Deanship of Graduate Studies
Al-Quds University

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

Hadi Mahmoud Khalilia

M.Sc. Thesis

Jerusalem – Palestine

1435/ 2014

Deanship of Graduate Studies
Al-Quds University

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

Prepared By

Hadi Mahmoud Khalilia

Supervisor: **Dr.Nidal Kafri**

Co-Supervisor: **Dr.Rezek Mohammad**

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Computer Science at Al-Quds University

1435/ 2014

Al-Quds University
Deanship of Graduate Studies
Master in Computer Science
Computer Science & Information Technology

Thesis Approval

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

Prepared By: Hadi Mahmoud Khalilia
Registration No: 21011433

Supervisor: Dr.Nidal Kafri

Master thesis submitted and accepted, Date: / / 2015
The names and signatures of the examining committee members are as follows:

1- Head of Committee: Dr.Jehad Najjar	Signature …………...
2- Internal Examiner:	Signature …………...
3- External Examiner:	Signature …………...
4- Committee Member: Dr.Nidal Kafri	Signature …………...
5- Co-Supervisor: Dr.Rezek Mohammad	Signature …………...

Jerusalem – Palestine

1435 / 2014

Dedication

To my beloved Country Palestine

To My Parents and family

To My wife and sons (Zaid and Mahmoud)

Declaration

I certify that this thesis submitted for the degree of Master, is the result of my own research, except where otherwise acknowledged, and that this study (or any part of the same) has not been submitted for a higher degree to any other university or institution.

Signed……………………………

Hadi Mahmoud Yousef Khalilia

Date:……………………………..

Acknowledgments

For his support and guidance, I would like to thank my supervisors, Dr. Nidal Kafri and Dr.Rezek Mohammad. Without their meticulous effort and support, this thesis would not have been the same.

I would also very much like to express my gratitude to all people in the Computer Science department for their endless help during study.

In addition, I would like to dedicate special and great thanks to my parents, brothers, sisters, wife, lovely son Zaid and friends for their inspiration and support throughout my academic life.

Abstract

Present and future multi-core computational system architecture attracts researchers as an adequate and inexpensive solution to achieve high performance computation for many problems. The multi-core architecture enables implementation of shared memory and/or message passing parallel processing paradigms. Therefore, there is a great need for standard libraries in order to utilize the resources 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 added features (total 9) impacted the MPICH2 results over MPICH1. On the other hand, the overheads of message passing and large data communication impact negatively on the performance of this paradigm against multithreading paradigm.

دراسة مقارنة الأداء والكفاءة بين تقنيات الخوارزميات المتوازية: تمرير الرسائل ومتعدد الخيوط على جهاز حاسوب متعدد المعالجات

إعداد : هادي محمود يوسف خليلية

إشراف : د.نضال الكفري

ملخص:

ان التطور في هيكلية انظمة الحاسوب متعددة المعالجات الحالية والمستقبلية تشكل عامل جذب وتحول نحو تكثيف استخدام المعالجة المتوازية في التطبيقات على اختلاف انواعها للوصول الى اداء افضل حيث يمكن تقسيم العمليات والبيانات الكبيرة الى عمليات/ومعالجات اصغر وتقسم البيانات على هذه المعالجات/العمليات. لذا يشهد البحث العلمي نشاطا مكثفا في البحث عن افضل الطرق لزيادة فعالية استخدام هذه البنية الحاسوبية بافضل ما يمكن. حيث يمكن استخدام النماذج والطرق الاساسية المتبعة في المعالجة المتوازية والتي تساعد في توفير التواصل والتفاعل بين العمليات المختلفة: المعالجة المتوازية بوجود الذاكرة المشتركة (Shared memory parallel processing paradigm) وكذلك باستخدام تبادل الرسائل والبيانات بين هذه العمليات (Message passing interface MPI) أو كلاهما معا (Hybrid approach). لذا توفر مراكز الابحاث والمؤسسات مكتبات برمجية لتسهيل تطوير برمجيات المعالجة المتوازية. ولتقييم الطرق المقترحة في تجارب الابحاث لاستخدام هذه البنية وطبيعة التطبيقات البرمجية لابد من استخدام برمجيات وبيانات قياسية معرفة في مجال البحث العلمي.

هناك العديد من المشاكل اليومية التي تم حلها باستخدام المعالجة المتوازية منها الحزمة الفيزيائية (WEIN2K)، هذه الحزمة تحتوي على بعض المشاكل منها استغراق الوقت الكبير في التشغيل، وهذا ناتج من أن الحزمة لازالت تستخدم القناة القديمة (MPICH1) في تبادل الرسائل للمعالجة المتوازية.

في هذا العمل البحثي قمنا بتمثيل الحزمة الفيزيائية على القناة الجديدة (MPICH2) لتبادل الرسائل، ومن ثم قمنا بتقييم أداء اصدارين متتاليان للحزمة البرمجية التي تدعم المعالجة المتوازية بطريقة تبادل الرسائل (MPI) بين العمليات وهي

بالتحديد (MPICH1) و (MPICH2). حيث قمنا بتنفيذ الحزمة (WIEN2K) المعروفة في أبحاث علم الفيزياء والكيمياء كحالة دراسية. وهي حزمة مخصصة لدراسة الخصائص الفيزيائية والكيميائية للمواد بالإعتماد على نظرية الكثافة الوظيفية الفيزيائية بواسطة المحاكاة كتطبيق قياسي لتجاربنا باستخدام (MPICH1) و (MPICH2).

وفي هذا البحث تم كذلك تقييم أداء (Multithreading) المتوفرة في حزمة (OpenMP) إعتماداً على خاصية الذاكرة المشتركة في هذه الهيكلية و (Multiprocesses) باستخدام تبادل الرسائل بين العمليات في تطبيقين مختلفين من حيث تبادل البيانات وحجمها. لذا قمنا بتنفيذ خوارزمية ضرب مصفوفتين كتطبيق فيه حجم تبادل البيانات كبير واخر وهو حساب الثابت الحسابي (π) وهي النسبة التقريبية لمحيط الدائرة إلى نصف قطرها. حيث إن تبادل البيانات يكاد يكون صغيراً جداً.

النتائج أظهرت أن آداء الإصدار الثاني من مكتبة واجهة تمرير الرسائل (MPICH2) أفضل من آداء الإصدار الأول (MPICH1)، وتعليل ذلك يرجع الى التحسينات/المميزات الإضافية التسعة التي تم إضافتها على (MPICH2). كما أظهرت النتائج أن آداء (Multithreading) في التطبيقات التي تتبادل فيها العمليات بيانات كبيرة يكون أفضل من (Message Passing) والعكس صحيح، وهذا يرجع إلى عمليات ذات حمولة كبيرة وحجم الرسائل الكبير.

Table of Contents

List of Tables

List of Figures

List of Appendices

List of Abbreviations

Abbreviation	Full Name
M.Sc.	Master Degree
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instruction Multiple Data
SPMD	Single Program Multiple Data
MPI	Message Passing Interface
MPICH1	Message Passing Interface Channel One
MPICH2	Message Passing Interface Channel Two
MPICH3	Message Passing Interface Channel Three
DFT	Density Functional Theory
FLOP	Floating-Point Operation
ILP	Instruction-Level Parallelism
VLIW	Very Long IinstructionWord
MMT	Matrix Multiplication
CISC	Complex Instruction Set Computer
RISC	Reduced Instruction Set Computer
LAN	Local Area Network
WAN	Wide Area Network
GPUs	Graphics Processing Units
UMA	Uniform Memory Access
NUMA	Non-Uniform Memory Access
DMA	Direct Memory Access
FIFO	First In First Out
ISA	Instruction Set Architecture
SMT	Simultaneous Multithreading
CMP	Chip Multiprocessor
HTT	Hyper Threading Technologies
QPI	Intel’s QuickPath Interconnect
Rsh	Remote Shell
Mpd	Message Passing Daemon
mpiexe	The mpiexec command
mpirun	The mpirun command
RMA	Remote Memory Access
PP	Pseudo potential method
TB	Tight binding method
BZ	Brillouin Zone
SC	Self-Consistent
LAPW	Linearized Augmented Plane Wave
HPC	High Performance Computing
ES-MPICH2	A Message Passing Interface with Enhanced Security
MBR	Mapped Block Row
MKL	Mathematical Kernel Library
FFTW	Fastest Fourier Transform in the West
IF	Improvement Factor
CMOS	Complementary Metal Oxide Semiconductor

Chapter One

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 [7]. 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 [25], [42].

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 and applications. 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, multithreading on shared 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 [15].

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 need 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 that form new materials. This is necessary for laboratory researchers who can produce desired materials such as drugs and medicine [8], [30]. 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 and it takes a lot of time to return the results of forming new material (around 30 days); so these results form a big problem. 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.

Based on the high efficiency of MPICH2 over MPICH1, in this work we implemented the WIEN2K package on MPICH2 and evaluated the performance of MPICH1 and MPICH2 by running the package that originally used MPICH1 and our new implementation of WIEN2K on MPICH2. Results show that MPICH2 increases the speed up of WIEN2K execution on each multicore by 3% which indicates decreasing one day of 30 work days to simulate producuction of new material. We believe that this improvement in performance is due to the added features to MPICH2. Some of these features are: dynamic spawning of tasks in LAPW (i.e., LAPW0, LAPW1 and LAPW2), different collective communication routines in LAPW, a number of one-sided and non-blocking routines in LAPW, and LCORE, And multiple threads in MIXER module.

Moreover, in order to evaluate and compare the performance of multithreding utilzation the shared memory property with multiprocessor using message passing techniques on multi-core architechture, we implemented a matrix multiplication (MMT) and the mathematical constat π (the ratio of a circle's circumference to its diameter) on both MPICH1 and MPICH2 message passing and OpenMP for testing multithreading technique.

In case of MMT (for largr size matrices), the results show multithreading execution time is lower than multiprocessing time. This is because of the processes schedulaing and large size of data chunks communication overheads. Nevertheless, in the second case (π) and MMT (for small size matrices), the results show that MPICH2 performes better than multithreading because of the small size of data chunks and the following features: Collective communication routines on master computer, a number of non-blocking routines on each client, and multiple threads on the master.

The thesis is organized as follows: Next chapter provides a background; chapter 3 introduces a literature review**.** In Chapter 4, we introduce our work. Chapter 5 explaines the experiment and discusses the results. Finally, Chapter 6 concludes this work and introduces future work.

Chapter Two

Background

In this chapter we present a background relevant to our work. Thus, we introduce parallel computing classification infrastructure by means of hardware and software supporting libraries such as MPICH1, MPICH2 and multithreading. Also, we will introduce/explain Density Functional Theory and WIEN2K package as benchmark for our experiments on MPICH1 vs. MPICH2 for comparison. Since this is one of the scientific problems (physical computation) that need high performance computing.

A. Parallel Approaches

In the last decade, a significant growth was achieved in performance and capability of computer systems. Applications need computers with high requirements for computing exploited this important event. Example applications include transaction processing, computer games and graphics, weather simulation, heat transfer, ray tracing and many others [7]. However, the traditional logical view of a sequential computer consists of a memory connected to a processor via a datapath. All three components – processor, memory, and datapath present bottlenecks to the overall processing rate of a computer system.

Number of architectural innovations over the years have addressed these bottlenecks. One of the most important innovations is multiplicity – in processing units, datapaths, and memory units. This multiplicity is either entirely hidden from the programmer as in the case of implicit parallelism or exposed to the programmer in different forms [40]: The first is **Data parallelism** [7], [27]: in this form of parallelization data is distributed on multiple processors environment, in a multiple system executing a single set of instructions (SIMD), data parallelism is achieved when each processor performs the same task on different pieces of distributed data. Second: **Bit level:** this form based on increasing processor work size. This will reduce the number of instructions the processor must execute [7], [29]. Third: **Instruction-Level Parallelism (ILP)**: ILP used in very long instruction word (VLIW) processors relies on the compiler to resolve dependencies and resource availability at compile time [24].

The previous styles depend on several parallel algorithm models such as *Data model, task model, work pool model etc.* The *data-parallel model* is one of the simplest algorithm models. In this model, the data is statically or semi-statically mapped on to processes and each processor performs similar operations on different data. In it, the decomposition of computations is done in two steps. In the first step, the data on which the computations are performed are partitioned, and in the second step, this data partitioning is used to induce a partitioning of the computations into tasks. The operations that these tasks perform on different data partitions are usually similar (*e.g.*, matrix multiplication) [7], [9]. But, we can calculate the PI (π) value using the task model which isn't need to decompose data because it depends on tasks decompositions. The third model, which is work pool model that is characterized by a dynamic mapping of tasks onto processes for load balancing in which any task may potentially be performed by any processes. Parallel tree search where the work is represented by a centralized or distributed data structure is an example of the use of the work pool model where the tasks are generated dynamically [40].

However, in parallelization of the computations or operations can often be achieved in two ways: by replicating the hardware components (processor, memory and bus) or by interleaving and organizing the single processor execution between multiple tasks [27].

The main intention in using the parallel systems is to support high execution speeds. The scope of parallelization of an application comes from the identification of multiple tasks of the same kind, which is a major source of speed up achieved by the parallel computers [39].

A.1 Parallel Hardware and Software

It is necessary to know about the parallel hardware before going deep into the study. The traditional uni-processor computer is said to follow Von-Neumann architecture, which consists of a single memory, connected to processor via data paths and works on the "stored memory concept". These kinds of architectures often represent a bottleneck for sequential processing and the performance associated with them is limited. Therefore, to relieve from these bottlenecks one possible way is to use the redundancy /duplication of the hardware components, which lead us to parallelism in order to achieve high speed and efficiency in processing.

We can calculate the speed up by calculate the ratio between the serial and the parallelism of the program. The maximum possible speed up of a program such as a result of parallelization is observed as Amdahl's law [12]. It states that a small portion of the program which cannot be parallelized will limit the overall speed up available from parallelization. A program that solves a large mathematical or engineering problem will typically consist of several parallelizable parts and several sequential parts. If α is the fraction of running time a sequential program spends on non-parallelizable parts, then:

$$S \leq \frac{1}{\alpha} \tag{2.1}$$

S: is the maximum speed up with parallelization of the program.

Efficiency is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to p and efficiency is equal to one. In practice, speedup is less than p and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. We denote efficiency by the symbol E. Mathematically, it is given by

$$E = \frac{S}{P} \tag{2.2}$$

If the speed up by parallel program is 3X and with four processors, we get efficiency value equals 75%.

Good speed and efficiency in parallel computing is due to replication of hardware components, thereby various types of parallel platforms that depend on duplication of hardware components designed to support the better parallel programming. The hardware used for parallel programming known as multiprocessors that introduce the classification of multi-core platforms. This classified into two types [7], [27]:

- SIMD architectures - involves multiple processors sharing the same instructions but rather executing them on multiple data.
- MIMD architectures – involves multiple processors each having its own set of instructions and data.

They are several designs and architectures that support parallelism such as RISC (Reduced Instruction Set Computer), cluster, grid, new architecture NVIDIA's GPUs (Graphics Processing Units) etc [41]. As a result, we are seeing the design that is best whose processors suited to parallel architecture become the performance leader as well.

A.2 Shared Memory and Distributed Memory Paradigms

Parallel programming models are not new and dates back to the cell processors. Several programming models have been proposed for multi-core processors. They can be classified based on the communication behaviour model used [39]. The communications can be applied on any one of these parallel architectures: the first is a shared memory architecture that shares the global address space under shared-memory multiprocessors. The multi-processors in these systems communicate with each other through global variables stored in a shared address space. They are several programming models that based on shared memory such as threading, tasking and directive models. The most important one of them is a threading model. It uses mutual exclusion locks and conditional variables for establishing communications and synchronizations between threads. This model distinguishes from others by: flexibility, more suitable for applications based on the multiplicity of data, easy to find tools related to the threading models and easy to develop parallel routines for it. Despite of threading model is the important one it includes several disadvantages such as hard to manage because of more errors can happen, the developer should be more careful in using global data otherwise this leads to data races, deadlocks and false sharing. Moreover, Threading models stand at low level of abstraction, which isn‘t required for a better programming model.

The second is a distributed memory architecture that each processor has its own memory module and the data at any time instant is private to the processors. These types of systems are constructed by interconnecting each component with a high-speed communications network. These architectures rely on the send/receive primitives for communication between multiple processors communicate to each other over the network. In addition, the distributed memory has the following advantages: low cost and a message passing models avoids the data races (no locks). But from its obstacles are: Development of applications on message passing models is hard and takes more time, the developer is responsible for establishing communication between processors and message passing models incur high communication overheads.

A comparison base characteristic using methods between shared vs. distributed is listed in Table 1 [44]. Knowing that a hybrid approach using both paradigms can also be implemented on both architectures.

In this research, we will concentrate on the ways of parallelism: message passing and shared memory approaches. We will go in details of message passing channel1 (MPICH1) and message passing channel2 (MPICH2) by using WIEN2K package with Density Functional Theory as first case study in the 1st part of the work. In addition, we will compare OpenMP and MPICH by using Matrix Multiplication and computing the mathematical constant π as a two cases study in the 2nd part.

Table 1: A Comparison between Shared vs. Distributed [44].

Architecture	**Distributed Memory MPI**	**Shared Memory Arch OpenMP**	**Hybrid Distributed & Shared Memory**
Creation mathematical model	Easy	Slightly complicated	Difficult
Balancing	Changeable with Difficulties	Changeable easily	Easily changeable
Simulation of parallel models	Advisable	Convenient	Useful
Synchronization models	Simple	Complicated	Complicated
Transfer dates between models	Large	Little	Intermediate
Power of large modules	Reasonable	Big	Big

The two parts of our research implemented and executed on a multi-core platform, which is the most common processor architectures available today and supports the two types of parallel paradigms: shared and distributed memory. Multi-core architecture implies to at least three aspects: there are multiple computational cores, there is a way by which these cores communicate and the processor cores have to communicate with the outside world.

So this platform based on several important processor architecture concepts such as (core organization, interconnects, memory architectures, support for parallel programming etc). The major vendors of multi-core are: Intel (supports the Hyper Threading Technologies (HTT) concept), IBM (which also supports thread priorities) and Oracle Sun (where as much as eight hardware threads are supported on each core). Knowing that machine specifications that we used in the experiment will be chapter 4 (Experiments and Results Analysis).

B. Message Passing Channel (MPICH)

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. The basic concept of message passing is processes communicating through messages. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm.

More recently, several systems have demonstrated that a message passing system can be efficiently and portably implemented. It is thus an appropriate time to try to know both the syntax and semantics of a core of library routines in MPI (Message Passing Interface) standards that will be useful to a wide range of users and efficiently implementable on a wide range of computers. MPI is a specification, not an implementation; there are multiple implementations of MPI. It is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C and Fortran-77 in the MPICH1 standard and which for C++ and Fortran-95 in the MPICH2.

The goal of the MPI simply stated is to develop a widely used standard for writing message-passing programs. As such, the interface should establish a practical, portable, efficient, and flexible standard for message passing. A complete list of goals follows [4], [28]:

- **Standardization** - MPI is the only message passing library which can be considered a standard.
- **Portability** - There is little or no need to modify your source code when you port your application to a different platform.
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.
- **Functionality** - There are over 440 new routines defined in MPICH2.
- **Availability** - A variety of implementations are available, both vendor and public domain.
- **Flexibility**: Define an interface, such as PVM, NX, Express, p4, etc
- **Communication Reliability**: The user need not cope with communication failures.
- **Thread-Safety:** The interface should be designed to allow for thread-safety.
- **Language Independent**: Semantics of the interface should be language independent.

All goals and basic rules in MPI applied on all versions of message passing channel releases, where each MPI channel (MPICH1 and MPICH2) has several releases as shown in next two tables (Table 2, Table 3).

Table 2: Message Passing Channel One (MPICH1) Versions [4], [18].

No	Version Name	Released Date
1	Version 1.0	May, 1994
2	Version 1.1	June, 1995
3	Version 1.2	July 18, 1997
4	Version 1.3	May 30, 2008

All MPICH1 versions focused on five areas: further corrections and clarifications, new datatype constructors and language interoperability, dynamic processes and one-sided communication, extensions to the Fortran 77 and C bindings and areas in which the MPI process and framework seem likely to be useful.

Table 3: Message Passing Channel Two (MPICH2) Versions [12], [20].

No	Version Name	Released Date
1	Version 2.0	May 20, 1998
2	Version 2.1	June 23, 2008
3	Version 2.2	September 4, 2009

All MPICH2 versions focused on extensions to the classical message-passing model. Those are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O.

Note: the major work of the current MPI Forum is the preparation and checking the stability of MPICH3 [18].

B.1 Differences between Two Channels: MPICH1 vs. MPICH2

If you have been using the latest version of MPICH2, you will find a number of things about MPICH1 that are different (and hopefully better in every case.) Your MPI application programs need not change, of course, but a number of settings and configurations about how you run them will be different.

MPICH2 is an all-new implementation of the MPI Standard, designed to implement all of the additions to MPICH1 such as (dynamic process management, one-sided operations, parallel I/O, and other extensions). If we apply the additions over MPICH1 in implementing MPICH2, we will get MPICH2 more robust, efficient, and convenient to use. So this motivates us to learn the changes between MPICH1 and MPICH2 as shown in Table 4 [4], [16].

Table 4: Different Changes that show the Differences between MPICH1 and MPICH2 [4], [16].

No	Changes	MPICH1	MPICH2
1	MPI Thread Multiple	Doesn't Support	Support
2	Configuration of MPICH	**./configure -cc=pgcc**	**./configure CC=pgcc**
3	Process Management and Communication	Process Management entagled with Communication Mechanism (Not Seperated)	Provides a Seperation of Process Management and Communication Mechanism
4	Collective Operations	Defined collective communication for intracommunicators.	Introduces extensions of the MPICH1 collective routines to intercommunicators. The two new collective routines: a generalized all-to-all and an exclusive scan.

No	Changes	MPICH1	MPICH2
5	Message Passing Daemon (mpd)	MPD is built in, so it doesn't need to start manually. **mpd**: establishes 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 and providing a fast and scalable startup mechanism when parallel jobs are started.	MPD is not built in, so it needs to start manually [13]. Some of commands that are used to daemon are" **mpd**: starts an mpd daemon. **mpdboot**: starts a set of mpd's on a list of machines. **mpdtrace**: lists all the MPD daemons that are running. **mpdlistjobs**: lists the jobs that the mpd's are running. **mpdkilljob**: kills a job specified by the name returned by mpd list jobs.
6	Starting Parallel Jobs	MPICH1 provided the **mpirun** command to start MPICH1 jobs.	MPICH2 provided the **mpiexec** command to start the jobs.
7	Command-Line Arguments	**MPICH1** required access to command line arguments in all application programs, and MPICH1's configure devoted some effort to finding the libraries that contained the right versions of iargc and getarg and including those libraries with which the mpif77 script linked MPI programs.	**MPICH2** does not require access to command line arguments to applications.
8	Arguments argc and argv	Needs to pass the arguments **argc** and **argv** by an application to MPI INIT and main functions.	Does not need to pass the arguments.
9	Error Handlers	Attached error handlers only to communicators.	Attached error handlers to three types of objects: communicators, windows and files.

No	Changes	MPICH1	MPICH2
10	Communicator Caching	**Doesn't** include functions for caching on communicators.	Includes several functions for caching on communicators.
11	Size-Specific MPI DataTypes	Optional	Required

They are several useful tools and components included in MPICH2 but not all of them included in MPICH1, these tools can be shown as in Table 5 [12].

Table 5: Different Criteria that show the differences between MPICH1 and MPICH2 [12].

No	Criteria	MPICH1	MPICH2
1	Point-to-point communication	Include	Include
2	Datatypes	Not Include	Include
3	Collective operations	Include	Include
4	Process groups	Include	Include
5	Communication contexts	Include	Include
6	Process topologies	Include	Include
7	Environmental Management and inquiry	Include	Include
8	The info object	Not Include	Include
9	Process creation and management	Not Include	Include
10	One-sided communication	Not Include	Include
11	External interfaces	Not Include	Include
12	Parallel file I/O	Not Include	Include
13	Language Bindings for Fortran, C and C++	Include Bindings for Fortran 77 and C	Include Bindings for Fortran 77, Fortran 95, C and C++
14	Profiling interface	Include	Include

MPICH2 includes C++ and Fortran 95 bindings, but MPICH1 provides the C and Fortran 77 bindings. So, the C++ binding matches the new C functions, datatypes and constants. That means the functions in C are replaced in C++. The FORTRAN 95 binding matches the new FORTRAN 77 functions [5], [12], [19]. (See Appendix 1). Moreover, MPICH2 replaced several MPICH1 constructors. (See Appendix 1)

Consequently, we can brief the differences that affect 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 [10].
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 [11], [12].
4. MPICH2 used generalized requests that are not used by MPICH1**.** These requests allow users to create new non-blocking operations with an interface [12].
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 [12].
6. MPICH1 defined collective communication for intra-communicators and two routines for creating new intercommunicators. But MPICH2 introduces extensions of many of the MPICH1 collective routines to intercommunicators, additional routines for creating intercommunicators, and two new collective routines: a generalized all-to-all and an exclusive scan [12].

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 [13].
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 get arg. 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 [13].

Therefore, in the conclusion we stated that MPICH2 extends most of the MPICH1 datatypes, routines, constants and constructors. It makes them more feasible and flexible in calling and implementation. But the extending takes into account the compatibility and portability of the applications.

Physicists, chemists, mathematicians, computer users and owners etc., benefit and achieve high performance when their applications and simulation softwares are implemented and built on new version of MPI channel (MPICH2) such as WIEN2K package that based on Density Functional Theory (DFT). WIEN2K used to simulate physical systems inorder to produce new materials such as medicine as we'll see in the next section.

C. Benchmarks

C.1 Density Functional Theory

Materials are build from atoms, atoms composed of a heavy positively charged nucleus and lighter particles called electrons. These particles interact with each other and also with their neighbors in the next atoms. In order to study the stability, structural, thermodynamic, mechanical, transport properties and electronic properties of these materials we have to solve many body second order deferential equation called equation of state, this equation obeys the laws of quantum mechanisms.

The equation of state composed of the kinetic energy operators for both the nucleus and electrons, potential energy resulted from interaction between electrons them self, nuclei's them self and nuclei's and electrons; these operators are measured by solving many-body Hamiltonian for the system, which is illustrated in equation (2.3) [8],[22]

This equation can be solved numerically after transforming it to a one body problem after some approximations, this method called Density Functional Theory (DFT) [21], [26].

$$\hat{H} = -\frac{h^2}{2}\sum_i \frac{\nabla^2_{\vec{R}}}{M_i} - \frac{h^2}{2}\sum_i \frac{\nabla^2_{\vec{r}}}{m_e} - \frac{1}{4\pi\epsilon_0}\sum_{i,j} \frac{e^2 Z_i}{\left| \vec{R}_i - \vec{r}_j \right|} - \frac{1}{8\pi\epsilon_0}\sum_{i\neq j} \frac{e^2}{\left| \vec{r}_i - \vec{r}_j \right|} + \frac{1}{8\pi\epsilon_0}\sum_{i\neq j} \frac{e^2 Z_i Z_j}{\left| \vec{R}_i - \vec{R}_j \right|} \qquad (2.3)$$

In Our work here, the Program packages like WIEN2K [8], using Full potential –Linear Augmented Plane Wave and Local Orbital's (FP-LAPW+Lo) technique is used. The WIEN2K can simulate physical and chemical systems supposed to form a new material, this is very necessary to the laboratory person, who can produce the desired material such as drug and medicine [21], [23]. It applied a parallel method to solve quantum mechanics equations based Density Functional Theory (DFT) to find the cohesive energy of any material.

In such studies we have two main factors controlling the calculation, these two factors are vice versa, the first factor is the time of calculation and the second is the sample actuality, the sample actuality means here the number of atoms constituting the sample, the bigger the number is the more actual case we have, and more complexity, this will cost a lot of calculation time. WIEN2K package composed of five modules, each module solve one of the equations from (2.4) to (2.7) sequentially [6], [8]:

- The first module is called **LAPW0**, in this process the V_{xc} is calculated in the crystal from the initial density P_0 using poisons equation:

$$\nabla^2 V_{xc} = \rho(r) \qquad (2.4)$$

- The second and third module is called **LAPW1**, **LAPW2** which are responsible for building and solving the Schrödinger equations (2.5) and (2.6), (setting up H and S matrix), and solves the generalized Eigen value problem for special point in the crystal. The number of these points is proportional to the reality of the study. The high number gives more accurate results and costs a lot of computational time, so Balanced is essential.

$$H_{ks}\Psi = E\,\Psi \qquad (2.5)$$

$$(-\nabla^2 + V_{xc})\,\Psi = E\,\Psi \qquad (2.6)$$

∇^2: is the second derivative with respect to space coordinates.	Ψ: is the wave function of this electron.
V_{xc}: is the effective attractive potential each electron feel.	**E**: is the energy of this electron in this crystal phase.

- The fourth module is called **LCORE**: from the density function, the electrons in the crystal are distributed on the lowest energy values, the density function for the core electrons is also calculated and in LCORE process as in equation (2.7):

$$\rho(r) = \int \Psi\,\Psi^* dr^3 \qquad (2.7)$$

- The fifth module is called **MIXER**: the new total density is compared with the old density, if the values are the same or the difference is less than an assigned value; the self consistent (SC) is finished as shown in Figure 1. The total energy and wave functions of the electrons are found. Otherwise, the new density is mixed with old density with a percentage decided at the beginning of the calculation to reproduce a new density to run another cycle to get faster convergence and recalculate V_{xc} using equation (2.4).

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 combined form and the sum of the free atom's energy in their free state as shown in equation (2.8)

$$E_{\text{cohesive energy}} = E_{\text{compound}} - \sum E_{\text{free atoms}} \qquad (2.8)$$

Each stable form of these atoms can produce positive value for the cohesive energy, the material normally can take more than one stable state, and the state with the highest cohesive energy is the most stable one [22].

To see more about density functional theory (DFT) and WIEN2K see Appendix 2.

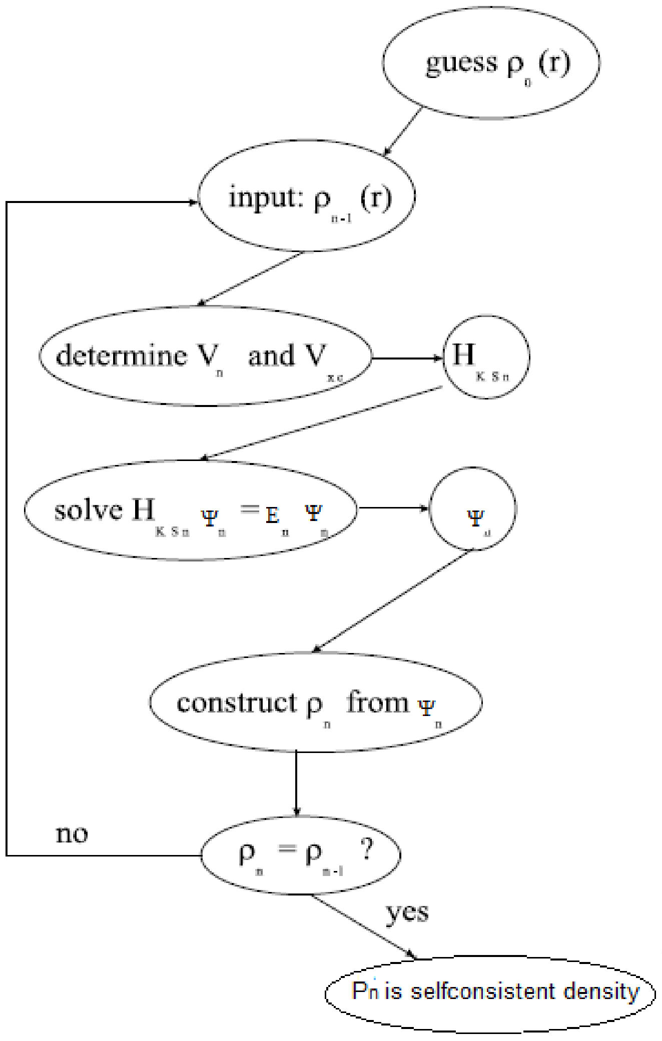
guess ρ_0 (r)
input: ρ_{n-1} (r)
determine V_n and V_{xc}
H_{KSn}
solve $H_{KSn} \Psi_n = E_n \Psi_n$
Ψ_d
construct ρ_n from Ψ_n
$\rho_n = \rho_{n-1}$?
no
yes
Pn is selfconsistent density

Figure 1: Physical Problem Solving Steps

The authors in [21] 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. But 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].

C.2 Matrix Multiplication

This section discusses parallel algorithms for multiplying two *n* × *n* dense, square matrices *A* and *B* to yield the product matrix *C* = *A* × *B*. Parallel matrix multiplication algorithm in this section is based on the conventional serial algorithm shown in Algorithm 1.

```
procedure MAT_MULT(A,B,C)
begin
for i:=0 to n-1 do
for j:=0 to n-1 do
begin
C[I,j] :=0;
for k :=0 to n-1 do
C[i,j] := C[i,j] + A[i,k] x B[k,j]
Endfor
end MAT_MULT
```

Algorithm1: The conventional serial algorithm for multiplication of two *n* × *n* matrices.

If we assume that an addition and multiplication pair (line 8) takes unit time, then the sequential run time of this algorithm is n^3. However, for the sake of simplicity and better performance, we take parallel matrix multiplication algorirhm, which based on the conventional best serial algorithm. A concept that is useful in matrix multiplication as well as in a variety of other matrix algorithms is that of block matrix operations.

The authors in [40] express a matrix computation involving scalar algebraic operations on all its elements in terms of identical matrix algebraic operations on blocks or submatrices of the original matrix. Such algebraic operations on the submatrices are called *block matrix operations*.

For example, an $n \times n$ matrix A can be regarded as a $q \times q$ array of blocks $A_{i,j}$ ($0 \leq i, j < q$) such that each bock is an $(n/q) \times (n/q)$ submatrix. The matrix multiplication algorithm in Algorithm 1 can then be rewritten as Algorithm 2, in which the multiplication and addition operations on line 8 are matrix multiplication and matrix addition, respectively.

Not only are the final results of Algorithm 1 and 2 identical, but so are the total numbers of scalar additions and multiplications performed by each. Algorithm 1 performs n^3 additions and multiplications, and Algorithm 2 performs q^3 matrix multiplications, each involving $(n/q)\times(n/q)$ matrices and requiring $(\frac{n}{q})^3$ additions and multiplications. We can use p processes to implement the block version of matrix multiplication in parallel by choosing $q = \sqrt{p}$ and computing a distinct $C_{i,j}$ block at each process.

```
Procedure BLOCK_MAT_MULT(A,B,C)
begin
for i:=0 toq-1 do
for j:=0 toq-1 do
begin
 Initialize all elements of Ci,j to zero;
for k :=0 toq-1 do
C[i,j] := C[i,j] + A[i,k] x B[k,j]
Endfor
End BLOCK_MAT_MULT
```

Algorithm 2: The block MMT algorithm for $n \times n$ matrices with a block size of $(n/q) \times (n/q)$.

C.3 Approximate Value/ Mathemetical Constant --PI (π)

PI is a name given to the ratio of the circumference of a circle to the diameter. That means, for any circle, you can divide the circumference (the distance around the circle) by the diameter and always get exactly the same number. It does not matter how big or small the circle is, PI remains the same.

The value of PI can be calculated in a number of ways. Consider the following method of approximating PI [28]:

1- Inscribe a circle in a square see Figure 2
2- Randomly generate points in the square.
3- Determine the number of points in the square that are also in the circle
4- Let r be the number of points in the circle divided by the number of points in the square
5- PI ~ 4 r
6- Note that the more points generated, the better the approximation

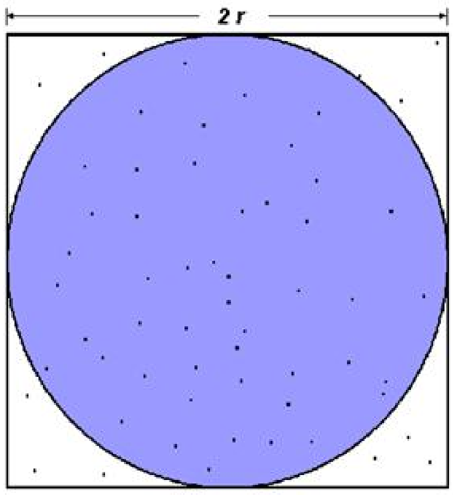

Figure 2: Inscribed circle in a square to calculate PI (π).

If the previous steps executed sequentially the pseudo code for this procedure can be as in Figure 3:

```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
xcoordinate = random1
ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```

Figure 3: Serial Pseudo Code to Calculate PI (π)

Note that most of the time in running this program would be spent executing the loop. Therefore, this leads us to check the parallel solution, which means: Computationally intensive, Minimal communication and Minimal I/O. however, Parallel strategy breaks the loop into portions that can be executed by the tasks. By the task of approximating PI in parallel way [28]:

- Each task executes its portion of the loop a number of times.
- Each task can do its work without requiring any information from the other tasks (there are no data dependencies).
- Uses the SPMD model. One task acts as master and collects the results.

If the previous steps executed in parallelized way, the pseudo code for this procedure can be as in Figure 4. Note that: ***Italic Font*** highlights changes for parallelism.

From parallel pseudo code to calculate PI, we conclude that the most of the time in running this program would be log(p). p is the number of processors. This indicates the performance is bigger that in serial way.

```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
xcoordinate = random1
ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER

  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)

else if I am WORKER

  send to MASTER circle_count

endif
```

Figure 4: Parallel Pseudo Code to Calculate PI (π)

Consequently, in this work, we evaluate the performance of two versions of the well-known massage passing interface (MPI) library: MPICH1 vs. MPICH2 and evaluate the performance between MPICH and OpenMP. In our experiments, we used three benchmarks. The first one is the WIEN2K application, which is based on Density Function Theory, the second is a Matrix Multiplication and the third is the approximate value PI.

Chapter Three

Literature Review

There are many studies and researches carried out on tasks distributing and system implementation in parallel processing systems. Applications based parallel processing used in a large number of fields: scientific, business, industrial and medical purposes. Implementation of tasks distributing via parallel algorithms using MPICH1, MPICH2 and OpenMP is important and very helpful in resources utilization and maximum throughput in minimum execution time. Many researches were conducted on comparison between parallelized implementations using different channels in several areas. In this chapter, we present related works and literature review relevant to our work.

A research by Erik Mc Clements (2006) implemented a Performance Comparison of Open Source MPI Implementations. They compared and contrasted various Open Source MPI implementations by using message size as key factor, Identifying their strengths and weaknesses across multiple machine architectures commonly used for HPC (High Performance Computing). Their results were as the following: MPICH performance is higher than OpenMP performance in the execution when a message size less than 5 kb. However, if it is more than 5 kb the OpenMP performance is better [29]. In Information Security of scientific computing, a study by Xiaojun Ruan and al proposed an optimization strategy for MPICH2 improvement by designing ES-MPICH2: A Message Passing Interface with Enhanced Security (2010). They integrated encryption algorithms into the

MPICH2 library so that data confidentiality of MPI applications could be readily preserved without a need to change the source codes of the MPI applications. Since they provide a security enhanced MPI-library with the standard MPI interface, data communications of a conventional MPI program can be secured without converting the program into the corresponding secure version. The results show ES-MPICH2 provides secured Message Passing Interface with a reasonable performance better than original MPICH2. Future work will implement some stronger and more efficient cryptographic algorithms like Elliptic Cureve Cryptography in ES-MPICH2 [31].

In parallel implementation area, Rahmadi Trimananda and Christoforus Yoga Haryanto performed a study of A Parallel Implementation of Hybridized Merge-Quicksort Algorithm on MPICH, study (2010). The paper indicated how the data elements are distributed to processors, sorted in smaller groups of data elements in parallel on each processor by using quicksort algorithm and later merged in parallel by using mergesort algorithm. The implementation results on MPICH1 platform are showing potential speedups since that the communication channel is adequate for large groups of data elements. In future work, the experiments are to be conducted on some other platforms, e.g. MPICH2, to compare the results with the ones obtained [16].

In addition, another research in parallelism of matrix multiplication by Sherihan Abu ElEnin, Mohamed Abu ElSoud (2011). The researchers implemented an Evaluation of Matrix Multiplication on an MPI Cluster by comparing between serial and parallel approaches for running Matrix Multiplication on MPICH1. The results show that the developed performance model checked and it showed that the parallel model is faster than the serial model and the computation time was reduced [1].

Finally, Rezek Mohammad, Areej Jabir, and Rashid Jayousi developed a comparison between distribute K-Point method and data distribution method for sparse matrix distribution over MPICH1, the two methods have been used to run WIEN2K package which is used to study the physical and chemical properties of the materials (2011). The result was as follows, the data distribution method gives better reduction in the time of calculation [21]. Table 6 presents a summary of the above literature review contributions.

Table 6: Summary of Literature Review Contributions According to Area of Research

Area of Research	Study Title	Author	Year	Main Contribution
Education	Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms	Samuel Williams, Leonid Oliker and Richard Vuduc	**2007**	Comparison between a multicore-specific Pthreads implementation versus a traditional MPI approach to parallelization across the cores. Results showed that the Pthreads strategy resulted in runtimes more than twice as fast as the message passing strategy [11].
	Design Considerations for Shared Memory MPI Implementati-ons on Linux NUMA Systems: An MPICH/MPI-CH2 Case Study.	Per Ekman and Philip Mucci	**2005**	The work is to make MPICH and MPICH2 more tolerant of Non Uniform Memory Access architectures (NUMA). The results showed that: the patched MPICH is efficient than the original mpich [33].
	Cilkvs MPI: Comparing Two Very Different Parallel Programming Styles	Sonny Tham and John Morris	**2003**	The results were: problems, which have simple dataflow solutions and involve transfer of large blocks of data are simpler and faster in Cilk, whereas MPI handles problems with iterative solutions and smaller messages better. MPI was clearly more efficient than Cilk only in the iterative, irregular Gaussian elimination problem [17].

Area of Research	Study Title	Author	Year	Main Contribution
Education	Hybrid Programming Fun: Making bzip2 Parallel with MPICH2 & pthreads on the Cray XD1	Charles Wright	**2006**	A reasonable approach would be to combine pthreads and MPI on the XD1. Using this hybrid model, the author was able to parallelize non-computational tasks such as I/O and communication easily. This study focuses on how pthreads were used to extend MPI in a natural way to improve the speed and efficiency of the program. The results were as the following : The combination of pthreads and MPICH2 can result in many benefits ranging from easier programming to more effective use of system resources. In the case of the parallel bzip program, the resulting improvements in both speedup and efficiency overshadow the lack of hardware support for MPICH2 currently available on the XD1 [32].
	NUMA-aware shared-memory collective communica-tion for MPI	Shigang Li, Torsten Hoefler and Marc Snir	**2013**	The results showed that: performance of HMPI dropped between the MPICH2 performance and OpenMP one. This is better than MPICH2 and lower than OpenMP performance [34].
Physics	Optimum Execution For WIEN2K using Parallel Programming Models (Comparison Study)	Rezek Mohammad, Areej Jabir, and Rashid Jayousi	**2011**	Development of data distribution method and compared between k-point method and data distribution. The results were, the data distribution method gives better reduction in the time of calculation and in case of large number of atoms or the complexity it is better to use data distribution method [21].

Area of Research	**Study Title**	**Author**	**Year**	**Main Contribution**
Education	A Parallel Implementation of Hybridized Merge-Quicksort Algorithm on MPICH	Rahmadi Trimananda and Christoforus Yoga Haryanto	**2010**	Showed how the data elements are distributed to processors, sorted in smaller groups of data elements in parallel on each processor by using quicksort algorithm, and later merged in parallel by using mergesort. The implementation results on MPICH1 showed potential speedups provided that the communication channel is adequate for large groups of data elements [16].
	Efficient Sparse Matrix Multiple-Vector Multiplication Using a Bitmapped Format	Ramaseshan Kannan	**2012**	The implemented algorithm achieves high-level advantage for very large problem sizes, e.g iterative solvers for linear systems. Moreover, its performance results proved that these performance optimizations could achieve good efficiency gains on all platforms by increasing register and cache reuse [14].
	Evaluation of Matrix Multiplication on an MPI Cluster	Sherihan Abu ElEnin, Mohamed Abu ElSoud	**2011**	In addition, a comparison between serial and parallel approaches for running Matrix Multiplication on MPICH1 was in [1].The results show that the developed model has been checked and it has been shown that the parallel model is faster than the serial and the computation time was reduced.
Scientific Computing	Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem	Darius Buntinas, Guillaume Mercier, and William Gropp	**2008**	They describe how we ported MPICH2 over Nemesis and show the performance benefits of MPICH2 Nemesis. The resulting MPICH2 software stack yields a very low latency and high bandwidth and compares favorably with previous competing software (MPICH1) [30].

Area of Research	**Study Title**	**Author**	**Year**	**Main Contribution**
Information Security	ES-MPICH2: A Message Passing Interface with Enhanced Security	Xiaojun Ruan, Qing Yang, Mohammed I. Alghamdi, Shu Yin, Zhiyang Ding, Jiong Xie, Joshua Lewis, and Xiao Qin	**2010**	They integrated encryption algorithms into the MPICH2 library so that data confidentiality of MPI applications could be readily Preserved without a need to change the source codes of the MPI applications. since they provide a security enhanced MPI-library with the standard MPI interface, data communications of a Conventional MPI program can be secured without converting the program into the corresponding secure version. The results were, ES-MPICH2 provides secured Message Passing Interface with a reasonable performance better than original MPICH2. In the future, they may implement some stronger and more efficient cryptographic algorithms like Elliptic Cureve Cryptography in ES-MPICH2 [31].
Scientific Computing	Blocking vs. Non-Blocking Coordinated Checkpoint-ing for Large-Scale Fault Tolerant MPI	Camille Coti, Thomas Herault, Pierre Lemarinier and Laurence Pilard	**2006**	A comparison between these two approaches (blocking and non-blocking) and a study of their scalability. Then they evaluate their impact on large-scale applications. The results were, the experimental study demonstrated that for high speed networks, the blocking implementation gives the best performance for sensible checkpoint frequency. On clusters of workstations and computational grids, the non-blocking implementation gives the best performance [35].

Area of Research	Study Title	Author	Year	Main Contribution
Scientific Computing	Adaptive Strategy for One-sided Communicati-on in MPICH2	Xin Zhao, Gopalakrishn-an Santhanaram-an, and William Gropp	**2012**	In this paper they describe their design and implementation of an adaptive strategy for one-sided operations and synchronization mechanisms (fence, post-start-complete-wait, lock-unlock) supported by MPICH2, which combines benefits from both lazy and eager approaches. Their performance results demonstrate that our approach performs as well as the lazy approach for small data transfers and achieves similar performance as the eager Approach for large data transfers [36].
	Multi-core Aware Optimization for MPI Collectives	BiboTu, Ming Zou, Jianfeng Zhan, Xiaofang Zhao and Jianping Fan	**2008**	The authors construct a portable optimization methodology over MPICH2 for collective operations on multicore clusters. In this study, collective algorithms with hierarchical virtual topology focus on the performance difference among different communication levels on multi-core clusters, simply for intra-node and inter-node communication; The results of performance evaluation show that the multi-core aware optimization methodology over MPICH2 is efficient [37].
	Asynchronous MPI for the Masses	Markus Wittmann, Georg Hager, Thomas Zeiser, and Gerhard Wellein	**2013**	They implemented non-blocking point-to-point communication. The results were, many applications show performance improvements when they use the new implemented approach [38].

Area of Research	**Study Title**	**Author**	**Year**	**Main Contribution**
Scientific Computing	Performance Comparison of Open Source MPI Implementati-ons	Erik McClements	**2006**	The main aim of this project is to compare and contrast various Open Source MPI implementations by using message size as key factor, Identifying their strengths and weaknesses across multiple machine architectures commonly used for HPC. The results were as the following: MPICH performance is higher than OpenMP performance in the execution when the message size less than 5 kb. But if it is more than 5 kb the OpenMP is higher [29].
Scientific Computing	Scheduling Dynamically Spawned Processes in MPI-2	M´arcia C. Cera1, Guilherme P. Pezzi, Maur´ıcio L. Pilla, Nicolas B. Maillard1, and Philippe O. A. Navaux, ,	**2006**	MPICH2 supports dynamic spawning of tasks. It provides primitives to spawn processes during the execution and to enable them to communicate together. This paper presents a scheduler module, that has been implemented with MPICH2, that determines, on-line (i.e. during the execution), on which processor a newly Spawned process should be run, and with which priority. The scheduling is computed under the hypotheses that the MPICH2 program follows a Divide and Conquer model. A clear improvement in the balance of the load is shown by the experiments [10].

It should be noted that in this research, we expanded on the work of Rahmadi Trimananda and Christoforus Yoga Haryanto [16]. The work of Sherihan Abu ElEnin, Mohamed Abu ElSoud[1]. And the work of Rezek Mohammad, Areej Jabir, and Rashid Jayousi [21]. The differences between our research and the other three researches are that our research will respond to future work of [16] that recommended, "Distributing the data elements, sorted in smaller groups of data elements in parallel on each processor by using quicksort algorithm and later merged in parallel by using mergesort algorithm on MPICH2 platform". Also it follows the recommended future work in [1] that recommended "to Evaluation of Matrix Multiplication on an MPICH2 Cluster". Furthermore, this research follows the proposed future work in [21] that recommended, "Studying the accuracy and the execution time of WIEN2K on MPICH2".

Our research main contributions are the evaluation of WIEN2K Performance on MPICH2 vs. MPICH1 and Evaluation of MMT and PI(π) Performances on MPICH vs. OpenMP used in this study . It should be noted that a comparison of partial results of our experiments is compared with the results of [1], [16], [21]. The next chapter introduces our work methodology and the environments of the experiments.

Chapter Four

Methodology

In this chapter, we present our research and work methodology. To achieve the objectives of this research, we started to prepare the environment, in order to conduct the experiment. We prepared a multi-core computer with Linux Fedora 14 operating system, MPICH1, MPICH2, Open MP files, WIEN2K packages modules and supported libraries. Moreover, a matrix multiplication program, mathmetical constant π program and other supported tools and programs as Mathematical Kernel Library (MKL), SCALAPACK and Secure Shell (SSH) program were installed and prepared for the experements. It should be noted that SCALPACK is needed for sparse matrices diagonalizating and Fastest Fourier Transform in the west (FFTW), whilest Secure Shell program is used for secure communication.

In the present work, two parts have been tested, in the first part (Part 1), we focused on implementing WIEN2K package on MPICH2 and distributing tasks of the package using MPICH1 and MPICH2 on multi-core machine (see Figure 5).

The experiments have been tested by running first module of WIEN2K package (LAPW0) as benchmark using MPICH1 and MPICH2 on one, two, three, and four processors of the quad multi-core machine. Each experiment has been repeated several times then the average of the elapsed time has been computed and recorded.

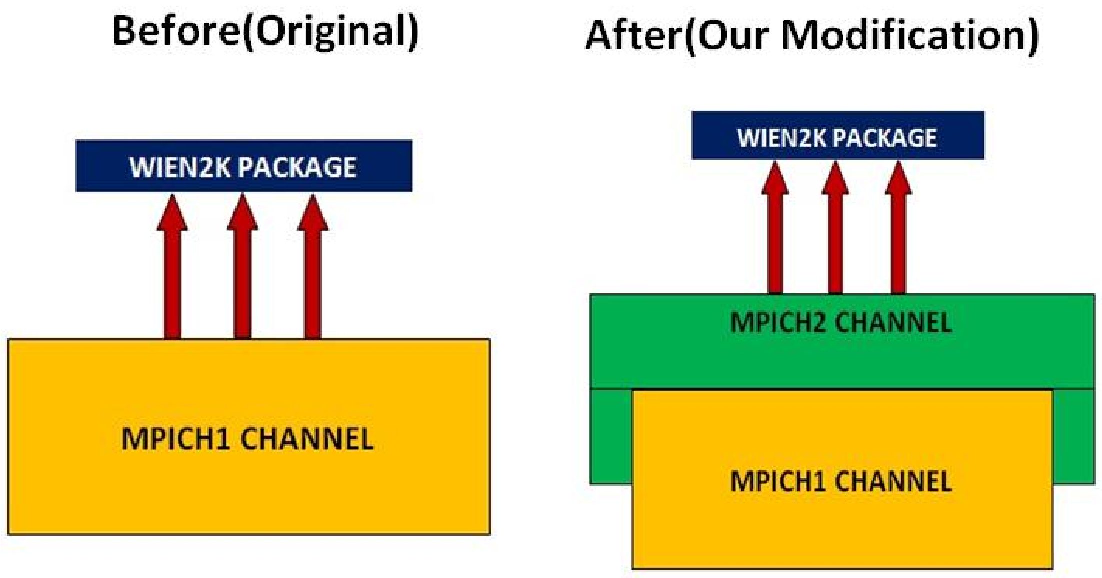

Figure 5: Possible Running for WIEN2K Package

MPICH2 included many new features, so we have focused on MPICH2 settings and configurations when we run MPI programs on the second channel. A complete focused list of changes follows:

1. **Dynamic process management**: MPICH2 presents a set of MPI interfaces that allow for a variety of approaches to process management while placing minimal restrictions on the execution environment. MPICH1 doesn't concern with communication rather than process management.
2. **One-sided operations**: put, get and accumulate routines.
3. **Machine file**: MPICH1 distribues CPUs for modules using machine file in different way than MPICH2 as shown in Figure 6.

MPICH1	MPICH2

`Lapw0: rezek-dell15:0`
`Lapw0: rezek-dell15:1`
`Lapw1:  rezek-dell15:2`
`Lapw1: rezek-dell15:3`
	`Lapw0: rezek-dell15:2`
	`Lapw1: rezek-dell15:2`

Figure 6: Sample of machine file shows CPUs distribution for modules in MPICH1 and MPICH2.

4. **Datatypes:** MPICH1 includes simple datatypes, but MPICH2 includes simple, advanced and derived dataypes.
5. **The Info Object:** MPICH2 includes info object, this object is used for several functions. Info is an opaque object with a handle of type MPI_Info in C, MPI::Info in C++, and INTEGER in FORTRAN. It stores an unordered set of (key, value) pairs (both key and value are strings). A key can have only one value. Each pair (key, value) is special for a determined function.
6. **External Interfaces:** MPICH2 used generalized requests that are not used by MPICH1**.** These requests allow users to create new non-blocking operations with an interface. A fundamental property of non-blocking operations is that progress toward the completion of this operation occurs asynchronously.
7. **I/O:** MPICH2 supports parallel I/O (e.g: grouping, collective buffering and disk-directed I/O) that added flexibility and expressiveness [12].
8. **Bindings:** MPICH2 includes C++ and FORTRAN 90 bindings, but MPICH1 provides the C and FORTRAN 77 bindings. Therefore, the C++ and FORTRAN 90 binding matches the new C and FORTRAN 77 functions respectively. The same deal with datatypes and constants.
9. **Arguments argc and argv:** MPICH1 needs to pass the arguments **argc** and **argv** by an application to MPI INIT and main functions. In MPICH2 does not need to pass them.
10. **Classes:** The members of the MPI namespace are those classes corresponding to objects implicitly used by MPI. An abbreviated definition of the MPICH1 namespace and its member classes is as follows:

```
namespace MPICH1 {
class Comm {...};
class Intracomm : public Comm {...};
class Graphcomm : public Intracomm {...};
class Cartcomm : public Intracomm {...};
class Intercomm : public Comm {...};
class Datatype {...};
class Errhandler {...};
class Exception {...};
class Group {...};
class Op {...};
class Request {...};
class Prequest : public Request {...};
class Status {...};
};
```

Additionally, the following classes defined for MPICH2:

```
namespace MPI {
class File {...};
classGrequest : public Request {...};
class Info {...};
class Win {...};
};
```

At the end, in the part 1 we have tested and concentrated with core changes between MPICH1 and MPICH2 to implement WIEN2K on MPICH2 and compare between the results WIEN2K MPICH1 running and MPICH2 one . However, we have looked forward to apply the additions over MPICH1 in implemention of MPICH2 in order to get MPICH2 more robust, efficient, and convenient to use. As a result, the performance of WIEN2K on MPICH2 will increase over MPICH1.

WIEN2K execution on OpenMP encountered by two factors and the same factors were the reseaons of our using other two benchmarks (Matrix Multiplication of different sizes and Mathmetical Constant): librariers that support WIEN2K running on OpenMP are not available and WIEN2K includes a large number of subroutines, cycles and modules. WIEN2K structure is complex and interleaved. Moreover, it is not clear in its commercial documentation. Therefore, we extended our experemnts using more benchmarks.

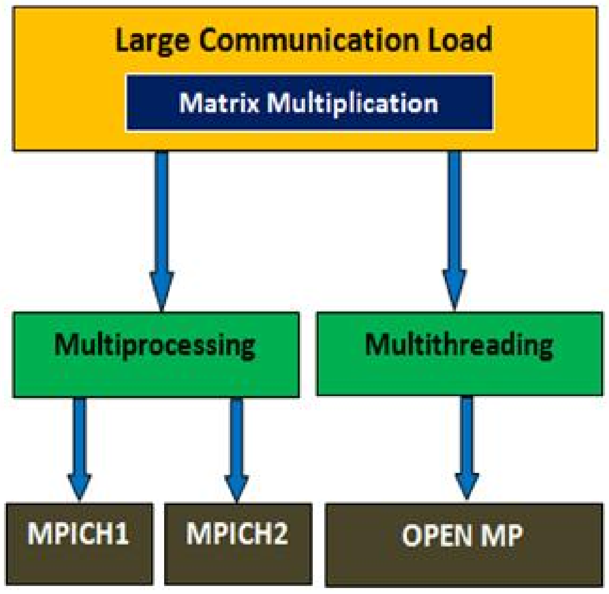

Figure 7: Possible Running for Matrix Multiplication

In the second part (Part 2) of experiments, two cases of experiments have been tested. In the first case (Case 1: example on large size of data chuncks) that presented heavy load communications and big data distributions; we tested the performance of parallel matrix multiplication using multi-processing (message passing) using MPICH1 and MPICH2, and multithreading paradigms using OpenMP (see Figure 7). In the second case (Case 2: example on small size of data chuncks) that presented light load communications and small data distributions; we tested the performance of parallel approximate value PI (π) using multi-processing (message passing) using MPICH1 and MPICH2, and multithreading paradigms using OpenMP (Figure 8).

In Case 1, the matrix multiplication has been implemented using MPICH1, MPICH2, and OpenMP by different matrix sizes that indicate twelve states (128, 256, 384, 512, 640, 768, 896, 1024, 2048, 3072, 4096 and 5120). Each state has acted a unique matrix size. In the other case (Case 2) of Part 2, the PI (π) has been computed using MPICH1, MPICH2 and OpenMP.

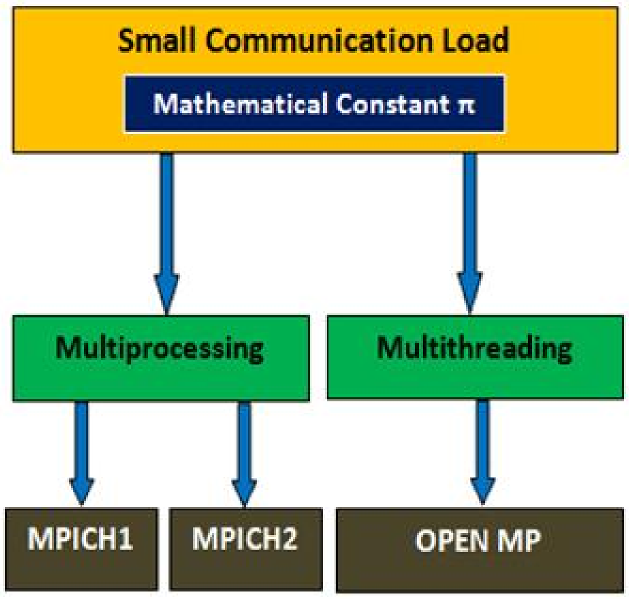

Figure 8: Possible Running for Mathmetical Constant π

Consequently, in Part 2 we have tested and concentrated with comparing and evaluating results between MPICH1, MPICH2 and OpenMP tests for matrix multiplication of different sizes and mathmetical constant.

Finally, in our research we encountered by number of obstacles but the most important of them are as follow:

1. We waited a round four months for preparing a cluster of computers in order run WIEN2K and evaluate MPICH1 and MPICH2.
2. Libraries that support running of WIEN2K on OpenMP are not available due to the lake of fund.
3. We waited a round two months for preparing MPICH2 standard version, that recommended for Linux Fedora 14.
4. We waited a round one month for preparing standard versions of FFTW and MKL programs, which is recommended for WIEN2K.

Chapter Five

Experiments and Results Analysis

In this work, two parts of experiments were carried out. In the first part (Part 1), we focused on distributing tasks of WIEN2K program using MPICH1 and MPICH2 on multi-core machine. Whereas in [21] the experiments were carried out on a cluster using MPICH1 to distribute WIEN2K task. In the second part (Part 2) of experiments, two cases of experiments were carried out. In the first case (Case 1) we tested the performance of parallel matrix multiplication using multi-processing (message passing) using MPICH1 and MPICH2, and multithreading paradigms using OpenMP. In the second case (Case 2) we tested the performance of parallel approximation of PI (π) value using the two paradigms: 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 7.

Table 7: 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, first we installed MPICH2 on Fedora Linux version 14 using specific steps as shown in Figure 9 [43]. Then a set of programs were installed and optimized with appropriate options together with WIEN2K. These programs are listed in Table 8.

We need the following prerequisites:

1. The tar file mpich2-1.0.5p3.tar.gz (which can be obtained from http://www-unix.mcs.anl.gov/mpi/mpich2/)
2. A C compiler (gcc is sufficient)
3. A Fortran compiler if Fortran applications are to be used (g77 is sufficient)

Both the C and Fortran compiler are present in Fedora Core 4 by default.

Step 1. Create a directory MPI (we can use any name) in the home directory.

```
$ cd $HOME
$ mkdir MPI
 $ cd $HOME
```

Step 2. Unpack the tar file.

```
$ tar xfz mpich2-1.0.5p3.tar.gz
```

The directory MPI will now contain a sub-directory mpich2-1.0.5p3.

Step 3. Choose an installation directory (the default is /usr/local/bin)

```
$ mkdir mpich2-install
```

Step 4. Choose a build directory

```
$ mkdir mpich2-1.0.5
```

Now the MPI directory will contain three sub-directories namely mpich2-1.0.5p3, mpich2-1.0.5 and mpich2-install.

Step 5. Configure MPICH2, specifying the installation directory and running the configure script in the source directory.

```
$ cd $HOME
$ cd MPI/mpich2-1.0.5
$/home/you/MPI/mpich2-1.0.5p3/configure --prefix=/home/you/MPI/mpich2-install
```

For other configure options please refer the MPICH2 Installer's Guide

Step 6. Build MPICH2

```
$ make
```

Step 7. Install the MPICH2 commands.

```
$ make install
```

Step 8. Add the bin directory to your path.

```
$ export PATH=/home/you/MPI/mpich2-install/bin:$PATH
```

(It is better to add this line in .bash_profile file present in the home directory so that this path gets permanently added once we reboot the system.

$ cd $HOME

$ vi *.bash_profile*

Then append the above command of step 8.)

We can check that everything is in order at this point by doing

```
$ which mpd
$ which mpicc
$ which mpiexec
$ which mpirun
```

All should refer to the commands in the bin subdirectory of our install directory.

The MPICH2 has been successfully installed now.

Figure 9: Installation Steps for MPICH2 on Fedora Linux Version 14

Recall that we continue the work of [21], where they installed and used MPICH1 to run WIEN2K program. For this work, we installed MPICH2 channels figure (9) then installed MPICH2 WIEN2K 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 (0,1,2,3). Each experiment was repeated several times then the average of the elapsed time was computed. After that, the calculation was recorded. The experiments were divided into two parts: the first one run LAPW0 for one cycle. In the second experiment (Part 2), in first case (Case 1), the matrix multiplication was implemented using MPICH1, MPICH2, and OpenMP by twelve states (128, 256, 384, 512, 640, 768, 896, 1024, 2048, 3072, 4096 and 5120). Each state acted a unique matrix size. But, in the second case (Case 2) of Part 2 the PI (π) was computed using MPICH1, MPICH2, and OpenMP.

Table 8: 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

Part 1:

MPICH1 does not need to run the daemon explicitly because it is built in the MPICH1 environment. Also, the command which is used in MPICH1 to execute programs is "mpirun". In other side, MPICH2 runs the daemon before any execution because MPICH2 separate the daemon from MPICH2 environment. In addition, MPICH2 use “mpiexec” to execute applications. For example, the steps of the LAPW0 execution on MPICH2 are shown in Figure 10. Moreover, Figure 11 shows the steps of the LAPW0 execution on MPICH1.

The results of the average running time for experiment 1 (LAPW0) are summarized in Table 9. 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}.

$$if = \frac{T_{MPICH1} - T_{MPICH2}}{T_{MPICH1}} \tag{4.1}$$

```
[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.

[rezek@rezek-dell15 examples]$ mpiexec -n 4 lapw0_mpi

lapw0_mpi has started with 4 tasks.

Initializing arrays...

Running Time = 19.001209

Done.
```

Figure 10 : Screen Shot of Running LAPW0 on MPICH2

```
[rezek@rezek-dell15~]$ cd/home/rezek/mpich1/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]$ mpirun -np 1 lapw0_mpi

lapw0_mpi has started with 1 tasks.

Initializing arrays...

Running Time = 64.764301

Done.

[rezek@rezek-dell15 examples]$ mpirun -np 2 lapw0_mpi

lapw0_mpi has started with 2 tasks.

Initializing arrays...

Running Time = 35.987721

Done.

[rezek@rezek-dell15 examples]$ mpirun -np 3 lapw0_mpi

lapw0_mpi has started with 3 tasks.

Initializing arrays...

Running Time = 26.880067

Done.

[rezek@rezek-dell15 examples]$ mpirun -np 4 lapw0_mpi

lapw0_mpi has started with 4 tasks.

Initializing arrays...

Running Time = 21.417534

Done.
```

Figure 11 : Screen Shot of Running LAPW0 on MPICH1

Table 9: 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

As shown in Figure 12, it is clear that MPICH2 performance is higher than MPICH1 performance by approximately 3%. In other words, MPICH2 increases the speed up of WIEN2K execution on each multicore by 3%. Consequently, the simulation of production a new material in our case which needs 30 working days will be decreased by one day. The figure shows the difference between the execution time on MPICH1 and MPICH2. In this figure the curves are decline when number of processors increase until it reaches 4. After that the speed up and efficiency approximately reach the stability then decreasing. But on all states MPICH2 performance is higher. Therefore, we believe that the following nine added features (mentioned in the background chapter) have positive impact on the performance of MPICH2:

1. 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 [10].

3. MPICH2 supports one-sided communication. It provides three communication calls, these operations are non-blocking [11], [12].
4. MPICH2 used generalized requests that are not used by MPICH1**.** These requests allow users to create new non-blocking operations with an interface [12].
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 [12].
6. MPICH2 introduces extensions of many of the MPICH1 collective routines to intercommunicators, additional routines for creating intercommunicators, and two new collective routines: a generalized all-to-all and an exclusive scan [12].
7. MPICH2 supports MPI THREAD MULTIPLE [5].
8. MPICH2 is concerned with communication rather than process management. In addition, it provides a fast and scalable startup mechanism when parallel jobs are started [13].
9. 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 [13].

It should be noted that the time unit in the experiments of Part 1 is in minutes, whereas it is in seconds in Part2.

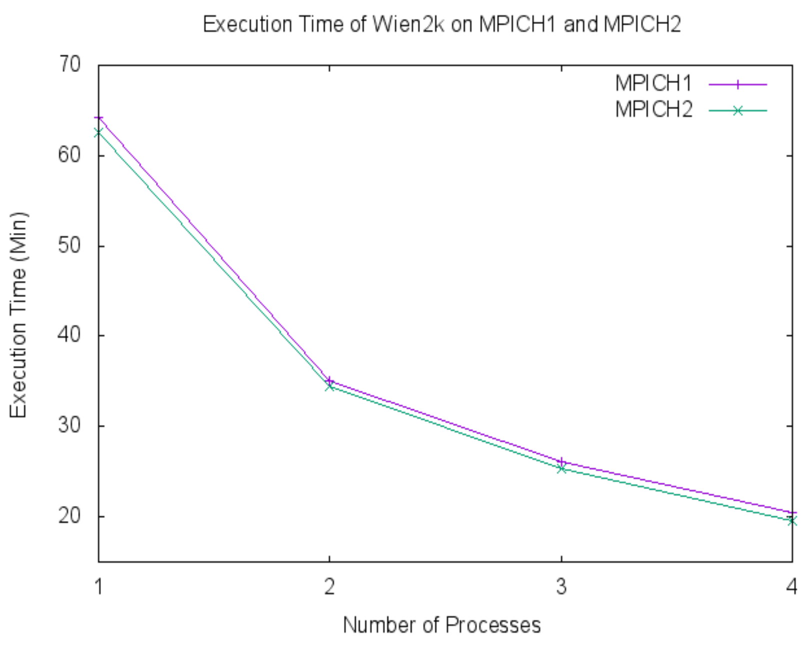

Figure 12: The WIEN2K Execution Time of MPICH1 vs. MPICH2.

Part 2:

Case 1:

In this case the experiments were implemented on a standard parallel matrix multiplication (MMT) of sizes 128x128, 256x256, 384x384, 512x512, 640x640, 768x768, 896x896, 1024x1024, 2048x2048, 3072x3072, 4096x4096 and 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.

However, the steps of the (MMT) execution on OpenMP are shown in Figure 13.

```
[rezek@rezek~]$ cd /home/rezek/OpenMP/examples

[rezek@rezek examples]$ icc -o mmtop -openmp mmtop.c

[rezek@rezek examples]$ ./mmtop

Starting matrix multiplication with 1 threads
```

```
Initializing matrices...
Time for parallel matrix multiplication: 151.24 s
Done.
[rezek@rezek examples]$ icc -o mmtop –openmp mmtop.c
[rezek@rezek examples]$ ./mmtop
Starting matrix multiplication with 2 threads
Initializing matrices...
Time for parallel matrix multiplication: 68.73 s
Done.
[rezek@rezek examples]$ icc -o mmtop –openmp mmtop.c
[rezek@rezek examples]$ ./mmtop
Starting matrix multiplication with 4 threads
Initializing matrices...
Time for parallel matrix multiplication: 51.99 s
Done.
[rezek@rezek examples]$ icc -o mmtop –openmp mmtop.c
[rezek@rezek examples]$ ./mmtop
Starting matrix multiplication with 8 threads
Initializing matrices...
Time for parallel matrix multiplication: 87.87 s
Done.
[rezek@rezek examples]$ icc -o mmtop –openmp mmtop.c
[rezek@rezek examples]$ ./mmtop
Starting matrix multiplication with 16 threads
Initializing matrices...
Time for parallel matrix multiplication: 104.60 s
Done.
[rezek@rezek examples]$
```

Figure 13: Screen Shot of Running MMT on OpenMP

The results of the average running time for all experiments in Case 1 (MMT) are summarized in Table 10. This table shows the execution time on MPICH and OpenMP. For 5120x5120 matrices the experiment results in Figure 14 shows that the performance and speed up using multithreading is higher than multiprocessing. Also the experiments with multiplier sizes larger than or equal 384x384 shows the same results, but the results are inverse when the matrix size is smaller than 384x384. Thus, in our experements environment 384x384 matrices size become as a conversion point (see Figure 15). This is caused by the overhead of processes management, data distribution and large size of data chunks communication in case of size larger; than 384x384.

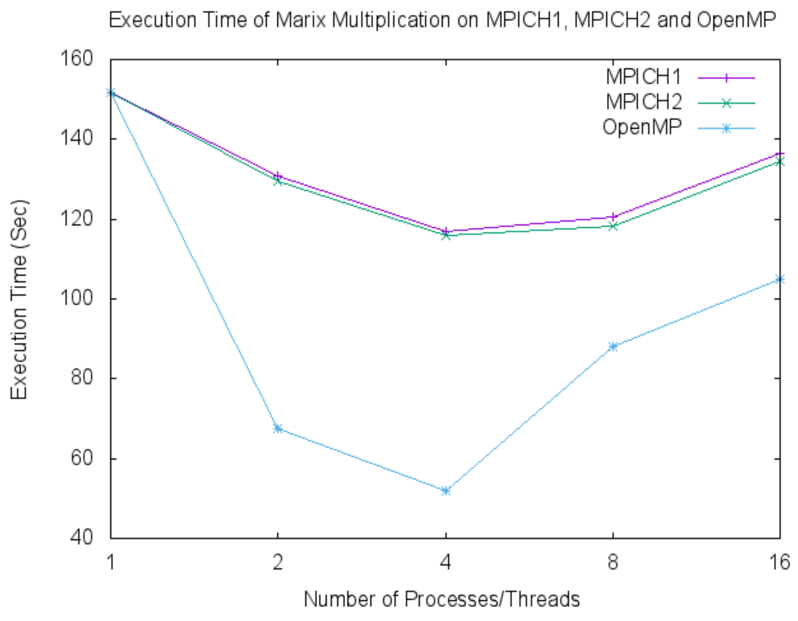

Figure 14: Execution Time of Matrix Multiplication (5120 X 5120) Using MPICH1 vs. MPICH2 vs. OpenMP

The experiment's platform has four processing elements. It is clear in Figure 14 that the curve declines (i.e. improving the efficiency and speed-up) until the number of processes/threads reaches 4. After that, 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).

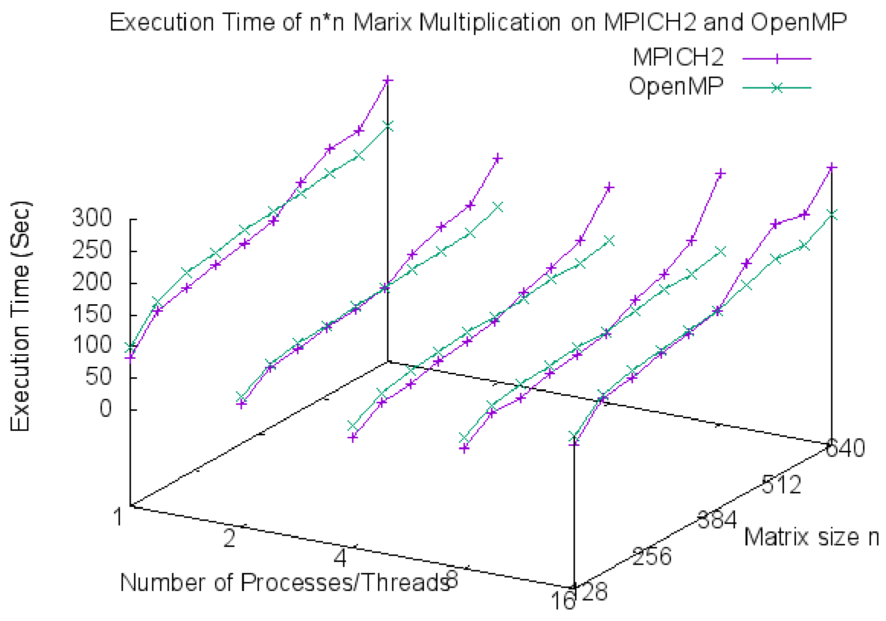

Figure 15: Execution Time of Matrix Multiplication (n x n) Using MPICH2 vs. OpenMP Shows the Conversion Point at (384x384) Matrix Size

Table 10: Execution Time of MMT on MPICH and OpenMP on Different # of Processors/Threads.			
Num of Processes / threads	***Exec. time on mpich1 (millisecond)***	***Exec. time on mpich2 (millisecond)***	***Exec. time on OpenMP (millisecond)***
Size = 128 x 128			
1	87.781	80.436	96.675
2	48.405	42.512	54.623
4	30.323	22.534	39.962
8	46.018	37.389	53.976
16	81.403	75.991	89.482
Size = 256 x 256			
1	147.129	138.769	168.845

Num of Processes / threads	*Exec. time on mpich1 (millisecond)*	*Exec. time on mpich2 (millisecond)*	*Exec. time on OpenMP (millisecond)*
2	077.532	072.879	086.129
4	054.039	050.269	079.136
8	067.022	060.786	089.033
16	128.763	122.648	141.881
Size = 384 x 384			
1	162.543	151.933	192.940
2	087.015	082.933	099.965
4	072.345	066.049	088.997
8	075.595	070.882	093.587
16	139.387	131.612	153.8717
Size = 512 x 512			
1	268.312	244.897	197.634
2	163.469	152.974	109.790
4	139.221	116.214	098.321
8	151.038	137.234	115.554
16	292.520	266.676	201.072
Size = 640 x 640			
1	319.654	293.109	220.154
2	213.574	203.027	126.761
4	193.101	190.285	107.609
8	253.465	243.779	123.609
16	306.825	285.076	212.001
Size = 768 x 768			
1	887.901	840.865	444.901
2	448.869	422.037	220.051
4	426.608	403.133	145.439
8	546.865	513.908	308.432
16	787.166	768.740	410.876
Size = 896 x 896			
1	1469.654	1393.109	680.154

Num of Processes / threads	***Exec. time on mpich1 (millisecond)***	***Exec. time on mpich2 (millisecond)***	***Exec. time on OpenMP (millisecond)***
2	731.463	719.163	341.234
4	642.388	624.498	249.765
8	756.627	717.417	388.801
16	1295.042	1113.259	579.121
Size = 1024 x 1024			
1	2000.129	1980.769	2027.845
2	1194.532	1058.879	0800.129
4	0972.039	0938.269	0421.136
8	0987.022	0961.786	0470.033
16	1098.763	1018.648	0951.881
Size = 2048 x 2048			
1	9495.1936	8165.320	9871.221
2	8234.1425	7705.432	4170.022
4	7374.4335	6501.234	3153.409
8	7478.1800	6887.654	3611.032
16	7684.7530	7192.301	3912.348
Size = 3072 x 3072			
1	33004.312	32654.897	34012.341
2	30012.343	28226.338	15683.412
4	26786.531	24889.059	10718.798
8	27612.391	25449.817	16313.106
16	27998.271	25884.934	19388.321
Size = 4096 x 4096			
1	71789.654	70003.109	73010.154
2	68712.106	66762.134	35032.178
4	62998.804	59250.207	27683.214
8	64660.081	61236.277	36367.731
16	66987.789	62067.714	48979.761
Size = 5120 x 5120			
1	141332.156	139870.865	151764.900
2	130771.310	129622.182	67590.8700

Num of Processes / threads	***Exec. time on mpich1 (millisecond)***	***Exec. time on mpich2 (millisecond)***	***Exec. time on OpenMP (millisecond)***
4	116896.998	115943.011	52098.7600
8	120509.880	118134.567	87958.6767
16	136567.899	134442.371	104934.567

Case 2:

Now we discus the results of the experemints in Case2. In this case the experiments were run to calculate by approximation the value of PI (π) with different number of points in the square ($1 \text{ X } 10^8$, $2 \text{ X } 10^8$, $4 \text{ X } 10^8$, $8 \text{ X } 10^8$ and $16 \text{ X } 10^8$) using multithreading by means of OpenMP and multi-processing (message passing) by 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 of the average running time for all experiments in Case 2 (PI) are summarized in Table 11. This table shows the execution time on MPICH and OpenMP.

Table 11 shows the execution time for computing (π) program running in all states ($1 \text{ X } 10^8$, $2 \text{ X } 10^8$, $4 \text{ X } 10^8$, $8 \text{ X } 10^8$ and $16 \text{ X } 10^8$) on three channels (MPICH1, MPICH2 and OpenMP) versus number of processors (1, 2, 4, 8 and 16) and number of threads (1, 2, 4, 8 and 16). In the five states the experiments where repeated and recorded the elapsed time.

The results in Figure 16 show that the performance using multiprocessing is higher than multithreading and MPICH2 performance is the best. This is due to the small size of data chunks in data distribution and recall the MPICH2 features that have impact on performance: Collective communication routines on master computer, a number of non-blocking routines on each client. And multiple threads on the master.

Table11: Execution Time of PI(π) Computation on MPICH and OpenMP on Different # of Processors/Threads and the Number of Points in the Square is (N)

Num of Processes / threads	*Exec. time on mpich1 (sec)*	*Exec. time on mpich2 (sec)*	*Exec. time on OpenMP (sec)*
$N = 1 X 10^8$			
1	0.891892	0.891885	0.896699
2	0.462869	0.462855	0.783978
4	0.443972	0.442911	0.450789
8	0.488757	0.446034	0.536067
16	0.503249	0.456917	0.544582
$N = 2 X 10^8$			
1	1.780018	1.771238	1.788288
2	0.961435	0.930937	0.996978
4	0.923319	0.884324	0.959076
8	0.930103	0.887004	0.965559
16	0.945534	0.897642	0.991138
$N = 4 X 10^8$			
1	3.541244	3.538235	3.552968
2	1.780001	1.773867	1.786492
4	1.783344	1.765872	1.799389
8	1.796789	1.770511	1.831845
16	1.811341	1.780981	1.854787
$N = 8 X 10^8$			
1	7.097942	7.070931	7.104267
2	3.748843	3.559537	3.976155
4	3.560004	3.530808	3.588692
8	3.579974	3.533668	3.614277
16	3.608152	3.546725	3.641683
$N = 16 X 10^8$			
1	14.784593	14.137508	15.202066
2	7.077461	7.077461	7.588175
4	7.099601	7.058459	7.167871
8	7.179459	7.062334	7.215497
16	7.266179	7.075695	7.429199

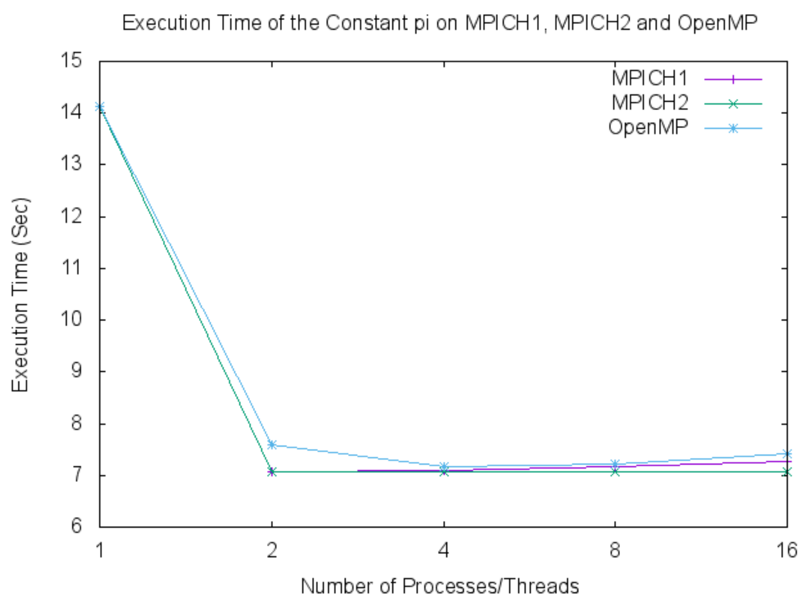

Figure 16: Execution Time of Mathmetical Constant PI (π) (N=16x10^8) Using MPICH1 vs. MPICH2 vs. OpenMP

In addition to the mentined features, the significant optimizations required for efficiency (e.g. asynchronous I/O, grouping and collective buffering) are supported by MPICH2 too. Thus, we can conclude that the added fatures in MPICH2 has positive impact on the performance as in in part 1 of the experiments.

On the same experiment's platform, that has four processing elements, it is clear in : Figure 16 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. Moreover, the execution time using OpenMP is longer than execution time using message passing on all processors. This is due to the overheads in scheduling the threads and processes in utilizing shared resources (i.e. processing elements and shared memories).

Chapter Six

Conclusion

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 and small data using message passing and multithreading.

As a result, we can conclude that MPICH2 speed up perform better than MPICH1 speed up in all cases and MPICH efficiency is higher than OpenMP efficiency when size of matrix A is less than 384 x 384 (18 KB) and vice versa. Because, if size of matrix A bigger than 384 x 384 then the transfer delay will increase, where many collective operations are used in parallel programs that increase execution time when researchers run programs using message passing. In addition, the added features in MPICH2 can affect the improvement possitively. Moreover, the results show that multithreading programming performance on multi-core architectures is higher than message passing when the parallel programs works on data size larger than (18 KB). Can this size be dependent of the computer on which the experiments carried out

So by using our research, if applications that work in parallel way implemented on MPICH2 instead of MPICH1 then researchers and labaratory persons will achieve higher performance and speed up in the computations.

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

Reference:

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

[2] *Science and Technology Support Group High Performance Computing.* 1224 Kinnear Road, Columbus : Ohio Supercomputer Center. OH 43212-1163.

[3] *Towards OpenMP Execution on Software Distributed Shared Memory Systems.* **Ayon Basumallik, Seung-Jai Min, Rudolf Eigenmann.** West Lafayette : School of Electrical and Computer Engineering Purdue University. http: // www.ece.purdue.edu/ParaMount. IN 47907-1285.

[4] *MPI: A Message-Passing Interface Standard, Message Passing Interface Forum.* by the Commission of the European Community through Esprit project P6643 : ARPA and NSF under grant ASC-9310330, the National Science Foundation Science and Technology Center Cooperative, Nov 15, 2003. Agreement No. CCR-8809615.

[5] *OpenMP compiler for a Software Distributed Shared Memory System SCASH.* **Mitsuhisa Sato, Hiroshi Harada and Yutaka Ishikawa.** Real World Computing Partnership, Tsukuba, Ibaraki 305-0032, Japan : s.n. E-mail: fmsato, h- harada, ishikawag@trc.rwcp.or.jp.

[6] WIEN2K. *http://www.wien2k.at/papers/index.html.* [Online]

[7] **David Culler. Jaswinder Pal Singh, Anoop Gupta.** *Parallel Computer Architecture A Hardware / Software Approach.* s.l. : University of California, Berkeley, Princeton University,Stanford University, Aug28, 1997.

[8] **Cottenier, S.** *Density Functional Theory the Family of (L)APW-methods:a step-by step introduction.* August 6, 2004. ISBN 90-807215-1-4.

[9] *an optimal migration algorithm for dynamic load balancing.* **Y.F.HU, R.J.Blake and R.D.Emerson.** Daresbury laboratory, Daresbury, warrington WA4 4AD : Y.F.HU, R.J.Blake and R.D.Emerson, 1998. 10(6), 467- 483.

[10] *Scheduling Dynamically Spawned Processes in MPI-2.* **M´arcia C. Cera1, Guilherme P. Pezzi, Maur´ıcio L. Pilla, Nicolas B. Maillard1, and Philippe O. A. Navaux.** Pelotas, Brazil : Universidade Federal do Rio Grande do Sul, Porto Alegre Brazil and Universidade, 2006.

[11] *Comparing One-Sided Communication with MPI, UPC and SHMEM.* **Maynard, C.M.** JCMB, Kings Buildings, Mayfield Road, Edinburgh : EPCC, School of Physics and Astronomy, University of Edinburgh. EH9 3JZ, UK.

[12] *MPI: A Message-Passing Interface Standard, Version 2.2, and Message Passing Interface Forum.* Sept 4, 2009.

[13] **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.

[14] *Efficient sparse matrix multiple-vector multiplication using a bitmapped format.* **Ramaseshan, Kannan.** s.l. : The University of Manchester, September 2012. ISSN 1749-9097.

[15] *Sparse representation of data.* **Thomas Villmann, Frank-Michael Schleif, and Barbara Hammer.** Germany : University of Applied Sciences Mittweida and Bielefeld University, 2010.

[16] *A Parallel Implementation of Hybridized Merge-Quicksort Algorithm on MPICH.* **Haryanto, Rahmadi Trimananda and Christoforus Yoga.** Indonesia : Computer Engineering Department, Universitas Pelita Harapan, 2010.

[17] *Cilk vs MPI: Comparing two very different parallel programming styles.* **Morris, Sonny Tham and John.** s.l. : International Conference on Parallel Processing, IEEE, 2003. (ICPP'03).

[18] *MPI: A Message-Passing Interface Standard, Version 2.1, and Message Passing Interface Forum.* June 23, 2008.

[19]*http://lists.mcs.anl.gov/pipermail/mpich-discuss/2007-December/002982.html.* [Online]

[20] *MPI-2: Extensions to the Message-Passing Interface, and Message Passing Interface Forum.* s.l. : This work was supported in part by NSF and DARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards, Nov 15, 2003. 21111.

[21] *Optimum Execution For WIEN2K using Parallel Programming Models (Comparison Study).* **Rezek Mohammad, Areej Jabir, and Rashid Jayousi.** Jerusalem, Palestine : Department of physics, Palestinian Technical University/Khadoorie, Middle East Technical University and department of Computer Science, Al-Quds University, 2011

[22] *A new Approximation Method in the Problem of Many Electrons.* **Hans, Hellmann** Moscow : Journal of Chemical Physics (Karpow-Institute for Physical Chemistry), 1935.

[23] *Electronic Structure and the Properties of Solids.* **Harrison, Walter Ashley.** 1989.

[24] **Kresse, G. and Furthmuller.** *Comp. Mat. Sci.6.* 1996. B 5, 11169.

[25] **Shoaib Akram, Rakesh Kumarand Deming Chen.** *Workload Adaptive Shared Memory Multicore Processors with Reconfigurable Interconnects.* Urbana Champaign : Department of Electrical and Computer Engineering, University of Illinois, 2009.

[26] *An Adulatory Theory of the Mechanics of Atoms and Molecules.* **E, Schrodinger** : Physical Review 28, 1926. 1049-1070.

[27] *A Survey of Parallel Computer Architectures.* **Duncan, Ralph.** s.l. : IEEE Compute, 1990.

[28] *Introduction to Parallel Computing.* **Blaise Barney, Lawrence Livermore National Laboratory.** s.l. : Available in the link https://computing.llnl.gov/tutorials/parallel_comp., 14 July 2014.

[29] *Performance Comparison of Open Source MPI Implementations.* **McClements, Erik.** s.l. : the University of Edinburgh, 2006.

[30] **Darius Buntinas, Guillaume Mercier, and William Gropp,.** *Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem.* s.l. : Mathematics and Computer Science Division, Argonne National Laboratory, 2008.

[31] **Xiaojun Ruan, Qing Yang, Mohammed I. Alghamdi, Shu Yin, Zhiyang Ding, Jiong Xie, Joshua Lewis, and Xiao Qin.** *ES-MPICH2: A Message Passing Interface with*

Enhanced Security. Al-Baha City, Kingdom of Saudi Arabia : Department of Computer Science and Software Engineering, Auburn University and Department of Computer Science, Al-Baha University, 2010. AL 36849-5347.

[32] **Wright, Charles.** *Hybrid Programming Fun: Making bzip2 Parallel with MPICH2 & pthreads on the Cray XD1.* s.l. : Alabama Supercomputer Center, 2006.

[33] *Design Considerations for Shared Memory MPI Implementations on Linux NUMA Systems: An MPICH/MPICH2 Case Study.* **Mucci, Per Ekman and Philip.** Stockholm, Sweden : PDC/KTH, 2005.

[34] *NUMA-aware shared-memory collective communication for mpi.* **Shigang Li, Torsten Hoefler and Marc Snir.** Beijing, Urbana-Champaign : School of Computer and Communication Engineering University of Science and Technology and Department of Computer Science, ETH Zurich and Department of Computer Science, University of Illinois and Argonne National Laboratory, 17 Jun 2013.

[35] *Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI.* **Camille Coti, Thomas Herault, Pierre Lemarinier and Laurence Pilard.** France : INRIA-Futurs / Grand-Large, Laboratoire de Recherche en Informatique, Universit´e Paris-XI, 2006.

[36] *Adaptive Strategy for One-sided Communication in MPICH2.* **Xin Zhao, Gopalakrishnan Santhanaraman, and William Gropp.** USA : University of Illinois, 2012. IL 61801.

[37] *Multi-core Aware Optimization for MPI Collectives.* **BiboTu, Ming Zou, Jianfeng Zhan, Xiaofang Zhao and Jianping Fan.** Beijing : Institute of Computing Technology, Chinese Academy of Sciences, 2008. 100190.

[38] *Asynchronous MPI for the Masses.* **Markus Wittmann, Georg Hager, Thomas Zeiser, and Gerhard Wellein.** Germany : Erlangen Regional Computing Center, University of Erlangen-Nuremberg, Martensstraße, 2013.

[39] *Comparison of Shared memory based parallel programming models.* **Ravela, Srikar Chowdary.** Sweden : School of Computing Blekinge Institute of Technology Box 520, 2010. SE – 372 25.

[40] **Ananth Grama, George Karpis, Vipin Kumar and Anshul Gupta.** *Introduction to Parallel Computing.* 2nd Edition. Feb, 2003. ISBN13 9780201648652, ISBN10 0201648652

[41] *Comparison between CISC and RISC.* **Yi Gao, Shilang Tang, Zhangli Ding.** 2000.

[42] *Multi-core and Many-core Processor Architectures.* **Brorsson, Mats.** s.l. : A. Vajda, Programming Many-Core Chips, Springer Science and Business Media, LLC, 2011. 1-4419-9739-5.

[43] *MPICH2 Installer's Guide, Version 1.5, Mathematics and Compute Science Division Argonne National Laboratory.* **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.** October 8, 2012

[44] *Simulation in Flight Simulator with the Hybrid Distributed-Shared Memory Architecture.* **Kvasnica, P., Páleník, T.** s.l. : ASIS, 2009. ISBN 978-80-86840-47-5.

Appendix 1: The Predefined MPI Datatypes and Their Corresponding C/ C++ Datatypes and the Replaced Constructs by MPICH2.

Table 12: The MPI Predefined Datatypes, and their Corresponding C/C++ Datatypes [5], [12], [19].

No	MPI DataTypes	C DataType	C++ DataType
1	MPI::CHAR	char	Char
2	MPI::SHORT	signed short	signed short
3	MPI::INT	signed int	signed int
4	MPI::LONG	signed long	signed long
5	MPI:: LONG_ LONG	signed long long	signed long long
6	MPI::SIGNED_CHAR	signed char	signed char
7	MPI::UNSIGNED_CHAR	unsigned char	unsigned char
8	MPI::UNSIGNED_SHORT	unsigned short	unsigned short
9	MPI::UNSIGNED_INT	unsigned int	unsigned int
10	MPI::UNSIGNED_LONG	unsigned long	unsigned long int
11	MPI::UNSIGNED_LONG_LONG	unsigned long long	unsigned long long
12	MPI::FLOAT	float	Float
13	MPI::DOUBLE	double	Double
14	MPI::LONG_DOUBLE	long double	long double
15	MPI::BOOL		Bool
16	MPI::COMPLEX		Complex<float>
17	MPI::DOUBLE_COMPLEX		Complex<double>
18	MPI::LONG_DOUBLE_COMPLEX		Complex<long double>
19	MPI::WCHAR	wchar_t	wchar_t
20	MPI::BYTE		
21	MPI::PACKED		

Table 13: The Replaced Constructs by MPICH2 [5], [12], [19].

	Deprecated	MPICH2 Replacement
1	MPI_ADDRESS	MPI_GET_ADDRESS
2	MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
3	MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
4	MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
5	MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
6	MPI_TYPE_UB	MPI_TYPE_GET_EXTENT
7	MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
8	MPI_LB	MPI_TYPE_CREATE_RESIZED
9	MPI_UB	MPI_TYPE_CREATE_RESIZED
10	MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
11	MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
12	MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
13	MPI_HANDLER_FUNCTION	MPI_COMM_ERRHANDLER_FUNCTION
14	MPI_KEYVAL_CREATE	MPI_COMM_CREATE_KEYVAL
15	MPI_KEYVAL_FREE	MPI_COMM_FREE_KEYVAL

	Deprecated	**MPICH2 Replacement**
16	MPI_DUP_FN	MPI_COMM_DUP_FN
17	MPI_NULL_COPY_FN	MPI_COMM_NULL_COPY_FN
18	MPI_COPY_FUNCTION	MPI_COMM_COPY_FUNCTION_ATTR
19	COPY_FUNCTION	COMM_ATTR_COPY_FN
20	MPI_DELETE_FUNCTION	MPI_COMM_DELETE_ATTR_FN
21	DELETE_FUNCTION	COMM_DELETE_ATTR_FN
22	MPI_ATTR_DELETE	MPI_COMM_ATTR_DELETE
23	MPI_ATTR_GET	MPI_COMM_ATTR_GET
24	MPI_ATTR_PUT	MPI_COMM_ATTR_PUT

Appendix 2: Density Functional Theory (DFT)

In physics, a collection of heavy positively charged particles (nuclei) and lighter negatively charged particles (electrons) is called a solid. Solids obey the laws of quantum mechanisms. By solving these equations, all of properties of solids like structural, thermodynamic, mechanical, transport properties and electronic properties are determined. If we have N nuclei and Z electrons for each nucleus then we will deal with a problem of N+ZN electromagnetically interacting particles. Any material composed of many atoms combined together according to the chemical bonding. These atoms can take many positions while keeping the same total number of atoms of the material. Each stable of combinations gives different properties [26]. This is a quantum many-body problem, and the particles are so light.
In science of material, stability of any material is measured via main scalable quantity, which is called cohesive energy. Cohesive energy equals the difference between the total energy of the material in combined form and the sum of the free atom's energy in their free state as shown in equation (2.1)

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

Each stable order of these atoms can produce positive value for the cohesive energy. For the material to match the stability it normally takes more than one phase and the phase with the highest cohesive energy is the most stable one, see Figure 17, which are drawn using WIEN2K package [26].

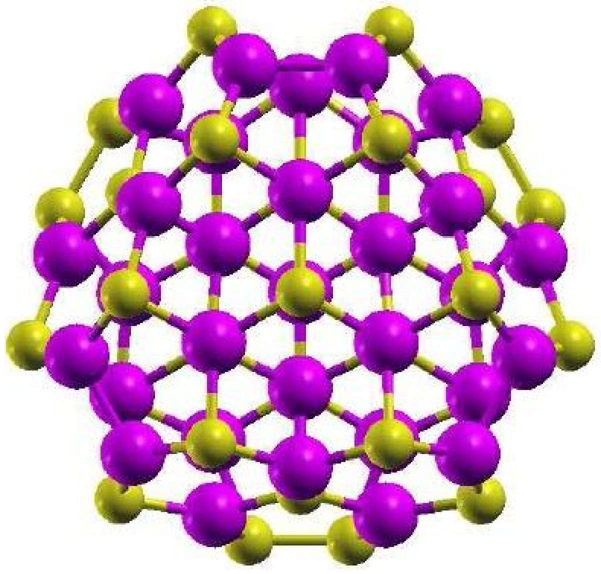

Figure 17: Schematic Diagram of Simple Cubic Phase along 111 Direction

In practice, applying quantum mechanisms in order to achieve stability is very hard, numerical task that consumes time even for idealized cases. In these calculations, all the atomic interactions can be done by scalar value model taken from experimental results. This model and others are used to explain properties of materials already exist in the laboratory: hence, some of famous methods were used to solve like this problem:

1- Pseudo potential method (PP) was first introduced by Hans Helman (1930) [22], in an attempt to replace the complicated effect of core electrons on the atomic potential. This is used to fit the experimental data about the material. In many cases many forms of potential can be used, for each form of the material; different potential can be used to give the experimental data.
2- Tight binding method (TB) was introduced in 1960 [23]. The value of the interaction between the **valence electrons**is replaced by a **numeric value**. The value of this number is predicted from already known experimental data, as in the PP method (pseudo potential). The value of the same interaction differs from form to form for the same material.

In density functional theory, the stability of a solid can be affected by: the kinetic energy operators for the nuclei and for the electrons, potential energy between electrons and nuclei and potential energy between nuclei and other nuclei; these factors are measured by exact many-particle Hamiltonian for the system, which is illustrated in [8]:

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

M_i: The mass of the nucleus at $\vec{R}$.
M_e: The mass of the electrons at $\vec{r}$.
The first term: is the kinetic energy operator for the nuclei.
The second term: is the kinetic energy operator for the electrons.
The third term: the Coulomb interaction (potential energy) between electrons and nuclei
The fourth term: the Coulomb interaction (potential energy) between electrons and other electrons.
The fifth term: the Coulomb interaction (potential energy) between nuclei and other nuclei.

In order to attain stability and find acceptable approximate eigenstates (eigenvalues and eigenvectors) for a system with a reasonable calculation time, we will need to make approximations at different levels:

1- Level 1: The Born-Oppenheimer Approximation

The nuclei are much heavier and therefore much slower than the electrons. Born and Oppenheimer can hence `freeze' them at fixed positions and assume the electrons to be in instantaneous equilibrium with them. In other words, only the electrons are kept as players in the many body problems. The nuclei are excluded from this status, reduced to a given source of positive charge and therefore become `external' to the electron cloud. After having applied this approximation, they are left with a collection of NZ interacting negative particles, moving in the (now external or given) potential of the nuclei.

The results of using Born-Oppenheimer approximation on the Hamiltonian (equation 2.2) are: The nuclei do not move any more, their kinetic energy is zero and the first term disappears. The last term reduces to a constant. We are left with the kinetic energy of the electron gas, the potential energy due to electron-electron interactions and the potential energy of the electrons in the (now external) potential of the nuclei. We write this as represented in the equation below [8]:

$$\hat{H} = \hat{T} + \hat{V} + \hat{V}_{\text{ext}} \qquad (3)$$

$\hat{T}$: The kinetic energy of the electron gas.

$\hat{V}$: The potential energy due to electron-electron interactions.

$\hat{V}$ext: The potential energy of the electrons in the (external) potential of the nuclei.

2- Level 2: Density Functional Theory Approximation

Together with the Development of theoretical schemes like Density Functional Theory (DFT) [8] by Hohenberg and Kohn and the fast cheap computers have helped to change the situation. Another name for such calculations is called ab-initio calculation. Such calculation forms the basic information like the form of material and the name of the atoms. Nowadays, many packages are using the DFT such as WIEN2K [6], VASP [24], Gaussian [25]….etc. In these packages and studies, we have two factors controlling such calculation:

1- The sample actuality.
2- The time of calculation.

Number of atoms constituting the sample and their distribution are called the sample actuality; the bigger number of atoms in study case will cost a lot of calculation time, so the relation between the two factors are vice versa.

In this study, we will focus on the WIEN2K program and on the order structure of atoms that are named “Crystal” in solid state physics as shown in Figure 17, the crystal is composed of a definite number of atoms, which has a definite position in space. The rest of the crystal is an empty space. The space between the atoms in the crystal is called interstitial region [22], as shown in Figure 18. This adaptation is achieved by dividing the unit cell into (I) non-overlapping atomic spheres (centered at the atomic sites) and (II) an interstitial region.

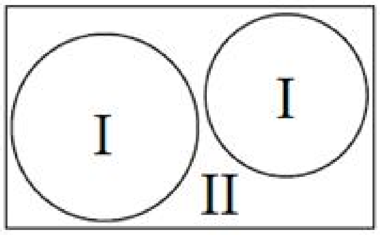

Figure 18: Partitioning of the Unit Cell into Atomic Spheres (I) and an Interstitial Region (II)

Experiments have proven that the outer shell electrons of the atoms are responsible to define the physical and chemical properties of the atoms and its compounds. The net interactions between the repulsive and attractive forces between different atoms (electrons and their nuclei) decide which phase these atoms will take to attain stability. Each atom composes of a big number of electrons and one nucleus, each electron interacts with all the other electrons and with each positive nucleus. These interactions can only be treated and analyzed using quantum mechanics treatment.

The quantum many body problems obtained after the first level approximation (Born-Oppenheimer) is much simpler than the original one, but is still far too difficult to solve. Several methods exist to reduce equation 2.3 to an approximate but tractable form. Such as Density Functional Theory (DFT). DFT has been formally established by two theorems due to Hohenberg and Kohn [8]. The traditional formulation of the two theorems of Hohenberg and Kohn is as follows [8]:

First theorem: *There is a one-to-one correspondence between the ground-state density* ρ(r) *of a many-electron system (atom, molecule, solid) and the external potential Vext. An immediate consequence is that the ground-state expectation value of any observable Ô is a unique functional of the exact ground-state electron density:*

$$< \Psi |\hat{O}| \Psi > = O[\rho] \qquad (4)$$

Second theorem: *For Ôbeing the Hamiltonian Ĥ, the ground-state total energy functional* *H*[ρ] = *EVext*[ρ] *is of the form:*

$$E\text{Vext}[\rho] \;=\; <\Psi\,|\hat{T} + \hat{V}|\,\Psi> + <\Psi\,|\hat{V}\text{ext}\,|\,\Psi> \qquad (5)$$

Many body problems can only be solved in DFT by making use of the translational symmetry, which cause the electronic wave functions to be of Bloch-type, labeled by k-vector in reciprocal space and the quantum number of the electron. Thus, the periodicity in real space is defined by k-vector in reciprocal space, whose unit cell is called Brillouin Zone (BZ). The latter becomes the smaller and the larger real space unit cell gets [21], [26], [30]. The interaction between the electrons and nucleus can be presented through the one electron Schrödinger Equation [26]:

$$H_{ks}\,\Psi = \text{E}\,\Psi \qquad (6)$$

$$(-\nabla^2 + V_{xc}\,)\,\Psi = \text{E}\,\Psi \qquad (7)$$

∇^2: is the second derivative with respect to space coordinates.
V_{xc}: is the effective attractive potential each electron feel.
E: is the energy of this electron in this crystal phase.
Ψ: is the wave function of this electron.

When Ψ is squared and summed over all the crystal space we get the density function of this electron as a function of position:

$$\rho(\text{r}) = \int \Psi\ \Psi^* dr^3 \qquad (8)$$

Adding this density function for all the electrons, the sum logically equals the total number of electrons in the interaction. The problem is that we do not know the actual V_{xc} and Ψ. This problem is treated in DFT by giving initial wave function Ψ and this wave function is extremely close to atomic wave function. Later we solve the Schrödinger equation and finding the V_{xc} from the equation [8]:

$$\nabla^2 V_{xc} = \rho(\text{r}) \qquad (9)$$

The exchange-correlation operator V_{xc} depends on the density ρ(r), which in turn depends on the Ψi that are being searched. This means we are dealing with a *self-consistency problem*. This V_{xc} is new V_{xc} and it entered again to the Schrödinger equation and again we solve for the new Ψ and so on. This cycle is kept repeated until the total energy reaches a minimum value. This minimum energy value is chosen at the beginning of the calculation; it should be suitable and comparable to the size of the problem. The value of this energy is directly related to the time of calculation through the number of cycles needed. At this optimum energy, the wave function Ψ and exchange correlation potential V_{xc} are the optimum representative for all the electrons see Figure 1, [8]. Some starting density $P0$ is guessed, and a Hamiltonian $HKS1$ is constructed with it. The eigenvalue problem is solved, and results in a set of $P1$ from which a density $P1$ can be derived. Most probably $P0$ will differ from $P1$. Now $P1$ is used to construct $HKS2$, which will yield a $P2$, etc. The procedure can be set up in such a way that this series will converge to a density Pf which generates a $HKSf$ which yields as solution again Pf: this final density is then *consistent* with the Hamiltonian [8].

These sequential operations of the WIEN2K program are divided into five modules:

1. The first module is called LAPW0, in this process the V_{xc} is calculated in the crystal from the initial.
2. The second module is called LAPW1, which is responsible for building the Schrödinger equation (setting up H and S matrix), and solves the generalized eigen value problem for special point in the BZ. These points are called K-points. The number of these points is proportional to the reality of the study. The high number gives results that are more accurate and costs a lot of computational time, so balanced is essential.
3. The third module in the program is called LAPW2. In this process and after solving the Eigen value problem, the Eigen vectors $\Psi1$ is calculated for each Eigen value and the new density is calculated according to Equation (2.5)
4. The fourth module is called LCORE: from the density function, the electrons in the crystal are distributed on the lowest energy values, the density function for the core electrons is also calculated and in LCORE process.
5. The fifth module is called MIXER: the new total density is compared with the old density, if the values are the same; the self-consistent (SC) is finished. The total energy and wave functions of the electrons are found. Otherwise, the new density is mixed with old density with a percentage decided at the beginning of the calculation to reproduce a new density to run another cycle.

The cycle (visit of the five modules) is repeated until we get the read difference between the total energy and the new total energy, less than a value already expected.

The Linearized Augmented Plane Wave (LAPW) method has proven to be one of the most accurate methods for the computation of the electronic structure of solids within density functional theory. A full-potential LAPW-code for crystalline solids has been developed over a period of more than twenty years. A first copyrighted software version for the computation of the electronic structure of solids within DFT was called **WIEN** and was published by P. Blaha, K. Schwarz, P. Sorantin, and S. B. Trickey [22]. After that significant improvements and updates were accomplished on the UNIX original version of WIEN2k. Consequently, sequence of versions were issued and known as WIEN 93, WIEN 95 and WIEN 97.

Now a new version, WIEN2K, is available, which is based on an alternative basis set. This allows a significant improvement, especially in terms of speed, universality, user-friendliness and new features. WIEN2Kis written in FORTRAN 90 and requires a UNIX operating system since the programs are linked together via C-shell scripts. It has been implemented successfully on the following computer systems: Pentium systems running under Linux, IBM RS6000, HP, SGI, Compac DEC Alpha, and SUN. It is expected to run on any modern UNIX (LINUX) system [22]. WIEN2K has the several features that are new with respect to **WIEN 97**.

In our work, the WIEN2K package is used to study the physical, chemical, electrical, structural and electronic properties of the materials, so when we run the WIEN2K then, we will compute the electronic structure of solids within DFT. The WIEN2K can simulate physical and chemical systems supposed to form a new material, this is very necessary to the laboratory person, who can produce the desired material such as drug and medicine.

Appendix 3: Publications

We published two papers in the thesis as the following citations and they are available in the next pages respectively:

1. Hadi Khalilieh, Nidal Kafri and Rezek Mohammad. 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). **Published**

2. Hadi Khalilieh, Nidal Kafri and Rezek Mohammad. The International Conference on Digital Information, Networking, and Wireless Communications (**DINWC 2014**) ISBN: 978-0-9891305-6-1 ©2014 SDIWC, VSB-Technical University of Ostrava, Czech Republic June 24-26, 2014. **Published**

First Publication in:

International Journal of New Computer Architectures and their Applications

(IJNCAA)

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, Multicore 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, this 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, where in 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 consequently 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 workshop 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	**Distributed Memory MPI**	**Shared Memory Arch OpenMP**	**Hybrid Dist. & Shared Memory**
Creation mathematical model	Easy	Slightly complicated	Difficult
Balancing	Changeable with Difficulties	Changeable-easily	Easily changeable
Simulation of parallel models	Advisable	Convenient	Useful
Synchronization Models	Simple	Complicated	Complicated
Transfer dates between models	Large	Little	Intermediate
Power of large modules	Reasonable	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 does not 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 from 50 years ago. Scientist knows 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].

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

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

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.

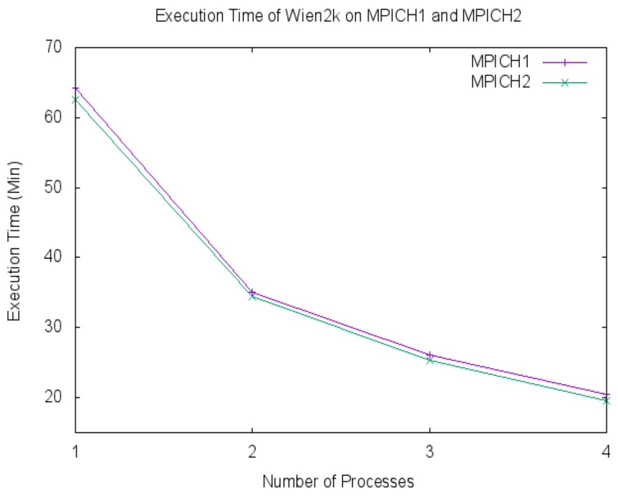

Figure 2: the WIEN2K execution time of MPICH1 vs. MPICH2.

Case 2:

In this case the experiments were implemented on a standard parallel matrix multiplication of size 5120 x 5120 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).

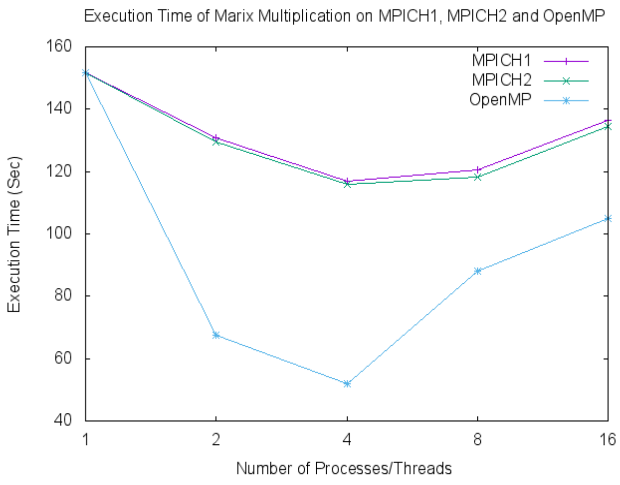

Fig 3: Execution Time of Matrix Multiplication Using MPICH1 vs. MPICH2 vs. OpenMP

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

<u>Second Publication in:</u>

The International Conference on Digital Information, Networking, and Wireless Communications

(DINWC 2014)

Evaluation of WIEN2K Performance on MPICH2 vs. MPICH1

Hadi Khalilieh and Nidal Kafri

Department of Computer Science, Al-Quds University, Jerusalem, Palestine

hkhalilia1@science.alquds.edu, nkafri@science.alquds.edu

Rezek Mohammad

Palestinian Technical University/Khadoorie, Middle East Technical University/physics department

esteteh@hotmail.com

ABSTRACT

The emerging multi-core computer architecture attracts the researchers to utilize this architecture as an adequate and inexpensive solution to achieve high performance computation for many problems. Where, the multi-core architecture enables us to implement shared memory and/or message passing parallel processing paradigms. Therefore, we need appropriate standard software libraries in order to utilize the resources efficiently for a given computational problem.

In this work, we evaluate the performance of two versions of the well-known massage passing interface (MPI) library: MPICH1 vs. MPICH2. In our experiments, we used two benchmarks. The first one is the WIEN2K application, which is based on Density Function Theory, and the second is a Matrix multiplication. The results show that we achieve better performance when MPICH2 is used than MPICH1.

KEYWORDS

Parallel Processing, Message Passing Interface MPI, MPICH1, MPICH2, performance, 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 scientific computing, engineering, multimedia application, industry, computer systems, statistical applications, and simulation. One of the important applications that need to speed up computation is WIEN2K application, which is base on Density Functional theory.

Usually parallel processing can be implemented on shared memory computer systems or distributed memory systems using message-passing paradigms. A hybrid approach using both paradigms also can be implemented. Parallel processing was usually carried out on expensive supercomputers and mainframes. After that, the emerging high performance computer network and protocols attracted the researcher to use the distributed memory parallel processing on clusters of on shelf computers and Grid computing.

In the past decade, the development of multicore Systems 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 single or a cluster of multi-core computers[2][3]. In order 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. Also Posix and OpenMP are used for multithreading on shared memory [3]. It should be noted that these libraries provide us with 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 evaluating the performance of different versions of MPI library namely MPICH1 and MPICH2. Since WIEN2K is currently using MPICH1.

The WIEN2K can simulate physical and chemical systems supposed to form a new material, this is very necessary to the laboratory person, who can produce the desired material such as drug and medicine [8]. The WIEN2K applied a parallel method to solve quantum mechanics equations based Density Functional Theory (DFT) to find the cohesive energy of any material.

In this work, we evaluated the performance of MPICH1 and MPICH2 by running WIEN2K that originally uses MPICH1 and the new implementation of WIEN2K on MPICH2 as benchmark. Moreover, we implemented a matrix multiplication on both MPICH1 and MPICH2.

This paper is organized as follows: Section 2 reviews the main difference between MPICH1 and MPICH2**.** In section 3, literature review and background are introduced. Next section (4) discusses the experiment and the results. Finally, a conclusion and future work are provided in section 5.

2. PRELIMINARIES

Multi-core systems and clusters become an interesting and affordable platform for running parallel processing to achieve a high performance computing for many applications and experiments. For instance: internet service, database, scientific computing and simulation. This is due to their scalability performance/cost ratio [1].

On the other hand, there are many Libraries to support the shared and distributed memory. The message passing interface (MPI) is a set of API functions that enable programmers to write parallel programs based on message passing paradigm. One of the well known APIs MPICH1 which established based on MPI standard that founded in April 29-30, 1992 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]. Of course, a number of changes about how you run them, dynamic spawning tasks and the nature of communication will be different. By new added features in MPICH2, we will get it 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 to enable 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,

MPICH2 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 does not support MPI THREAD MULTIPLE [5].
8. **MPICH1** does not concern with communication rather than process management. But, **MPICH2** concerns 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 and providing 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, so MPICH1’s configure devoted 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 you need them in your applications, you will have to ensure that they are available in the environment you are using [15].

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

3. LITERATURE REVIEW AND BACKGROUND

Materials are build from atoms, atoms composed of a heavy positively charged nucleus and lighter particles called electrons. These particles interact with each other and with their neighbors in the next atoms. In order to study the stability, structural, thermodynamic, mechanical, transport properties and electronic properties of these materials we have to solve many-body second order deferential equation called equation of state, this equation obeys the laws of quantum mechanisms.

The equation of state composed of the kinetic energy operators for both the nucleus and electrons, potential energy resulted from interaction between electrons them self, nuclei’s them self and nuclei’s and electrons; these operators are measured by solving many body Hamiltonian for the system, which is illustrated in equation (1) [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].

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

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

In Our work, the program packages like WIEN2K [7], using Full potential –Linear Augmented Plane Wave And Local Orbital’s (FP-LAPW+Lo) technique is used, in such studies we have two main factors controlling the calculation, these two factors are vice versa, the first factor is the time of calculation and the second is the sample actuality, the sample actuality means here the number of atoms constituting the sample, the bigger the number is the more actual case we have, and more complexity, this will cost a lot of calculation time. WIEN2K package composed of five modules, each module solve one of the equations from (2) to (5) sequentially:

- The first module is called **LAPW0**, in this process the V_{xc} is calculated in the crystal from the initial density P_0 using poisons equation:

$$\nabla^2 V_{xc} = \rho(r) \qquad (2)$$

- The second and third module is called **LAPW1**, **LAPW2** which are responsible for building and solving the Schrödinger equations (3) and (4), (setting up H and S matrix), and solves the generalized Eigen value problem for special point in the crystal. The number of these points is proportional to the reality of the study. The high number gives more accurate results and costs a lot of computational time, so Balanced is essential.

$$H_{ks}\Psi = E\,\Psi \qquad (3)$$

$$(-\nabla^2 + V_{xc})\,\Psi = E\,\Psi \qquad (4)$$

∇^2: is the second derivative with respect to space coordinates.
V_{xc}: is the effective attractive potential each electron feel.
E: is the energy of this electron in this crystal phase.
Ψ: is the wave function of this electron.

- The fourth module is called **LCORE**: from the density function, the electrons in the crystal are distributed on the lowest energy values, the density function for the core electrons is also calculated and in LCORE process as in equation (5):

$$\rho(r) = \int \Psi\,\Psi^* dr^3 \qquad (5)$$

- The fifth module is called **MIXER**: the new total density is compared with the old density, if the values are the same or the difference is less than an assigned value; the self-consistent (SC) is finished as shown in Figure 1. The total energy and wave functions of the electrons are found. Otherwise, the new density is mixed with old density with a percentage decided at the beginning of the calculation to reproduce a new density to run another cycle to get faster convergence and recalculate V_{xc} using equation (2).

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 combined form and the sum of the free atom's energy in their free state as shown in equation (6)

$$\boldsymbol{E}_{\text{cohesive energy}} = \boldsymbol{E}_{\text{compound}} - \sum \boldsymbol{E}_{\text{free atoms}} \qquad (6)$$

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

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. But 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].

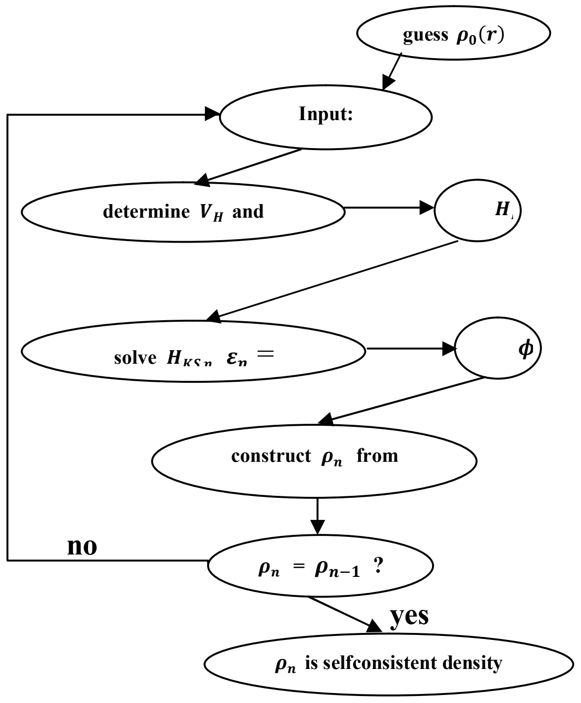

Figure 1: Physical problem solving steps

4 EXPERIMENT AND RESULTS DISCUSSION

In our study, 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. The main contribution in our work depends on the comparison between the results of these experiments.

Our experiments were running on Linux (Fedora 14) installed on multi-core (quad) machine (Intel Core i5 3GHz processor); the specification details of the experiments platform/machine are listed in Table 1.

Table 1: 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 2.

Table 2: 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

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 experiment was carried out by running the programs (LAPW0 and Matrix Multiplication) using MPICH1 and 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 were recorded. The experiments in divided into two cases: the first one is running LAPW0 for one cycle, and in the second case is the running of Matrix multiplication.

```
[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.
```

Fig 2 : Screen Shot of Running LAPW0 on MPICH2

It should be noted that for running the experiments on MPICH1 we use "mpirun" command and “mpiexec” for running it on MPICH2. For example, the steps of the LAPW0 execution on MPICH2 are shown in Figure (2).

The results of the average running time for case 1 (LAPW0) are summarized in table 3. This table shows the execution time on MPICH1 and MPICH2 and the improvement factor (*if*) by the number of processors. Where 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., $(T_{MPICH1}-T_{MPICH2})/ T_{MPICH1}$.

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

It is clear that the performance of MPICH2 is better than MPICH1 by approximately 3%. Also, Figure 3 shows the difference between the execution time on MPICH1 and MPICH2.

Table 3: 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

Recall that in case 2 matrix multiplication program for matrices of size (5120 x 5120) were running using MPICH1 and MPICH2 on one, two, three, and four processors. The results of the average running time are summarized in table 4 and depicted in Figure 4. Again it is clear that the performance of MPICH2 is better than MPICH1.

The results of the experiments in case 1 and case 2 assess the improvement of MPICH2 over MPICH1, which has significant results on the performance and efficient utilization of resources. Note that the time units in case 1 are in minutes, whereas it is in seconds in case 2.

Consequently, in all cases MPICH2 is better than MPICH1. 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.

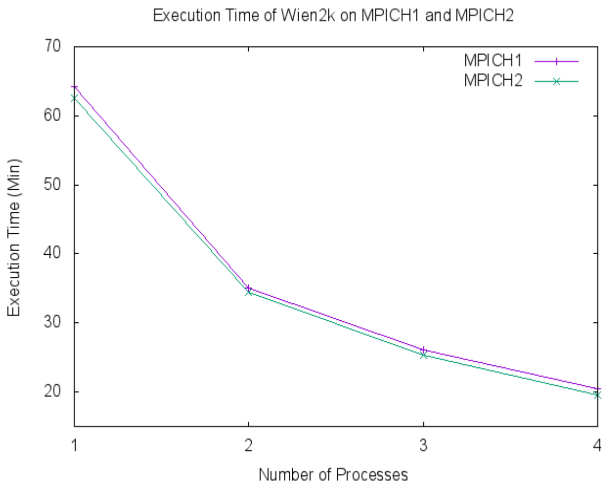

Fig 3: the WIEN2K execution time of MPICH2 vs. the execution time of MPICH1.

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

# of Proc	*Exec. time on mpich1 (sec)*	*Exec. time on mpich2 (sec)*	*If*
1	92.357	89.562	0.030263
2	63.109	61.776	0.021122
3	60.910	59.113	0.029503
4	57.965	55.935	0.035021

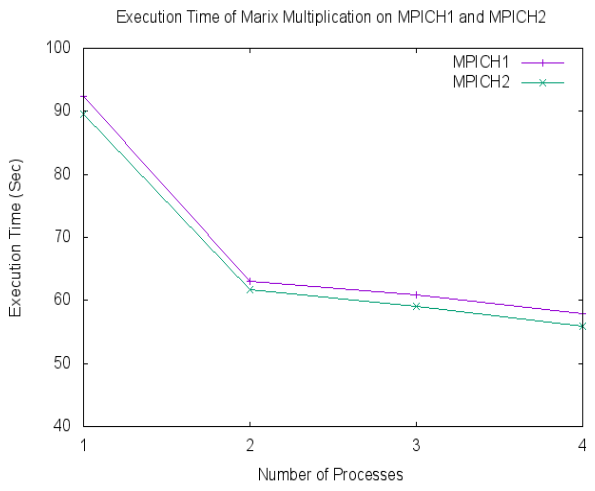

Fig 4: Execution Time of Matrix Multiplication Using MPICH1 vs. MPICH2

CONCLUSION AND FUTURE WORKS

The goal of this work is to evaluate and compare the performance of MPICH1 and MPICH2 using different cases running on one, two, three, and four processors. As a result, we can conclude that MPICH2 perform better than MPICH1. This is due to the collective improvement and added features in MPICH2.

Finally, as a future work we intend to extend our experiment to test the performance of newly issued MPICH3 using different tasks.

REFERENCES:

[1] Sherihan Abu ElEnin, Mohamed Abu ElSoud,” Evaluation of Matrix Multiplication on an MPI Cluster” Faculty of computers and Information, Mansourauniversity, 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