İSTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

PERFORMANCE ANALYSIS OF PDE BASED PARALLEL ALGORITHMS ON DIFFERENT COMPUTER ARCHITECTURES

M.Sc. Thesis by
Ilker KOPAN

Department : Informatics Institute

Programme : Computational Science and Engineering

DECEMBER 2009

İSTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

PERFORMANCE ANALYSIS OF PDE BASED PARALLEL ALGORITHMS ON DIFFERENT COMPUTER ARCHITECTURES

M.Sc. Thesis by
İlker KOPAN
(702051008)

Date of submission : **06 June 2009**
Date of defence examination: **07 October 2009**

Supervisor (Chairman) : **Prof. Dr. M. Serdar ÇELEBİ (ITU)**
Members of the Examining Committee : **Prof. Dr. Hasan DAĞ (KHAS)**
Yrd.Doç.Dr. Lale TÜKENMEZ ERGENE (ITU)

DECEMBER 2009

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

PARALEL KISMİ FARK DENKLEMLERİ FARKLI BİLGİSAYAR MİMARİLERİNDE PERFORMANS ANALİZİ

YÜKSEK LİSANS TEZİ
Ilker KOPAN
(702051008)

Tezin Enstitüye Verildiği Tarih : 06 Haziran 2009
Tezin Savunulduğu Tarih : 07 Ekim 2009

Tez Danışmanı : Prof. Dr. M. Serdar ÇELEBİ (ITU)
Diğer Jüri Üyeleri : Prof. Dr. Hasan DAĞ (KHAS)
Yrd.Doç.Dr. Lale TÜKENMEZ ERGENE (ITU)

ARALIK 2009

FOREWORD

I would like to express my deep appreciation and thanks for my advisor and my teachers. Also I want to thank NCHPC support team, our assistants and my classmates for encouraging me. Our friendship with Erdem Üney, Sayat Baronyan, Deniz Güvenç and Barış Avşaroğlu started as grad students, after graduation it will become a life time friendship. The last but certainly not the least person I would like to acknowledge is my love Şirin Özdilek. I am very grateful for your presence and continuous support since the beginning.

SEPTEMBER 2009

İlker Kopan

(Computer Engineer)

TABLE OF CONTENTS

Page

ABBREVIATIONS

TAU : Tuning and Analysis Utilities
MPI : Message Passing Interface
App : Appendix
PDE : Partial Differential Equation
SMP : Symmetric MultiProcessing
NCHPC : National Center for High Performance Computing Turkey
NUMA : Non-Uniform Memory Access
SOR : Successive over-relaxation
HPC : High Performance Computing
OpenMP : Open Multi-Processing
CPU : Central Processing Unit
UMA : Uniform Memory Access
NUMA : Non-Uniform Memory Access
TLB : Translation Look aside Buffer
IBA : Infiniband Architecture
RDMA : Remote Direct Memory Access
SDP : Sockets Direct Protocol
PDT : Program Database Toolkit

LIST OF TABLES

Page

LIST OF FIGURES

Page

PERFORMANCE ANALYSIS OF PDE BASED PARALLEL ALGORITHMS ON DIFFERENT COMPUTER ARCHITECTURES

SUMMARY

In last two decades, use of parallel algorithms on different architectures increased the need of architecture and application independent performance analysis tools. Tools that support different communication methods and hardware prepare a common ground regardless of equipments provided.

Partial differential equations (PDE) are used in several applications (such as propagation of heat, wave) in computational science and engineering. These equations can be solved using iterative numerical methods. Problem size and error tolerance effects iteration count and computation time to solve equation. PDE computations take long time using single processor computers with sequential algorithms, and if data size gets bigger single processors memory may be insufficient. Thus, PDE's are solved using parallel algorithms on multiple processors. In this thesis, elliptic partial differential equation is solved using Gauss-Seidel and Successive Over-Relaxation (SOR) methods parallel algorithms.

Performance analysis and optimization basically has three steps; evaluation, analysis of gathered information, defining and optimizing bottlenecks. In evaluation, performance information is gathered while program runs, then observations are made on gathered information by using visualization tools. Bottlenecks are defined and optimization techniques are researched. Necessary improvements are made to analyze the program again. Different applications in each of these stages can be used but in this thesis TAU is used, which collects these applications under one roof.

TAU (Tuning and Analysis Utilities) supports many hardware, operating systems and parallelization methods. TAU is an open source application and collaborates with other open source applications at different levels.

In this thesis, differences based on performance analysis of an algorithm in different two architectures are investigated. In performance analysis and optimization there is no golden rule to speed up algorithm. Each algorithm must be analyzed on that specific architecture. In this context, the performance analysis of a PDE algorithm on two architectures has been interpreted.

FARKLI PLATFORMLARDAKİ PDE TABANLI PARALEL ALGORİTMALARIN PERFORMANS ANALİZİ VE ENİYİLEMESİ

ÖZET

Son yıllarda dağıtık algoritmaların farklı platformlarda kullanılabilmesi platform ve uygulama bağımsız performans analizi uygulamaları ihtiyacını arttırmıştır. Farklı donanımları ve haberleşme metodlarını destekleyen uygulamalar kullanıcılara donanım ve yazılımdan bağımsız ortak bir zemin hazırladıkları için kolaylık sağlamaktadır.

Kısmi fark denklemleri hesaplamalı bilim ve mühendisliğin bir çok alanında kullanılmaktadır (ısı, dalga yayılımı gibi). Bu denklemlerin sayısal çözümü yinelemeli yöntemler kullanılarak yapılmaktadır. Problemin boyutu ve hata değerine göre çözüme ulaşmak için gereken yineleme sayısı ve buna bağlı olarak süresi değişmektedir. Kısmi fark denklemelerinin tek işlemcili bilgisayarlardaki çözümü uzun sürdüğü ve yüksek boyutlarda hafızaları yetersiz kaldığı için paralelleştirilerek birden fazla bilgisayarın işlemcisi ve hafızası kullanılarak çözülmektedir. Tezimde eliptik kısmi fark denklemlerini Gauss-Seidel ve Successive Over-Relaxation (SOR) metodlarını kullanarak çözen paralel algoritmalar kullanılmıştır.

Performans analizi ve eniyilemesi kabaca üç adımdan oluşmaktadır; ölçüm, sonuçların analizi, darboğazların tespit edilip yazılımda iyileştirme yapılması. Ölçüm aşamasında programın koşarken ürettiği performans bilgisi toplanır, toplanan bu veriler görselleştirme araçları ile anlaşılır hale getirilerek yorumlanır. Yorumlama aşamasında tespit edilen dar boğazlar belirlenir ve giderilme yöntemleri araştırılır. Gerekli iyileştirmeler yapılarak program yeniden analiz edilir. Bu aşamaların her birinde farklı uygulamalar kullanılabilir fakat tez çalışmamda uygulamaları tek çatı altında toplayan TAU kullanılmıştır.

TAU (Tuning and Analysis Utilities) farklı donanımları ve işletim sistemlerini destekleyerek farklı paralelleştirme metodlarını analiz edebilmektedir. Açık kaynak kodlu olan TAU diğer açık kaynak kodlu uygulamalar ile uyumlu olup birçok seviyede bütünleşme sağlanmıştır.

Bu tez çalışmasında, iki farklı platformda aynı uygulamanın performans analizi yapılarak platform farkının getirdiği farklılıklar incelenmektedir. Performans analizinde bir algoritmanın eniyilemesini yapmak için genel bir kural olmadığından her algoritma her platformda incelenerek gerekli değişiklikler yapılmalıdır. Bu bağlamda kullandığım PDE algoritmasının her iki sistemdeki analizi sonucu elde edilen bilgiler yorumlanmıştır.

1. INTRODUCTION

In the past, processor design trends were dominated by adding new instruction sets and increasing clock speeds. Recently, clock speeds have reached to maximum speed. Processor manufacturers are making multiple core designs to correspond demand for increasing performance. Consider clock frequency, which was on an exponential trend in the mid 90's. From about 1993 with the Intel Pentium processor and continuing through mid 2003 with the Intel Pentium IV processor, clock frequency doubled every 18 months to 2 years. This was a driving force for increasing performance of microprocessors during this timeframe. However, due to increased dynamic power dissipation and design complexity, this trend tapered with maximum clock frequencies around 4GHz [1].

Since sequential algorithms use only one processor (core), makes need of parallel algorithms on the increase. Especially, some algorithms need more processing power that cannot be satisfied using single processor. Considering that, hardware trends are making multiple core processors instead of speeding up a single core, algorithms making intensive calculations will not be satisfied with sequential algorithms.

In parallel computing, a program is split up into parts that run simultaneously on multiple computers communicating over a network. Distributed computing is a form of parallel computing, but parallel computing is most commonly used to describe program parts running simultaneously on multiple processors in the same computer. Both types of processing require dividing a program into parts that can run simultaneously, but distributed programs often must deal with heterogeneous environments, network links of varying latencies. There are different types of distributed computer architectures based on communication, memory and computation distribution. In this thesis, parallel architectures; cluster computing and symmetric multiprocessing (SMP) architectures has been studied.

Parallel algorithms are designed to run on computer hardware constructed from interconnected processors. Parallel algorithms are used in various application areas, such as scientific computing.

Parallel algorithms are typically executed concurrently, with separate parts of the algorithm being run simultaneously on independent processors, and having limited information about what the other parts of the algorithm are doing. One of the major challenges in developing and implementing parallel algorithms is successfully coordinating the behavior of the independent parts of the algorithm. The choice of an appropriate parallel algorithm to solve a given problem depends on both the characteristics of the problem, and characteristics of the system, the kind of inter-process communication that can be performed, and the level of timing synchronization between separate processes [2].

Performance analysis tools used for parallel algorithms are different from sequential algorithm performance analysis tools. Data gathered from distinct nodes must be merged together in the conscious of cooperative basis between nodes. On the other hand, performance analysis tool must be compatible with the hardware, operating system and software languages. This is why developers who are developing software on different architectures and software languages are in demand of a highly portable performance analysis tool.

Performance analysis tools generate output data, which is collected when program runs. Generated output data can be interpreted by visualization tools. If data can be transformed into different formats, different visualization tools can be used for different purposes.

On the other hand, portability looks for common abstractions in performance methods and how these can be supported by reusable and consistent techniques across different computing environments (software and hardware). Lack of portable performance evaluation environments forces users to adopt different techniques on different systems, even for common performance analysis.

Given the diversity of performance problems, evaluation methods, and types of events and metrics, the instrumentation and measurement mechanisms needed to support performance observation must be flexible, to give maximum opportunity for configuring performance experiments, and portable, to allow consistent cross-platform performance problem solving [3].

1.1 Objectives of the Study

Parallel algorithms achieved more popularity by the increase of HPC (High Performance Computing) systems and widespread use of algorithms for these systems. Like sequential algorithms, parallel algorithms need to be analyzed for performance. However, the increasing complexity of parallel systems is an issue for a portable and robust performance analysis tool. TAU (Tuning and Analysis Utilities) satisfies parallel systems requirements. In this thesis, TAU is used for performance analysis.

Complex scientific calculations requires significant amount of computational power that cannot be done or done on time with sequential algorithms. Parallel algorithms are inevitable for some calculations. To achieve high performance computing software developer must be aware of the computing architecture. Because of the parallel algorithms characteristics, program performance may vary on different architectures. Today's computing centers have different types of parallel computing servers. ITU National Center for High Performance Computing of Turkey (NCHPC) has three super computers with different architecture. Differences of systems achieve advantage to some parallel algorithms and disadvantage for some. These three systems have two distinct architecture types; symmetric multiprocessing (SMP) and cluster.

Purpose of this thesis is to compare two architectures by making performance analysis of a parallel PDE algorithm. By this experiment, software developer can choose either of the architectures by looking at the parallel algorithm characteristics like communication and synchronization. By defining pros and cons of two parallel architectures, developer can select best suitable system for algorithm. Although knowing advantages of the parallel computing architecture, developers can write algorithms that are more efficient.

Unfortunately, there is no golden recipe to speed up an algorithm. Hence, each algorithms performance analysis must be done individually to define bottleneck and find solutions for speeding up algorithm. Concordantly, this thesis is also a guideline for analyzing performance of a parallel algorithm and finding bottlenecks. Steps of performance analysis are common and described in details but finding solutions for bottlenecks are algorithm specific.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. In NCHPC, there are two different types of parallel computers, a cluster and a symmetric multiprocessor. A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus. Difference of two architectures makes them preferable on some applications. A parallel PDE algorithms performance analysis is made and performance effects of two systems are defined by the data gathered. This work will help parallel algorithm developers to write software by knowing the performance characteristics of computer architecture.

1.2 Background

Here are some studies comparing parallel programming models and for PDE algorithms and making performance analysis on different architectures. In addition, performance analysis tools are criticized for their competency. TAU is used in many applications and architectures.

Scalability of performance analysis software is important as much as scaling of the tested algorithm. TAU performance systems scalability in terascale systems has been proven [4]. In conclusion, the need of a performance observation framework that supports a wide range of instrumentation and measurement strategies for terascale systems is pronounced.

The goals of a performance system in terascale is defined as:

- greater dynamics and flexibility in performance measurements,
- improved methods for performance mapping in multi-layered and mixed model software, and
- more comprehensive application/system performance data integration

TAU supports MPI at library level instrumentation [4].

Programming models has been compared on four different architectures for solving implicit finite-element method [5]. Four parallel architectures were used in this

study: (1) IBM SP with 184 4-way SMP nodes (Winterhawk I or WH I) each with four 375 MHz Power 3 processors, (2) IBM SP with 144 8-way SMP nodes (Nighthawk II or NH II) each with eight 375 MHz Power 3 processors, (3) Compaq/Alpha SC server with 64 4-way SMP nodes each with four 667 MHz CPU's, (4) SGI Origin 2000 with 256 250 MHz processors. The performance analyses that were performed in this context showed that the pure MPI performance was usually better than the pure OpenMP performance for all architectures except for the case of two processors in which case the performances were close. This limitation in the pure OpenMP model also extends to the hybrid model, which performs best only when two OpenMP threads are used.

Also another work on SGI Origin 2000 with 300MHz R12000 showed that some algorithms scale better on pure MPI implementation and some on OpenMP [6]. Especially if MPI implementation suffers from pure scaling due to poor load balance or memory limitations due to the use of replicated data strategy, OpenMP strategy may perform better [6].

In addition, iterative PDE solvers performance has been studied on elder architectures. PDE algorithms performance analysis on Digital Alpha-Server 8400 with Alpha 21164 processor showed the inefficiency of programs [7]. Using red-black decomposition made data level parallelization. Also, loop fusing was used for instruction level parallelism and to enable re-use of cache. When two or four iterations are fused together this two methods increased efficiency of the algorithm. Modern compilers can do these optimizations if algorithm supports optimization.

When selecting parallelization method and its implementation, computers network connection must be considered. Implementations performance varies on different network architectures. MVAPICH is an MPICH2 based MPI implementation for Infiniband network infrastructure. MVAPICH uses Infiniband's Remote Direct Memory Access (RDMA) and low latency features. With optimizations such as piggybacking, pipelining and zero-copy, MPICH2 is able to deliver good performance to the application layer. For example, MVAPICH designs achieves 7.6 microsecond latency and 857MB/s peak bandwidth, which come quite close to the raw performance of InfiniBand [8].

2. SELECTION OF PARALLELIZATION METHODS

2.1 Introduction

Algorithm development is a critical component of problem solving using computers. A sequential algorithm is a sequence of basic steps for solving a given problem using a serial computer. Similarly, a parallel algorithm is a recipe that tells us how to solve a given problem using multiple processors. However, specifying a parallel algorithm involves more than just specifying the steps. At the very least, a parallel algorithm has the added dimension of concurrency and the algorithm designer must specify sets of steps that can be executed simultaneously. In practice, specifying a nontrivial parallel algorithm may include some or all of the following:

- Identifying portions of the work that can be performed concurrently.
- Mapping the concurrent pieces of work onto multiple processes running in parallel.
- Distributing the input, output, and intermediate data associated with the program.
- Managing accesses to data shared by multiple processors.
- Synchronizing the processors at various stages of the parallel program execution.

Typically, there are several choices for each of the above steps, but usually, relatively few combinations of choices lead to a parallel algorithm that yields sufficient performance with the computational and storage resources employed to solve the problem. Often, different choices yield the best performance on different parallel architectures or under different parallel programming paradigms [9].

Below Figure 2.1 shows basic four steps for parallelizing a problem. These steps are explained individually.

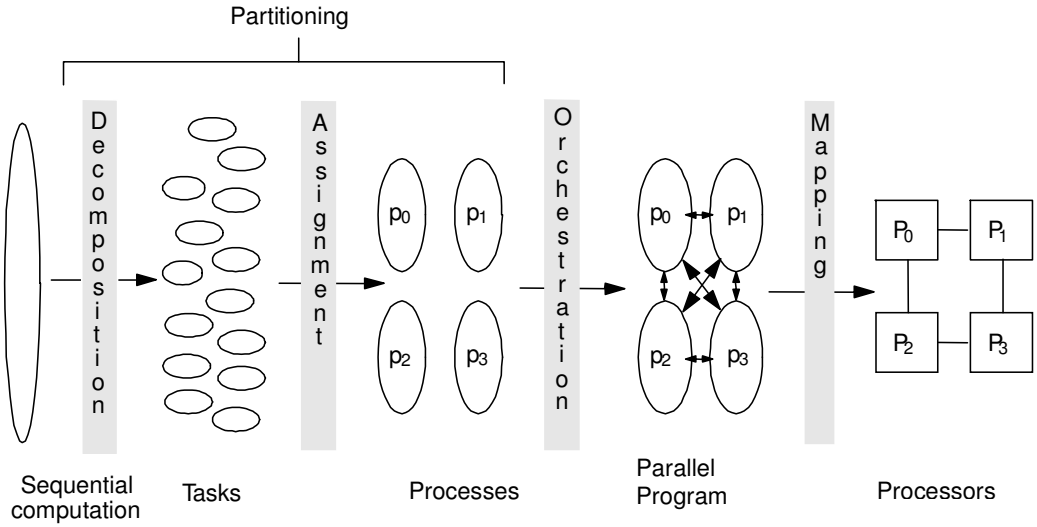

Figure 2.1 : Steps for parallelizing a problem [9]

Dividing a computation into smaller computations and assigning them to different processors for parallel execution are the two key steps in the design of parallel algorithms. The process of dividing a computation into smaller parts to be executed in parallel is called decomposition. The main computation is divided into tasks, which are programmer-defined units of computation. Simultaneous execution of multiple tasks is the key to reducing the time required to solve the entire problem.

The number and size of tasks into which a problem is decomposed determines the granularity of the decomposition. Decomposition into a large number of small tasks is called fine-grained and decomposition into a small number of large tasks is called coarse-grained [9].

The tasks run on physical processors. A process uses the code and data to produce the output of that task within a finite amount of time after the task is activated by the parallel program. The mechanism by which tasks are assigned to processes for execution is called assignment.

The task-dependency and task-interaction graphs that result from a choice of decomposition play an important role in the selection of a good assignment for a parallel algorithm. A good assignment should seek to maximize the use of concurrency by assigning independent tasks onto different processes. Assignment stage is important for balancing workload and reducing communication between processes.

During computation, a process may synchronize or communicate with other processes, if needed. In order to obtain any speedup over a sequential implementation, a parallel program must have several processes active simultaneously, working on different tasks. Designing this communication and synchronization structure is called orchestration. Reducing the cost of communication, and preserving locality of data is the important goals of this stage.

Mapping is the process of mapping processes into processors that we have. There are situations where mapping is done by Operating System (centralized multiprocessor), and there are situations where we manually do the mapping (distributed memory system). Maximizing processors utilization and minimizing interprocessor communication are the main goals of this stage.

2.2 Parallelization Methods

Parallel programming model is a set of software technologies to express parallel algorithms and match applications with the underlying parallel systems. A programming model must allow the programmer to balance the competing goals of productivity and implementation efficiency.

Parallel models are implemented in several ways: as libraries invoked from traditional sequential languages, as language extensions, or complete new execution models.

It is typically concerned with either the implicit or explicit specification of the following program properties:

- The computational tasks – How is the application divided into parallel tasks?
- Mapping computational tasks to processing elements – The balance of computation determines how well utilized the processing elements are.
- Distribution of data to memory elements – Locating data to smaller, closer memories increases the performance of the implementation.
- The mapping of communication to the inter-connection network – interconnect bottlenecks can be avoided by changing the communication of the application.

- Inter-task synchronization – The style and mechanisms of synchronizations can influence not only performance, but also functionality.

There are several different forms of parallel computing:

- bit-level
- instruction level
- data parallelism
- task parallelism

Bit-level parallelism is a form of parallel computing based on increasing processor word size. From the advent of very-large-scale integration (VLSI) computer chip fabrication technology in the 1970s until about 1986, advancements in computer architecture were done by increasing bit-level parallelism [10].

A computer program is, in essence, a stream of instructions executed by a processor. These instructions can be re-ordered and combined into groups, which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism [11].

Data parallelism is parallelism inherent in program loops, which focuses on distributing the data across different computing nodes to be processed in parallel.

Task parallelism is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data" [11]. This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data.

There are several parallel programming models in common use:

- Shared Memory
- Threads
- Message Passing
- Data Parallel
- Hybrid

Shared Memory Model: In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.

Various mechanisms such as locks / semaphores may be used to control access to the shared memory. An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. Program development can often be simplified.

An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality. Keeping data local to the processor that works on it conserves memory accesses, cache refreshes and bus traffic that occur when multiple processors use the same data.

Unfortunately, controlling data locality is hard to understand and beyond the control of the average user.

Threads Model: In the threads model of parallel programming, a single process can have multiple, concurrent execution paths. Perhaps the simplest analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines:

The main program is scheduled to run by the native operating system. Main program performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently. Each thread has local data, but also, shares the entire resources of main program. This saves the overhead associated with replicating a program's resources for each thread. Each thread also benefits from a global memory view because it shares the memory space of main program.

Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to insure that more than one thread is not updating the same global address at any time.

Threads can come and go, but main program remains present to provide the necessary shared resources until the application has completed.

Threads are commonly associated with shared memory architectures and operating systems. OpenMP is an implementation of threaded parallel programming model.

Message Passing Model: In the message-passing model a set of tasks use their own local memory during computation. Multiple tasks can reside on the same physical

machine as well across an arbitrary number of machines. These tasks exchange data through communications by sending and receiving messages. Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation. Message Passing Interface (MPI) is an implementation of message passing model.

Data Parallel Model: In the data parallel model most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube. A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure. Tasks perform the same operation on their partition of work. On shared memory architectures, all tasks may have access to the data structure through global memory. On distributed memory architectures, the data structure is split up and resides as "chunks" in the local memory of each task.

Hybrid Model: Hybrid model is the collection of different parallel models. By combining two or more parallel models, parallelization of the program can be increased. This technique also helps to increase the parallel part of the algorithm.

2.2.1 Message Passing Interface (MPI)

MPI is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communications are supported. MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today [12].

Most MPI implementations consist of a specific set of routines (i.e., an API) callable from FORTRAN, C, or C++ and from any language capable of interfacing with such routine libraries. The advantages of MPI over older message passing libraries are portability (because MPI has been implemented for almost every distributed memory architecture) and speed (because each implementation is in principle optimized for the hardware on which it runs) [13].

The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few features that are language-specific. MPI programs always work with processes, but programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multicore machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or mpiexec.

The MPI library functions include, but are not limited to, point-to-point rendezvous-type send/receive operations. MPI supports a Cartesian or graph-like logical process topology for exchanging data between process pairs (send/receive operations). MPI combines partial results of computations (gathering and reduction operations), synchronizes nodes (barrier operation) as well as obtaining network-related information such as the number of processes in the computing session. Point-to-point operations come in synchronous, asynchronous, buffered, and ready forms, to allow both relatively stronger and weaker semantics for the synchronization aspects of a rendezvous-send.

2.2.2 OpenMP

The OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and FORTRAN on many architectures, including UNIX and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and MPI.

OpenMP is an implementation of multithreading, a method of parallelization whereby the master "thread" (a series of instructions executed consecutively) "forks" a specified number of slave "threads" and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

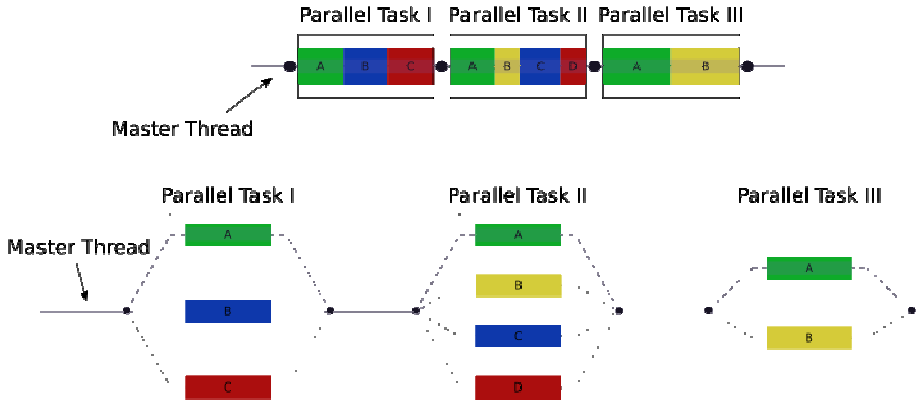

Figure 2.2 : OpenMP Thread Model [9]

By default, each thread executes the parallelized section of code independently. "Work-sharing constructs" can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both Task parallelism and Data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The number of threads can be assigned by the runtime environment based on environment variables or in code using functions. The OpenMP functions are included in a header file labeled "omp.h" in C/C++.

Getting N times less wall clock execution time (or N times speedup) when running a program parallelized using OpenMP on an N processor platform, is seldom due to the other limitations. A large portion of the program may not be parallelized by OpenMP, which means that the theoretical upper limit of speedup is according to Amdahl's law [14]. One other limitation is; N processors in an SMP may have N times the computation power, but the memory bandwidth usually does not scale up N times. In addition, many other common problems affecting the final speedup in parallel computing also apply to OpenMP, like load balancing and synchronization overhead.

2.2.3 Mixed Programming (MPI+OpenMP)

We can mix MPI and OpenMP if architecture has SMP nodes connected with a network. Most of the clusters have nodes connected to each other via communication network. However, inside nodes there are multiple processing units (cores). In NCHPC, cluster nodes have 4 or 8 cores. Parallelizing algorithm using OpenMP inside nodes and using MPI for inter node connection can be advantageous.

Multiple levels of parallelism can be achieved by combining message passing and OpenMP parallelization. Which programming paradigm is the best will depend on the nature of the given problem, the hardware components of the cluster, and the network.

Hybrid programming avoids the extra communication overhead with MPI within node. However, OpenMP has thread creation overhead, and explicit synchronization is required.

3. PARALLEL COMPUTER ARCHITECTURES

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes.

3.1 Flynn's Taxonomy

There are different ways to classify parallel computers. One of the more widely used classifications is called Flynn's taxonomy. Flynn's taxonomy is a classification of computer architectures, proposed by Michael J. Flynn in 1966 [15]. Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of Instruction and Data. Each of these dimensions can have only one of two possible states: Single or Multiple.

S I S D Single Instruction, Single Data	**S I M D** Single Instruction, Multiple Data
M I S D Multiple Instruction, Single Data	**M I M D** Multiple Instruction, Multiple Data

Figure 3.1 : Flynn's taxonomy

The four classifications defined by Flynn are based upon the number of concurrent instruction and data streams available in the architecture:

Single Instruction, Single Data stream (SISD): A sequential computer, which exploits no parallelism in either the instruction or data streams. This corresponds to the von Neumann architecture. Examples of SISD architecture are the traditional uniprocessor machines like a PC or old mainframes.

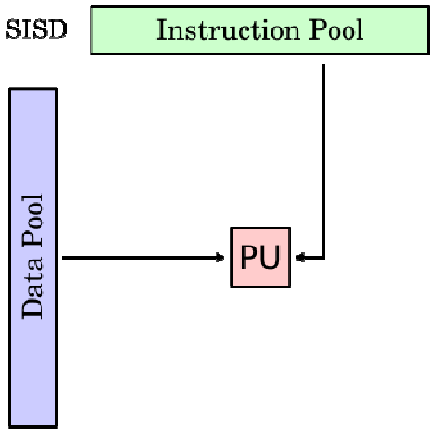

Figure 3.2 : SISD Model

Single Instruction, Multiple Data streams (SIMD): A computer, which exploits multiple data streams against a single instruction stream to perform operations, which may be naturally parallelized. SIMD (Single Instruction, Multiple Data; colloquially, "vector instructions") is a technique employed to achieve data level parallelism. Each processing unit can operate on a different data element, thus SIMD suits for specialized problems characterized by a high degree of regularity, such as graphics/image processing. Since the release of MMX, all the desktop CPU manufacturers have released chips with SIMD instructions (MMX, SSE, 3DNow!). As SIMD on the desktop becomes both more common and more technically advanced, the number of cases where it can be used has increased dramatically [16].

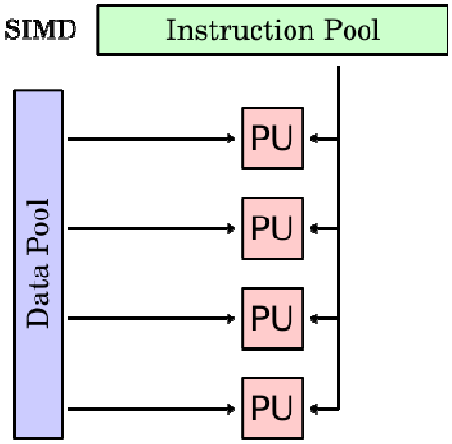

Figure 3.3 : SIMD Model

Multiple Instructions, Single Data stream (MISD): Multiple instructions operate on a single data stream. Few actual examples of this class of parallel computer have ever existed.

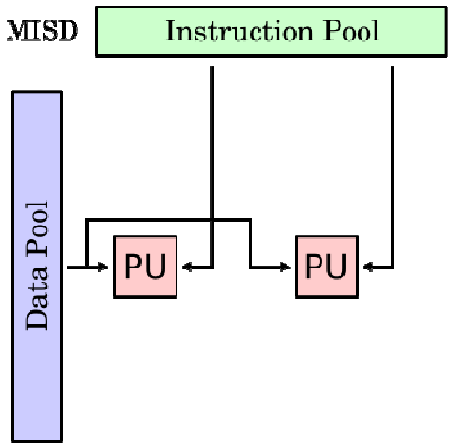

Figure 3.4 : MISD Model

Multiple Instructions, Multiple Data streams (MIMD): Multiple autonomous processors simultaneously executing different instructions on different data. Parallel systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space. Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data.

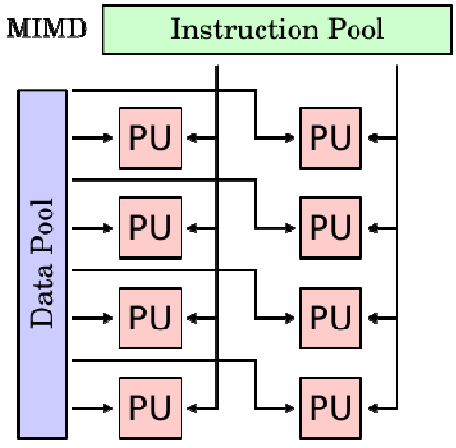

Figure 3.5 : MIMD Model

MIMD computers support higher-level parallelism (subprogram and task levels) that can be exploited by "divide and conquer" algorithms organized as largely independent subcalculations (for example, searching and sorting) [17].

3.2 Parallel Computer Memory and Communication Architectures

Main memory in a parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space) [18]. Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well. Distributed shared memory is a combination of the two approaches, where the processing element has its own local memory and access to the memory on non-local processors. Accesses to local memory are typically faster than accesses to non-local memory.

Computer systems have caches which are small, fast memories located close to the processor which store temporary copies of memory values. Parallel computer systems have difficulties with caches that may store the same value in more than one location, with the possibility of incorrect program execution. These computers require a cache coherency system, which keeps track of cached values and strategically purges them, thus ensuring correct program execution. Designing large, high-performance cache coherence systems is a very difficult problem in computer architecture. As a result, shared-memory computer architectures do not scale as well as distributed memory systems do [18].

3.2.1 Shared Memory

Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space. A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location. In shared memory architecture, multiple processors can operate independently but share the same memory resources. The issue with shared memory systems is that many CPUs need fast access to memory and will likely cache memory, which has two complications:

- CPU-to-memory connection becomes a bottleneck. Shared memory computers cannot scale very well.
- **Cache coherence:** Whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors; otherwise, the different processors will be working with incoherent data. Coherence protocols can provide extremely high-performance access to shared information between multiple processors. On the other hand, they can sometimes become overloaded and become a bottleneck to performance [19].

Computer architectures in which each element of main memory can be accessed with equal latency and bandwidth are known as Uniform Memory Access (UMA) systems. Typically, that can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as a Non-Uniform Memory Access (NUMA) architecture.

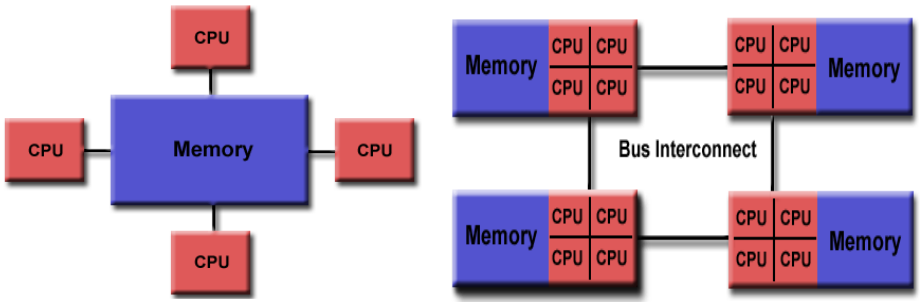

Figure 3.6 : UMA and NUMA Architectures [37]

Main advantages of shared memory system are user-friendly programming perspective to memory and data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs. Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.

In NCHPC a ccNUMA symmetric multiprocessing (SMP) computer, HP Integrity Superdome, is used for computational calculations. HP Integrity Superdome has cache coherency to imply this property its memory architecture is referred as ccNUMA, which means that processors have shorter access times for their cell's memory but longer access times for other cell's memories, and data items are allowed

to be replicated across individual cache memories but are kept coherent with one another by cache coherence hardware mechanisms [20].

3.2.2 Distributed Memory

Distributed memory refers to a multiple-processor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors.

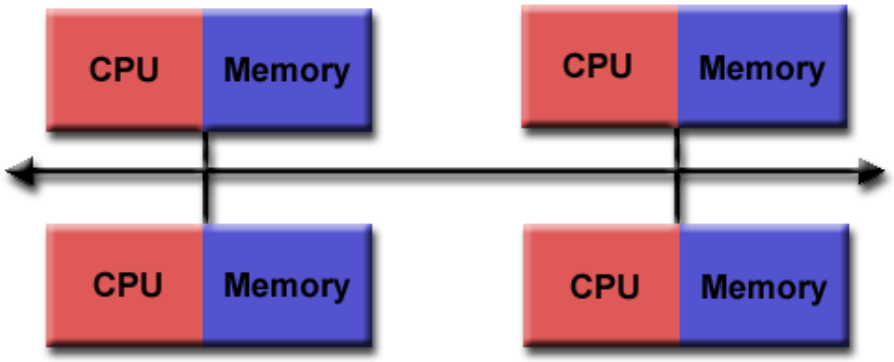

Figure 3.7 : Distributed Memory Architecture [37]

In a distributed memory system, there is typically a processor, a memory, and some form of interconnection that allows programs on each processor to interact with each other. The interconnect can be organized with point-to-point links or separate hardware can provide a switching network.

Main advantage of distributed memory system is its scalability. Increase the number of processors and the size of memory increases proportionately. In addition, each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency. Main disadvantage is low communication speed and higher latency (compared with shared memory) which causes more wait time at synchronization points.

3.2.3 Hybrid Distributed-Shared Memory

Hybrid memory is a mixture of distributed and shared memory systems. In hybrid memory, each compute node has its own address space, which is used by multiple processors. These processors have their own caches and implement cache coherency protocol. Nodes have fewer processors and more cost effective compared to a shared

memory system. Nodes have multiple processors, a global memory and an interconnection that allows nodes to communicate with each other.

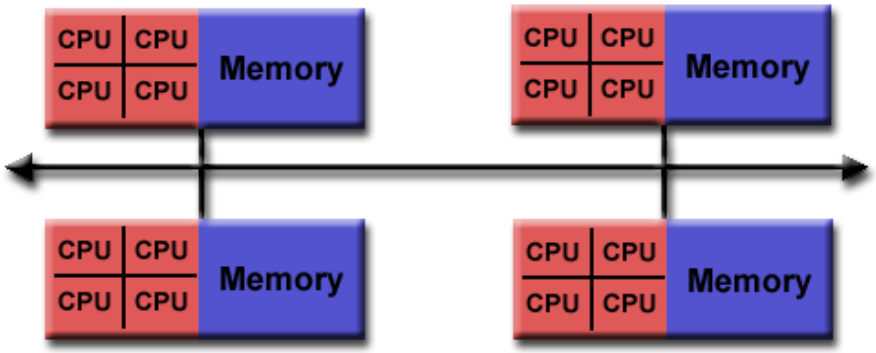

Figure 3.8 : Hybrid Memory Architecture [37]

NCHPC also has a HP DL360 G5 Cluster, which has hybrid memory architecture. Cluster has 192 nodes and 1004 cores.

3.3 CPU Cache Memory Hierarchy

Improvements in technology do not change the fact that microprocessors are still much faster than main memory. Memory access time is increasingly the bottleneck in overall application performance. As a result, an application might spend a considerable amount of time waiting for data [21]. To overcome this problem CPU caches are used. A CPU cache is used by the central processing unit of a computer to reduce the average time to access memory. The cache is a smaller, faster memory, which stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory. When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. The application can take advantage of this enhancement by fetching data from the cache instead of main memory. Of course, there is still traffic between memory and the cache, but it is minimal.

Figure 3.9 shows general cache memory hierarchy model.

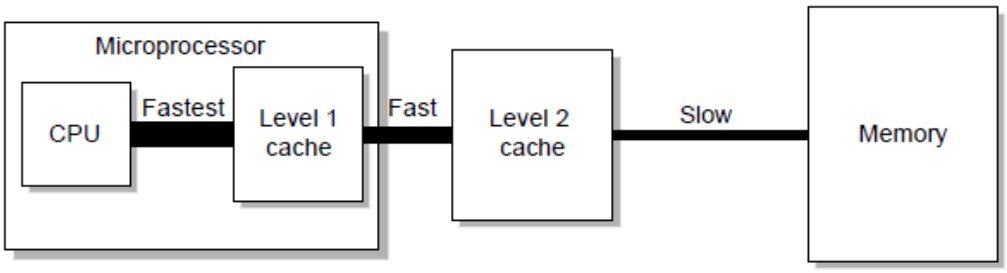

Figure 3.9 : General Memory Hierarchy [38]

In a modern microprocessor, several caches are found. They not only vary in size and functionality, but also their internal organization is typically different across the caches. Common caches are instruction, data, and Translation Lookaside Buffer (TLB) cache.

The instruction cache is used to store instructions. This helps to reduce the cost of going to memory to fetch instructions.

A data cache is a fast buffer that contains the application data. Before the processor can operate on the data, it must be loaded from memory into the data cache. The element needed is then loaded from the cache line into a register and the instruction using this value can operate on it. The resultant value of the instruction is also stored in a register. The register contents are then stored back into the data cache.

Translating a virtual page address to a valid physical address is rather costly. The

TLB is a cache to store these translated addresses.

Each entry in the TLB maps to an entire virtual memory page. The CPU can only operate on data and instructions that are mapped into the TLB. If this mapping is not present, the system has to re-create it, which is a relatively costly operation. The larger a page, the more effective capacity the TLB has. If an application does not make good use of the TLB (for example, random memory access) increasing the size of the page can be beneficial for performance, allowing for a bigger part of the address space to be mapped into the TLB.

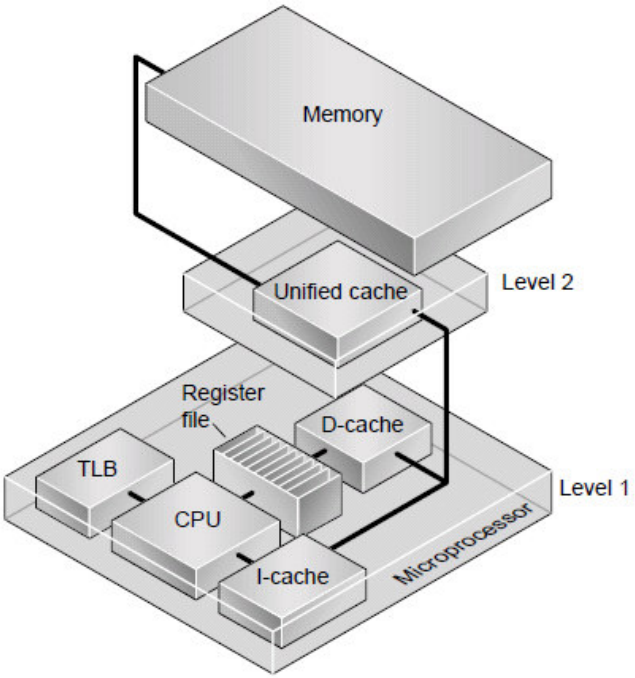

Figure 3.10 : Generic System Architecture [38]

Figure 3.10 shows unified cache at level two. Both instructions and data are stored in this type of cache. The cache at the highest level is often unified and external to the microprocessor. The cache architecture shown in figure 3.10 is rather generic. There are other types of caches in a modern microprocessor. In NCHPC two types of processors are used. HP Integrity Superdome is a RISC-based ccNUMA SMP system and uses Intel Itanium 2 processors. Another HP cluster uses Intel XEON processor. Below is the block diagram of Intel Itanium processor.

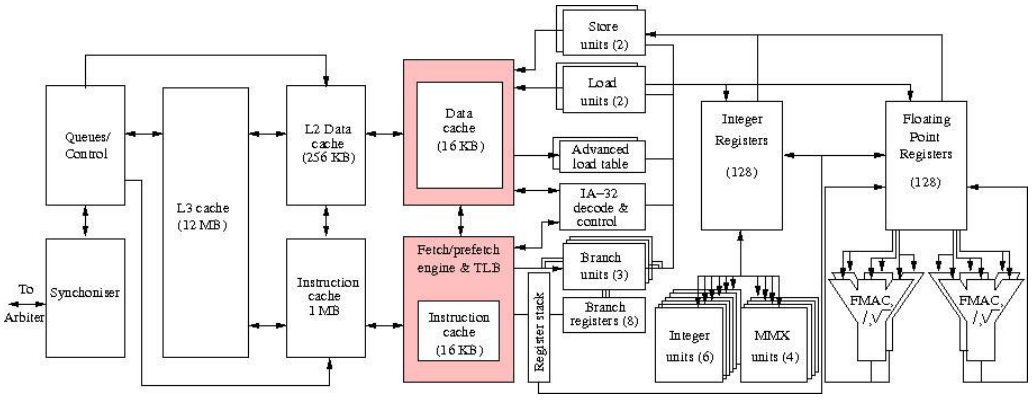

Figure 3.11 : Block diagram of an Intel Itanium 2 core [22]

As can be seen from Figure 3.11 there are four floating-point units capable of performing Fused Multiply Accumulate (FMAC) operations. However, two of these

work at the full 82-bit precision, which is the internal standard on Itanium processors, while the other two can only be used for 32-bit precision operations. When working in the customary 64-bit precision the Itanium has a theoretical peak performance of 6 Gflop/s at a clock frequency of 1.5 GHz [22]. Using 32-bit floating arithmetic, the peak is doubled. In addition, four MMX units are present to accommodate instructions for multi-media operations, an inheritance from the Intel Pentium processor family. For compatibility with this Pentium family there is a special IA-32 decode and control unit.

Because now two cores are present on a chip, some improvements had to be added to let them cooperate without problems. The synchronizers in the core feed their information about read and write requests and cache line validity to the arbiter. The arbiter filters out the unnecessary requests and combines information from both cores before handing the requests over to the system interface.

Intel Xeon processors play a major role in the cluster community as the majority of compute nodes in Beowulf clusters are of this type.

In Figure 3.12, a block diagram of the processor is shown with one of the cores in some detail. Note that the two cores share one second-level cache while the L1 caches and TLBs are local to each of the cores.

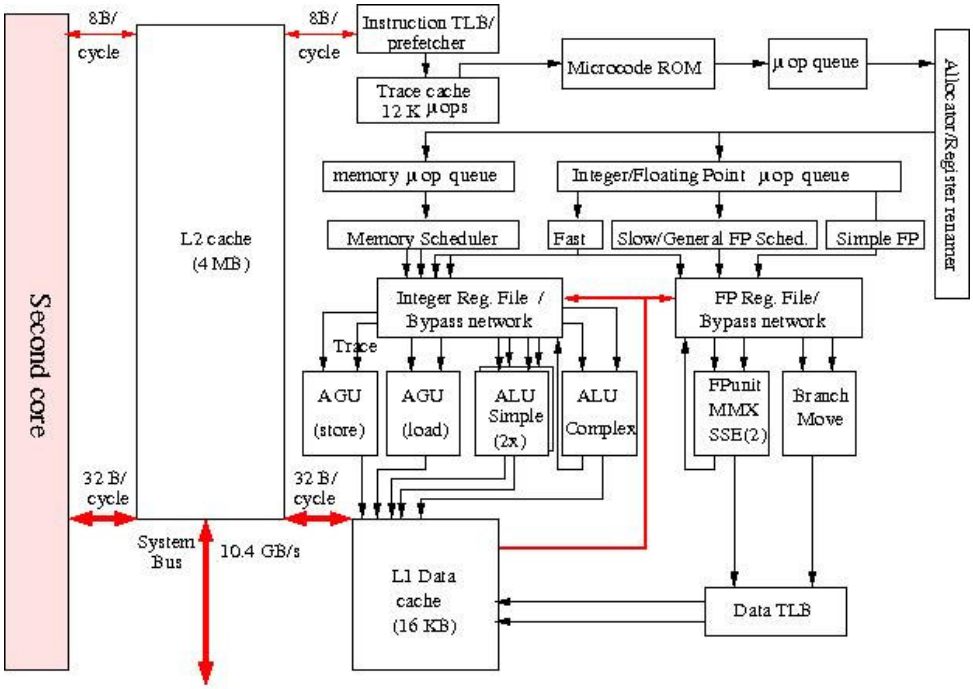

Figure 3.12 : Block diagram of the Intel Xeon processor [22]

The floating-point units, depicted in Figure 3.12, contain also additional units that execute the Streaming SIMD Extensions 2 and 3 (SSE2/3) instructions, a 144-member instruction set, that is especially meant for vector-oriented operations like in multimedia, and 3-D visualization applications but which will also be of advantage for regular vector operations as occur in dense linear algebra. The length of the operands for these units is 128 bits. The throughput of these SIMD units has been increased by a factor of two in the core architecture, which greatly increase the performance of the appropriate instructions. The Intel compilers have the ability to address the SSE2/3 units. This makes it in principle possible to achieve a 2-3 times higher floating-point performance [22].

3.4 Network Interfaces

Cluster computers are connected through network devices. There are several types of network devices. Each device has different speed and latency. Speeds of these devices are listed in Table 3.1 [23].

Table 3.1 : Local Area Network Device Bandwidths

Device	Speed (bit/s)	Speed (byte/s)
Token Ring IEEE 802.5t	100 Mbit/s	12.5 MB/s
Fast Ethernet (100base-X)	100 Mbit/s	12.5 MB/s
FDDI	100 Mbit/s	12.5 MB/s
FireWire (IEEE 1394) 400	393.216 Mbit/s	49.152 MB/s
HIPPI	800 Mbit/s	100 MB/s
Token Ring IEEE 802.5v	1,000 Mbit/s	125 MB/s
Gigabit Ethernet (1000base-X)	1,000 Mbit/s	125 MB/s
Myrinet 2000	2,000 Mbit/s	250 MB/s
Infiniband SDR 1X	2,000 Mbit/s	250 MB/s
Quadrics QsNetI	3,600 Mbit/s	450 MB/s
Infiniband DDR 1X	4,000 Mbit/s	500 MB/s
Infiniband QDR 1X	8,000 Mbit/s	1,000 MB/s
Infiniband SDR 4X	8,000 Mbit/s	1,000 MB/s
Quadrics QsNetII	8,000 Mbit/s	1,000 MB/s
10 Gigabit Ethernet (10Gbase-X)	10,000 Mbit/s	1,250 MB/s
Myri 10G	10,000 Mbit/s	1,250 MB/s
Infiniband DDR 4X	16,000 Mbit/s	2,000 MB/s
Scalable Coherent Interface (SCI) Dual Channel SCI, x8 PCIe	20,000 Mbit/s	2,500 MB/s

Infiniband SDR 12X	24,000 Mbit/s	3,000 MB/s
Infiniband QDR 4X	32,000 Mbit/s	4,000 MB/s
Infiniband DDR 12X	48,000 Mbit/s	6,000 MB/s
Infiniband QDR 12X	96,000 Mbit/s	12,000 MB/s
100 Gigabit Ethernet (100Gbase-X)	100,000 Mbit/s	12,500 MB/s

NCHPC cluster network interface is Infiniband DDR 4X. InfiniBand is a switched fabric communications link primarily used in high-performance computing. Its features include quality of service and failover, and it is designed to be scalable. The InfiniBand architecture specification defines a connection between processor nodes and high performance I/O nodes such as storage devices. It is a superset of the Virtual Interface Architecture.

Like Fibre Channel, PCI Express, Serial ATA, and many other modern interconnects, InfiniBand is a point-to-point bidirectional serial link intended for the connection of processors with high speed peripherals such as disks. It supports several signaling rates and, as with PCI Express, links can be bonded together for additional bandwidth.

Infiniband architecture (IBA) defines a System Area Network (SAN) for connecting multiple independent processor platforms (i.e., host processor nodes), I/O platforms, and I/O devices (see Figure 6). The IBA SAN is a communications and management infrastructure supporting both I/O and interprocessor communications (IPC) for one or more computer systems. An IBA system can range from a small server with one processor and a few I/O devices to a massively parallel supercomputer installation with hundreds of processors and thousands of I/O devices. Furthermore, the internet protocol (IP) friendly nature of IBA allows bridging to an internet, intranet, or connection to remote computer systems. IP over InfiniBand (IPoIB) is implemented for using IP communication on IBA [24].

IBA defines a switched communications fabric allowing many devices to concurrently communicate with high bandwidth and low latency in a protected, remotely managed environment. An end node can communicate over multiple IBA ports and can utilize multiple paths through the IBA fabric. The multiplicity of IBA ports and paths through the network are exploited for both fault tolerance and increased data transfer bandwidth.

IBA hardware off-loads from the CPU much of the I/O communications operation. This allows multiple concurrent communications without the traditional overhead associated with communicating protocols. The IBA SAN provides its I/O and IPC clients zero processor-copy data transfers, with no kernel involvement, and uses hardware to provide highly reliable, fault tolerant communications [24].

The serial connection's signaling rate is 2.5 gigabit per second (Gbit/s) in each direction per connection. InfiniBand supports double (DDR) and quad data (QDR) speeds, for 5 Gbit/s or 10 Gbit/s respectively, at the same data-clock rate [24].

Links use 8B/10B encoding — every 10 bits sent carry 8bits of data — so that the useful data transmission rate is four-fifths the raw rate. Thus single, double, and quad data rates carry 2, 4, or 8 Gbit/s respectively [24].

Links can be aggregated in units of 4 or 12, called 4X or 12X. A quad-rate 12X link therefore carries 120 Gbit/s raw, or 96 Gbit/s of useful data. Most systems today use either a 4X 2.5 Gbit/s (SDR) or 5 Gbit/s (DDR) connection. Larger systems with 12x links are typically used for cluster and supercomputer interconnects and for inter-switch connections.

The single data rate switch chips have a latency of 200 nanoseconds, and DDR switch chips have a latency of 140 nanoseconds. The end-to-end latency range is from 1.07 microseconds MPI latency (Mellanox ConnectX HCAs) to 1.29 microseconds MPI latency (Qlogic InfiniPath HTX HCAs) to 2.6 microseconds (Mellanox InfiniHost III HCAs). Various InfiniBand host channel adapters (HCA) exist in the market today, each with different latency and bandwidth characteristics. InfiniBand also provides RDMA capabilities for low CPU overhead. The latency for RDMA operations is less than 1 microsecond (Mellanox ConnectX HCAs) [24].

InfiniBand uses a switched fabric topology, as opposed to a hierarchical switched network like Ethernet. Like the channel model used in most mainframe computers, all transmissions begin or end at a channel adapter. Each processor contains a host channel adapter (HCA) and each peripheral has a target channel adapter (TCA). These adapters can also exchange information for security or quality of service.

Data is transmitted in packets of up to 4 kB that are taken together to form a message. A message can be:

- a direct memory access read from or, write to, a remote node (RDMA)
- a channel send or receive
- a transaction-based operation (that can be reversed)
- a multicast transmission.
- an atomic operation

Sockets Direct Protocol (SDP): The Sockets Direct Protocol (SDP) is a networking protocol originally defined by the Software Working Group (SWG) of the InfiniBand Trade Association. Originally designed for InfiniBand, SDP now has been redefined as a transport agnostic protocol for Remote Direct Memory Access (RDMA) network fabrics. SDP defines a standard wire protocol over an RDMA fabric to support stream sockets (SOCK_STREAM) network. SDP utilizes various RDMA network features for high-performance zero-copy data transfers. SDP is a pure wire-protocol level specification and does not go into any socket API or implementation specifics.

The purpose of the Sockets Direct Protocol is to provide an RDMA accelerated alternative to the TCP protocol on IP. The goal is to do this in a manner, which is transparent to the application.

Today, Sockets Direct Protocol for the Linux operating system is part of the OpenFabrics Enterprise Distribution (OFED), a collection of RDMA networking protocols for the Linux operating system. OFED is managed by the OpenFabrics Alliance. Many standard Linux distributions include the current OFED.

Sockets Direct Protocol only deals with stream sockets, and if installed in a system, bypasses the OS resident TCP stack for stream connections between any endpoints on the RDMA fabric. All other socket types (such as datagram, raw, packet etc.) are supported by the Linux IP stack and operate over standard IP interfaces (i.e., IPoIB on InfiniBand fabrics). The IP stack has no dependency on the SDP stack; however, the SDP stack depends on IP drivers for local IP assignments and for IP address resolution for endpoint identifications.

IP over IB: InfiniBand is an emerging standard intended as an interconnect for processor and I/O systems and devices. IP is one type of traffic that could use this interconnect. InfiniBand would benefit greatly from a standardized method of handling IP traffic on IB fabrics. It is also important to be able to manage InfiniBand

devices in a common way. IPoIB enables advanced functionalities such as mapping IP QOS into IB-specific.

Direct Access Provider Library (kDAPL/uDAPL): Direct Access Provider Library is a transport-independent, platform-independent, high-performance API for using the remote direct memory access (RDMA) capabilities of modern interconnect technologies such as InfiniBand, the Virtual Interface Architecture, and iWARP.

The Kernel Direct Access Programming Library (kDAPL) defines a single set of kernel-level APIs for all RDMA-capable Transports [25]. The User Direct Access Programming Library (uDAPL) defines a single set of user-level APIs for all RDMA-capable Transports. Both kDAPL and uDAPL mission are to define a Transport-independent and Platform-standard set of APIs that exploits RDMA capabilities, such as those present in IB, VI, and RDDP WG of IETF [26].

Latency and bandwidth are most used network performance parameters. These two parameters affect MPI performance too. Latency is a dominant factor for network performance on small sized messages and synchronization points. Bandwidth becomes dominant on heavy data transfers. IBA's low latency and high bandwidth increases its performance. Thus, MPI implementations latency and bandwidth vary and they cannot achieve theoretical values of IBA. MVAPICH2 (MPI over InfiniBand and iWARP) is MPICH2 based MPI implementation for IBA. MVAPICH2 designs achieves 7.6 microsecond latency and 857MB/s peak bandwidth, which come quite close to the raw performance of InfiniBand [8].

MVAPICH is an MPICH2 based MPI implementation for Infiniband network infrastructure. MVAPICH uses Infiniband's Remote Direct Memory Access (RDMA) and low latency features. With optimizations such as piggybacking, pipelining and zero-copy, MPICH2 is able to deliver good performance to the application layer. For example, MVAPICH designs achieves 7.6 microsecond latency and 857MB/s peak bandwidth, which come quite close to the raw performance of InfiniBand [8]. IBA's high-speed infrastructure delivers high bandwidth compared to other network architectures. InfiniBand can outperform other interconnects if the application is bandwidth-bound [27].

The Superdome has a 2-level crossbar processor interconnection: one level within a 4-processor cell and another level by connecting the cells through the crossbar

backplane. Every cell connects to the backplane at a speed of 8 GB/s and the global aggregate bandwidth for a fully configured system is therefore 64 GB/s.

Another parallel architecture used in this work is HP Integrity Superdome in ccNUMA architecture SMP computer. HP Integrity Superdome has crossbar connection between cells. Crossbar connection throughput per cell is 27.3 GB/s, which is much higher than any network connection device [28].

The basic building block of the Superdome is the 4-processor cell. All data traffic within a cell is controlled by the Cell Controller. It connects to the four local memory subsystems at 16 GB/s, to the backplane crossbar at 8 GB/s, and to two ports, that each serves two processors at 6.4 GB/s/port. As each processor houses two CPU cores, the available bandwidth per CPU core is 1.6 GB/s [28].

4. PERFORMANCE ANALYSIS

Performance analysis is the investigation of a program's behavior using information gathered as the program runs. The usual goal of performance analysis is to determine which parts of a program to optimize for speed or memory usage.

A profiler is a performance analysis tool that measures the behavior of a program as it runs, particularly the frequency and duration of function calls. The output is a stream of recorded events (a trace) or a statistical summary of the events observed (a profile). Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, operating system hooks, and performance counters.

Performance analysis tools generates data while program runs, and data size is related to code size and run time. To keep pace with the growing complexity of large-scale parallel supercomputers, performance tools must handle effective instrumentation of complex software and the correlation of runtime performance data with system characteristics. In addition, workload characterization is an important tool for understanding the the nature and performance of the workload submitted to a parallel system.

In this thesis, TAU (Tuning and Analysis Utilities) is used for performance analysis. TAU parallel performance system is the product of seventeen years of development to create a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications. The success of the TAU project represents the combined efforts of researchers at the University of Oregon and colleagues at the Research Centre Juelich and Los Alamos National Laboratory. [3]

4.1 Performance Evaluation and Objectives

In general, the objective of performance analysis is to define and reduce the consumption of sources. Performance analysis of a parallel algorithm is used to determine which sections of an algorithm to optimize. Optimization is made either to

increase speed or decrease communication time (or both). In this thesis, both computation and communication time tried to be decreased.

This flow chart (Figure 4.1) is the general approach in performance evaluation [29].

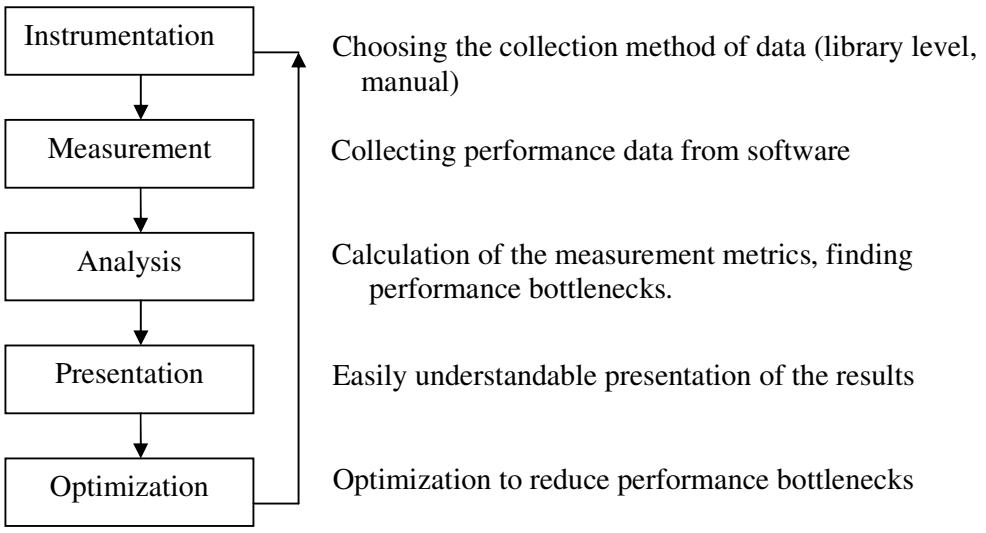

Figure 4.1 : Performance Evaluation

Most performance problems are unique. The metrics, workload, and evaluation techniques used for one problem generally cannot be used for the next problem. Nevertheless, there are steps common to all performance evaluation projects that help you avoid the common mistakes. These steps are as follows [30].

State Goals and Define the System: The first step in any performance evaluation project is to state the goals of the study and define what constitutes the system by delineating system boundaries. Given the same set of hardware and software, the definition of the system may vary depending upon the goals of the study. The goal is to find bottlenecks and reduce wall clock time.

Select Metrics: The next step is to select criteria to compare the performance. These criterias are called metrics. In general, the metrics are related to the speed, accuracy, and availability of services. The performance of a network, for example, is measured by the speed (throughput and delay), accuracy (error rate), and availability of the packets sent. The performance of a processor is measured by the speed of (time taken to execute) various instructions. Metrics used in this thesis are, time taken to execute a part of program, speedup and network throughput.

List Parameters: The next step in performance projects is to make a list of all the parameters that effect performance. The list can be divided into system parameters and workload parameters. System parameters include both hardware and software parameters. Workload parameters are characteristics of users' requests. In this work parameters were architecture, number of processers, data size and PDE error tolerance.

Select Factors to Study: The list of parameters can be divided into two parts: those that will be varied during the evaluation and those that will not. The parameters to be varied are called factors and their values are called levels. In general, the list of factors, and their possible levels, is larger than what the available resources will allow. Otherwise, the list will keep growing until it becomes obvious that there are not enough resources to study the problem. It is better to start with a short list of factors and a small number of levels for each factor and to extend the list in the next phase of the project if the resources permit. In this thesis, different number of processors and data size used to show scalability.

Select Evaluation Technique: The three broad techniques for performance evaluation are analytical modeling, simulation, and measuring a real system. The selection of the right technique depends upon the time and resources available to solve the problem and the desired level of accuracy. In this work, real system values measures with TAU.

Select Workload: The workload consists of a list of service requests to the system. Depending upon the evaluation technique chosen, the workload may be expressed in different forms. For analytical modeling, the workload is usually expressed as a probability of various requests. For simulation, one could use a trace of requests measured on a real system. For measurement, the workload may consist of user scripts to be executed on the systems. In PDE algorithms data size defines programs workload. Various workloads used to show scalability.

Design Experiments: The goal is to determine the relative effect of various factors. In most cases, this can be done with fractional factorial experimental designs. In the second phase, the number of factors is reduced and the number of levels of those factors that have significant impact is increased. Experiments are done by running program and collecting information using TAU.

Analyze and Interpret Data: In comparing two experiments, it is necessary to take into account the variability of the results. Simply comparing the means can lead to inaccurate conclusions. It must be understood that the analysis only produces results and not conclusions. The results provide the basis on which the analysts or decision makers can draw conclusions. TAU has visualization tools for interpreting data. Paraprof is used for profiling visualization and Jumpshot is used for trace visualization.

Present Results: The final step of all performance projects is to communicate the results to other members of the decision-making team. It is important that the results be presented in a manner that is easily understood. This usually requires presenting the results in graphic form and without statistical title.

4.2 Instrumentation

In order to observe performance, additional instructions or probes are typically inserted into a program. This process is called instrumentation. As events execute, they activate the probes, which perform measurements. Thus, instrumentation exposes key characteristics of an execution. Instrumentation can be introduced in a program at several levels of the program transformation process. Instrumentation methods below are used in this thesis.

Selective Instrumentation: Selective instrumentation is based on definitions of listed events to be included or excluded for measurement. TAU supports this feature by using an instrumentation file. The idea is to record a list of performance events to be included or excluded by the instrumentation in a file. The file is then used during the instrumentation process to restrict the event set. The basic structure of the file is a list of names separated into include and exclude lists. File names can be given to restrict instrumentation focus. Exclusion is used to eliminate unwanted performance events, such as high frequency, small routines that generate excessive measurement overhead, and provide easy event configuration for customized performance experiments. Selective instrumentation is used for defining calculation areas of algorithm.

Preprocessor-Based Instrumentation: The source code of a program can be altered by a preprocessor before it is compiled. This approach typically involves parsing the

source code to infer where instrumentation probes are to be inserted. As an example of automatic instrumentation through the preprocessing built into a compiler, TAU's memory allocation/deallocation tracking package can be used to re-direct the references to the C malloc/free calls. The preprocessor invokes TAU's corresponding memory wrapper calls with the added information about the line number and the file. The atomic event interface can then track the size of memory allocated and deallocated to help locate potential memory leaks. Preprocessor-based instrumentation is also commonly used to insert performance measurement calls at interval entry and exit points in the source code. To support automatic performance instrumentation at the source level, the TAU project has developed the Program Database Toolkit (PDT) [31]. The purpose of PDT, shown in Figure 4.2 is to parse the application source code and locate the semantic constructs to be instrumented. PDT is comprised of commercial-grade front-ends that emit an intermediate language (IL) file, IL analyzers that walk the abstract syntax tree and generate a subset of semantic entities in program database (PDB) ASCII text files, and a library interface (DUCTAPE) to the PDB files that allows to write static analysis tools. When the application is executed subsequently, performance data is generated. TAU also supports OpenMP instrumentation using a preprocessor tool called Opari [32]. Opari inserts POMP [32] annotations and rewrites OpenMP directives in the source code. TAU's POMP library tracks the time spent in OpenMP routines based on each region in the source code. To track the time spent in user-level routines, Opari instrumentation can be combined with PDT based instrumentation as well. Opari is used with TAU to measure OpenMP performance.

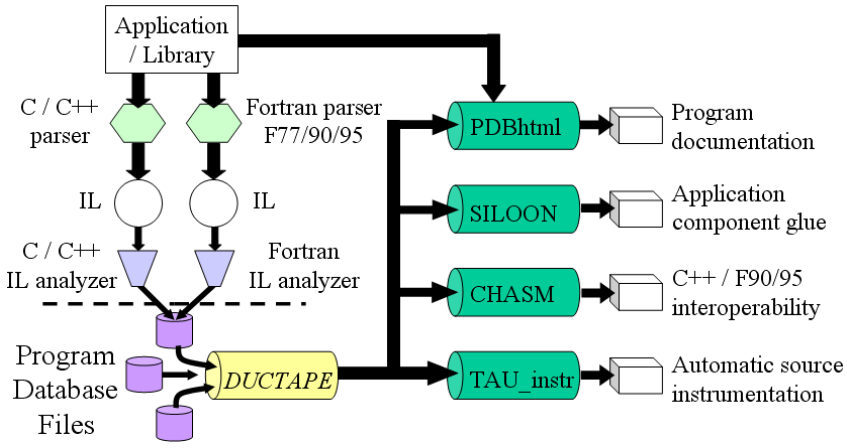

Figure 4.2 : Program Database Toolkit Diagram [3]

Wrapper Library-Based Instrumentation: A common technique to instrument library routines is to substitute the standard library routine with an instrumented version, which in turn calls the original routine. The problem is that you would like to do this without having to develop a different library just to alter the calling interface. MPI provides an interface [33] that allows a tool developer to intercept MPI calls in a portable manner without requiring a vendor to supply proprietary source code of the library and without requiring the application source code to be modified by the user. This is achieved by providing hooks into the native library. The advantage of this approach is that library level instrumentation can be implemented by defining a wrapper interposition library layer that inserts instrumentation calls before and after calls to the native routines. TAU has a TAU MPI wrapper library that intercepts calls to the native library by defining routines with the same name, such as MPI_Send. These routines then call the native library routines with the name shifted routines, such as PMPI_Send. Wrapped around the call, before and after, is TAU performance instrumentation. An added advantage of providing such a wrapper interface is that the profiling wrapper library has access to not only the routine transitions, but also to the arguments passed to the native library. This allows TAU to track the size of messages, identify message tags, or invoke other native library routines. This type of instrumentation is used for MPI functions in this thesis.

4.3 Measurement

The instrumentation layer is responsible for defining the performance events for an experiment, establishing relationships between events, and managing those events in the context of the parallel computing model being used. Measurement is done through the probes inserted in instrumentation.

Figure 4.3 shows TAU instrumentation and measurement architecture.

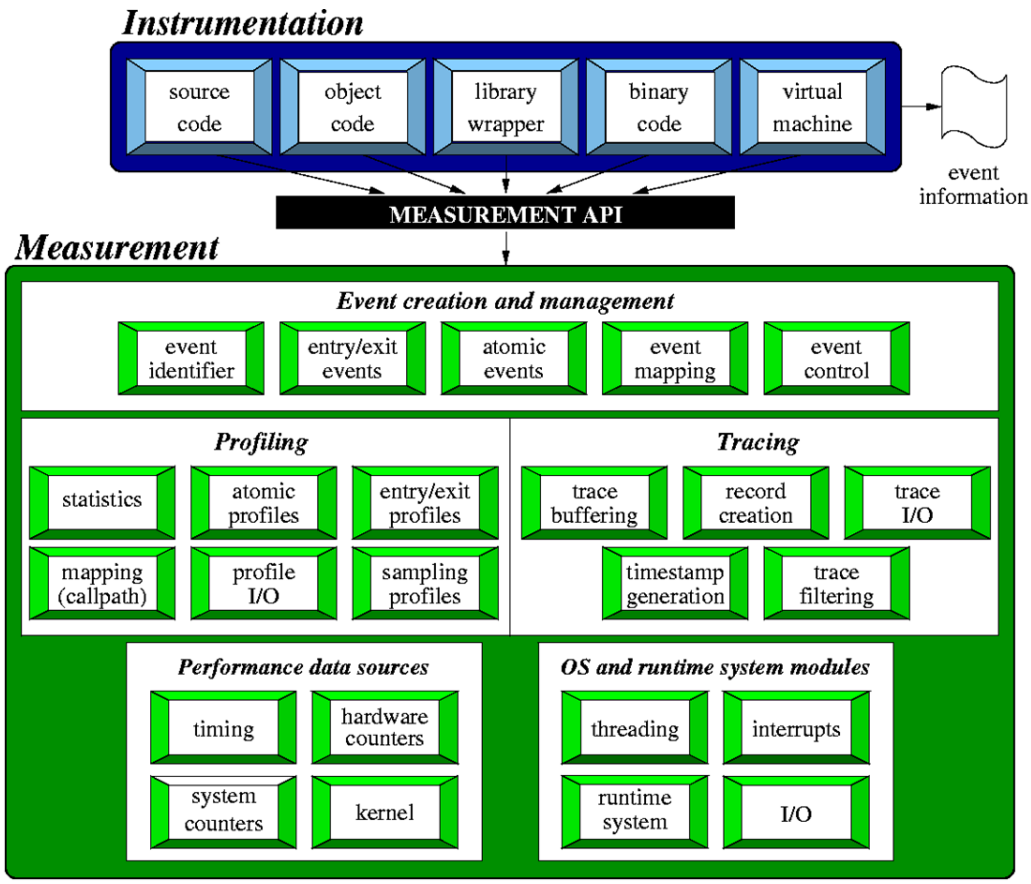

Figure 4.3 : Architecture of TAU (Instrumentation and Measurement) [3]

4.3.1 Profile of an Algorithm

Profiling characterizes the behavior of an application in terms of aggregate performance metrics. Profiles are typically represented as a list of various metrics (such as wall-clock time) and associated statistics for all performance events in the program. There are different statistics kept for interval events (such as routines or statements in the program) versus atomic events. For interval events, TAU profile measurements compute exclusive and inclusive metrics spent in each routine.

The TAU profiling system supports several profiling variants [3]. The most basic and standard type of profiling is called flat profiling. If TAU is being used for flat profiling, performance measurements are kept for interval events only. For instance, flat profiles will report the exclusive performance (e.g. time) for a routine, say A, as the amount of time spent executing in A exclusively. Any time spent in routines called by A will be represented in A's profile as inclusive time, but it will not be differentiated with respect to the individual routines A called. Flat profiles also keep

information on the number of times A was called and the number of routines (i.e. events) called by A. Again, TAU will keep a flat profile for every node/ /context/thread of the program's execution.

Depth of flat profiling is one hence TAU can be configured for deeper profiling. Depth of profiling can be increased.

4.3.2 Trace of an Algorithm

While profiling is used to get aggregate summaries of metrics in a compact form, it cannot highlight the time varying aspect of the execution. Event tracing usually results in a log of the events that characterize the execution. Each event in the log is an ordered row typically containing a time stamp, a location (e.g. node, thread), an identifier that specifies the type of event (e.g. routine transition, user-defined event, message communication, etc.) and event-specific information. With tracing enabled, every node/context/thread will generate a trace for instrumented events. TAU will write traces in its modern trace format as well as in VTF3 format. Support for a counter value to be included in event records is fully implemented. In addition, certain standard events are known by TAU's tracing system, such as multi-threading operations and message communication [3].

TAU also supports runtime trace reading and analysis, it is important to understand what takes place when TAU records performance events in traces. Also in case of a program crash traces generated so far will remain, this can help the user to find point of crash.

4.4 Analysis

Analysis is interpretation of collected performance data. Several tools can be used to visualize performance data. TAU gives the ability to track performance data in widely diverse environments, and thus provides a wealth of information to the user. The usefulness of this information, however, is highly dependent on the ability of analysis toolsets to manage and present the information. As the size and complexity of the performance information increases, the challenge of performance analysis and visualization becomes more difficult. TAU supports different visualization tools. In this thesis, ParaProf is used for profile analysis and Jumpshot is used for trace data. Below Figure 4.4 shows analysis architecture of TAU [3].

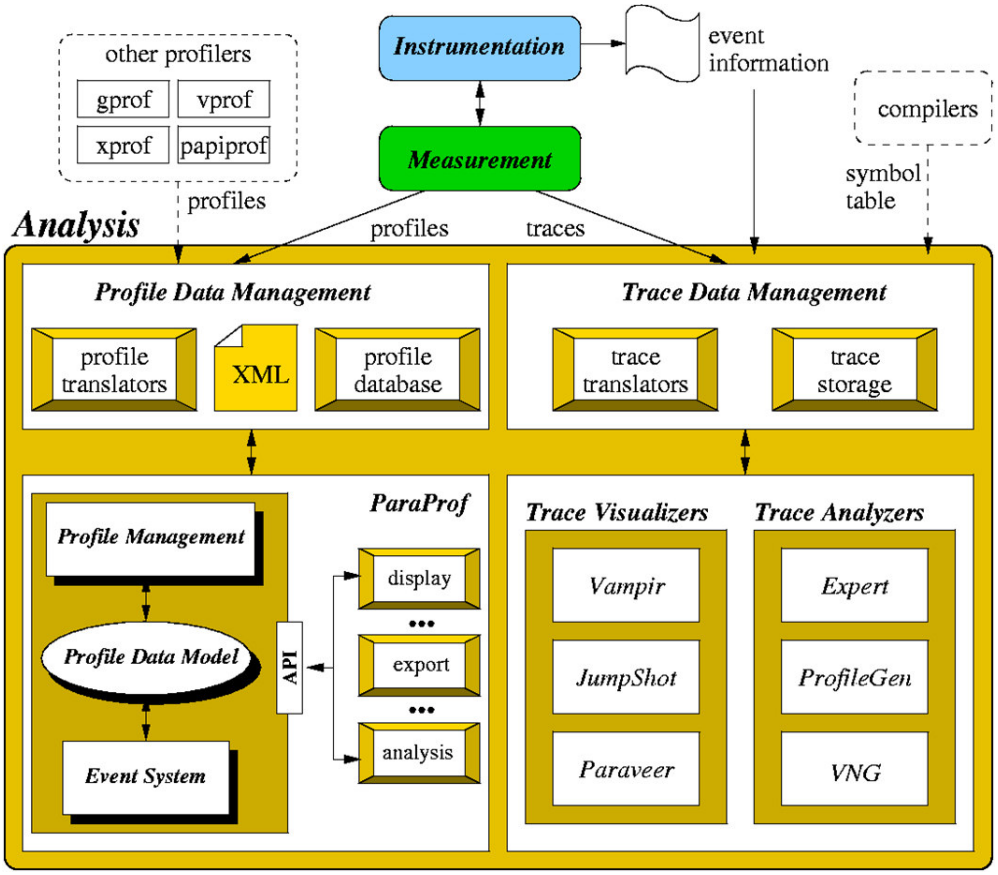

Figure 4.4 : Architecture of TAU (Analysis and Visualization) [3]

Both ParaProf and Jumpshot are capable of handling large size of performance data. TAU supports different profile and trace file formats with file converters.

5. COMPUTATION OPTIMIZATIONS

Parallel algorithms, used in High Performance Computing (HPC) are making intensive floating-point calculations that take long time. For this reason, any optimization in parallel algorithm saves significant time even if the percentage of optimization is low. Computational optimizations are algorithm specific; algorithms are modified to eliminate branches and used alternative instructions, which take less computational time.

5.1 Objectives

One might reduce the amount of time that a program takes to perform some task at the price of making it consume more memory. In an application where memory space is at a premium, one might deliberately choose a slower algorithm in order to use less memory. Often there is no "one size fits all" design which works well in all cases, so engineers make trade-offs to optimize the attributes of greatest interest. Additionally, the effort required to make a piece of software completely optimal is almost always not needed when more than significant speedup left; so the process of optimization may be halted before a completely optimal solution has been reached. Fortunately, it is often the case that the greatest improvements come early in the process.

In this thesis, overall wall clock time of the PDE algorithm is tried to be reduced. Performance evaluation starts with finding most time consuming the part of the algorithm. After improving that part, another section of code is selected for performance improvement.

5.2 Optimization Levels

Techniques used in optimization can be broken up among various levels, which can affect anything from a single statement to the entire program. In addition to scoped optimizations, there are two further general categories of optimization:

Programming language-independent vs. language-dependent: Most high-level languages share common programming constructs and abstractions; decision (if, switch, case), looping (for, while, repeat.. until, do.. while), encapsulation (structures, objects). Thus, similar optimization techniques can be used across languages. However, certain language features make some kinds of optimizations difficult. For instance, 2D matrix data order in C is row wise but in FORTRAN it is column wise. This is important if matrix data is processed in nested two loops; loops order changes cache performance. Software developer must be aware of the programming languages characteristics.

Machine independent vs. machine dependent: Many optimizations that operate on abstract programming concepts (loops, objects, structures) are independent of the machine targeted by the compiler, but many of the most effective optimizations are those that best exploit special features of the target platform. RISC and CISC processors have different instruction sets. Software must be compiled for its processor architecture. Easy way of machine dependent optimization is leaving it to the compiler and forcing compiler to use machine dependent optimizations.

For instance, in the case of compile-level optimization, platform independent techniques are generic techniques such as loop unrolling, reduction in function calls, memory efficient routines, reduction in conditions, etc., that impact most CPU architectures in a similar way. Generally, these serve to reduce the total instruction path length required to complete the program and/or reduce total memory usage during the process. On the other side, platform dependent techniques involve instruction scheduling, instruction level parallelism, data level parallelism, cache optimization techniques, i.e. parameters that differ among various platforms; the optimal instruction scheduling might be different even on different processors of the same architecture.

In this thesis, platform dependent optimizations are based on compiler level using optimization level O3. Otherwise, CPU dependent optimization must be done using assembler, which is not available in IA64 architecture compilers. Compilers for specific architectures like IA64 are capable of doing CPU dependent optimization. Compilers analyses code and decides optimization. Compilers need simple, clear and data independent algorithms for better optimization. In this thesis compiler cannot

optimize code version 1 but after changing loop properties compiler is able to make optimization.

Optimization can occur at a number of 'levels'. These levels are described below.

5.2.1 Design level

At the highest level, the design may be optimized to make best use of the available resources. The implementation of this design will benefit from a good choice of efficient algorithms and the implementation of these algorithms will benefit from writing good quality code. The architectural design of a system overwhelmingly affects its performance. The choice of algorithm effects efficiency more than any other item of the design. In some cases, however, optimization relies on using fancier algorithms, making use of special cases and special tricks and performing complex trade-offs; thus, a fully optimized program can sometimes, if insufficiently commented, be more difficult for less experienced programmers to comprehend and hence may contain more faults than unoptimized versions.

In this thesis, a PDE solver algorithm has been analyzed and optimized. In the algorithm, Gauss-Seidel method is used for solving PDE. 2-D PDE algorithms are commonly used, and simple to understand. Algorithms matrix data distribution is row-wise block stripped 1D decomposition. To achieve Gauss-Seidel method multicoloring algorithm is used with three colors and nine stencils. Algorithm will be explained in details later.

5.2.2 Source code level

Avoiding bad quality coding can also improve performance, by avoiding obvious slowdowns. Parallel algorithm used in this thesis has three calculation blocks for three colors. Each block contains two nested loops to calculate new values of matrix at each iteration. However, each block has to calculate its color not all points. To test the points color if branch was used; from now on, this version of algorithm will be called as version 1. This part of the code is altered to eliminate if condition test. After eliminating if condition, calculation block speeds up nearly five times. Below is the code snippet of algorithms version1 and version 2.

Version 1:

```
for(i=2; i<rows_local-2; i++)
    for(j=2; j<cols_local-2; j++){
        if((i+j-global_start) % 3 == colorC){
            temp = A(i,j);
            A(i,j) = 0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2)+A(i-1,j)
                    +A(i+1,j)+A(i,j-1)+A(i,j+1) );
            if( fabs(temp - A(i,j)) > tol && iter%10 == 0)
                  done = FALSE;
      }
}
```

Version 2:

```
start=(colorC+global_start-2)%3;
if (start<2)
   start+=3;

for(i=2; i<rows_local-2; i++) {
   for (j=start;j<cols_local-2;j=j+3) {
      temp = A(i,j);
      A(i,j) = 0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2) + A(i-1,j)
                  +A(i+1,j)+A(i,j-1)+A(i,j+1) );
      if(iter%10 == 0 && fabs(temp - A(i,j)) > tol  )
            done = FALSE;
   }
   if (start > 2)
      start--;
   else start+=2;
}
```

5.2.3 Compiler level

Compiler optimization is the process of tuning the output of a compiler to minimize or maximize some attribute of an executable computer program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. In this thesis, compiler optimization levels O2 and O3 are used to achieve compiler level optimization. If source code is compiled with –O3 flag the optimization is will be CPU dependent. Programs, compiled at optimization level O3, may not run on different processors. Compilers optimization levels does not always decrease programs run time, this is shown with the experiments.

5.2.4 Assembly level

At the lowest level, writing code using an assembly language designed for a particular hardware platform will normally produce the most efficient code since the programmer can take advantage of the full repertoire of machine instructions. Unfortunately, c compilers in ia64 architecture do not allow using inline assembly in c. No assembly level optimizations have been made in this work.

6. COMMUNICATION OPTIMIZATIONS

Parallel algorithms exchange their data using communication methods. This communication can be point-to-point or collective. In MPI algorithms, the programmer must design these communication steps. In contrast, OpenMP has a seamless processor communication. Programmer does not use special communication functions, because all memory is accessible to all processors.

6.1 Objectives

In PDE based iterative high performance parallel computing algorithms, communication takes place at each iteration. Iteration count effects communication time. If communication time can be reduced, whole program will benefit from this optimization.

Two parameters effect communication time, latency and bandwidth. Latency refers to any of several kinds of delays typically incurred in processing of network data. Mostly latency is referred as time taken for transfer of a zero sized packet. Bandwidth is a measure of available or consumed data communication resources expressed in bit/s. Bandwidth effects data packets travel time.

In this work, MPI communication time is tried to reduce using persistent communication methods.

6.2 Communication Methods

MPI supports both point-to-point and collective communications.

6.2.1 Point-to-Point Communication

MPI provides many ways to send and receive messages. Each routine has different types based on blocking, non-blocking [9]. These routines give flexibility to the programmer.

Send routines (match any receive, probe; non-blocking can match any completion/testing)

- Blocking - standard, buffered, ready, synchronous
- Non-blocking - standard, buffered, ready, synchronous
- Persistent - standard, buffered, ready, synchronous

Receive routines (match any send)

- Blocking
- Non-blocking
- Persistent

Probe routines (match any send)

- Blocking
- Non-blocking

Completion / Testing routines (match any non-blocking send/receive)

- Blocking - one, some, any, all
- Non-blocking - one, some, any, all

6.2.2 Collective Communication

Collective communication must involve all processes in the scope of a communicator. There are three types of collective operations [9].

- **Synchronization:** All processes wait until all members of the group have reached the synchronization point.
- **Data Movement:** broadcast, scatter/gather, all to all.
- **Collective Computation (reductions):** One member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

6.3 Hardware Based Optimizations

Network devices performance directly effects communication time. We can analyze network hardware with two parameters; latency and bandwidth.

If latency is high small messages becomes more expensive. In high latency when message size is small enough latency becomes dominant in communication time. In very low latency, communication can be split into smaller parts to make immediate computation.

If bandwidth is too low data to be transmitted takes more time. In low bandwidth networks, data compression can be used. Data compression decreases data size but requires extra CPU time. If processing power is higher than bandwidth data compression decreases wall clock time. There are several types of data compression algorithms for different data types [34].

In NCHPC both two architectures has low latency high bandwidth interconnection. Cluster network device is Infiniband DDR 4X. Infiniband DDR 4X has 2000MB/s bandwidth and 140 nanosecond latency. HP Integrity Superdome has 27.3 GB/s bandwidth [28]. No hardware-based optimization is made. Instead algorithm based optimizations are experimented.

6.4 Algorithm Based Optimizations

MPI supports different types of communication methods. These methods have better performances in different sizes of messages. Figure 6-1, 6-2, 6-3 shows point-to-point performance of these methods [35].

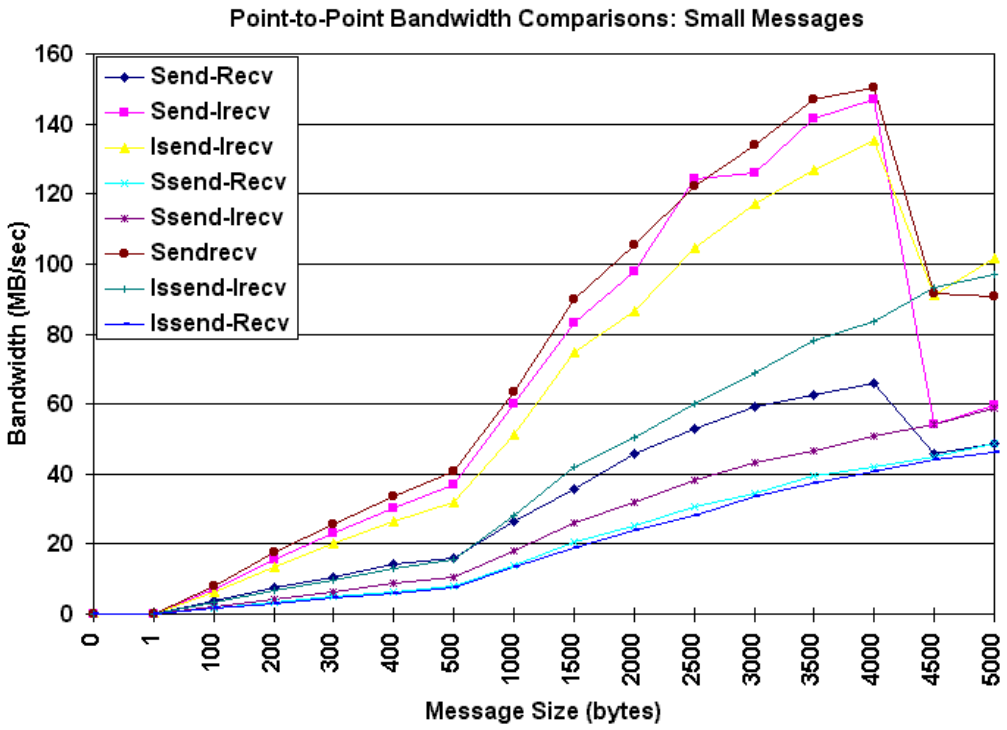

Figure 6.1 : Small Messages Performance

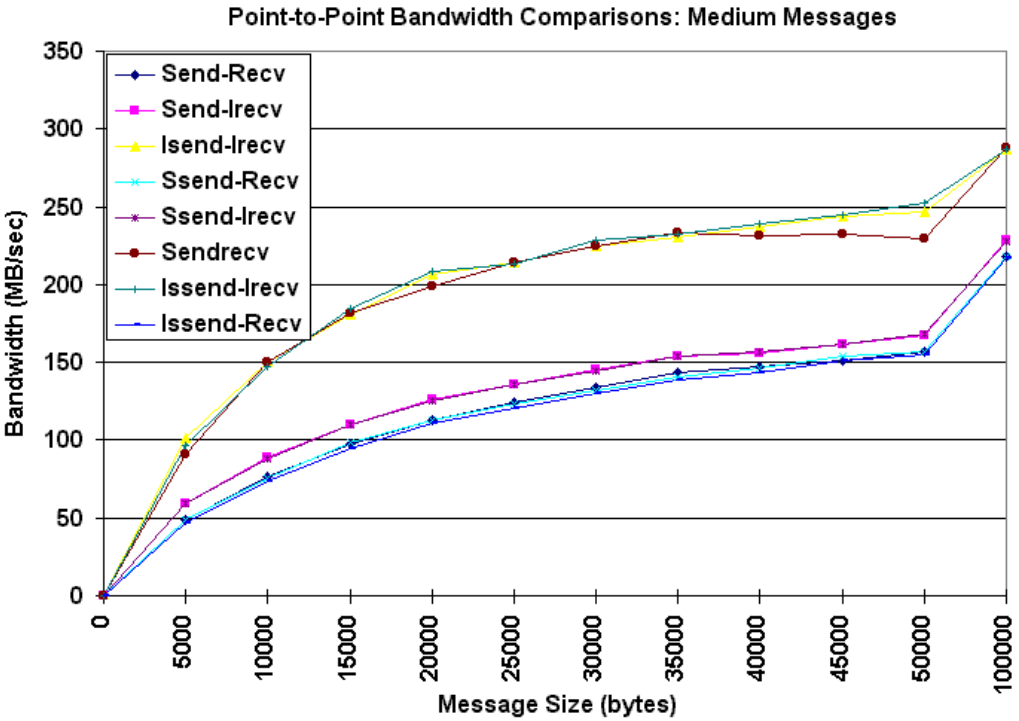

Figure 6.2 : Medium Messages Performance

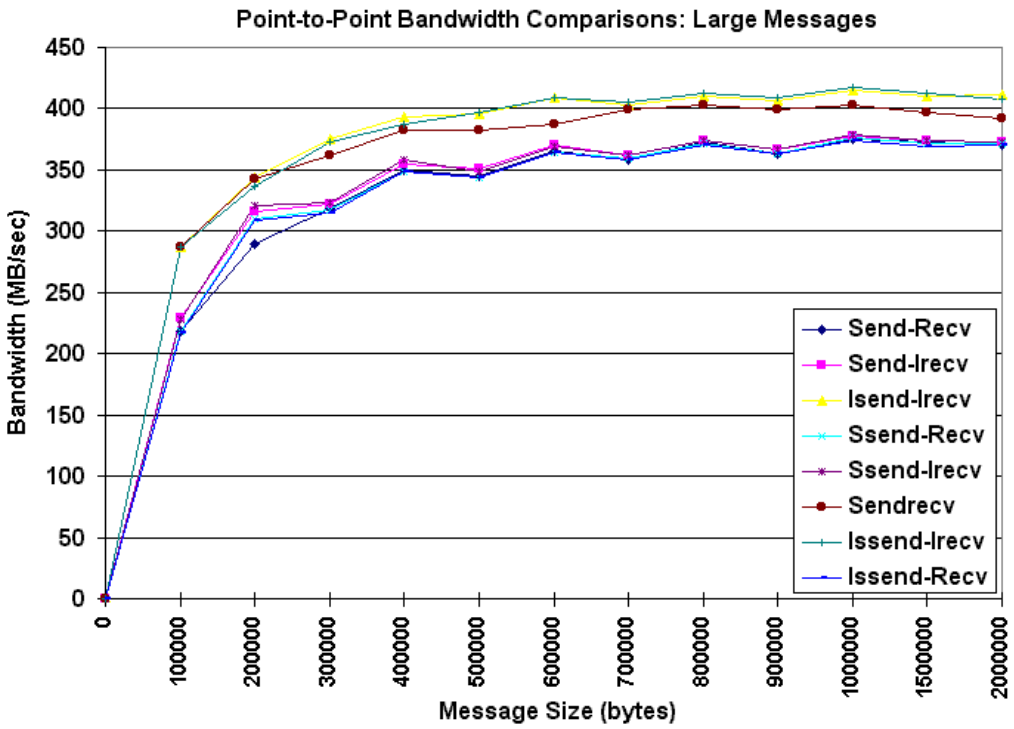

Figure 6.3 : Large Messages Performance

MPI Sendrecv method has high performance on every size of messages. For scalability and easy of programming MPI Sendrecv is used for point-to-point communication. In PDE like iterative methods, communication takes place with same nodes in each iteration. MPI has a persistent connection method for these types of connections. MPI persistent communications can be used to reduce communication overhead for repeatedly called point-to-point message passing routines with the same arguments. Persistent communications improvement is shown in the Figure 6.4 below [35].

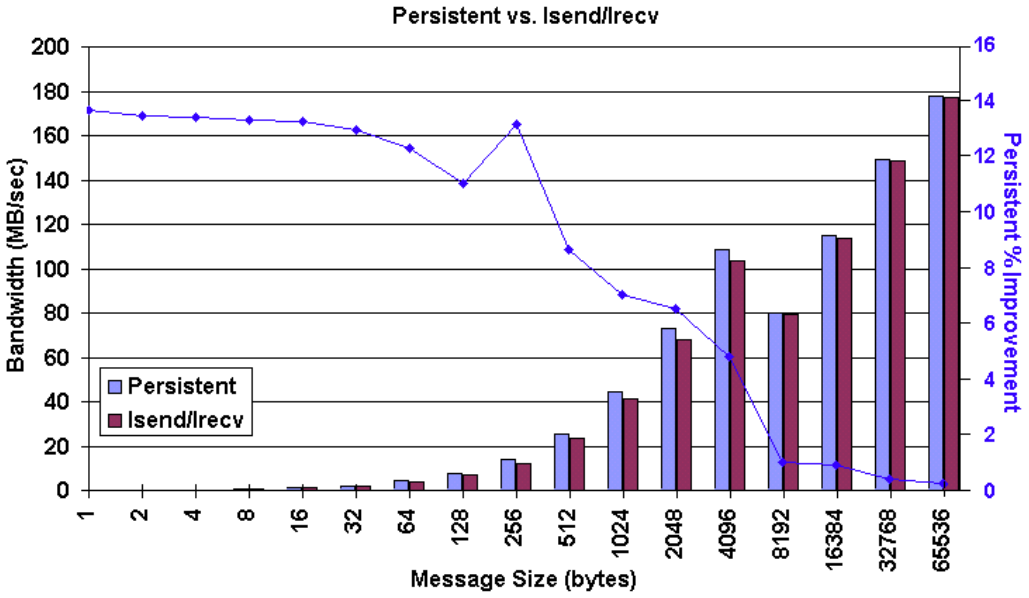

Figure 6.4 : Persistent vs Isen/Irecv

7. PARALLELIZATION OF PARTIAL DIFFERENTIAL EQUATIONS

In mathematics, partial differential equations (PDE) are a type of differential equation, i.e., a relation involving an unknown function of several independent variables and its partial derivatives with respect to those variables. Partial differential equations are used to formulate, and thus aid the solution of, problems involving functions of several variables; such as the propagation of sound or heat, electrostatics, electrodynamics, fluid flow, and elasticity.

The solution procedure of a partial differential equation depends on the type of the equation. Partial differential equations can be classified as linear or nonlinear. In a linear PDE, the dependent variable and its derivatives enter the equation linearly. On the other hand, a nonlinear PDE contains a product of the dependent variable and/or a product of its derivatives [36].

Some linear, second-order partial differential equations can be classified as parabolic, hyperbolic or elliptic.

Mathematically, a partial differential equation of the form

$$Au_{xx} + Bu_{xy} + Cu_{yy} + Du_x + Eu_y + F = 0 \qquad \textbf{(7-1)}$$

Parabolic PDE: A parabolic partial differential equation is a type of second-order partial differential equation, describing a wide family of problems in science including heat diffusion and stock option pricing. These problems, also known as evolution problems, describe physical or mathematical systems with a time variable, and which behave essentially like heat diffusing through a medium like a metal plate.

If equation satisfies $B^2 - 4AC = 0$ it is called parabolic.

Hyperbolic PDE: The wave equation is an example of a hyperbolic partial differential equation.

If equation satisfies $B^2 - 4AC > 0$ it is called hyperbolic.

Elliptic PDE: It can be defined on spaces of complex-valued functions, or some more general function-like objects. What is distinctive is that the coefficients of the highest-order derivatives satisfy a positivity condition. An important example of an elliptic operator is the Laplacian.

If equation satisfies $B^2 - 4AC < 0$ it is called elliptic.

7.1 Finite Difference as a Discretization Model

An important application of finite differences is in numerical analysis, especially in numerical differential equations, which aim at the numerical solution of ordinary and partial differential equations respectively. The idea is to replace the derivatives appearing in the differential equation by finite differences that approximate them. The resulting methods are called finite difference methods.

Typical elliptic equations in a two-dimensional Cartesian system are Laplace's equations,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \qquad \textbf{(7-2)}$$

and Poisson's equation,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \qquad \textbf{(7-3)}$$

These model equations are used to investigate a variety of solution procedures.

Of the various existing finite difference formulations, the so-called "five-point formula" is the most commonly used. In this representation of the PDE, central differencing which is second order accurate is utilized. Therefore, model Equation (7-2) is approximated as

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2} = 0 \qquad \textbf{(7-4)}$$

The corresponding points are shown in Figure 7.1

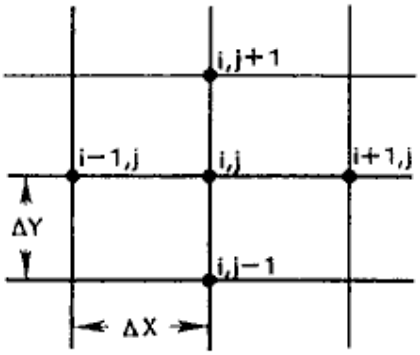

Figure 7.1 : Grid points for a five point formula

A higher order formulation is the nine-point formula, which uses a fourth-order approximation for the derivatives. With this formulation, the PDE of model Equation is:

$$\frac{-u_{i-2,j}+16u_{i-1,j}-30u_{i,j}+16u_{i+1,j}-u_{i+2,j}}{12(\Delta x)^2} + \frac{-u_{i,j-2}+16u_{i,j-1}-30u_{i,j}+16u_{i,j+1}-u_{i,j+2}}{12(\Delta y)^2}=0 \qquad \textbf{(7-5)}$$

The grid point involved in Equation (7-5) is shown in Figure 7.2

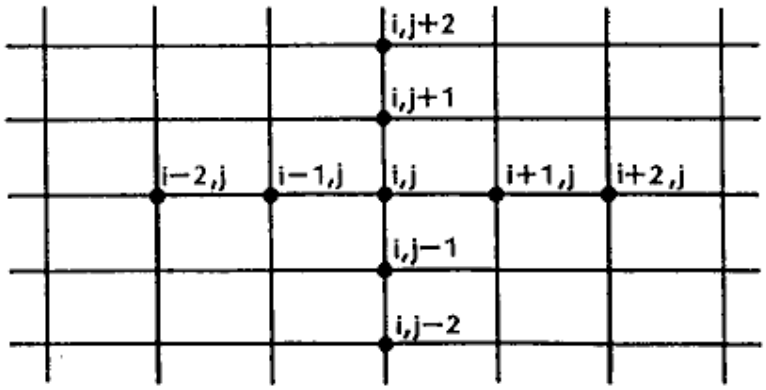

Figure 7.2 : Grid points for a nine-point formula

One obvious difficulty with the application of this formula is the implementation of the boundary conditions. Thus, for problems where higher accuracy is required, it is easier to use the five-point formula with small grid sizes than the fourth-order accurate nine-point formula. Due to its simplicity, the five-point formula represented by Equation (7-4) will be considered. Rewrite Equation (7-4) as

$$u_{i+1,j} - 2u_{i,j} + u_{i-1,j} + \left(\frac{\Delta x}{\Delta y}\right)^2 \left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right) = 0 \qquad \textbf{(7-6)}$$

Define the ratio of step sizes as β, so that $\beta = \Delta x / \Delta y$. By rearranging the terms in Equation (7-6), one obtains

$$u_{i+1,j} + u_{i-1,j} + \beta^2 u_{i,j+1} + \beta^2 u_{i,j-1} - 2(1+\beta^2)u_{i,j} = 0 \qquad \textbf{(7-7)}$$

In order to explore various solution procedures, first consider a square domain with Diriclet boundary conditions. For instance, a simple 6x6 grid system (Figure 7-3) subject to the following boundary conditions is considered:

$$x = 0 \quad u = u_2 \;, \qquad y = 0 \quad u = u_1$$
$$x = L \quad u = u_4 \;, \qquad y = H \quad u = u_3$$

Applying Equation (7-7) to the interior grid points produces sixteen equations with sixteen unknowns. The equations are:

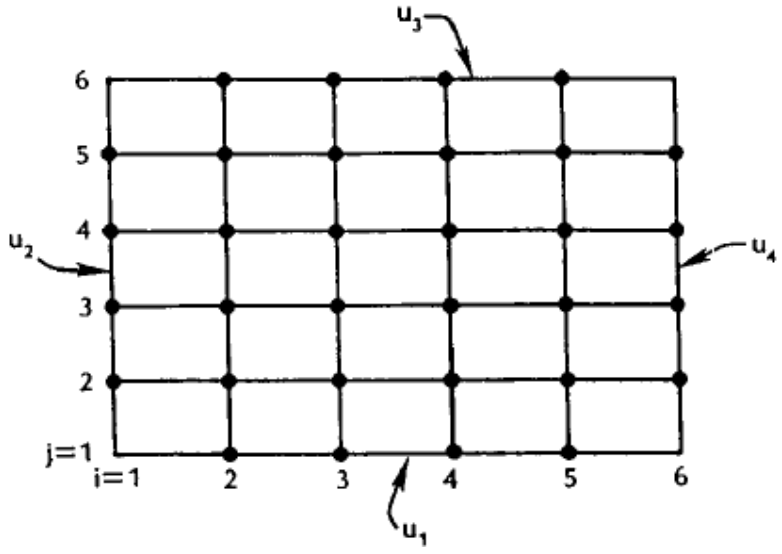

Figure 7.3 : Grid system used for solution of Equation (7-7)

$$u_{3,2} + u_{1,2} + \beta^2 u_{2,3} + \beta^2 u_{2,1} - 2(1+\beta^2)u_{2,2} = 0$$

$$u_{4,2} + u_{2,2} + \beta^2 u_{3,3} + \beta^2 u_{3,1} - 2(1+\beta^2)u_{3,2} = 0$$

$$u_{5,2} + u_{3,2} + \beta^2 u_{4,3} + \beta^2 u_{4,1} - 2(1+\beta^2)u_{4,2} = 0$$

$$u_{6,2} + u_{4,2} + \beta^2 u_{5,3} + \beta^2 u_{5,1} - 2(1+\beta^2)u_{5,2} = 0$$

$$u_{3,3} + u_{1,3} + \beta^2 u_{2,4} + \beta^2 u_{2,2} - 2(1+\beta^2)u_{2,3} = 0$$

$$u_{4,3} + u_{2,3} + \beta^2 u_{3,4} + \beta^2 u_{3,2} - 2(1+\beta^2)u_{3,3} = 0$$

$$u_{5,3} + u_{3,3} + \beta^2 u_{4,4} + \beta^2 u_{4,2} - 2(1+\beta^2)u_{4,3} = 0$$

$$u_{6,3} + u_{4,3} + \beta^2 u_{5,4} + \beta^2 u_{5,2} - 2(1+\beta^2)u_{5,3} = 0$$

$$u_{3,4} + u_{1,4} + \beta^2 u_{2,5} + \beta^2 u_{2,3} - 2(1+\beta^2)u_{2,4} = 0$$

$$u_{4,4} + u_{2,4} + \beta^2 u_{3,5} + \beta^2 u_{3,3} - 2(1+\beta^2)u_{3,4} = 0$$

$$u_{5,4} + u_{3,4} + \beta^2 u_{4,5} + \beta^2 u_{4,3} - 2(1+\beta^2)u_{4,4} = 0$$

$$u_{6,4} + u_{4,4} + \beta^2 u_{5,5} + \beta^2 u_{5,3} - 2(1+\beta^2)u_{5,4} = 0$$

$$u_{3,5} + u_{1,5} + \beta^2 u_{2,6} + \beta^2 u_{2,4} - 2(1+\beta^2)u_{2,5} = 0$$

$$u_{4,5} + u_{2,5} + \beta^2 u_{3,6} + \beta^2 u_{3,4} - 2(1+\beta^2)u_{3,5} = 0$$

$$u_{5,5} + u_{3,5} + \beta^2 u_{4,6} + \beta^2 u_{4,4} - 2(1+\beta^2)u_{4,5} = 0$$

$$u_{6,5} + u_{4,5} + \beta^2 u_{5,6} + \beta^2 u_{5,4} - 2(1+\beta^2)u_{5,5} = 0$$

The equations gathered from this grid are expressed in a matrix form as

$$
\begin{bmatrix}
\alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & \alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & \alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & \alpha & 0 & 0 & 0 & \beta^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\beta^2 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \beta^2 & 0 & 0 & 1 & \alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha & 0 & 0 & 0 & \beta^2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \beta^2 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha & 1 & 0 & 0 & \beta^2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha & 0 & 0 & 0 & \beta^2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \beta^2 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha
\end{bmatrix}
\begin{bmatrix} u_{2,2} \\ u_{3,2} \\ u_{4,2} \\ u_{5,2} \\ u_{2,3} \\ u_{3,3} \\ u_{4,3} \\ u_{5,3} \\ u_{2,4} \\ u_{3,4} \\ u_{4,4} \\ u_{5,4} \\ u_{2,5} \\ u_{3,5} \\ u_{4,5} \\ u_{5,5} \end{bmatrix}
=
\begin{bmatrix} -u_{1,2} - \beta^2 u_{2,1} \\ -\beta^2 u_{3,1} \\ -\beta^2 u_{4,1} \\ -u_{6,2} - \beta^2 u_{5,1} \\ -u_{1,3} \\ 0 \\ 0 \\ -u_{6,3} \\ -u_{1,4} \\ 0 \\ 0 \\ -u_{6,4} \\ -u_{1,5} - \beta^2 u_{3,6} \\ -\beta^2 u_{3,6} \\ -\beta^2 u_{4,6} \\ -u_{6,5} - \beta^2 u_{5,6} \end{bmatrix}
$$

(7-8)

Where $\alpha = -2(1+\beta^2)$.

The matrix formulation has two noteworthy features. First, it is a pentadiagonal matrix with nonadjacent diagonals; and second, the elements in the main diagonal in each row are the largest. These features are important when developing solution procedures [36].

7.2 Gauss-Seidel and SOR

The Gauss–Seidel method is a technique used to solve a linear system of equations. It is defined on matrices with non-zero diagonals, but convergence is only guaranteed if the matrix is either diagonally dominant, or symmetric and (semi) positive definite. In this method, the current values of the dependent variable are used to compute the neighboring points as soon as available. This will certainly increase the convergence rate dramatically over the Jacobi method (about 100%) [36]. The method is convergent if the largest elements are located in the main diagonal of the coefficient matrix, as in the case of the formulation that produced (7-8). The formal requirement (sufficient condition) for the convergence of the method is,

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \qquad \textbf{(7-9)}$$

And, at least for one row,

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \qquad \textbf{(7-10)}$$

Since this is a sufficient condition, the method may converge even though the condition is not met for all rows. Now the formulation of the method is considered. The finite difference equation is given here:

$$u_{i,j} = \frac{1}{2(1+\beta^2)}\left[u_{i+1,j} + u_{i-1,j} + \beta^2 (u_{i,j+1} + u_{i,j-1})\right] \qquad \textbf{(7-11)}$$

In order to solve for the value of u at grid point i,j, the values of u on the right-hand side must be provided. This procedure is easy to understand if the application of Equation (7-11) considered as a few grid points. For the computation of the first point, example (2,2), as shown in Figure 7.4, it follows that

$$u_{2,2}^{k+1} = \frac{1}{2(1+\beta^2)}\left[u_{3,2} + u_{1,2} + \beta^2 (u_{2,3} + u_{2,1})\right] \qquad \textbf{(7-12)}$$

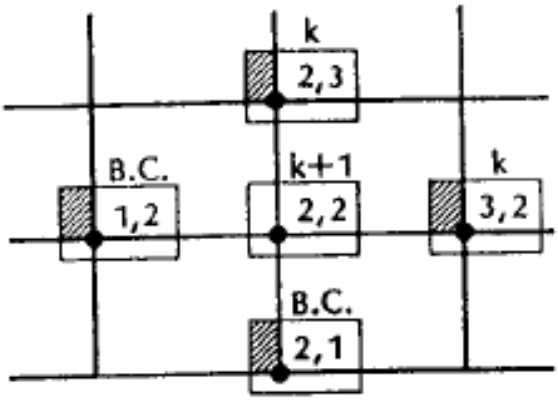

Figure 7.4 : Grid points for Equation (7-13)

In this equation, $u_{2,1}$ and $u_{1,2}$ are provided by the boundary conditions. Only two values, namely $u_{3,2}$ and $u_{2,3}$, use the values from the previous iteration at k. Thus, in terms of the iteration levels,

$$u_{2,2}^{k+1} = \frac{1}{2(1+\beta^2)}\left[u_{3,2}^{k} + u_{1,2} + \beta^2(u_{2,3}^{k} + u_{2,1})\right] \qquad \textbf{(7-13)}$$

Now, for point (3,2), one has

$$u_{3,2}^{k+1} = \frac{1}{2(1+\beta^2)}\left[u_{4,2}^{k} + u_{2,2}^{k+1} + \beta^2(u_{3,3}^{k} + u_{3,1})\right] \qquad \textbf{(7-14)}$$

In this equation, $u_{3,1}$ is provided by the boundary condition, and $u_{4,2}$ and $u_{3,3}$ are taken from the previous computation; but $u_{2,2}$ is given by Equation (7-13).

Finally, the general formulation provides the equation

$$u_{i,j}^{k+1} = \frac{1}{2(1+\beta^2)}\left[u_{i+1,j}^{k} + u_{i-1,j}^{k+1} + \beta^2(u_{i,j+1}^{k} + u_{i,j-1}^{k+1})\right] \qquad \textbf{(7-15)}$$

The solution is to find a set of linear equations, expressed in matrix terms as

$A\vec{x} = \vec{b}$ The Gauss-Seidel iteration is

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j\geq i} a_{ij}x_j^{(k)}\right), i = 1,2,....,n \qquad \textbf{(7-16)}$$

Note that the computation of $x_i^{(k+1)}$ uses only those elements of $x^{(k+1)}$ that have already been computed and only those elements of $x^{(k)}$ that have yet to be advanced to iteration k + 1. This means that no additional storage is required, and the computation can be done in place ($x^{(k+1)}$ replaces $x^{(k)}$). While this might seem like a rather minor concern, for large systems it is unlikely that every iteration can be stored. Thus, unlike the Jacobi method, one does not have to do any vector copying should one want to use only one storage vector. The iteration is generally continued until the changes made by an iteration are below some tolerance.

Successive over-relaxation (SOR) is a numerical method used to speed up convergence of the Gauss–Seidel method for solving a linear system of equations. A similar method can be used for any slowly converging iterative process.

A similar technique can be used for any iterative method. Values of $\omega > 1$ are used to speedup convergence of a slow-converging process, while values of $\omega < 1$ are often used to help establish convergence of a diverging iterative process.

There are various methods that adaptively set the relaxation parameter ω based on the observed behavior of the converging process. Usually they help to reach a super-linear convergence for some problems but fail for the others.

No general guideline exists for computing the optimum value of the relaxation value ω [36].

We seek the solution to a set of linear equations, expressed in matrix terms as

$A\vec{x} = \vec{b}$ The successive over-relaxation (SOR) iteration is defined by the recurrence relation

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j\geq i} a_{ij}x_j^{(k)}\right), i = 1,2,....,n \qquad \textbf{(7-17)}$$

This iteration reduces to the Gauss–Seidel iteration for $\omega = 1$. As with the Gauss–Seidel method, the computation may be done in place, and the iteration is continued until the changes made by iteration are below some tolerance.

7.3 Red-Black and Multi-coloring Scheme

In Gauss-Seidel method calculated values are used immediately, this is not a problem in sequential algorithms. Thus, in parallel algorithms processors need calculated values of neighbors. Red-black decomposition is used if calculated values are immediately used in the neighbor points calculations. Red-black decomposition separates points with two colors red, black. This decomposition is like a checkerboard. Red-black is the simplest version of multi-coloring scheme.

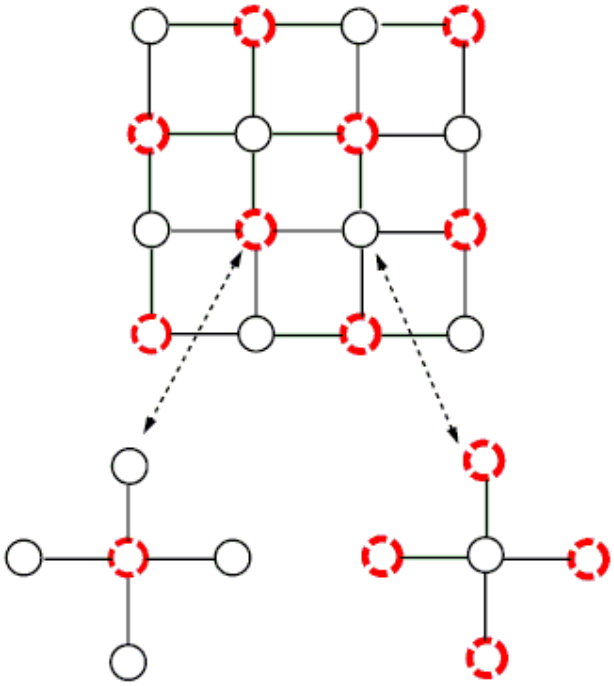

Figure 7.5 : Red and Black Stencils

The key idea is to group the grid points into two groups, identified as black and red nodes, and observe that for Cartesian differencing the black nodes are surrounded by red nodes only, and the red nodes are surrounded by black nodes only. This is shown schematically in Figure 7.5 . The figure is 2D Cartesian topology and has five stencils if nine stencils needed three or four colors can be used.

The implementation of the Gauss-Seidel method by means of the red-black ordering of the grid points is limited to rather simple partial differential equations, such as Poisson's equation, and rather simple discretization.

Consider the equation

$$u_{xx} + u_{yy} + au_{xy} = 0 \qquad \textbf{(7-18)}$$

Again in the unit square, where a is a constant. This is just Laplace's equation with an additional term. The standard finite difference approximation is

$$u_{xy} = \frac{1}{4h^2}\left[u_{i+1,j+1} - u_{i-1,j+1} - u_{i+1,j-1} + u_{i-1,j-1}\right] \qquad \textbf{(7-19)}$$

Which combined with the previous approximation (7-2) for Laplace's equation gives the system of equations

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} + \frac{a}{4}\left[u_{i+1,j+1} - u_{i-1,j+1} - u_{i+1,j-1} + u_{i-1,j-1}\right] = 0 \qquad \textbf{(7-20)}$$

Red-Black ordering for equation 7-20 is shown in Figure 7-6 below.

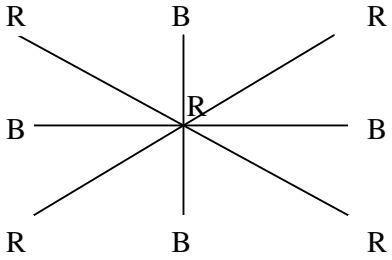

Figure 7.6 : Red-Black ordering for equation 7-20

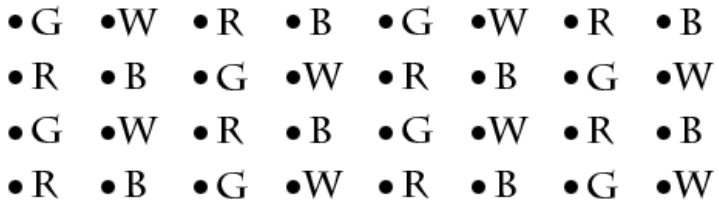

Figure 7.7 :A Four-Color Ordering for equation 7-21

The system may be written in the matrix form as in 7-21

$$\begin{bmatrix} D_1 & B_{12} & B_{13} & B_{14} \\ B_{21} & D_2 & B_{23} & B_{24} \\ B_{31} & B_{32} & D_3 & B_{34} \\ B_{41} & B_{42} & B_{43} & D_4 \end{bmatrix} \begin{bmatrix} u_R \\ u_B \\ u_G \\ u_W \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \qquad \textbf{(7-21)}$$

Where the diagonal D_i are diagonal. The Gauss-Seidel iteration can then be written as

$$\mathrm{D}_1 u_R^{n+1} = -B_{12}u_B^n - B_{13}u_G^n - B_{14}u_W^n + b_1 \qquad \textbf{(7-22)}$$

$$\mathrm{D}_2 u_B^{n+1} = -B_{21}u_R^n - B_{23}u_G^n - B_{24}u_W^n + b_2 \qquad \textbf{(7-23)}$$

$$\mathrm{D}_3 u_G^{n+1} = -B_{31} u_R^n - B_{32} u_B^n - B_{34} u_W^n + b_3 \qquad \textbf{(7-24)}$$

$$\mathrm{D}_4 u_W^{n+1} = -B_{41} u_R^n - B_{42} u_B^n - B_{43} u_G^n + b_4 \qquad \textbf{(7-25)}$$

Similarly, for the other two colors. Since the D_i are diagonal, the solution off the triangular system to carry out a Gauss-Seidel iteration has again reduced to matrix multiplication.

The four colors ordering of Figure 7-7 was based on the coupling of grid points illustrated in Figure 7-6, and such a pattern is called a stencil. A stencil shows the connection of a grid point to its neighbors and depends on both the differential equation and the discretization. The determination of the number of colors needed is

simplified if the stencil is the same at all points of the grid. Then the criterion for a successful coloring is that when the stencil is put at each point of the grid, the center point has a color different from that of all other points to which it is connected. It is this "local uncoupling" of the unknowns that allows a matrix representation of the problem that in general has the form

$$\begin{bmatrix} D_1 & B_{12} & \cdots & B_{1c} \\ B_{21} & D_2 & & \vdots \\ \vdots & & \ddots & B_{c-1,c} \\ B_{c1} & \cdots & B_{c,c-1} & D_c \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_c \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_c \end{bmatrix} \qquad \textbf{(7-26)}$$

The Gauss-Seidel iteration can be carried out, analogously to Equation 7-27

$$u_i^{n+1} = D_i^{-1} \left[b_i - \sum_{j<i} B_{ij} u_j^{n+1} - \sum_{j>i} B_{ij} u_j^n \right], \; i = 1, \ldots, c \qquad \textbf{(7-27)}$$

The solution of the triangular system is reduced to matrix-vector multiplications.

7.4 Pseudo Code for Parallel PDE

In this work, a nine-stencil multicolor Gauss-Seidel method is used to solve PDE. Minimum three colors are needed for calculating nine-stencil equation. In each iteration a colors calculation is made and calculated color is exchanged between neighbors. Here is the pseudo code of the algorithm:

A,B,C are the three colors of multicolor algorithm.

```
while do until convergence
   Calculation  of A points
   for i=2 step until n-2 do
      for j=2 step until n-2 do
         if (point is A)
            temp=A(i,j)
            A(i,j) = 0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2)+
                     A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1) );
         End if
      End j loop
   End I loop
   End of A points calculation
   Exchange ghost points
   Calculate B points
   Exchange ghost points
   Calculate C points
   Exchange ghost points
   Check if convergence is reached
End while

Exchange ghost points
   Send two upper row blocks to the upper neighbor
   Receive two upper row blocks from the upper neighbor
   Send two lower row blocks to the lower neighbor
   Receive two lower row blocks from the lower neighbor
End of exchange ghost points
```

Neighbor processors exchange two neighboring rows with each other. Since decomposition is row wise, no derived data type is used to conserve sequential access to data. In the ghost point exchange, two row blocks are exchanged with both upper and lower neighbors. This means if a processor has two neighbors it has to send four row blocks and receive four blocks.

7.5 Decomposition an Topolgy of PDE Matrix

Dividing data into parts is called decomposition. In the algorithm, 1D row wise decomposition is used. A 1D Cartesian non-periodic topology is created for defining neighbors. MPI methods are used to create the topology. Topology usage makes communication routines simpler for the developer. If topology is created then neighbors are known, no need to deal with processor ranks to find who is neighbor.

8. IMPLEMENTATION AND RESULTS

A parallel PDE solver algorithm using Gauss-Seidel method is used in this work. This algorithm is written by Gülnur Demir, who is a graduate student in ITU Computational Science and Engineering programme. This algorithm was developed for Parallel Programming lecture project assignment. She implemented Gauss-Seidel with three colors and nine stencils. In the algorithm, PDE equations variable matrix is meshed to a two dimensional matrix. If PDE has 100 variables then this mesh matrix size will be 10x10. This mesh matrix is solved iteratively. Algorithm is parallelized using MPI. After analyzing this algorithm with TAU, bottlenecks have been defined. There were two major bottlenecks one is computation of points other one is communication for sharing ghost points between processors. First computational bottleneck is analyzed.

Below is the v1 algorithm of colorA colorB and colorC computation. These computations are done in each iteration.

```
//Color C calculation
for(i=2; i<rows_local-2; i++)
   for(j=2; j<cols_local-2; j++){
      if((i+j-global_start) % 3 == colorC){
         temp = A(i,j);
         A(i,j) = 0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2)+A(i-1,j)+
                  A(i+1,j)+A(i,j-1)+A(i,j+1) );
         if( fabs(temp - A(i,j)) > tol && iter%10 == 0)
                  done = FALSE;
      }
}
UpdateGhosts(A, rows_local, cols_local, rowType, neigh, cartcomm);

//Color B calculation
for(i=2; i<rows_local-2; i++)
   for(j=2; j<cols_local-2; j++){
      if((i+j-global_start) % 3 == colorB){
         temp = A(i,j);
         A(i,j) = 0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2)+A(i-1,j)+
                  A(i+1,j)+A(i,j-1)+A(i,j+1) );
         if( fabs(temp - A(i,j)) > tol && iter%10 == 0)
                  done = FALSE;
      }
}
UpdateGhosts(A, rows_local, cols_local, rowType, neigh, cartcomm);
```

```
//Color A calculation
for(i=2; i<rows_local-2; i++)
   for(j=2; j<cols_local-2; j++){
      if((i+j-global_start) % 3 == colorA){
         temp = A(i,j);
         A(i,j) = 0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2)+A(i-1,j)+
                  A(i+1,j)+A(i,j-1)+A(i,j+1) );
         if( fabs(temp - A(i,j)) > tol && iter%10 == 0)
                  done = FALSE;
      }
}
UpdateGhosts(A, rows_local, cols_local, rowType, neigh, cartcomm);
```

Two nested loops have an if condition inside. This has two performance effects. One if branch prediction miss predicts CPU wastes its pipelined instruction. Another thing is, compiler cannot be aware of data independency in calculation of point A(i,j). If inner loops j is incremented by three not by one compiler can detect data independency. Data independency is important for massive parallelism. Multicoloring is used for data independency between processors, but with the first implementation, data is dependent inside one processor.

Below is the v2 algorithm, which eliminates if branch inside nested loops. This gains performance by eliminating if branch and makes this loop data independent.

```
//colorC
start=(colorC+global_start-2)%3;
if (start<2)
   start+=3;
for(i=2; i<rows_local-2; i++) {
   for (j=start;j<cols_local-2;j=j+3) {
      temp = A(i,j);
      A(i,j) = 0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2) + A(i-1,j)+
               A(i+1,j)+A(i,j-1)+A(i,j+1) );
      if(iter%10 == 0 && fabs(temp - A(i,j)) > tol  )
            done = FALSE;
   }
   if (start > 2)
      start--;
   else start+=2;
}
UpdateGhosts(A, rows_local, cols_local, rowType, neigh, cartcomm);

//colorB
start=(colorB+global_start-2)%3;
if (start<2)
   start+=3;
for(i=2; i<rows_local-2; i++) {
   for (j=start;j<cols_local-2;j=j+3) {
      temp = A(i,j);
      A(i,j) = 0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2) + A(i-1,j)+
               A(i+1,j)+A(i,j-1)+A(i,j+1) );
      if(iter%10 == 0 && fabs(temp - A(i,j)) > tol  )
            done = FALSE;
   }
   if (start > 2)
      start--;
   else start+=2;
}
UpdateGhosts(A, rows_local, cols_local, rowType, neigh, cartcomm);
```

```
//colorA
start=(colorA+global_start-2)%3;
if (start<2)
   start+=3;
for(i=2; i<rows_local-2; i++) {
   for (j=start;j<cols_local-2;j=j+3) {
      temp = A(i,j);
      A(i,j) = 0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2) + A(i-1,j)+
               A(i+1,j)+A(i,j-1)+A(i,j+1) );
      if(iter%10 == 0 && fabs(temp - A(i,j)) > tol  )
         done = FALSE;
   }
   if (start > 2)
      start--;
   else start+=2;
}
```

In this implementation, inner loops start value is important it must be the color we are computing. Thus, start value is computed and modified for next start. Another optimization was in the if condition. If a condition has more than one value anded, the first false invalidates this if condition. In v1 code, if conditions first condition was a hard computation compared with mod operation. So mod operation changed to be the first condition. If mod does not satisfies if condition the condition is not calculated. By the way mod operations calculation time can be reduced by using twos power in unsigned numbers. However, it was not used in this works implementations.

Another hot spot was communication. MPI supports different communication methods. In point-to-point communication, MPI Send-recv has high performance at every size of messages [35]. One chance to reduce communication can be using MPI persistent connection communication. In this algorithm, two neighbor processors communicate with each other, so persistent connection can be used. Persistent connection reduces connection overhead of communication, and stays connected unless MPI finalized or persistent connection freed. In persistent connection, the communication channel is initialized at the beginning of the algorithm and at each iteration MPI_Startall() method is used for making transfer. Before ending program persistent connection is terminated using MPI_Request_free() method. Algorithm version is named v3 and algorithm changed as follows.

Algorithm v1 communication UpdateGhosts method:

```
MPI_Isend( &A(rows_local-4,0), 1, rowType, neigh[DOWN], 99, cartcomm, &rq1 );
// send down
MPI_Recv( &A(0,0), 1, rowType, neigh[UP], 99, cartcomm, &st1 );
// recv from up
MPI_Wait( &rq1, &st1 );
MPI_Isend( &A(2,0), 1, rowType, neigh[UP], 999, cartcomm, &rq2 );    // send up
MPI_Recv( &A(rows_local-2,0), 1, rowType, neigh[DOWN], 999, cartcomm, &st2 );
// recv from down
MPI_Wait( &rq2, &st2);
```

Algorithm v3 changes this with persistent communicaiton methods:

```
//Before beginning iterations
/* Setup persistent requests for both the send and receive */
MPI_Send_init(&A(rows_local-4,0),1,rowType,neigh[DOWN],99,cartcomm,&reqs[0]);
MPI_Recv_init(&A(0,0), 1, rowType, neigh[UP], 99, cartcomm, &reqs[1]);
MPI_Send_init(&A(2,0), 1, rowType, neigh[UP], 999, cartcomm, &reqs[2]);
MPI_Recv_init(&A(rows_local-2,0),1,rowType,neigh[DOWN],999,cartcomm,&reqs[3]);
……….
while(iter<1000000 && !alldone)
{
   Calculate color C
   MPI_Startall (count, reqs);
   MPI_Waitall (count, reqs, stats);
   Calculate color B
   MPI_Startall (count, reqs);
   MPI_Waitall (count, reqs, stats);
..
}
/* Free persistent requests */
MPI_Request_free (&reqs[0]);
MPI_Request_free (&reqs[1]);
MPI_Request_free (&reqs[2]);
MPI_Request_free (&reqs[3]);
```

On the other hand, Successive over relaxation (SOR) can be used instead of Gauss-Seidel for faster convergence. Algorithm has been modified for SOR method. Only calculation part has changed and w parameter is added.

```
for(i=2; i<rows_local-2; i++) {
   for (j=start;j<cols_local-2;j=j+3) {
      temp = A(i,j);
      A(i,j) = ((1-w)*temp)+(w*0.125*( A(i-2,j)+A(i+2,j)+A(i,j-2)+A(i,j+2) +
               A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1) ));
      if(iter%10 == 0 && fabs(temp - A(i,j)) > tol  )
         done = FALSE;
   }
   if (start > 2)
      start--;
   else start+=2;
}
```

Three w values has been tested w= 0.5, 1, 1.5. When w equals 1 this method is same with the Gauss-Seidel method.

8.1 Runs and Results

Runs are made in two different architectures. There are three different versions of algorithm v1 is unoptimized, v2 is computation sections optimized, v3 communication sections optimized. Below Figure 8.1 are the analyses of three different version codes using 800x800 matrix size with 2 CPUs on HP Superdome Infinity ccNUMA server with Intel Itanium processor. Measurements are made using TAU performance analysis system. Library level instrumentation and selective instrumentation for calculation sections are used. Performance analyses are made

both using profiling and tracing. ParaProf visualizes profiles. Jumpshot visualizes traces. Linux timers are used measurement.

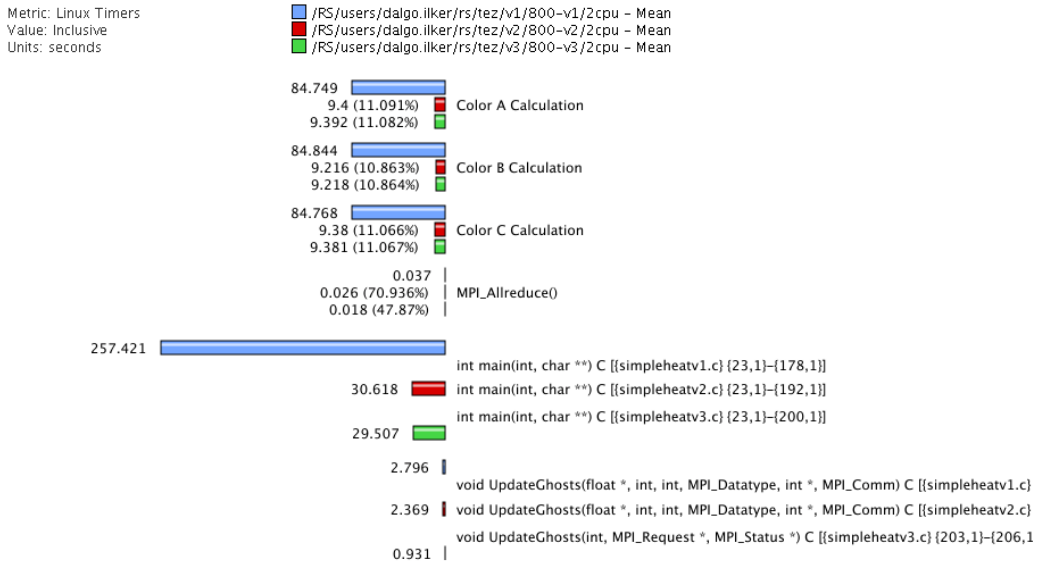

Figure 8.1 : Profile result of v1,v2,v3 algorithms; 800x800 matrix on 2 CPUs

As seen on graphics calculation optimization speed up is approximately 9x. Communication optimization (UpdateGhosts function) has approximately 2x speed-up in this matrix size. Overall performance speed-up is almost 9x.

Below Figure 8.2 is Cluster computers performance analysis. Cluster has Intel Xeon CPU and connected via InfiniBand. Figure 8.2 experiment parameters are 800x800-matrix size, two cpus, three version codes.

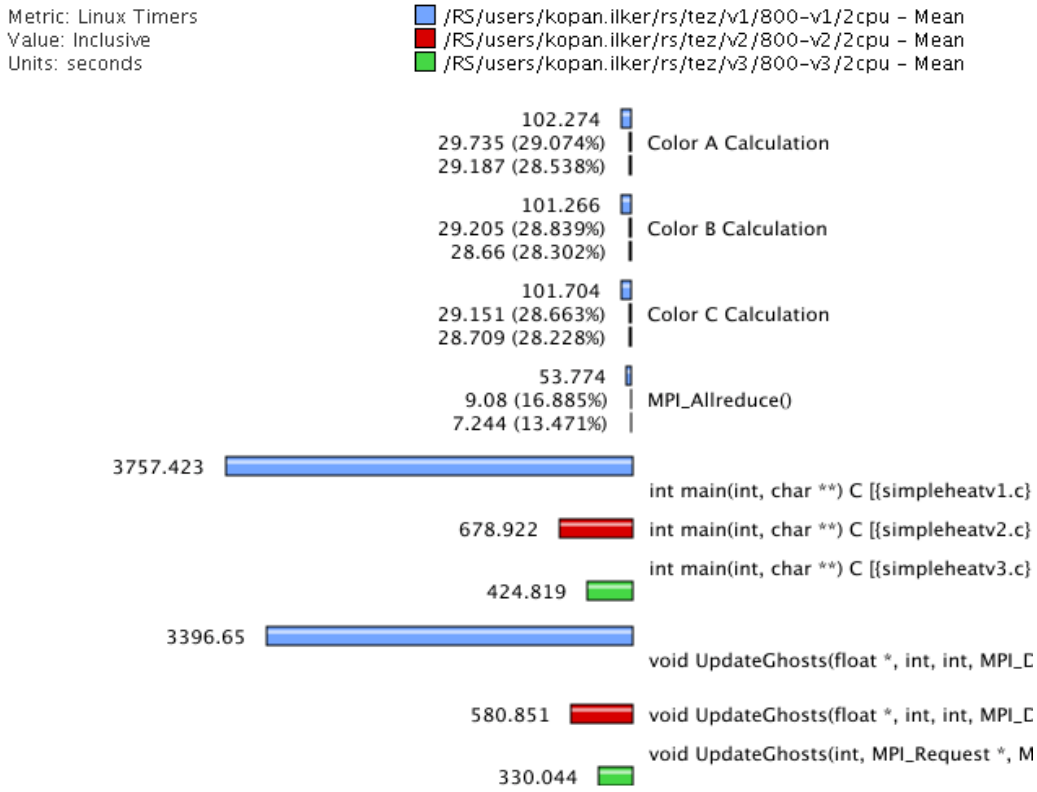

Figure 8.2 : Profile result of v1,v2,v3 algorithms; 800x800 matrix on 2 CPUs

Intel Xeon and Itanium have similar performance at computation section of version 1 algorithm. However, after making optimizations Intel Itanium performance is 3 times better than Xeon. Architecture based optimizations are done at compiler level. Compiler optimization flags O2 and O3 are used. With Intel Xeon processor O2 optimization and O3 optimization were the same. On the other hand, O3 optimization speed-ups program on Itanium processor. Below Figure 8.3 shows this detail. Only v1 and v2 algorithms are compared.

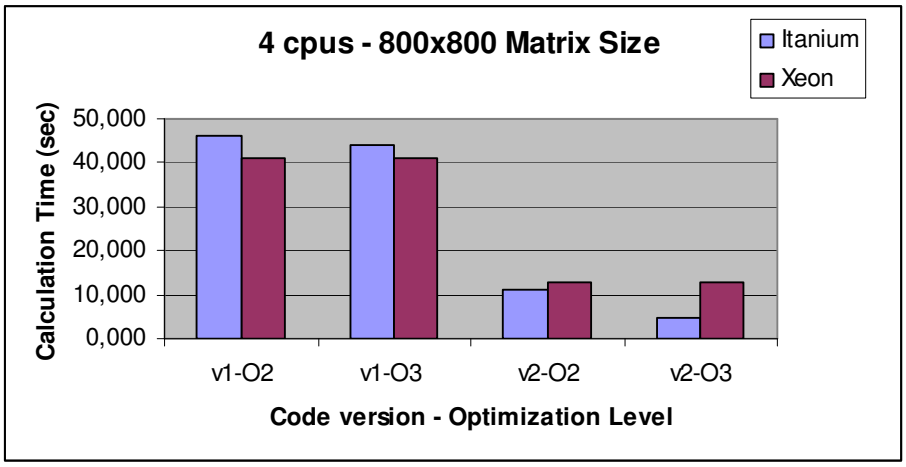

Figure 8.3 : Optimization comparison of two processors

HPC systems are used shared. Different nodes can run different application algortihms at the same time for maximum efficiency. In National Center for High Performance Computing of Turkey (NCHPC) clusters nodes are shared between projects, many independent programs are running on different nodes simultaneously. Since these nodes communicate with each other using underlying network infrastructure, different nodes communication effects others communication performance. This is called network contention [9]. While running these tests, NCHPC cluster with Xeon processors was having many other applications running on other nodes. Because of this, communication times vary depending on the other nodes communication. Figure 8.4 is a part of trace file output; network contention effect can be seen on this figure. In Figure 8.4 four processors are communicating with each other at each iteration but at some time network performance throughput decreases dramatically. Two different time's communications are seen on Figure 8.4. At one time, 3200byte communication took 12 microsecond and at next iteration, same sized communication took 4.493 millisecond. However, on SMP server tests are done one by one. SMP server was not running any other application. Thus, communication performance tests are done on SMP server. Figure 8.5 is a part of trace output, which, shows contention effect in HP shared memory system. In HP server communication performance is more consistent because it is only running test algorithm.

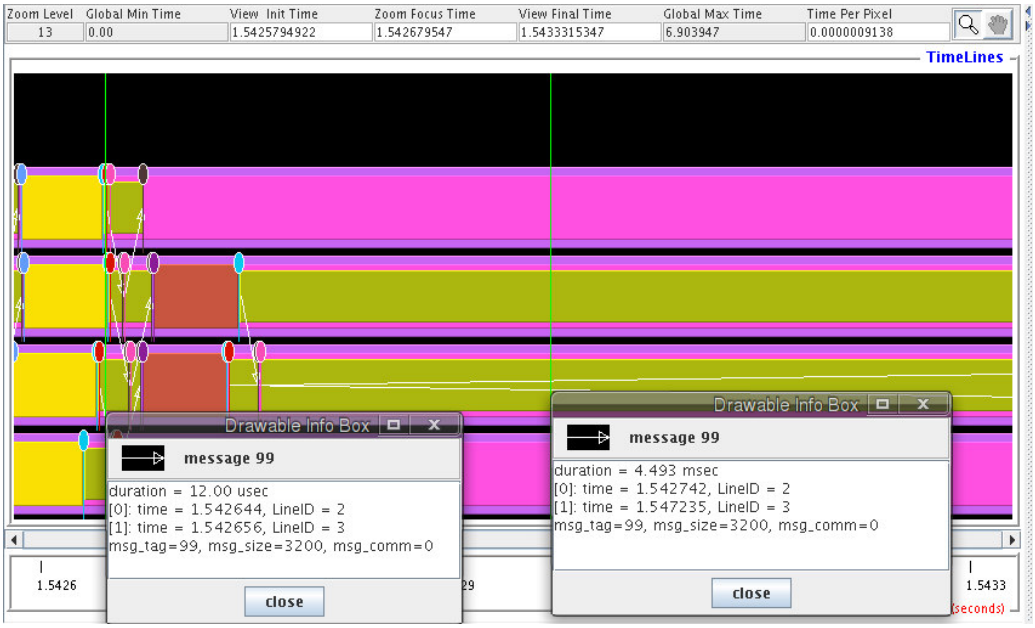

Figure 8.4 : Trace output showing cluster network performance variety

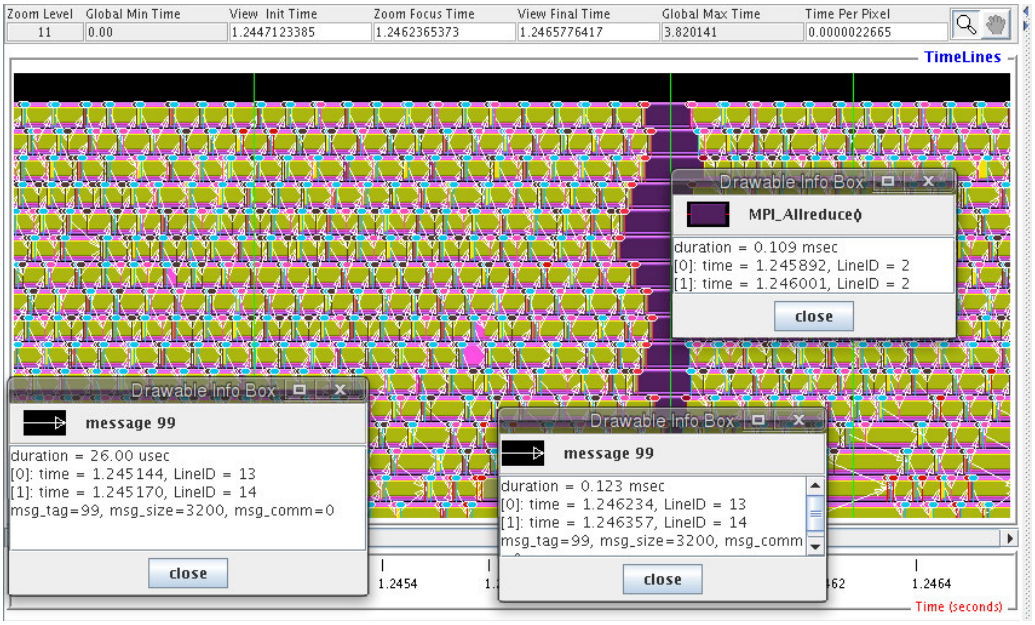

Figure 8.5 : Trace output showing SMP computer communication performance

When more processors are involved in computation, communication becomes a dominant factor. Time taken for calculation becomes smaller when processor number increases. Below Figure 8.6 is an example of this. In Figure 8.6 three versions of algorithms run on 64 cores using 1600x1600 matrix size. Increase of cores increases message count and makes communication a bottleneck.

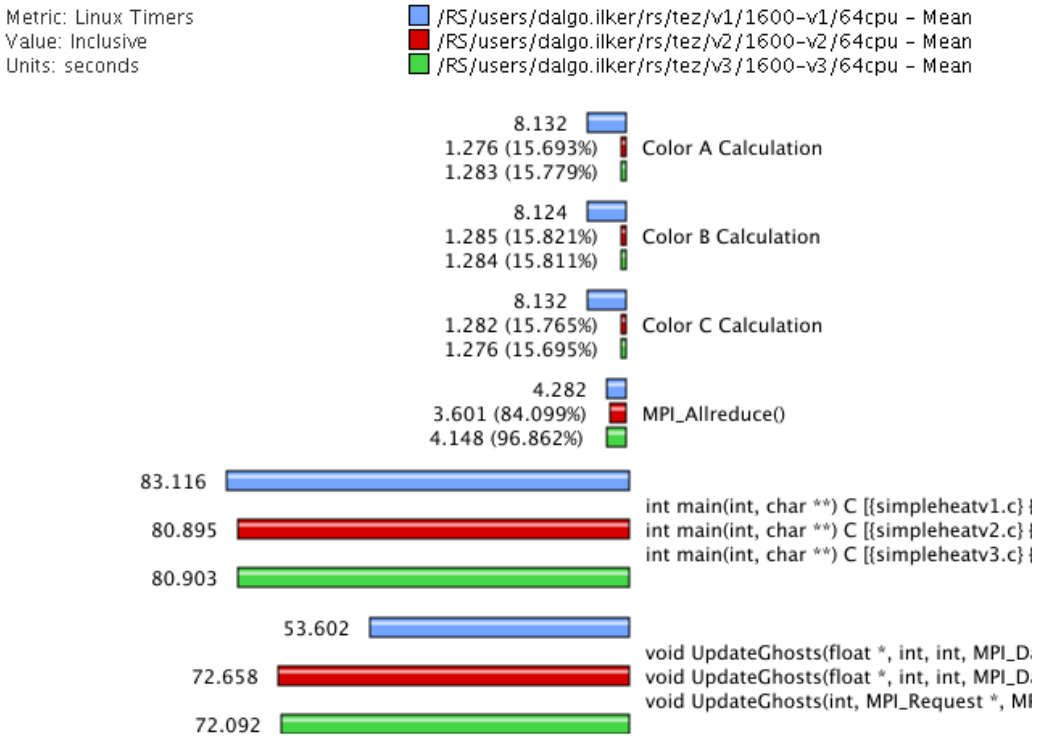

Figure 8.6 : Profile output of 64 processor communication bottleneck

In iterative methods, convergence rate determines iteration count. SOR is used to increase the convergence of Gauss-Seidel. SOR method is also implemented to see effects. SOR methods convergence can be showed by iteration counts, Figure 8.7 shows iteration counts for w=0.5 w=1.0 w=1.5 values.

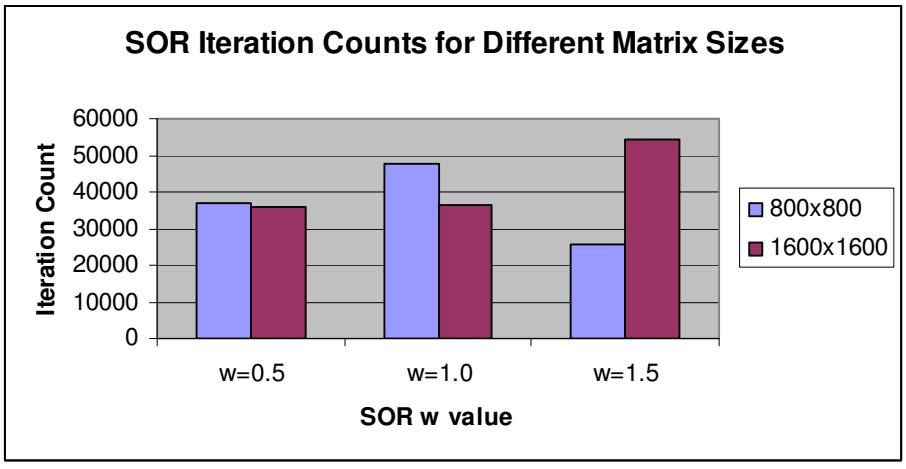

Figure 8.7 : SOR Iteration Counts for Different Relaxation Values

Figure 8.8 shows measured wall clock times for different relaxation (w) values. As seen, iteration count effects wall clock time.

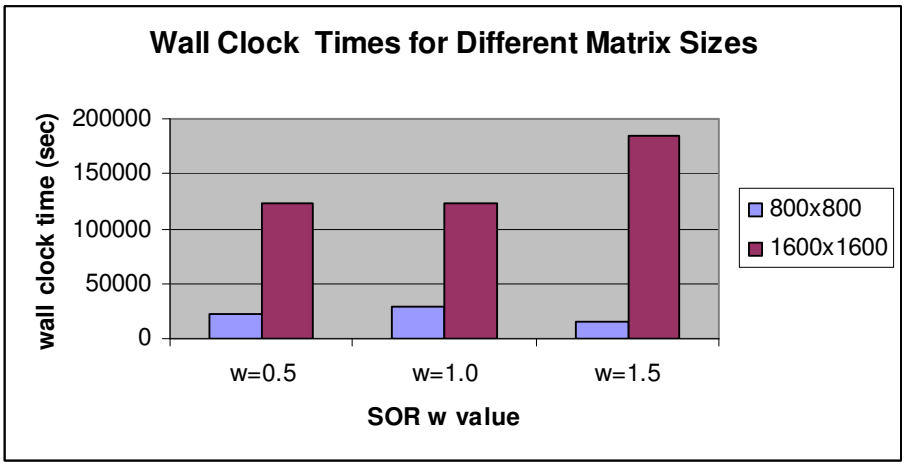

Figure 8.8 : Wall Clock Times for Different Relaxation Values

Network performance optimization shows becomes important when message size is small and iteration count increases. For example if tolerance value gets smaller iteration count increases. This means there will be many communications with small message sizes. Algorithm v1 uses MPI_Isend() and MPI_Recv(), v2 uses MPI_Sendrecv(), v3 uses MPI_Send_init() and MPI_Recv_init() persistent connection communication methods. Figure 8.9 shows relation between error tolerance value and communication time of different algorithms. As seen when error rate gets lower v3 algorithms performance gain increases. Matrix size is 400x400 core count is four in Figure 8.9.

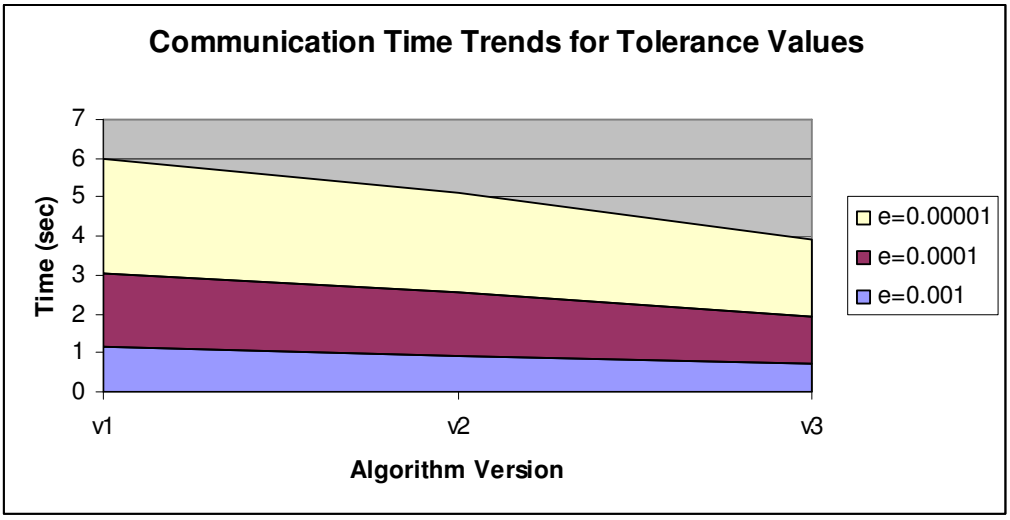

Figure 8.9 : Communication Time for Different Error Tolerance Values

Scalability of algorithms for different matrix sizes is shown in the figures below. As seen on Figures 8.10 – 8.13 at some point (related to the data size) wall clock time increases when more processors are used. The reason is, communication becomes a bottleneck when processors do less computation due to the increasing size of processors. In addition, it seen that increasing data size number of cores need for minimum wall clock time slightly shifts to the right.

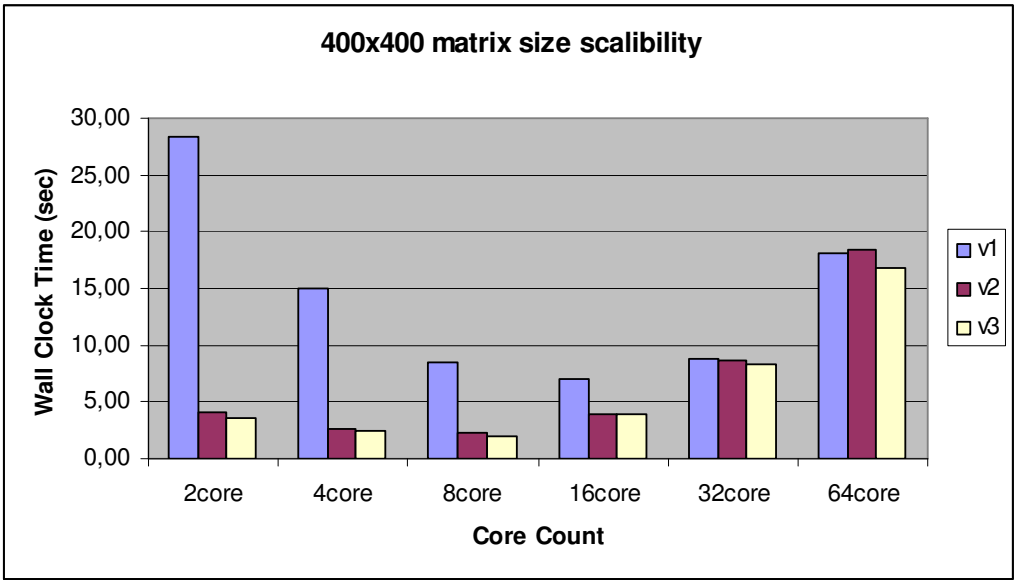

Figure 8.10 : Scalability for 400x400 matrix size

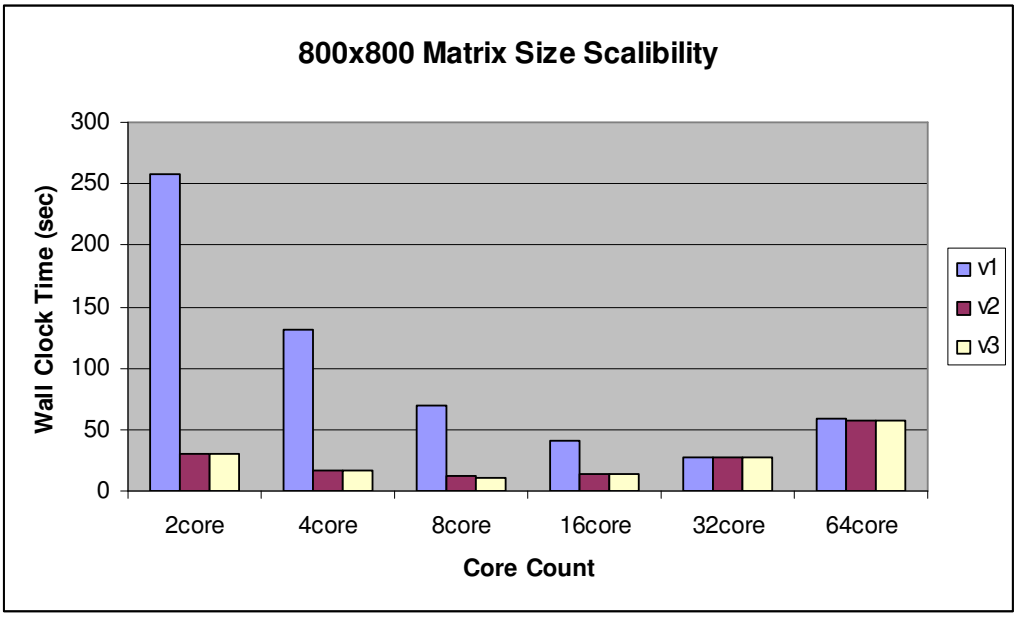

Figure 8.11 : Scalability for 800x800 matrix size

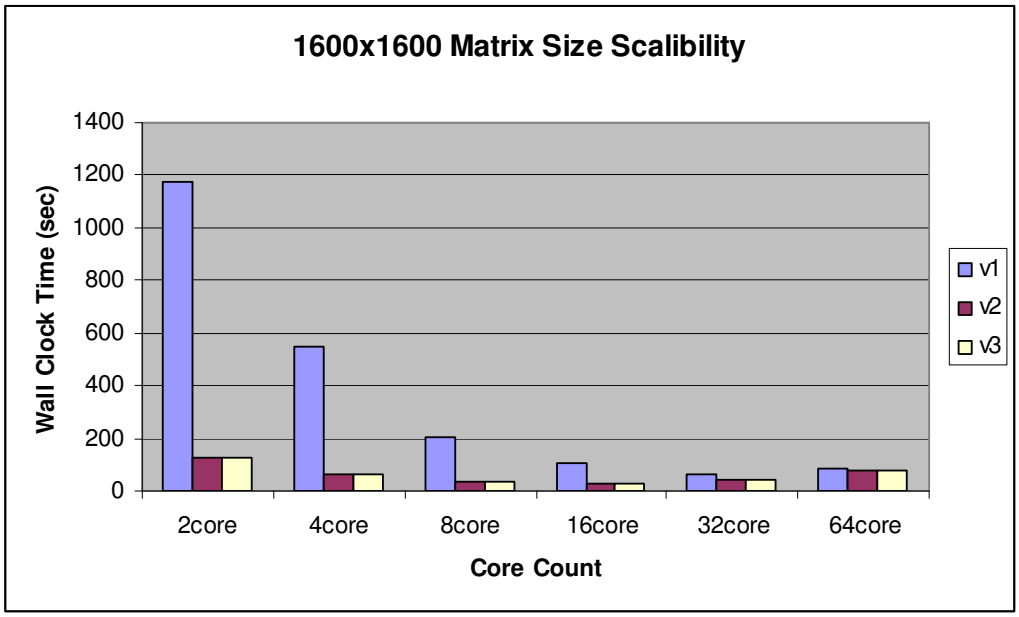

Figure 8.12 : Scalability for 1600x1600 matrix size

In Figure 8.13, v1 algorithm does not used due its insufficient performance, this is why wall clock time seems decreased but it increases.

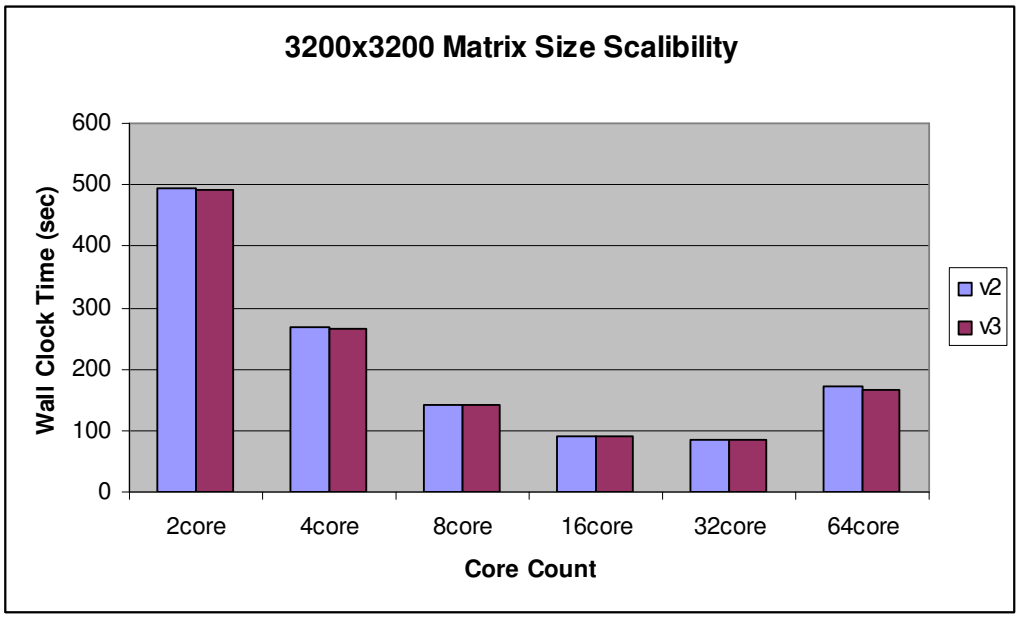

Figure 8.13 : Scalability for 3200x3200 matrix size

As seen on Figures 8.10 to Figure 8.13 processor scalability is higher at big matrix sizes. When matrix size is not big enough communication becomes dominant. Thus, communication contention effects when communication is made at the same time between all processors.

The relation between communication-calculation time and processor count can be seen in the in the Figure 8.14 and 8.15. If communication and calculation times are balanced, optimum wall-clock time is gained. Figure 8.14 shows balance effect of a 400x400 sized matrix scaling on different core counts.

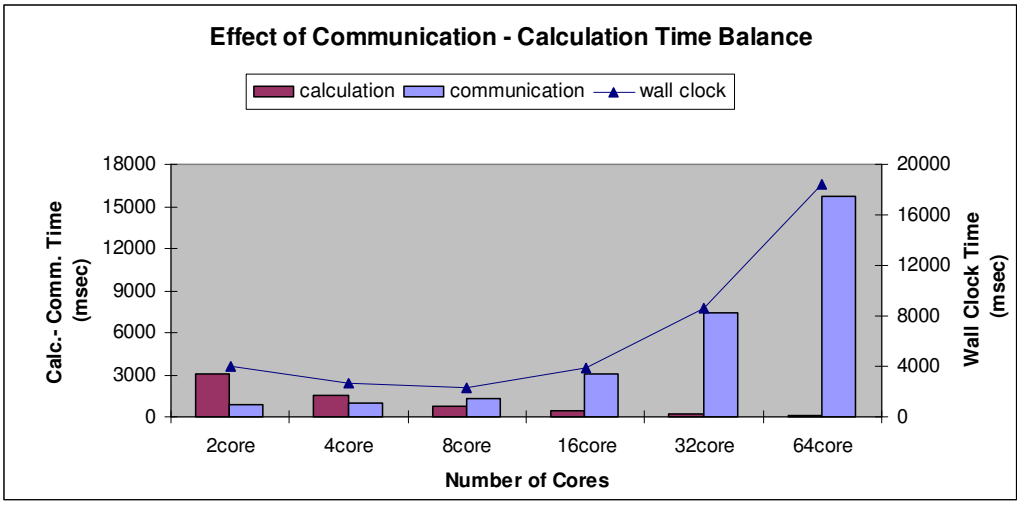

Figure 8.14 : 400x400 Matrix Balance Effect

If the matrix size increases calculation increases too. This shifts the communication-calculation balanced core count. In 400x400 matrix size core count that satisfies communication and calculation time is between 4 and 8 cores. However, in 3200x3200 matrix size balance is satisfied between 16-32 cores. Figure 8.15 shows 3200x3200 matrix size balance shift.

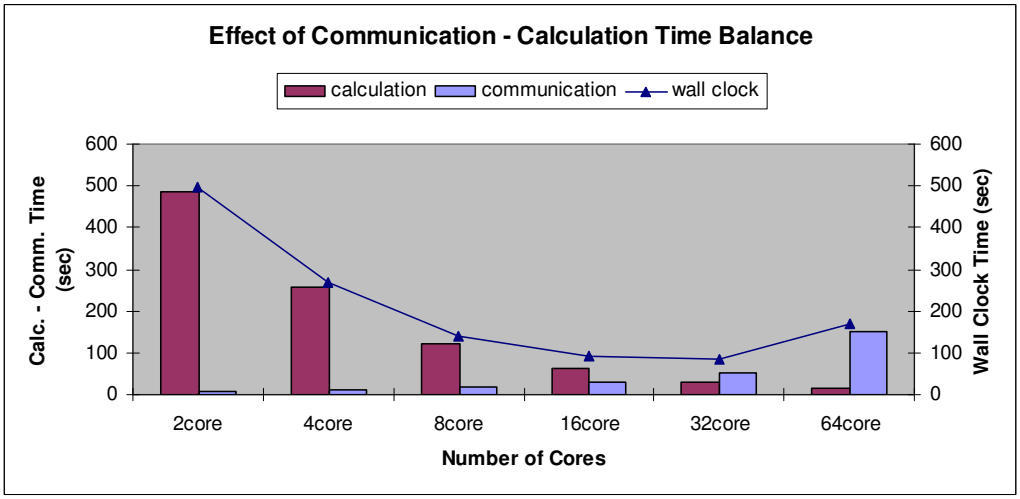

Figure 8.15 : 3200x3200 Matrix Balance Shift

9. CONCLUSION AND RECOMMENDATIONS

Processor types have different characteristics, for example, Xeon processor has higher clock rate than Itanium processors. Thus, Itanium is a RISC processor and Xeon is a CISC processor, CPU clock rate is not a pure performance determining parameter. Processors internal hardware like cache size and floating-point registers is other factors of processors computing power. For gaining maximum performance from a processor, algorithm must be efficient and right compiler parameters must be used. It is proven with experiments that, unoptimized algorithm runs faster on Xeon compared to Itanium processor but when right compiler parameters are used on efficient algorithm Itanium processor shows much better performance. Itanium processors performance is high on floating point intensive applications.

In addition, SOR algorithm issued on two architectures, SOR algorithm only effects convergence, if convergence rate decreases iteration count algorithm completes faster. However, defining optimal relaxation parameter is another work.

Communication optimization is done by using MPI persistent connection. Persistent connection removes connection initiation overhead at each iteration. Performance gain of persistent connection is maximum at small size messages. Moreover, when iteration count increases with small sized messages (like lowering error tolerance) time gained using persistent connection increases. MPI persistent communication methods can be preferred if same nodes are communicating at each iteration.

Using more processors for fixed data sizes does not always speed-up program. If computation takes less then communication time, then increasing processor count will raise overall time due to the increased communication count. Since communication medium is shared, more communication means slower communication.

In these entire measurements, TAU framework is used. Instrumentation is done using automatic library level instrumentation and for calculation sections, selective instrumentation is used. In profiling TAU tracks inclusive and exclusive times and

function/sections call counts. Function call count values are used for determining iteration counts. In addition, trace output is enabled for tracing. Since profiling only has statistical information, profile files are always small but trace files linearly increase as iteration count increases. That is why tracing must be used for test purposes on small data sizes, otherwise overhead of tracing raises.

9.1 Application of The Work

This work shows differences of two architectures with experiments. These experiments can guide HPC software developers for making performance analysis and optimization.

Optimizations made in these experiments are algorithm specific. Each algorithm has different characteristics and bottlenecks. Since iterative methods have similar characteristics, these optimization techniques can be applied to different iterative algorithms.

For processor-based optimization, using compilers related flag is the easiest and most effective way. Software developers may not be aware of processor internals, but they can develop highly parallelizable algorithms. In this works experiments, key point in writing parallel algorithm was to satisfy data in-dependency inside loops and avoiding unpredictable branches. Compilers know processor architecture and they can benefit from it if algorithm has fewer branches inside loops. Nonetheless, HPC algorithms are compiled from source code for the specific system. We can benefit from this by compiling program optimized for that specific architecture. Compilers -O3 level optimization and -fast parameters does this. If source code is compiled using these parameters compiler will do processor specific optimizations, which may limit the binary program to run only that specific processor.

Using more CPUs does not directly speed up program. It is seen that at low data sizes using more processors generates a communication bottleneck. Communication is related to processor count if more processors are involved more communication is needed. Roughly if calculation time equals communication time it is the best point of processor scalability.

In addition, to reduce communication contention, communication and calculation can be overlapped. Alternatively, while one group of processors is making calculation, others can communicate with each other. However, this cannot be achieved in PDE algorithm do the dependency of calculated values.

In addition, persistent connection becomes important when error tolerance is low on small sized matrix. When error tolerance is low, iteration count increases and persistent connection shows significant speed up on small messages.

9.2 Future Work

In this work, measurements are made using Linux timers. Due to the incompatibility between hardware counter patches and Lustre file system, hardware counters cannot be used on real system yet. Hardware counters can be used to see L1,L2 ad L3 cache hit rates of the algorithm. Also hardware counters show correctly predicted and unpredicted branch counts. One more important counter is TLB (Translate Lookaside Buffer) hit count. If these are known algorithms efficiency can be compared in this respect.

As seen making blocking communication decreases performance when many processors are used. Algorithm improvements can be researched to overlap communication and computation for based iterative numerical methods. Overlapping would improve scalability.

REFERENCES

[1]**Jeff Parkhurst, John Darringer, Bill Grundmann**, 2006: From Single Core to Multi-Core: Preparing for a new exponential

[2]**Lynch, Nancy,** 1997: Distributed Algorithms (1st ed.). San Francisco, CA: Morgan Kaufman Publishers.

[3]**Sameer S. Shende and Allen D. Malony,** 2006: The Tau Parallel Performance System. International Journal of High Performance Computing Applications

[4]**Jack Dongarra, Allen D. Malony, Shirley Moore, Philip Mucci, and Sameer Shende,** 2003: Performance Instrumentation and Measurement for Terascale Systems. *International Conference on Computational Science (ICCS 2003)*

[5]**Mahinthakumar G., Saied F.,** 2002: A Hybrid MPI-OpenMP Implementation of an Implicit Finite-Element Code on Parallel Architectures

[6]**Smith L.A,**.2002: Mixed Mode MPI / OpenMP Programming

[7]**Ulrich Rfide,** 1997: Iterative Algorithms on High Performance Architectures

[8]**Jiuxing Liu, Weihang Jiang, and others**, 2004: Design and Implementation of MPICH2 over InfiniBand with RDMA Support

[9]**Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar,** 2003: Introduction to Parallel Computing, Second Edition.

[10]**David E. Culler, Jaswinder Pal Singh, Anoop Gupta,** 1999: Parallel Computer Architecture - A Hardware/Software Approach. Morgan Kaufmann Publishers.

[11]**B. Ramakrishna Rau1, Joseph A. Fisher ,** 1993: Instruction-level parallel processing: History, overview, and perspective

[12]**Patt Yale,** 2004: The Microprocessor Ten Years From Now: What Are The Challenges, How Do We Meet Them?

[13]**Sayantan Sur, Matthew J. Koop, Dhabaleswar K. Panda,** 2006: High-performance and scalable MPI over InfiniBand with reduced memory usage

[14]**Gene Amdahl,** 1967: Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities

[15]**Flynn, M.,** 1972: Some Computer Organizations and Their Effectiveness, *IEEE Trans. Comput.*

[16]**Josh Stewart,** 2005: An Investigation of SIMD instruction sets

[17]**Ralph Duncan,** 1990: A Survey of Parallel Computer Architectures

[18]**Patterson and Hennessy,** 2008: Computer Organization and Design

[19]**Handy, Jim,** 1998: The Cache Memory Book. *Academic Press, Inc*.

[20]**HP,** 2009: < http://h20338.www2.hp.com/hpux11i/downloads/WP_ccnuma_final.pdf>

[21]**Ruud van der Pas,** 2002: Memory Hierarchy in Cache-Based Systems, *Sun BluePrints*.

[22]**Utrecht University, <** http://www.phys.uu.nl/~steen/web06/architecture.html>, accessed at 20.04.09.

[23]**Wikipedia,** <http://en.wikipedia.org/wiki/List_of_device_bandwidths>, accessed at 20.04.09.

[24]**InfiniBandTM,** 2007: InfiniBandTM Architecture Specification Volume 1, Release 1.2.1

[25]**DAT Collaborative,** 2007: kDAPL: Kernel Direct Access Programming Library, Version: 2.0

[26]**DAT Collaborative,** 2007: uDAPL:User Direct Access Programming Library, Version: 2.0

[27]**Jiuxing Liu, Sushmitha Kini and others**, 2003: Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics

[28]**HP,** 2007: HP Integrity Superdome Data sheet, 5982-9830EEW Rev. 5

[29]**International Supercomputer Conference (ISC)**, 2006: Tutorial, Performance Measurement and Analysis of Parallel Programs

[30]**Raj Jain, 1991:** Art of Computer Systems Performance Analysis Techniques For Experimental Design Measurements Simulation And Modeling, Wiley Computer Publishing, John Wiley & Sons, Inc.

[31]**Lindlan, K., Cuny, J., Malony, A. D., Shende, S., Mohr, B., Rivenburgh, R., and Rasmussen, C.** 2000: A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. *Proceedings of the SC'2000 Conference*

[32]**Mohr, B., Malony, A., Shende, S., and Wolf, F.** 2002: Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*.

[33]**Forum, M.P.I.** 1994: MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications* (Special Issue on MPI)

[34]**Martin Burtscher, Paruj Ratanaworabhan,** 2009: FPC: A High-Speed Compressor for Double-Precision Floating-Point Data

[35]**Lawrence Livermore National Laboratory,** <https://computing.llnl.gov/tutorials/mpi_performance/>, accessed at 20.04.09.

[36]**Klaus A. Hoffman, Steve T. Chiang,** Computational Fluid Dynamics (4. Edition)

[37]**Lawrence Livermore National Laboratory,** <https://computing.llnl.gov/tutorials/parallel_comp/>, accessed at 20.04.09

[38]**Ruud van der Pas,** 2002: Memory Hierarchy in Cache-Based Systems

CURRICULUM VITA

Candidate's full name:	İlker Kopan
Place and date of birth:	İstanbul, 15/07/1981
Permanent Address:	Denizköşkler Mah. Dr Sadık Ahmet Cad. Maritim Sitesi No 83 Daire: 13 Avcılar/ İstanbul
Universities and Colleges attended:	Yıldız Technical University (Computer Engineering)
Publications:	PERFORMANCE ANALYSIS OF PDE BASED PARALLEL ALGORITHMS ON DIFFERENT COMPUTER ARCHITECTURES at ICSCCW 2009