examples]$ mpicc -c
lapw0_mpi.c
[rezek@rezek-dell15 examples]$ mpicc -o
lapw0_mpi lapw0_mpi.o
[rezek@rezek-dell15 examples]$ mpd &
[1] 3929
[rezek@rezek-dell15 examples]$ mpiexec -
n 1 lapw0_mpi
lapw0_mpi has started with 1 tasks.
Initializing arrays...
Running Time = 62.005132
Done.

[rezek@rezek-dell15 examples]$ mpiexec -
n 2 lapw0_mpi
lapw0_mpi has started with 2 tasks.
Initializing arrays...
Running Time = 34.002134
Done.

rezek@rezek-dell15 examples]$ mpiexec -n
3 lapw0_mpi
lapw0_mpi has started with 3 tasks.
Initializing arrays...
Running Time = 25.141348
Done.
```

Figure 1 : Screen Shot of Running LAPW0 on MPICH2

Table 4: Execution Time of LAPW0 on MPICH1 and MPICH2 on Different # of Processors.

# of Proc	*Exec. time on mpich1 (min)*	*Exec. time on mpich2 (min)*	*If*
1	64.25	62.54	0.026615
2	35.05	34.38	0.019116
3	26.03	25.37	0.025355
4	20.5	19.52	0.047805

It is clear that the performance of MPICH2 is better than MPICH1 by approximately 3%. Also, Figure 2 shows the difference between the execution time on MPICH1 and MPICH2. Therefore, we believe that the nine added features have positive impact on the performance. The most important added features in MPICH2 are the

collective communications, the support of one sided communication, MPI Thread Multiple, and its concern on communication rather than process management. It should be noted that the time unit in the experiments of case 1 is in minutes, whereas it is in seconds in case 2.

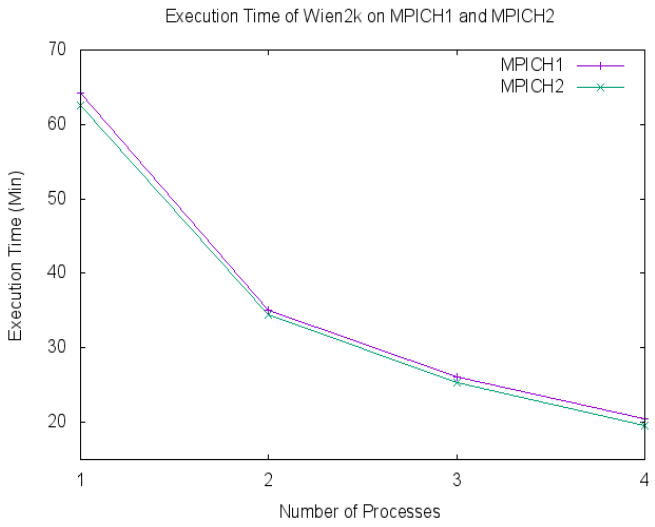

Figure 2: the WIEN2K execution time of MPICH1 vs. MPICH2.

Case 2:

In this case the experiments were implemented on a standard parallel matrix multiplication of size 5120x5120 using multithreading by means of OpenMP and multi-processing (message passing) using MPICH1 and MPICH2. Also, in these experiments we utilized 1, 2, 4, 8 and 16 processes. The experiments where repeated by using multithreading with 1, 2, 4, 8, and 16 threads. The results in Figure 3 show that the performance using multithreading is better than multiprocessing. This is because of the overhead processes and data distribution.

Recall that the experiment's platform has four processing elements. It is apparent from Figure 3 that the curve declines (i.e. improving the efficiency and speed-up) until the number of processes/threads reaches 4. Afterwards, the curve begins to incline, which indicates a decrease in performance and efficiency. This is due to the overheads in scheduling the threads and processes in utilizing shared resources (i.e. processing elements and shared memories).

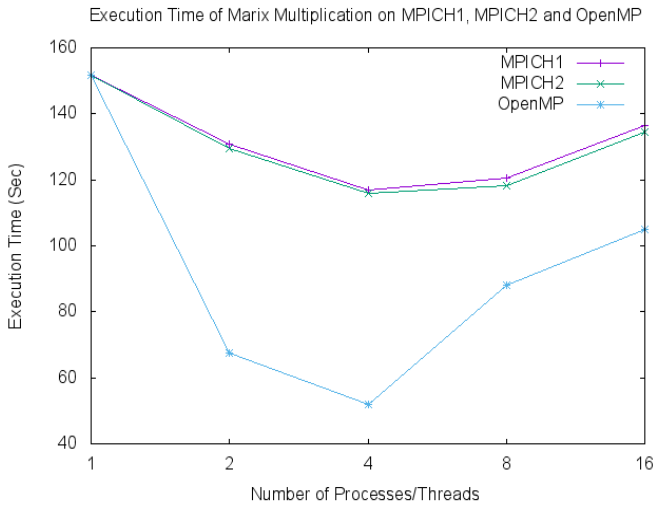

Fig 3: Execution Time of Matrix Multiplication Using MPICH1 vs. MPICH2 vs. OpenMP

4 CONCLUSION AND FUTURE WORKS

The goal of this work is twofold. The first is to evaluate and compare the performance of MPICH1 and MPICH2 using different cases running on one, two, three, and four processors. The second aim is to evaluate the performance of running parallel programs with big data using message passing and multithreading. As a result we can conclude that MPICH2 perform better than MPICH1 in all cases. It is due to the collective improvement and added features in MPICH2. Moreover, the results show that multithreading programming on multi-core architectures perform better than message passing when the parallel programs works on big data.

Finally, for future work, we intend to extend our experiment to test the performance of newly issued MPICH3 and Graphical Processing Units (9999999GPU) using different tasks.

5 REFERENCES:

1. Sherihan Abu El-Enin, Mohamed Abu El-Soud,” Evaluation of Matrix Multiplication on an MPI Cluster” Faculty of computers and Information, Mansoura University, Egypt. 2011.

2. Dami´an A. Mall´on, Guillermo L. Taboada, Carlos Teijeiro, Juan Touri˜no, Basilio B. Fraguela, Andr´es G´omez1, Ram´on Doallo, and J. Carlos Mouri˜no1,*" Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures".*Galicia Supercomputing Center (CESGA), Santiago de Compostela, Spain. Computer Architecture Group, University of A Coru˜na, A Coru˜na, Spain. 2009.
3. David Culler. Jaswinder Pal Singh, Anoop Gupta.*" Parallel Computer Architecture A Hardware / Software Approach ".* University of California, Berkeley, Princeton University, Stanford University, Aug 28, 1997, Pages 40 -127.
4. *"*MPI: A Message-Passing Interface Standard, Message Passing Interface Forum*".* ARPA and NSF under grant ASC-9310330, the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, by the Commission of the European Community through Esprit project P6643. Nov 15, 2003.
5. *"*MPI: A Message-Passing Interface Standard, Version 2.1, and Message Passing Interface Forum*".* June 23, 2008.
6. EDOUARD BUGNION, SCOTT DEVINE, KINSHUK GOVIL, and MENDEL ROSENBLUM, "*Disco: Running Commodity Operating Systems on Scalable Multiprocessors*", Stanford University, November 1997, Vol. 15, No. 4, Pages 412–447.
7. S. Cottenier, "Density Functional Theorythe Family of (L)APW-methods: a step-by-step introduction", August 6, 2004, ISBN 90-807215-1-4.
8. Rezek Mohammad, Areej Jabir, and Rashid Jayousi, *"Optimum Execution For WIEN2K using Parallel Programming Models (Comparison Study)"*. Department of physics, Palestinian Technical University/Khadoorie, Middle East Technical University, and department of Computer Science, Al-Quds University, Jerusalem, Palestine. 2011
9. Schrodinger¸ E. *"An Adulatory Theory of the Mechanics of Atoms and Molecules"*. Physical Review 28 (26): 1049-1070. 1926.
10. Hellmann¸ Hans, *"A new Approximation Method in the Problem of Many Electrons"*. Journal of Chemical Physics (Karpow-Institute for Physical Chemistry,Moscow), 1935.
11. M´arcia C. Cera1, Guilherme P. Pezzi, Maur´ıcio L. Pilla, Nicolas B. Maillard1, and Philippe O. A. Navaux, , *"Scheduling Dynamically Spawned Processes in MPI-2"*. Universidade Federal do Rio Grande do Sul, Porto Alegre Brazil and Universidade Cat´olica de Pelotas, Pelotas, Brazil).
12. C.M. Maynard, *"Comparing One-Sided Communication with MPI, UPC and SHMEM"*. EPCC, School of Physics and Astronomy, University of Edinburgh, JCMB, Kings Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK.
13. Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, William Gropp and Rajeev Thakur, *"High Performance MPI-2 One-Sided Communication over InfiniBand"*. Computer and Information Science The Ohio State University Columbus, OH 43210 Mathematics and Computer Science Division Argonne National Laboratory Argonne, IL 60439.
14. *"*MPI: A Message-Passing Interface Standard, Version 2.2, and Message Passing Interface Forum*"*. **Sept 4, 2009**
15. William Gropp, Ewing Lusk, David Ashton, Pavan Balaji, Darius Buntinas, Ralph Butler, Anthony Chan, Jayesh Krishna, Guillaume Mercier, Rob Ross, Rajeev Thakur, and Brian Toonen,*" MPICH2 User's Guide,* Version 1.0.6, Mathematics and Computer Science Division Argonne National Laboratory*"*. **September 14, 2007**
16. Srikar Chowdary Ravela, *"Comparison of Shared memory based parallel programming models"*. School of Computing Blekinge Institute of Technology Box 520 SE – 372 25 Ronneby Sweden, **2010.**
17. Kvasnica, P., Páleník, T "*Simulation in Flight Simulator with the Hybrid Distributed-Shared Memory Architecture*" In: ASIS 2009, s. 19 – 24. ISBN 978-80-86840-47-5. **2009**