Master Thesis
Computer Science
Thesis no: MSC-2010-01
Month Year

Comparison of Shared memory based parallel programming models

Srikar Chowdary Ravela

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Srikar Chowdary Ravela
Address: If applicable
E-mail: rchowdarys@gmail.com

University advisor(s):
Håkan Grahn
School of Computing

External advisor(s):
Firstname Lastname
Company/Organisation name, do not forget AB
Address:
Phone: Only exchange number, international standard, e.g. use +

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 102 45

ABSTRACT

Parallel programming models are quite challenging and emerging topic in the parallel computing era. These models allow a developer to port a sequential application on to a platform with more number of processors so that the problem or application can be solved easily. Adapting the applications in this manner using the Parallel programming models is often influenced by the type of the application, the type of the platform and many others. There are several parallel programming models developed and two main variants of parallel programming models classified are shared and distributed memory based parallel programming models.

The recognition of the computing applications that entail immense computing requirements lead to the confrontation of the obstacle regarding the development of the efficient programming models that bridges the gap between the hardware ability to perform the computations and the software ability to support that performance for those applications [25][9]. And so a better programming model is needed that facilitates easy development and on the other hand porting high performance. To answer this challenge this thesis confines and compares four different shared memory based parallel programming models with respect to the development time of the application under a shared memory based parallel programming model to the performance enacted by that application in the same parallel programming model. The programming models are evaluated in this thesis by considering the data parallel applications and to verify their ability to support data parallelism with respect to the development time of those applications. The data parallel applications are borrowed from the Dense Matrix dwarfs and the dwarfs used are Matrix-Matrix multiplication, Jacobi Iteration and Laplace Heat Distribution. The experimental method consists of the selection of three data parallel bench marks and developed under the four shared memory based parallel programming models considered for the evaluation. Also the performance of those applications under each programming model is noted and at last the results are used to analytically compare the parallel programming models.

Results for the study show that by sacrificing the development time a better performance is achieved for the chosen data parallel applications developed in Pthreads. On the other hand sacrificing a little performance data parallel applications are extremely easy to develop in task based parallel programming models. The directive models are moderate from both the perspectives and are rated in between the tasking models and threading models.

Keywords: Parallel Programming models, Distributed memory, Shared memory, Dwarfs, Development time, Speedup, Data parallelism, Dense Matrix dwarfs, threading models, Tasking models, Directive models.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to express my sincere gratitude towards my thesis supervisor Håkan Grahn for his patient guidance, his invaluable comments, and support at every stage of my Master's Thesis for such a long time. All the work presented in this thesis was possible only because of his esteemed guidance at every stage during the course. I would like to express my great thanks to him not only for answering my questions, but also for guiding me with invaluable support. Without his help and encouragement this thesis would never have been completed.

Many thanks to my thesis examiner at BTH, Prof: Dr. Guohua Bai for his continuous guidance and for commenting the report.

My loving thanks to all of my family members. They provided continuous encouragement and support during my thesis. I would like to express my special thanks to my brother for encouraging me during my work and for many great memories and still many more to come.

Most of all, I would like to express my warmest possible thanks by dedicating this thesis to parents and my brother for supporting me in such an excellent and unselfish way in harder times.

Finally, I would like to express my special thanks to May-Louise Anderson and the other BTH staff making the study environment being very co-operative and for their continuous support.

INTRODUCTION

This chapter presents an introduction to parallel programming models. An ensemble of parallel programming models with their connections, existence and entanglements in using the parallel programming models is presented.

Parallel programming refers to the computational form where applications achieve high performance by parallelizing the operations and executing them simultaneously. Parallelization of the computations or operations can often be achieved in two ways.

- By replicating the hardware components (processor, memory, bus).
- By interleaving and organizing the single processor execution between multiple tasks.

1.1 Parallel programming

This thesis deals with the former concept mentioned above which is prominently known as parallel processing. This word can be used interchangeably with the term parallel programming. Parallel programming means performing multiple tasks in parallel using the duplication of the functional components. Parallel programming is used extensively for a wide range of applications ranging from scientific applications to the commercial applications. Example applications include transaction processing, computer games and graphics, weather simulation, heat transfer, ray tracing and many others.

1.2 Parallel Hardware

It is necessary to know about the parallel hardware before going deep into the study. The traditional uni-processor computer is said to follow Von-Neumann architecture which consists of a single memory connected to processor via data paths and works on the "stored memory concept". These kinds of architectures often represent a bottle neck for sequential processing and the performance associated with them is limited. So to relieve from these bottle necks one possible way is to use the redundancy /duplication of the hardware components. Various types of parallel platforms are designed to support the better parallel programming. The hardware used for parallel programming is known as multiprocessors. [27], [3] provides a very good introduction to the classification of multi-core platforms. They are in general classified into two types

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

Parallel data structures are the ones that benefit more by using the SIMD architectures. Usually these types of computations are known as structured computations. Often it is necessary to selectively turn of operations on certain data items. SIMD paradigms are known for the usage of an activity mask where a central control unit turns off selectively the operations on certain data items done by a processor array. Such conditional execution can be complex by nature, detrimental to the performance and can be used with care. In contrast to SIMD architectures, MIMD computers have the capability of executing independent instruction and data streams. A simple variant of this model is called the Single program multiple data (SPMD) model, relies on multiple instances of the same program executing on different data. SPMD model is widely used by many parallel platforms and requires minimal architectural support.

1.3 Parallel Programs

This idea of parallelizing the tasks was first encountered in the year 1958 by Gill [35]. Later in 1959 Holland pointed about a computer capable of running an arbitrary of sub programs simultaneously [35]. Conway presented the design of a parallel computer and its programming in 1963[35]. In 1980's a naive idea is that the performance of computers is increased by creating fast and efficient processors. But as the hardware developments progressed and this was challenged by the parallel processing by encompassing multiple cores in a single die. Thus leading to the development of the parallel programming models and paradigm shift has been encountered from the traditional uni-processor software to the revolutionary parallel programming software.

1.4 Why are parallel computers important?

Parallel computing allows solving problems that usually pose high requirements on the memory and processing power i.e. applications that can't be run using a single CPU. Also some problems are time critical; they need to be solved in a very limited amount of time. Using uni-processors for those applications can't achieve the desired performance and therefore parallel computers are used to solve that sort of problems. By using parallel computers we can run large problems, faster, and many more cases. The main intention in using the parallel systems is to support high execution speeds. The scope of parallelization of an application comes from the identification of multiple tasks of the same kind, which is a major source of speedup achieved by the parallel computers. Parallel computers are also used to overcome the limits of sequential processing like with sequential processing the available memory, performance is often a limitation to the processing for achieving high performance.

1.5 Problem Domain

The research discipline is confined to the software based shared memory parallel programming models. These models are designed for achieving considerable gain in performance by running the applications efficiently on the underlying platforms. Even though a comparison of parallel programming models is done by few researchers, this study is new of its kind and also no definite literature is available for this study. This area of research is quiet dominant and emerging because of the challenges posed in section (Chapter 2). This thesis considers the chance of answering those research challenges faced by the most of the industries in this field.

1.6 Problem and Research Gap

The true objective of the parallel computing is to solve the problems that require large resources and to achieve high performance of the application. But achieving the high performance is not the only challenge faced in the field of parallel computing. One of the main challenges faced by the parallel computing industry is to bridge the gaps between the parallel hardware and parallel software with respect to the increasing capabilities of parallel hardware [30]. Modern computer architectures (Multi-core and CMPs) tend to follow the computational power argument [3] and reaching the limits of the Moore's law. Software for parallel computers is slowly but steadily increasing [3] need to be bridged with the increasing capabilities of the parallel hardware. Software applications are vast from various disciplines and must be adapted to execute in a diverse set of parallel environments. One obvious solution to face this challenge is to arrive at a solution of a problem with minimal effort and time [30]. Thus an important metric that for evaluating various approaches to code development is the "**time to solution**" also called the "**development time**" metric. On the

other hand the time to solution has its impact on affecting the performance of the application. One obvious follow-up question drawn from the above statement is that; how the performance of the parallel application is achieved with respect to the development time. The answer for this depends on the support for development of an application under a parallel programming model while at the same time the performance offered by using the model specific primitives. For a parallel programming model to face the above challenges, it is a tradeoff between the development time of the application and the speedup of an application [30]. The main goal is to reduce the time to solution, by sacrificing either of the parameters, the speedup of an application or the development time of the application or both if necessary. For example the expending the development effort for tuning the performance of the application may leads to the decrease in orders of magnitude of the execution times or may lead to less improvement in the execution times, which has its effects on achieving greater speedups of the parallel application. Barriers need to be identified for development time and speedup for a parallel programming model, such that time to solution is minimized. These values will differ based on the type of the problem, the type of the parallel programming model used and also on the type of the language used for development. If the code will be executed many times, many of the costs of the increasing development time can be amortized across the multiple runs of the software and balancing against the cumulative reduced execution time conversely, if the code will be executed only once the benefit of increasing effort in tuning the code may not be as large [30].

1.7 Research Questions

This thesis is mainly aimed to compare the shared memory based parallel programming models. (Chapter 2) describes the goals and objectives in detail.

The following research questions are studied in the thesis.

- **RQ1.**What is the speedup of the application achieved under the chosen parallel programming model?
- **RQ2.** What is the development time of the application under each parallel programming model?
- **RQ3.** Which model is best when compared to the relative speedup and the total development time of the application?

The research methodology followed for this study in order to answer the research questions is a quantitative methodology. The methodology is presented in detail in Chapter 3.

1.8 Structure of the thesis

Chapter 1 explains the background of the study; chapter 2, explains the problem definitions and goals identified for this study; chapter 3, presents the methodology; chapter 4, presents the Benchmarks used for this study; Chapter 5 describes the Qualitative work; chapter 6, the empirical case study; Chapter 7, the results; Chapter 8 the Analysis and Discussions.

CHAPTER 2: BACKGROUND

2.1 Parallel hardware

Parallel hardware is equally important for a parallel application developer because there are several details that the developer has to consider while developing applications in a parallel programming model like for example the underlying processor architecture, the memory architecture, the cache organization and so-on. Parallel hardware incorporates details about the processors, memory hierarchy, and the type of inter-connection used. Parallel hardware is used to provide a framework on which the parallel software resides.

2.1.1 Processors

The hardware used for the parallel systems comes in a varying number of cores and complexities. The performance achieved depends on the underlying processor architecture. For high efficiency, the programming model must be abstracted from the underlying processors and is independent of the number of processors and the design complexity involved (like the interconnections used and the number of threads per processor core) [21] [20].

2.1.2 Memory hierarchy

Memory organization is an important issue that supports the levels of parallelism and takes a different form for parallel architectures on which the parallel programming model is relying on, than in uni-processor architecture. Memory consistency issues must be dealt with more care; otherwise they could introduce long delays, ordering and visibility issues. Having memory on chip improves the efficiency of the programming model Rather having the off chip memory [20], [21]. For this high speed and smaller memory components called caches are used. Programming models need to be designed for the efficient use of those caches. Otherwise it is the responsibility of the programmer to optimize the program for the cache efficiency. Often there are few parallel programming models like Cilk++ and Intel's TBB (threading building blocks) that are designed to optimize for the cache usage.

2.1.3 Interconnection network

Interconnects specify how the processor nodes in a parallel computers are connected. The information about various topologies are defined in [27], [3] few of the most widely used are ring, bus, hypercube and tree based topologies. The type of the interconnection used is affected in terms of the cost and scalability. Some topologies scale well in terms of area and power while increasing the cost whereas some other topologies comes at low cost but not well scalable. The choice of the interconnection used is often a trade-off between these two factors. Also the inefficient way of interconnecting the processor nodes in a multiprocessor leads to performance dropdown by introducing high latency in the communications even though the architecture is scalable. The best interconnection model must scale with the number of the processor nodes, with introducing low latency during communications (fully discussed in section 2.1.4). Often it is suggested that processor nodes should be interconnected in terms of small ring or bus topologies and interconnecting them using mesh[21].

2.1.4 Communication

The communication models used in the parallel architecture classifies the type of the multiprocessors [10]. Multiple processors that share the global address space are classified

under **shared-memory multiprocessors.** The multi-processors in these systems communicate with each other through global variables stored in a shared address space. Another complete variant architecture for the one mentioned above is known as **distributed-memory multiprocessors** where each processor has its own memory module and the data at any time instant is private to the processors. These types of systems are constructed by interconnecting each component with a high-speed communications network. These architectures rely on the send/receive primitives for communication between multiple processors communicate to each other over the network. Parallel programming models that rely on shared memory based multiprocessors are called shared memory based parallel programming models and parallel programming models that rely on distributed memory architectures are called distributed memory based parallel programming models (discussed in the section 2.3).

2.2 System Software

The system software includes the compilers and operating systems that support the parallel programming models. These help to span the gap between the applications and the hardware [21]. Each of these has the potentiality in affecting the performance of the programming models. This is because the parallel code generated by a computer often depends on the system software employed where the responsibility is ensured by the compilers and autotuners [21]. Again there is a tradeoff between autotuners and compilers about which software to use for parallelizing of the applications.

2.3 Shared memory vs. Distributed memory based parallel programming

Parallel programming models are not new and dates back to the cell processors [20]. There are several programming models that have been proposed for multi-core processors. They can be classified based on the communication model used. A good literature on Share memory versus Distributed memory parallel programming is found in [5].

2.3.1 Shared memory based parallel programming

These are the models that rely on the shared memory multiprocessors. Shared memory based parallel programming models communicate by sharing the data objects in the global address space. Shared memory models assume that all parallel activities can access all of memory. Consistency in the data need to be achieved when different processors communicate and share the same data item, this is done using the cache coherence protocols used by the parallel computer. All the operations on the data items are implicit to the programmer. For shared memory based parallel programming models communication between parallel activities is through shared mutable state that must be carefully managed to ensure correctness. Various synchronization primitives such as locks or transactional memory are used to enforce this management. The advantages with using shared memory based parallel programming models are presented below.

- Shared memory based parallel programming models facilitate easy development of the application than distributed memory based multiprocessors.
- Shared memory based parallel programming models avoids the multiplicity of the data items and the programmer doesn't need to be concerned about that which is the responsibility of the programming model.
- Shared memory based programming models offer better performance than the distributed memory based parallel programming models.

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

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

2.3.2 Distributed memory based parallel programming

These kind of parallel programming models are often known as message passing models, that allows communication between processors by using the send/receive communication routines**. Message passing models** avoids communications between processors based on shared/global data [20]. Typically used to program clusters, where each processor in the architecture gets its own instance of data and instructions. The advantages of distributed memory based programming models are shown below.

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

The disadvantages with distributed memory based parallel programming model.

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

2.4 Shared memory based parallel programming models

A diverse range of shared memory based parallel programming models are developed till date. They can be classified into mainly three types as described below.

2.4.1 Threading models

These models are based on the thread library that provides low level library routines for parallelizing the application. These models use mutual exclusion locks and conditional variables for establishing communications and synchronizations between threads. The advantages with threading models are shown below.

- More suitable for applications based on the multiplicity of data.
- The flexibility provided to the programmer is very high.
- Threading libraries are most widely used also it is easy to find tools related to the threading models.
- Performance can still be improved by using conditional waits and try locks.
- Easy to develop parallel routines for threading models.

The disadvantages associated with threading models are described below.

- Hard to write applications using threading models, because establishing a communication or synchronization incurs code overhead which is hard to manage and thereby leaving more scope of causing errors.
- The developer should be more careful in using global data otherwise this leads to data races, deadlocks and false sharing.
- Threading models stand at low level of abstraction, which isn't required for a better programming model.

2.4.2 Directive based models

These models use the high level compiler directives to parallelize the applications. These models are an extension to the thread based models. The directive based models takes care of the low level features like partitioning, worker management, synchronization and communication among the threads. The advantages with directive models are presented below.

- Directive based models emerged as a standard.
- It is easy to write parallel application.
- Less code overhead is incurred and it is easy manage the code developed using directives.
- This model stands at low level of abstraction.
- The programmer doesn't need to consider issues like data races false sharing, deadlocks.

The disadvantages associated with using directive based models are presented below.

- These models are less popularly used.
- The flexibility provided to the programmer is low.
- Also the support of the tools for development is low.

2.4.3 Tasking models

Tasking models are based on the concept of specifying tasks instead of threads as done by other models. This is because tasks are of short span and more light weight than threads. Tasks in general are 18X faster than threads in UNIX implementations and 100X faster than threads in windows based implementations [14]. One difference between tasks and threads is that tasks are always implemented at user mode [20]. Also tasks are parallel but not concurrent as a result they are non-preemptive and can be executed sequentially. The advantages with task based models are presented below.

- Tasks are proved to lessen the overhead associated with communications as is incurred in other models.
- Facilitates easy development of the parallel applications.
- Easy to find tools related to task based models. As a consequence it is easy to debug errors faced in task based models.

The disadvantages associated with using task based models are presented below.

- Task based models are not emerged as standard.
- Task based models are hard to understand.
- The flexibility associated with them is very low.

2.5 Programming Models Evaluated

This section describes the parallel programming models that are evaluated during this study. In this study, only the shared memory based parallel programming models were considered. There are several reasons for this, but few of the important reasons are discussed below.

- The shared memory platforms were dominant in the near future that poses a challenge of proposing new shared memory based parallel programming models that ease the development of parallel applications on those architectures.
- Programming for shared memory platforms is easier rather than the distributed parallel programming models which delve the mind of the developers making them responsible for establishing the parallel features.
- Shared memory based parallel programming models are more convenient to achieve good performance scales on the architecture employed with less effort than distributed memory parallel programming models.
- Various parallel programming models for shared memory platforms based on different features exist till date it is necessary to evaluate those models regarding the performance and development time (as mentioned in the challenges).

This thesis considers the shared memory parallel programming models based on the features of expressing the parallelism. Three of the various types of expressing the shared memory based parallelism are presented below. These models are selected for the study because of strictly confining the study only to the parallel programming libraries.

- Threading models.
- Directive based models.
- Task based models.

A detailed description of the above models is presented in the Chapter-3. The models selected based on these features are under threading model is the Pthreads model, under directive based model comes the OpenMP model and under task models the selected models are Intel's TBB and Cilk++.

2.5.1 Pthreads

Portable Operating System Interface (POSIX) threads [8] are an interface with a set of C language procedures and extensions used for creating and managing threads. It is first developed for uni-processor architectures to support for the multi-tasking/multi-threading on UNIX platforms and later emerged as an ANSI/IEEE standard POSIX 1003.1 C in 1995. They can be easily extended to multiprocessor platforms and are capable for realizing potential gain in performance of parallel programs. It is raw threading model that resides on a shared memory platform leaving most of the implementation details of the parallel programs and more flexible to developer. Pthreads has very low level of abstraction and hence developing the application in this model is hard from the developer perspective. With Pthreads the parallel application developer has more responsibilities like work load partitioning, worker management, communication and synchronization & task mapping. It defines a very wide variety of library routines categorized according to the responsibilities presented above.

2.5.2 OpenMP

Open Message passing or Open specification for multiprocessing [6] is an application program interface, which defines a set of program directives, Run time library routines and environment variables that are used to explicitly express direct multi-threaded, shared memory parallelism. It can be specified in C/C++/FORTRAN. It is an open parallel programming model defined by major computer vendor companies. OpenMP stand at high level of abstraction which eases the development of parallel applications from the perspective of the developer. OpenMP hides and implements by itself the details like work load partitioning, worker management, communication and synchronization. The developers only need to specify the directives in order to parallelize the application. It is the best programming model known till date. OpenMP is not widely used as Pthreads and is not emerged as a standard. It is hard to find the tools related to OpenMP. Moreover, the flexibility with this model is less compared to Pthreads. OpenMP is best suitable for task based applications.

2.5.3 TBB

Threading building blocks [14] is a parallel programming library developed by the Intel Corporation. It offers a very highly sophisticated set of parallel primitives to parallelize the application and to enhance the performance of the application on many cores. Intel's threading building blocks is a high level and supports the task based parallelism to parallelize the applications; it not only replaces the threading libraries, but also hides the details about the threading mechanisms for performance and scalability. Creating and managing threads are slower than that of the tasks in general tasks are 18X faster than threads on a Linux based system and almost 100X faster than a windows based system [14]. Intel TBB relies on offering the scalable data parallel programming which is much harder to achieve by making use of the performance as the number of processor cores increases. Threading building blocks is a library that supports the scalable parallel programming by using the C++ code and does not require any special languages or compilers. Threading building blocks provides the standard templates for common parallel iteration sequences to attain increased speed from multiprocessor cores without having to be experts in synchronization, load balancing, cache optimization. Programs using TBB will run on systems with a single processor, as well as on systems with multiple processor cores. Additionally it fully supports nested parallelism, so that you can develop larger parallel components form smaller components easily. To efficiently use the library the developer has to specify tasks instead of probing into the library by managing threads. The underlying library is responsible for mapping the tasks on to the threads in an efficient manner. As a result TBB enables you to specify parallelism more efficiently than using other models for scalable data parallel programming. In general as one goes deep into the TBB parallel programming model it is hard to understand the model and in some cases increases the development time this is because TBB stands at a high level of abstraction. The benefits of TBB are discussed below.

- Threading building blocks works at a higher level of abstraction than raw threads yet does not require exotic languages or compilers. You can use it with any compiler supporting ISO C++. This library differs from typical threading packages in these ways.
- Threading building blocks enables the developer to specify tasks instead of threads. This model has the effect of compelling the developer to think about the application in terms of modules.

- Threading building blocks targets threading for performance and is compatible with other threading packages, like it can be successively used with other packages like Pthreads and/or OpenMP.
- Threading building blocks emphasizes scalable data parallel programming and relies on the concept of generic programming.
- Threading building blocks is the first to implement the concepts like recursive splitting, algorithms and task stealing. Task stealing is proved to be popular and a dominant model to achieve greater speedups for task based applications.

2.5.4 Cilk++

Cilk++ [34] developed by Cilk arts was founded by a student of MIT, but was later over taken by Intel. It is a task based parallel library. It facilitates the fastest development of the parallel applications by just using the three Cilk++ components and a runtime system that are capable of extending to the realms of the parallel programming. Other than this a little change to the serial code is required to convert it into a parallel program. Cilk++ source retains the serial source semantics making the programmers easier to use the tools that exist in the serial domain making Cilk++ easier to learn and use. This model ensures no introduction of new serial bugs by unchanging the serial debug and regression systems and there by introducing no more newer ones. Serial correctness is assured. Cilk++ is a new technology that enables the creation of parallel programs by writing a new application or by parallelizing the serial application. It is based on the C++. It is well suited for problems based on the divide and conquer strategy. Recursive functions are often used that are well suitable for the Cilk++ language. The Cilk++ keywords identify function calls and loops that can run in parallel. The Intel Cilk++ runtime schedules these tasks to run efficiently on the available processors. Eases the development of the applications by imposing the developer to create tasks and also enables to check for the races in the program. The benefits in using Cilk++ are discussed below.

- Eases the development of parallel applications using a simple model and concentrates more on the programming methodology. Like for example with only three keywords to learn, C++ developers can migrate quickly into the Cilk++ parallel programming domain.
- Cilk++ is designed to optimize for the performance it is the first to implement the concept of Hyper object libraries. These libraries are used to resolve race conditions without the impacting the performance by increasing the overhead encountered by traditional locking solutions, and the scalability analyzer which is a new concept used to predict the performance of the applications when migrating to the systems with many cores.
- Cilk++ is ported with built in tools that eases the development of parallel applications. Like for example the leverage of existing serial tools which means that the serial semantics of Cilk++ allows the parallel program developer to debug in a familiar serial debugger.
- Proposes the use of race detector which is also a rather new concept that decreases the scope of the errors encountered during development. This race detector guarantees the operation of the parallel program in a race – free manner that eliminates the tension about the bugs associated with the application.
- Provides good scaling of the applications performance when migrating to the systems with respect to the increasing number of the cores.

2.6 Summary

This chapter presents a qualitative survey about the models that are evaluated under this thesis. The domain of the models studied in this thesis are confined to shared memory based parallel programming models and the models selected are Pthreads, OpenMP, Intel's TBB and Intel's Cilk++. Also the reasons for using the shared memory based parallel programming models and the benefits of using each of the models are highlighted in this chapter.

CHAPTER 3: PROBLEM DEFINITION / GOALS

3.1 Purpose of the study

The main intention of the research study is to compare the shared memory based parallel programming models. The comparison of the models is based on the development time of an application implemented under a specific programming model and the speedup achieved by an application implemented in a programming model. The purpose of the study is to answer the problems faced by the parallel programming community in the form of the research questions posed in the Chapter 1 and section 1.8.

3.2 Challenges

This study compares the models based on the speedup and development time. The use of these two independent metrics for the comparison is because of the challenges faced by the organizations due to the conflicting requirements speedup and development time while developing the applications. Here are few key challenges faced by the HPC industry regarding the development time and speedup.

- All the metrics for the HPC are based on the estimations of certain factors, there is very less evidence [15] and do not hold much in the HPC community.
- Development of parallel application software is constrained by the time and effort [3]. It is necessary to empirically study the trade-offs associated with the development time of parallel applications.
- Requirements often need sophisticated empirical models to be defined [24] for calculating the development metrics.
- The main intention of a better programming model is to support better performance on the other hand assist easy development of the reliable and scalable applications. [9] Presents this as three key challenges of the multi-core revolution in the form of a "*software triad*".
- The outcome of a HPC project is influenced mainly by two factors the type of the problem and the programming model [36]. [16] Describes this as a growth of computing requirements of the application and the potential of the programming model to support those requirements.
- Finally there are no efficient metrics to compare the development time and speedup of a programming model this is because very less work has been done on this concept.

3.3 Goals and Objectives

The main goal of the study is to compare the shared memory based parallel programming models from the perspective of two independent dimensions. Here are the objectives and goals of the study.

- One important goal of the study is to determine the impact of the individual parallel programming model on the development time and speedup.
- To characterize the programming models by comparing the models for the speedup and development time from the model proposed in this paper.
- To identify the model specific barriers/trade-offs for development time and speedup of the parallel programming models proposed in this study.

3.4 Problem statement and Research Questions

Software programmers tend to use the parallel programming models to develop applications that are effective and scalable. The problem is that which model to use for developing applications? A plethora of parallel programming models exist in the field of high performance computing. The models are categorized based on several factors like their working environment, the type of applications they are going to deal with, the level of parallelism they can implement and so on. But however the aim of each model is to improve the productivity, efficiency and correctness of the applications running over it [21]. All the three goals are very important and are also conflicting for a programming model. This is because of the dependency of the models on the above mentioned factors. It is hard to define the best model for all kinds of problems because of the conflicting goals the parallel programming model has to satisfy. The landscape of parallel computing research a view from Berkley has defined some ideal characteristics for a better parallel programming model. They are the parallel programming model must satisfy at least the following factors [21].

- Be independent of the number of nodes.
- Support rich set of data sizes.
- Support proven styles of parallelism.
- Goal of the problem.

"*The aim of the thesis is to experimentally compare and evaluate the software implementations of shared memory based parallel programming models in terms of the speedup and development time*[30][36][15]".

The following research questions are to be studied in the thesis in order to achieve the aims described above.

- **RQ1.** What is the speed up of the application supported by a shared memory based parallel programming model?

- **RQ2.** What is the development time for an application in each of the shared memory based parallel programming model?

- **RQ3.** Which programming model is best in terms of speedup and development time among the three models?

3.5 Expected outcomes

The expected outcome of the research study is to compare the shared memory based parallel programming models in terms of the speedup and development time and to identify the speedup and development time trade-offs for the parallel programming models evaluated in this study. This is done by comparing the results obtained from the empirical study in terms of development time of the application under a specific programming model versus the speedup of the application under the same programming model.

3.6 Summary

This chapter presents the purpose of the study, the challenges, the problem definition, research questions, goals and expected outcomes of the thesis.

CHAPTER 4: BENCHMARKS

4.1 Why using Benchmarks?

Generally a benchmarking suite is considered to evaluate the parallel computation models. A benchmark is usually a standard process or a program used to objectively evaluate the performance of an application. But this has few problems like the pace of the expressing the innovations in parallelism at its best is obscure with current benchmarking models. Bench marks don't consider any architectural or parallel application requirements which are very important to consider while evaluating parallel programming models; they are most of the time generic. Therefore a need for more high abstraction is identified in [21] where the result is a term called "Dwarf".

The main intention is to portray the application requirements in a manner that is not confined to scientific applications (or) overly specific to some individual applications (or) the optimizations used for certain hardware platforms so that we can draw broader conclusions about hardware requirements. This type of classification of the standards is preferred because the parallel applications and methods of parallel computations change over time but the underlying patterns remain constant and therefore comes the use of the concept.

4.2 Dwarfs

Dwarfs are first proposed by Phil Colella [21]. "*A dwarf is an algorithmic method that captures a pattern of computation and communication [21]*". A dwarf specifies in high level of abstraction about the behavior and requirements of the parallel computations. Proposing the dwarfs is often challenged by two factors described below.

- How well they present the abstraction for capturing the applications?
- What are the dwarfs added over time to cover the missing areas beyond High Performance Computing?

The eligibility of an application as a dwarf can be decided based on the similarity in computation and data movement. Dwarfs can be classified based on their distribution on a platform in two different ways and can be defined by using the communication between dwarfs. The dwarfs included in a parallel application can be mapped on two different environments [21]. They can be classified into temporal distribution environment (based on the concept of sharing the same dwarf) (or) spatial distribution environment (based on the concept of running separate dwarfs on each of the processors). Various kinds of dwarfs are identified till now by [21].

The composition of dwarfs selected for the study is based on the model set used and the implementation of the dwarfs. The model set is described in the methodology Chapter 6 and the sections 4.4 to 4.7 present the proposed implementation of the dwarfs. The dwarfs selected for the thesis are the dense linear algebra application. This particular set of dwarf constitutes algorithms based on the dense matrices. Dense matrices are the matrix where the behavior of each and every element of the matrix is defines, i.e. there are no zero entries in the matrix. Few of the Algorithms which are based on this concept are classified as shown below.

- Matrix - Matrix multiplication.
- Matrix - Vector Multiplication.
- Gaussian Elimination.
- Jacobi iteration.

- Laplace heat Distribution.

These kinds of applications tend to follow a regular structure and are called data parallel algorithms. These algorithms are best suitable for data parallelism and facilitate the data decomposition.

4.3 Problem Domains

The dwarfs used in the study are identified to be used in the following areas.

4.3.1 Linear Algebra

A matrix represents a collection of elements; where each element defines the behavior of the matrix at that position. Matrices are most popularly used in linear Algebra. One important use of matrix – matrix multiplication is to propose the linear transformations to represent the higher dimensional simplifications and similarities of linear functions of the form f(x) = cx, where c corresponds to the constant. For the Multiplication of matrices various methods are proposed for matrix multiplication. The algorithm selected in this category is a dense matrix-matrix multiplication.

4.3.2 Linear System of Equations

A set of mathematical equations are said to be linear if they involve n unknown variables in the equation. The linear system of equation takes the form as shown below [7].

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots\ldots\ldots + a_{n-1,n-1}\,x_{n-1} = b_{n-1}$$

$$\vdots$$

$$a_{2,0}x_0 + a_{2,1}x_1 + \ldots\ldots\ldots + a_{2,n-1}\,x_{n-1} = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 + \ldots\ldots\ldots + a_{1,n-1}\,x_{n-1} = b_1$$

$$a_{0,0}x_0 + a_{0,1}x_1 + \ldots\ldots\ldots + a_{0,n-1}\,x_{n-1} = b_0$$

For example

$3x_1 + 2x_2 - x_3 = 0$ --------- (1)
$x_1 + 5x_2 - 3x_3 = 5$ ---------- (2)
$5x_1 + 2x_2 - 7x_3 = 16$ --------- (3)

The above equations can be solved easily by writing them in the matrix form AX = B as shown below.

$$\begin{matrix} 3 & 2 & -1 \\ 1 & 5 & -3 \\ 5 & 2 & -7 \end{matrix} * \begin{matrix} x1 \\ x2 \\ x3 \end{matrix} = \begin{matrix} 10 \\ 5 \\ 16 \end{matrix}$$

Solving a system of linear equations AX = B. The system of linear equations can be solved by using direct solvers or by using iterative solvers. The algorithm used is based on iterative solvers and is the Jacobi iteration method.

4.3.3 Convergence methods

Convergence methods are more frequently encountered in numerical analysis techniques. This involves applications which concern about the study of an effect on a set of data points

residing on a mesh. This involves computation of the new values or points from already defined boundary values. For example, we compute the values sequentially in natural order from like $x_1, x_2, x_3, \ldots\ldots x_n$. When x_i has to be computed it can be done by iteratively calculating the value from previously computed $x_1, x_2, \ldots\ldots\ldots\ldots x_{i-1}$ Values. This kind of applications are more frequently used in computer graphics, image processing and weather forecasting techniques and heat distribution e.t.c. Gauss – Seidel relaxation method is used to solve the fast convergence series.

The algorithms selected for the thesis are based on the concept of reflecting the communication and computation patterns posed by different algorithms and the algorithms selected are presented below. These algorithms are selected based on the factor of facilitating the easy development which could be an added benefit for this study.

- Matrix multiplication.
- Jacobi iteration problem.
- Laplace heat distribution.

4.4 Matrix - Matrix Multiplication

4.4.1 Sequential Application

This section presents the details about the matrix multiplication algorithm that is implemented in this thesis. A Matrix defines a behavior of elements or a set of elements where every element can be individually accessed by the coordinates row number and column number (i, j). For example $x_{i,j}$ refers to the element at i^{th} row and j^{th} column. Matrix multiplication can be done on any two dense matrices of same size (for square matrices) n X *n* or two matrices (for rectangular matrices) where the number of rows in one matrix must necessarily be equal to the number of columns in the second matrix. For example a matrix of *n* X *p* and another of size *p* X *m* then the resultant matrix is of size *n* X *m*. This thesis considers only the multiplication of the square matrices. This is because if the implementation is successful with the square matrices they can be easily adaptable to rectangular matrices and also sparse matrices as well. Let A, B be two square matrices of size *n* X *n* then the result of the matrix multiplication is another matrix of the size *n* X *n*. The procedure for multiplication of two matrices is shown below where a single element $c_{i,j}$ of the resultant matrix is computed by the adding the partial sums obtained by multiplying each element in matrix A at i^{th} row with the corresponding element in Matrix B at the j^{th} column. And the asymptotic time complexity for this is O (n^3).

$$\begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} * \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix} = \begin{bmatrix} a11*b11+a12*b21 & a11*b12+a12*b22 \\ a21*b11+a22*b21 & a21*b12+a22*b22 \end{bmatrix}$$

Figure 4.1: General Matrix - Matrix Multiplication [7].

4.4.2 Implemented parallel solution

The idea is to compute the algorithm in parallel by allowing each thread to compute only n/q X n/q rows of the resultant matrix. Row wise partitioning is used for decomposing the algorithm. In this algorithm the matrices A and B are placed in global memory and are shared among the threads and each thread is confined to only computing the values of n/p rows of the resultant matrix. This can be shown below.

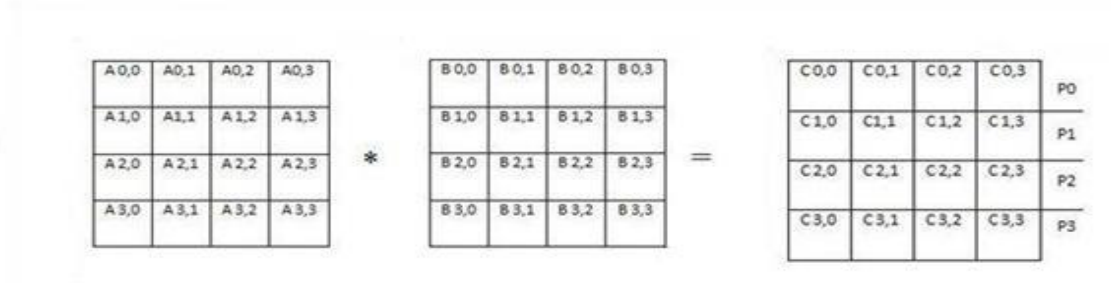

Figure 4.2: Implemented parallel Solution for Matrix – Matrix Multiplication [3].

4.5 Jacobi Iteration

4.5.1 Sequential Jacobi Iteration

It is an iterative solver for solving a system of linear equations. The traditional way of solving the Jacobi iteration is as shown below. Where a set of equations in n unknowns is given and the unknowns involved in the n equations need to be solved. In the iterative based solution the linear system of equations are represented in the form of matrices as shown below with the co-efficient of the unknowns in a 2-dimensional square matrix A and set of unknowns in the form of a vector X. And the result of the equations in the resultant vector B. This notation can now be abstractly represented using the matrices as AX =B. The Jacobi iteration is done in three phases and starts by computation of the new values of the unknowns into the matrix x_*new* using the formula shown below. This is the first phase of the algorithm.

$$x\_new_i = \frac{1}{a_{i,i}}\left[b_i - \sum_{j!=i} a_{i,j} * x_j\right] \text{ [7]}$$

The second phase of the algorithm to compare all the new values of the unknowns in the X vector are verified for acceptance values, which specifies the convergence of the solution and if the values reach the convergence the iteration stops otherwise the old values are swapped with the new values which is the last phase of the computation and the iteration proceeds until the convergence. A variety of solutions for measuring the convergence are presented in [7]. And the convergence selected in this solution is based on the approximate value obtained by substituting the new values in the equation and the difference between the results must be less than the selected tolerance value. The termination condition is specified below

$$\left|\sum_{j=0}^{n-1} a_{i,j}\, x_j^t - b_j\right| < \text{Tolerance [7]}$$

Apart from this the reading of the values in the matrix is done in such a way that the co-efficient matrix is diagonally dominant. This is the necessary and sufficient condition for the matrix to converge i.e,

$$\sum_{j!=i}\left|a_{i,j}\right| < |a_{i,i}| \text{ [7]}$$

4.5.2 Implemented parallel solution

The sequential time complexity for this problem is $O(n^2)$. This is the problem based on global communications. Where the coefficient matrix, the vector with unknowns and the

resultant vector is shared among the threads and the values of the unknowns calculated are confined to only the n/p rows of the new resultant vector [7].

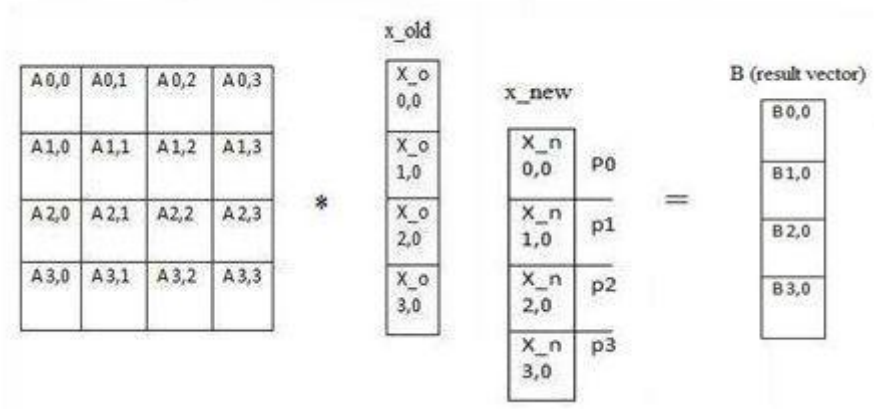

Figure 4.3: Implemented parallel solution to Jacobi Iteration.

4.6 Laplace heat distribution

4.6.1 Sequential Laplace Heat Distribution

This method is used to calculate the effect of heat distribution. This algorithm consists of a global matrix which represents the number of cells of a metal surface or simply the surface whose effect of heat on those cells need to calculated by performing the computations on the matrix. For example let A is the matrix to represent the surface on which the distribution of heat has to be calculated. The effect of heat is calculated using the formula shown below.

$$X_{i,j} = \frac{x_{i,j+1} + x_{i,j-1} + x_{i+1,j} + x_{i-1,j}}{4} \quad [7]$$

This method calculates the new values of the matrix by using the most recent values of the matrix already computed. This method of using the most recent values in computing the new values is called Gauss-Seidel Relaxation. And the newly computed values are verified for the convergence where the largest change encountered by the cell is compared with the acceptance value.

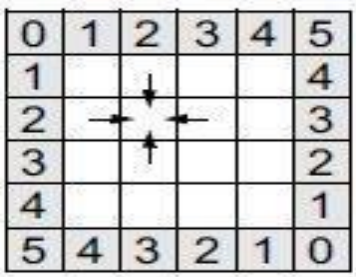

Figure 4.4: Sequential Laplace Heat Distribution [7].

4.6.2 Implemented parallel solution

This is the problem based on the local communications where threads are communicated using their maximum local value of heat change occurred in a cell. This computation uses the most recently computed values for the computation of new $x_{i,j}$ and therefore uses the red-black ordering algorithm pointed in [7]. This procedure can be depicted below. The

partitioning of the matrix is done wise to ensure the computation in parallel. Therefore each process is limited to compute the new values only up to n/p * n rows.

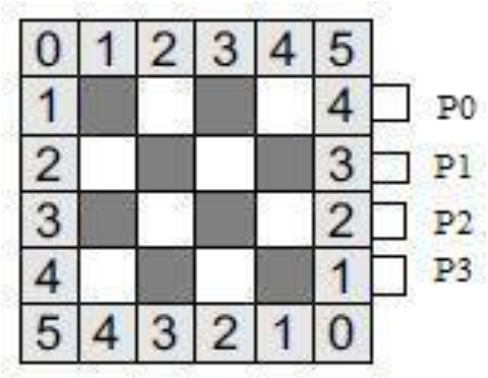

Figure 4.5: Implemented parallel solution for Laplace Heat Distribution [7].

4.7 Summary

This chapter describes in detail about the dwarfs and the need for using the dwarfs in parallel computing rather than using the benchmark standards. This chapter also describes in detail the dwarfs considered for this thesis, the type of algorithms selected, and the requirements for those algorithms and the implemented solution for those algorithms.

CHAPTER 5: RESEARCH METHODOLOGY

5.1 Research Approach

The research approach used is a quantitative study and the process starts with the collection of data about the shared memory based parallel programming models that are considered for the evaluation of the thesis. For this research first a theoretical study is done on the programming models and the dwarfs selected for the thesis. Later an empirical study will be conducted for developing the dwarfs selected on each parallel programming model and calculate practically the factors development time and speedup for an application. The results obtained from the experiments are used to analyze and compare the parallel programming models.

5.1.1 Quantitative Research

Answering the above research questions involves conducting quantitative research procedures in the concerned field of study. In order to understand how the application's development time and performance are affected by a specific parallel programming model a series of controlled human-subject experiments are conducted. The experiments are conducted for four different parallel programming models and for three different dwarfs (can be explained in detail in Chapter- 6). These experiments are used to collect data about the factors involved in this study and to later evaluate and compare those factors. This research study involves answering the research questions drawn from the problem statement as mentioned in Chapter-2. All the research questions are drawn with shared memory based parallel programming models into consideration and focuses on comparing the directive based models, thread based models and task based models.

For answering the research question RQ1; during the empirical study the performance measurements are taken and the required parameters are calculated. After calculating the results an investigation is done on the performance results obtained from the empirical study and are used for the comparison. Section 5.4.1 presents the procedure followed for the calculation of the performance parameter and the comparison of the programming models Chapter - 9 presents the answer to the RQ1.

Answering the research question RQ2 involves investigating the development time results gathered from the empirical study. The empirical study involves coding the application for all parallel programming models and data regarding the calculation of the development time are taken. The methods for calculating the development time are presented in the section 5-7. The data thus collected is used to calculate the overall development time according to the model proposed in this study. The same experiment is done for all the applications for each parallel programming model. The final development times of the applications thus obtained are used to compare the parallel programming models.

RQ3. This question can be answered by using the results obtained by conducting the experiments from the two perspectives i.e., from the development time perspective and the performance perspective. An investigation is done on the experimental results obtained and the comparison is done based on the trade-off between these two different factors. Section 5.6 presents a qualitative description to this question and the answer to this question is presented in Chapter-9.

5.2 Variables

This study involves collection of data about the variables from the controlled human-subject experiments conducted for the thesis. It is necessary to classify those variables or parameters

that are involved in the study. As pointed by Creswell [19] the variables in this study are identified to be as independent variables and are categorized as shown below.

- **Speedup -** The speedup of an application is defined as the ratio of serial execution time to the parallel execution time of the algorithm [36] and can be explained in detail in Section 5.4.
- **Development time -** It is simply the total time taken to implement and execute a parallel solution for the given sequential application and can be explained in detail in Section 5.5.

5.3 Instrumentation

The instrumentation as guided by Wohlin et al [11] classified depending on the two factors; they are the objects and the measurement instruments. The Objects used in this study are parallel application code document and person time diaries. The measurement instruments are compilers and data recorded from human subject used in this experiment. The values collected from the instruments are the direct and objective measures which include from the performance perspective are the execution times which are calculated by the compiler instrument and from the development perspective are the manual data collected from the subjects.

The factors collected from the code document specification are the SLOC, the number of model specific routines, the types of defects and are identified to be as internal attributes collected from the product object which is maintained using the code document. And the attributed collected from the person time diaries object are identified to be as internal attributes collected from the process workflow as part of the development process. From the personal time diaries the data collected are the time taken to establish a model specific activity, time taken for writing the SLOC and the time taken to rectify those defects. All these measures are also direct and objective measures extracted from the code specification document. Apart from these parameters another independent and indirect parameter that has to be calculated is the Speedup parameter which is calculated using the execution times.

5.4 Speedup

Speedup deals with the execution time or time complexity of the parallel application and is an important metric used in parallel computing for measuring the performance. Speedup tells us how fast is the implemented parallel solution implemented for the sequential application compared to the original sequential version of the application. Parallel execution time of an application must always be the fraction of the serial application's execution time and speedup is the ratio of the execution times of the serial program to the parallel program. A plethora of literature can be figured out for the speedup metric and can be found in [1], [3], [13], [17], [18], [28], [29], [31]- [33]. A general notation for the speedup is presented below.

$$\frac{run\ time\ of\ the\ sequential\ program}{run\ time\ of\ the\ parallel\ program} \qquad [3]$$

Some common definitions for speedup defined by different authors are described below. The definitions presented below are based on the variants employed for the serial and parallel execution times to calculate the speedup. The different types are shown below [28].

- Relative speedup – it is ratio of the time required to solve a parallel application on one processor to the execution time of the parallel program on P processors. This relative speedup of an application depends on many factors like the input size, the type of the problem and number of processors. In general it is not a fixed number.

- Real speedup – it is the ratio of the time required to run the best sequential application on a 1processor of the parallel processor to the execution time obtained by solving the parallel problem on P processors.
- If the parallel execution is defined in the form of a fraction of the best serial algorithm in a best serial computer then it is termed as absolute speedup.
- The asymptotic speedup is the ratio of the asymptotic time complexities of the best sequential algorithm to the parallel time complexity of the same algorithm. This can be in turn divided into types [28] Relative and real asymptotic speedup.

A speedup limit is defined by Gene Amdahl also popularly known as Amdahl's law or fixed size speedup [18], [32]; which means that the speedup achieved by the parallel application depends on the serial fraction of the parallelized application. [1] Presents a metric to estimate the serial fraction in the parallel application. All the speedup types presented above are based on fixed work load instances. There is another form of speedup defined by Gustafson [17] and later generalized by Sun Xian [32]. This type of speedup is for varying work load instances. This speedup is known as scaled speedup and is explained below.

- Scaled speedup is the performance model that is used to study the performance of the system under varying workloads and is calculated by making observations on the speedup obtained by scaling the size of the workload instance with respect to the number of the processor nodes used. Scaling the problem instance or the work load instance can be done in two ways which defines two variants in this model. One method is to vary the problem size with respect to the number of the processor nodes, keeping the execution time constant this is called the fixed time speedup. And in other model the memory size is scaled by using the problem size with respect to the processing elements; this method is called the scaled size or memory bounded speedup [3], [28], [32], [17]. These factors are used to study how the speedup varies when either or all of the following factors are changed; like the problem size, the number of processors, the memory size e.t.c. This method of study is used to analyze the scalability of the system.

other types of speedup are also defined they are; **super linear speedup** [13], [32] – also termed as anomalous speedup where the speedup bounds greater than the processors number used and **cost normalized speedup**[28] – used to define the speedup in terms of the accomplishments in the performance of the system with respect to the cost expended. Among all the speedups mentioned above the asymptotic speedups doesn't require any empirical measures where as all the rest need empirical measurements to calculate the speedup. The former method of calculating the speedup is known as analytical speedup and the latter is known as measured speedup.

5.4.1 Calculating the speedup of the parallel applications

The base for the performance results is the execution time measurements of the applications that are developed under a parallel programming model. The measurements of the execution times are taken for increasing number of nodes by powers of 2 on the selected platform. The results that obtained are used to calculate the speedup of an application for all the programming models. The type of the speedup used in this study is self relative speedup which is the speedup of the application achieved under increasing number of nodes; which is developed under parallel programming model. This is used to calculate the growth of the speedup of the same application with increasing number of nodes. The speedup of parallel programming models is affected by the overhead encountered by that programming model

while parallelizing the serial application. This study uses the Relative Speedup parameter for the comparison because of the advantages associated with this parameter described below.

- It considers only the algorithms which are easy to implement, rather than the best known algorithms that are considered for real and absolute speedups which is a disadvantage for them.
- It is trivially possible to select a bad and easy to parallelize algorithm and achieve good speedups as we scale through the processors and data sizes [28], rather than expending time on parallelizing the best algorithms known.
- It is also possible to estimate the overhead associated with the problem when it scales to increasing number of the processors.

The speedup measures are calculated for the parallel application on 2-nodes, 4-nodes and 8 – nodes using the execution times of the application obtained for 1-node, 2-nodes, 4-nodes and 8- nodes. The speedup thus obtained is used to compare it with the development time of the application.

5.5 Development time

It is the effort expended for the parallelizing the application and to make it run successful on the parallel computer. The literature related to the development is found in [30], [23], [15], [4], [36], [24], [12], [2]. This includes the time expended to develop the parallel application from the serial version. It is the time spent on the programming model related issues while parallelizing the application. Development time of a programming model is influenced by the following factors.

- The total number of source lines of code for a parallel application in a parallel programming model (development time by code volume).
- The number of defects of the application under a specific parallel programming model (explained in detail in section 5.6).
- The number of model specific functions used in the parallelizing of the application and the frequency of those routines encountered during the development.

The first issue is used to study the impact of the development time on achieving a model specific activity. This kind of studying the development time with respect to the effort spent in writing the code also known as development time by code volume. Whereas factor 2 is used to study the impact of the model in producing defects and the time spent debugging those defects and factor 3 is used to study the overall development activity of a parallel programming model. Studying all the above three factors requires the usage of the directional study based on the data collected from the past which is noted due to achievement of a model specific activity.

5.5.1 Methods for Calculating the Development time

Analyzing and estimating the development time of a parallel programming model can be done in three ways [12]. The first way is to employ an archeological study, second is to use the Daily Activity Dairy and the last method is to conduct an observational study for calculating the development time.

- In the archeological study the development activities are concerned and the time taken by the developer for each activity is calculated. The archeological study focuses on the time elapsed between different phases of the development process and it was a sort of directional study of developer’s time expended in implementing one feature; by using the data collected from the past [12].

- The main intention of Daily Activity dairy approach is to support the marking of the time elapsed for an activity accurately by using the object called personal time dairies. These are maintained by the programmer during the development of the applications to improve recording efficiency while at the same time minimizing the overhead to the subject. This approach has the advantages of measuring the development time in an accurate manner than the archeological study and more over it can be used to conduct a wide range of studies by varying the parameters like the number of developers and the features of the application and or the development environments e.t.c.
- The third method is the direct observational study which stems from the notion of observing the results collected for a set of developers based on the self reporting. This can be achieved by periodically conducting conversations among a group of developers who take part in the study. The times taken by the programmers at each phase of the development are recorded and are later subjected to observation and conclusions are drawn from the observation. This type of approach is mostly seen in pilot studies [30].

5.5.2 Manual Data Reporting vs. Deductive Analysis

In general two types of data reporting techniques are proposed by classifying them based on either purely manual or purely automated techniques [2].

5.5.2.1 Manual Data reporting

The is the first method of capturing the data related to the development time during the development process and relies on the data reported by the programmer that is recorded by considering an accomplishment of a set of activities or enactments. The manual data is recorded after the completion of the activity. This type of notion has few advantages mentioned below.

- This type of data capturing method assists in the study of the behavior of the process under study.
- This type of data capturing method can be used to make cross referential studies and can be easily extended to complex systems.
- This study can be used to comprehend their behavior of egregious problems and to avoid their forth comings in the near future.
- Also this study is helpful for the proposal of more convenient analysis techniques.

5.5.2.2 Deductive Analysis

This is the second type of data capturing techniques for the software process [2]. This method relies on using the notion of a logical specification for the capture of the process data rather than relying on the manual data. So far a number of formal notations had been proposed and the method used for the study depends on the notation used to study the process. The advantages associated with this method are presented below.

- They can be easily adapted into other methods for process specification and analysis.
- These models provide flexibility by clubbing them with other models.

Apart from the techniques mentioned above hybrid methods are also proposed for the capturing of the process data.

5.5.3 Distribution of effort for different programming models

The effort expended for parallelizing an application under a parallel programming model is hard to quantify, because this usually depends on the skill of the programmer, the model specific routines used the type of the problems, the type of the model used and many more issues. For the distribution of effort a variation of the programming models is considered. One suggested method is to use the time expended in performing the activities which are common to all the programming models. The effort expended by the developer for developing an application under a programming model is distributed among the factors described as in the above section 5.5 is summarized as shown below.

- The time taken for developing the model related code it is the total source code.
- The time taken to develop the model specific functionalities.
- The time taken for testing and debugging the developed code (this includes the time taken for rectifying the errors, enhancing new facilities if necessary also called performance tuning).

5.5.4 Scope of the programming model for calculating the Development time

The applications developed under the parallel programming models are used to estimate the impact of the programming models on development time. This reporting of the development time data mentioned above can be extracted from three main sources. One way is to collect the data regarding the development time using the source code analysis, second is through the collection of data from a version control repository and the third way is to analyze the results obtained after the regression testing of the system as a whole. This thesis uses the source code analysis as a major source for the data collection about the development times for effort distribution mentioned above in section 5.5.3. The remaining two reporting techniques will be used for large scale software projects. The impact of the programming models on development is calculated and confined to the source code of the application implemented under the parallel programming model. As part of source code analysis a code base is maintained for each application implemented under each parallel programming model which is used for estimating the development time based on the effort distribution mentioned in section 5.5.1. The source code analysis approach is selected as a source for calculating the development times because of the following reasons.

- Studying the development time of the code can also helps to understand the performance of the code of the parallel application being developed.

- Code Development time is an issue when we port applications from one system to another.

5.5.5 Calculating the Development time

In this thesis development time is calculated in three steps. First is to identify the type of the development time study and the second step is to identify the sources for calculating the development time. Third step is to distribute effort among the sources and to report the data about the effort expended in developing the parallel application. The approach followed in this thesis is to estimate the development time using the daily activity dairies where the data collected is from the developer by making a note of the time taken during the development and from the above qualitative analysis the source code analysis is identified as a source for calculating the development times. The effort distribution is done as mentioned in section 5.5.3 and the reporting technique used is the manual data reporting. Using these methods in the study has some advantages shown below.

- This technique is used to best reveal the activities involved in the process like the wasted intervals the learning effects and so-on.

- This technique is used to understand the behavior of the programmer and also to study the impact of the programming models on the development time.

- This whole study is equally important where the development time of an application expended by one programmer can be compared with the development time taken by another programmer, typically a form of multi-developer study as described above.

5.6 Defect Studies

It is necessary to study the defects encountered during the development of the applications in different parallel programming models. This is so required because the defects encountered during the programming model depends on many factors like the type of the programming model used, the type of the model specific constructs used and the skill of the developer e.t.c. Defect studies are done in the debugging phase of the application development (as shown in the work flow model in Chapter - 6). Debugging the applications under a parallel programming model requires the detection of the errors and modifying those defects. Defects in an application may arise because of the following problems.

- Error in the logic.
- Errors in the syntax.
- Errors that occur in runtime.

Errors in the logic are due to the wrong logic while parallelizing the applications. These can be the defects in establishing inter-process communication between the processes and synchronizing the processes. In general in thread based programming models the level of abstraction for parallelizing the applications is less compared to a directive based parallel programming model. In a thread based programming model the issues like inter and intra process communications are exposed and it is up to programmer to achieve these functionalities in parallelizing the applications. Where as in the directive based programming model these issues are governed by the underlying compiler. The user is relieved from establishing the complex communication and synchronization routines between processes.

Errors in the syntax are due to the errors that occurred in expressing the semantics of the language in parallelizing the applications. In a thread based programming model, since all the functionalities are exposed to the programmer i.e. the programmer is responsible for establishing the complex routines in parallelizing the applications, this leads to a complex way of expressing the parallel application and is quite common to encounter more errors than a directive based model. Runtime errors have some common properties for all kinds of parallel programming models, and need to be examined carefully. This can be done by examining the output of the application executed under a parallel programming environment. This requires examining the intermediate values and final output values for the functionality to be achieved by an application provided by the parallel programming models. The rate at which the defects occur varies between parallel programming models and is different to that for a sequential application. No model based prediction can be done for detecting and rectifying the defects as for a sequential application. And this depends on many factors the skill of the programmer, the abstraction provided to the programmer by the underlying programming model.

5.7 Comparing Development time and Speedup

The development time calculated from the above procedure is used to compare with the relative speedup of the applications. This can be done as follows. Initially a parallel version of the algorithm is developed from the sequential version. The total development time for that algorithm is noted using the daily activity dairies. These development times are noted for the implementation of the same algorithm under all the parallel programming models. On contrast the execution times are collected for measuring the performance and the speedup is calculated from execution times. The relative speedup is calculated for 2, 4, and 8-nodes respectively. By the end of the thesis with the results obtained a qualitative comparison is build-up by showing the % of the overall time taken for the implementation of a dwarf under a programming model 'X' to the overall development time under programming model 'Y', with the development of the dwarf in model 'X' consumes a % of time for rectifying the defects to the time taken for the rectification of the defects in programming model 'Y'. On the other hand we quantitatively compare the performance of the application by showing that the relative speedup achieved by an application implemented under one parallel programming model 'X' is approximately --% of the relative speedup achieved by the same application in programming model 'Y'. This result gives the development time consumed and the relative speedup of an application in all parallel programming models and this guides the developers about when to switch for an alternate parallel programming model with respect to the development time and / or relative speedup [22].

5.8 Summary

This chapter presents the type of the research methodology required for comparing the parallel programming models. This chapter also describes the qualitative study about the speedup and development time parameters, methodology for calculating those parameters and the instrumentation used for the comparison by using the results obtained from the empirical study.

CHAPTER 6: EXPERIMENTAL CASE STUDY

6.1 Experiment Design

A frame work is designed that describes a procedure to test the models empirically to validate the models against the research questions. Table 6.1 presents the frame work designed for this chapter. This frame work is used for the family of studies and in this case is used for a shared memory based parallel programming models. The models used in this study are Pthreads, OpenMP, TBB and Cilk++. The experimental methodology involves implementing the parallel computing dwarfs on these models and a data collection is done on these models for calculating the speedup and development time. The collected data is then analyzed and validated against the research questions. The results thus obtained are used to compare the parallel programming models. The types of algorithms examined in this study are based on the dwarfs that are used extensively in the field of parallel computing research as explained in Chapter - 4. And the type of parallel applications is selected based on the data parallelism concept. This is because data parallel applications are more widely used, flexible and more performance oriented. Also they tend to follow a regular structure and computationally intensive. The subject selection for the experiments is shown in section 6.2.

Table 6.1: Design Frame work.

Serial implementation of Matrix-Matrix multiplication	Pthreads implementation of Matrix-Matrix multiplication. OpenMP implementation of Matrix–Matrix multiplication. TBB implementation of Matrix–Matrix multiplication. Cilk++ implementation of Matrix-Matrix multiplication.
Serial implementation of Jacobi Iteration.	Pthreads implementation of Jacobi Iteration. OpenMP implementation of Jacobi Iteration. TBB implementation of Jacobi Iteration. Cilk++ implementation of Jacobi Iteration.
Serial implementation of Laplace Heat Distribution.	Pthreads implementation of Laplace Heat Distribution. OpenMP implementation of Laplace Heat Distribution. TBB implementation of Laplace Heat Distribution. Cilk++ implementation of Laplace Heat Distribution.

The model set required for executing the experiments is shown below in table 6.2. The serial implementations mentioned in the above framework are only used to take the execution times and output values which are used to verify the parallel implementations for correctness and accuracy purposes. All the parallel codes are written and tested for 8-cpu based shared memory machine architecture with 16 GB ram each processor sharing 2 GB. The platform employed for the study is an open source Linux version. All codes are written in C, the size of the code lies in between 70 to 350 lines. While all the codes are written to run on this architecture, none of them are optimized to take advantage of the architecture, it is assumed that the vendor specific implementations of the library and architecture are already optimized and further effort on improving leads to poor performance, excess use of resources and is not considered. The attributes and considerations for the system under study are shown below.

Table 6.2: The empirical study model set.

Programming language	C
Types of models	Shred memory based models
Dwarfs	Dense matrix applications
No of algorithms	3
Types of algorithms	Data parallel applications
Code Size	70-350 SLOC
Processor	Intel
Platform	Ubuntu Linux
No of cpus	8
Ram	16gb
Staff	1

6.2 Experiment Workflow

The subject selected for the study is the student conducting a thesis in the concerned field of study and is observed to be categorized under the novice programmer category. This categorization is based on the skill of the developer where a novice programmer and the eligibility of the subject are based on the interest of the subject in the field of study. The implementation of the dwarfs is carried out this programmer. First an in-depth study of the models and the algorithms being implemented and later designs and implements the algorithms for all the models. The algorithms are tackled by dividing the problem definition into individual or cooperative tasks and each are individually coded and at later point of time are used to include interactions between the tasks. A standard procedure is followed for designing the parallel formulations of the dwarfs that are selected for this study. This involves four phases like partitioning the work load among the workers, establishing communication between them, synchronizing multiple workers and mapping tasks on to the workers. Since this research involves only the use of small parallel algorithms all the studies that are conducted doesn't contain any high level project management or configuration management. Only the implementation and execution of the applications are concentrated by the developer. The generalized development work flow is given below by the Markov process models from the point of view of lonely developer that is being followed for all the algorithms considered for the study. The usage of work flow models is first described in [8]. Figure 6.1 outlines the development work flow for the empirical study. The workflow of the experiments employed in this study involve designing, developing, running, testing and tuning the algorithms on the selected target parallel architecture by a single individual no cooperation is made with any other members during the experiments, and coordinate the efforts.

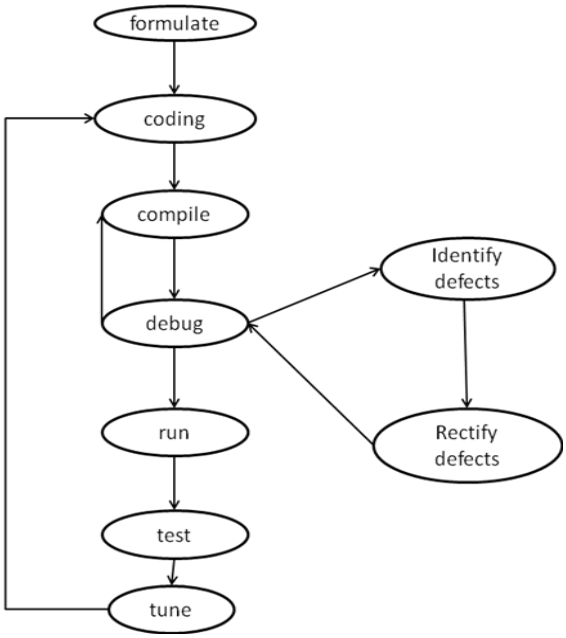

Figure 6.1: Lone Programmer Work flow [8].

6.3 Experiment Execution

The execution of the experiment consists of mainly two-steps. In the first step the dwarfs are implemented under each parallel programming model and development time of the working dwarfs are noted. In this phase all the information related to the development time and performance and speedup is collected. In the second step the speedup is calculated from the information collected in the first step and then the programming models are compared for the development time in contrast to the speedup achieved under the specific parallel programming model.

6.4 Data Analysis and Validation

The study involves few data items to be collected from the experiments conducted above. This data that has to be analyzed and validated can be categorized into two types Qualitative factors and quantitative factors. Table 6.3 and 6.4 shows the data that has to be collected from the measurement objects mentioned in Section 5.3. All the data collected from the empirical study are identified as objective measures and direct measures as pointed by [11]. Apart from the data mentioned in the table the parallel programming models are compared by taking only the final development time which is the overall development time of the application and the indirect measure speedup. The rest of the data mentioned in section 5.3 is used to study the inter code development activities and to classify the programming models as mentioned in the chapter 5.

Table 6.3: Collection of data from the experiments for a single programming model.

Parameter	D1	D2	D3
Parallel run time on 1 node			
Parallel run time & speedup on 2-nodes			
Parallel runtime & speedup on 4-nodes			
Parallel runtime & speedup on 8-nodes			

Table 6.4 Data collection by Person time diaries.

Parameter	Dwarf 1	Dwarf 2	Dwarf 3
Time for model specific activity.			
Time taken for total SLOC (overall development time)			
Time for rectifying total Defects.			
Total SLOC.			
Total Number of Defects.			
Total no of Library routines.			
Frequency of Library Routines.			

All the data that is collected from the experiments is used to calculate the parameters used for the study.

6.4.1 Validity

During the collection of data, there may be a chance that the data for the experiment does not meet the requirements of the experiment (i.e. Applications with poor quality which caused by the occasional bad coding of the dwarfs using less efficient routines etc.). As a result, a validity check must be done on the data collected. This includes the data collected from the speedup and development time perspective. For validating of the development time data the ratio scale is used to guide in selection/rejection of the development time data. Similarly for the speedup the scale used is absolute scale where the speedup value is selected or rejected based on whether the value is meaningful or not. The speedup calculated and collected is compared against a notable value for all the dwarfs. If any of the algorithms fail to satisfy the desired performance again a tuning for performance is done as mentioned in section. Also in this case the development times are altered. The inclusion of the data related to development times are taken after the enactments in establishing a parallel activity. If in any case the tuning of performance is done the modified development times are replaced on to the old development times.

6.5 Execution times

Calculating the speedup requires the collection of the execution times during the operation of the experiments as mentioned above. The user is allowed to run the benchmarks and make a note of the execution times. The user specifies varying types and amounts of the input and is responsible for executing the benchmark initially. Finally, since the speedup studied is fixed size relative speedup, the execution times thus obtained by the user are noted for fixed and notable size of the input data and studied for the features required. The execution times are calculated for the serial implementation of the dwarf, the parallel implementation of the dwarf on one node and the execution times for 8-nodes. The execution time is manually calculated by using the built in functions provided by the parallel programming library or the functions provided by the underlying language compiler. The execution time is typically measured in seconds, and is calculated from the moment the parallel threads are created and after the moment all the threads are finished. For the serial implementation the unit of measurement is the same and the execution time is calculated from the moment the computational task begins and the moment the task ends.

6.6 Examining the output

Testing the algorithm is a challenging issue [23]. Examining the output include the result collected after the execution of each application. For verifying the overall correctness of the algorithm, the collected data is used to verify with the output obtained for the sequential version of the application. Verifying the output doesn't makes ensures that the implemented algorithm will work correctly. The values computed by individual workers need to be verified. This can be done using the selection of small test cases to ensure that the algorithm works for small data sizes. Later the algorithm is adapted for large data sizes. The algorithms need to be tested for the performance this involves a qualitative analysis on the execution times of the algorithm and depending on the nature of the problem. The algorithm is executed for fixed input size to get notable performance measurements; after ensuring the correct execution of the application. The execution times noted are used to calculate the parameters required for the comparison like speedup. Performance tuning is done if necessary. Defect studies are done on the models. During the testing process the defects raised are modified and the effort for this is calculated and noted.

6.7 Debugging

Debugging involves the defects that arise in the codes developed. Defects however are important for the studies are capable of imparting the model specific drawbacks on the parallel algorithms employed. This effect in turn has the drawback of consuming more development times during the process. Therefore defects are equally important as do the code performance and have to be extensively studied because of their impact in deciding the performance of the parallel software applications. The rate at which the defects arises and the effort required for rectifying the defects are two major issues of concern. In this study the programming models are monitored for defects and thus identified can be rectified in this case by debugging the algorithm. This involves the detection of bugs like data races, deadlocks and model specific factors. In each case a data collection is maintained about the number of defects identified for each programming model for an algorithm and the effort required to rectify the defects and validation of the data against the model developed is also studied. This is studied as part of the development process, the resources required for this are calculated.

6.8 Tuning

Performance tuning is required for the algorithm to work efficiently, it is the effort spent in enhancing the performance and alleviating the defects that occur for a specific algorithm.

6.9 Threats to Validity

6.9.1 Internal Threats

The following internal threats are identified during the study [19].

When to stop the work flow? This is regarded as the threats related to the workflow procedures employed as part of the experimental study. It is not exactly sure how many iterations of the workflow have to be performed for a given dwarf. This is because the factors like performance tuning are done for the dwarfs only if necessary. But however this threat will not affect the study because the outcome is assumed to be perfect only if the dwarf is considered to be efficient.

The characteristics of the developer are a threat. The developers of the software are novice and didn't have any experience before regarding this particular field. The skill of the developer, implementation of the dwarfs takes more time typically than from the expert programmer's perspective. Also the performance of the applications achieved is less when compared to an expert programmer. This threat is relaxed by establishing strict constraints during the development and at the same time trying to achieve better performance as best as possible.

6.9.2 External Threats

The following external threats are identified according to the classification specified by [19].

- The experimentation procedure and generalization of the research questions are studied only for small dwarfs and from a novice perspective. Therefore this study doesn't involve any high level metrics, only the metrics related to the implementation and performance are considered.
- The use of the diverse programming models is an external threat. This is so because, since the study is done from the perspective of a novice programmer, learning of the programming models may lead to the threat known as learning effect, that the programmer be familiar with the parallel models during the study.

6.9.3 Statistical Conclusion Validity

The data collection related to development time and speedup is not accurate. This is because inadequate procedures available related to development time and also calculating the parallel execution time for one specific data size is not sufficient. The threat related to development time can be avoided by making accurate measurements as efficient as possible and related to parallel execution time the threat can be avoided by taking execution times till specific number of intervals.

6.10 Summary

This chapter presents the experimental methodology that is followed for this thesis.

CHAPTER 7: RESULTS - DEVELOPMENT PHASE

7.1 Development time from Source Code Analysis

7.1.1 Matrix-Matrix Multiplication

The development times for the Matrix – Matrix multiplication application are shown in the table 7.1.

Table 7.1: Development times for Matrix - Matrix multiplication Algorithm.

Programming model	Development time in man- hours
Pthreads	30 hours
OpenMP	16 hours
TBB	10 hours
Cilk++	6 hours

7.1.1.1 Pthreads

For the implementation of the matrix - matrix multiplication as mentioned in chapter - 4 is partitioned row wise among the threads and each thread is responsible for making the computations local to its rows. The time taken for the Worker creation and management is approximately 47% of the total development time for the matrix multiplication algorithm and the time taken to develop the partition activity for allotting equal workload among the threads is approximately 53% of the total amount of time spent in developing the application. This version of the algorithm with equal partitions among the threads occupies 96 SLOC in volume. The total time taken for the development of the matrix-matrix multiplication algorithm is 30 man-hours.

7.1.1.2 OpenMP

For the implementation of the matrix multiplication under OpenMP the same type of the partitioning method is employed but the difference is that work is allocated among the threads done by the compiler. The developer only has to specify the size of the partition. The time taken for the Worker creation and management is approximately 37% of the total development time for the matrix multiplication algorithm and the time taken to develop the partition activity for allotting equal workload among the threads is approximately 63% of the total amount of time spent in developing the application. This version of the algorithm occupies 79 SLOC in volume. The more development time for the work allocation of the algorithm is more because it is not sure as a novice programmer which variables to be declared as shared or private while partitioning. The time taken for the development of the matrix-matrix multiplication algorithm in OpenMP is about 16 man-hours.

7.1.1.3 TBB

The Matrix-Matrix multiplication developed for this model consists of the specification of only one task and the number of SLOC is 91 lines. The time taken for the development of a single task is around 85% of the total time taken for the development of the algorithm. And the time taken for the initialization and termination of the threads is around 15% of the total time taken for the development. This parallel version consists of only one task which is the task developed to specify the parallel for iterations which is shared among the threads. And hence the percentage of the development of this single task is high. All the data partition models provided by the TBB were used during the development of applications in TBB and time taken for the development is included in the parallel _for () which consumes the time as

mentioned above. The total time taken for the development of the matrix-matrix multiplication algorithm in TBB is about 10 hours.

7.1.1.4 Cilk++

The matrix multiplication developed for this model consists of the specification of only one task as in TBB and the number of SLOC is 66 lines. The time taken for the development of a single task is around 90% of the total time taken for the development of the algorithm and the remaining time is spent for the other model independent functionality. Cilk++ avoids the possibility of creating and terminating the external threads as done by using the initialization object in TBB. Rather Workers in Cilk++ can be specified in command line which almost takes negligible amount of time for managing the workers. The task developed in this application consists of only one cilk_for () loop which is shared among the threads. Two kinds of partitioning methods are considered one is the dynamic partitioning done by the underlying system and the other is by using the #pragma cilk_grainsize directive to specify the chunks externally by the user. The total time taken for the development of the matrix-matrix multiplication algorithm in TBB is about 6 hours.

7.1.2 Jacobi iteration

The development times for the Jacobi iteration algorithm in all the programming models are depicted in table 7.2.

Table 7.2: Development times for Jacobi iteration algorithm.

Programming model	**Development time in man- hours**
Pthreads	75 hours
OpenMP	46 hours
TBB	22 hours
Cilk++	18 hours

7.1.2.1 Pthreads

The development activity of the Jacobi iteration encompasses two phases; the first phase is to simply develop the algorithm using the mutual exclusion locks for shared writes and by using barriers for synchronization. This development for establishing the basic thread synchronization activity takes approximately 26% of the total development activity of the total development time for the Jacobi iteration algorithm. While the other implementation phase uses conditional variables along with the mutual exclusion locks for guarding the shared writes takes 43%. The implementation of the last phase consumed most of the development activity this is because embedding condition variables when ever doing shared writes are hard to identify and more over difficult to achieve. The first phase of the algorithm occupies 189 SLOC, where as the second phase takes around 217 lines of code respectively. The total time taken for the development of the Jacobi iteration algorithm in Pthreads is about 75 man-hours.

7.1.2.2 OpenMP

The development activity of the Jacobi iteration encompasses only a single version. In this version the shared variables are ensured for correctness by guarding them by using critical directives and using barrier directives for synchronization where ever necessary. The synchronization barrier only occupies one line of code whereas that in Pthreads the barrier occupies at least 8 lines of code. These two activities consumed 64% of the total time for the development of the algorithm with the implementation of shared writes using the critical section and atomic directives takes 28% of the development time. And the time taken for the

development of the barriers at the pre-analyzed synchronization points takes approximately 36% of the total time taken for the development. This version of the algorithm occupies 118 SLOC. The total time taken for the Jacobi iteration algorithm in OpenMP is about 46 man-hours.

7.1.2.3 TBB

The Jacobi iteration consists of three tasks one for each parallel_for () with each having three different functionalities. This development is done in phases where in the first phase the development of the first task is considered which is the parallelization of the computation of new values for the Jacobi iteration. This consumes 30% of the total time for the development of the algorithm. The second task is to develop the parallel tolerance function which takes the approximately 45% time for the development. This is because of the use of the shared writes in the task development, which is done using the mutual exclusion locks and scoped locks provided by the TBB library. The third task is to map the new values to the old array which takes the approximately 25% time for the development. The partitioning details are similar to that of the matrix multiplication. What is new in this application is the use of mutual exclusion locks the task of establishing and using locks for guarding the data items where the threads share among them. This subtask alone incurs extra development time and hence the development time of task 2 is more than the other tasks. The total source code is about 155 lines. The total time taken for the Jacobi iteration algorithm in TBB is about 22 man-hours.

7.1.2.4 Cilk++

The Jacobi iteration consists of three tasks one for each cilk_for () same as in TBB with each having three different functionalities and also the development of the algorithm proceeds in phases where in the first phase the development of the first task is considered which are the parallelizing of the computation of new values for the Jacobi iteration. This consumes 30% of the total time for the development of the algorithm. The second task is to develop the parallel tolerance function which takes approximately 40% of the total development time. The third task is to map the new values to the old array which also takes the same time as for the task1. The partitioning details considered for the development are similar to that of the matrix multiplication. This application uses mutual exclusion locks the task of establishing and using locks for guarding the data items is easy in Cilk++. This subtask incurs the extra development time which is the reason for the high development time of task 2 than others. The total source code is about 118 lines. The total time taken for the Jacobi iteration algorithm in Cilk++ is about 18 man-hours.

7.1.3 Laplace Heat Distribution

The development times for the Laplace heat distribution algorithm are depicted in table 7.3.

Table 7.3: Development times for the Laplace heat distribution.

Programming model	Development time in man- hours
Pthreads	90 hours
OpenMP	58 hours
TBB	29 hours
Cilk++	24 hours

7.1.3.1 Pthreads

The development activity of the algorithm follows the same procedure as the Jacobi algorithm and has two phases, but the only difference is that the previous algorithm is based on synchronizing and communicating the threads using global variables and this algorithm is

based on synchronizing and communicating the threads using local variables. This phase of the algorithm consumes 30% of the development activity with 287 lines of code. Whereas the second phase among all consumed more time than others with 346 lines of code volume and consuming 48% of the development activity. The total time spent for the development of the Laplace heat distribution algorithm is about 90 man-hours.

7.1.3.2 OpenMP

The development activity of the algorithm follows the same procedure as the Jacobi algorithm in OpenMP. This version of this algorithm contains the communications of local variables also there are situations where only one thread is ensured to execute that particular region. The latter activity uses barrier directives along with single directive for establishing this sort of structured communication. These two activities consume a total of 63% of the total development time with establishing the first activity consumed 24% of the total development time and is coded using the single directive. These writes are also coded using the atomic, critical directives as a supplement to the single directive as in Jacobi iteration. And the second activity consumes 39% of the total development activity. The more development time for this is because of the second activity. The total time taken for the development for this algorithm in OpenMP is about 58 man-hours. This version of the code occupies 219 lines of code.

7.1.3.3 TBB

The development of the application begins in the same way as that of the above matrix multiplication and Jacobi iteration. As this resembles a modular approach, the development of the application starts with identification of the tasks which have a scope of parallelizing the operations using parallel_for and parallelizing each task separately. The parallelization of the algorithm consists of three tasks one for the computation of the even points, odd points and a common maximum value computation task. Each of the above two tasks consumes 30% of the total time taken for the development and the rest of the time is consumed by the tolerance activity. The total source code of the algorithm is about 212 lines. The total time spent for the development of the Laplace heat distribution algorithm in TBB is about 29 man-hours.

7.1.3.4 Cilk++

The development of the application proceeds the same way as that of the above two applications that is a modular approach, to identify the tasks of data parallel applications which have a scope of parallelizing the operations using cilk_for and parallelizing each task separately. The parallelization of the algorithm consists of three tasks one for the computation of the even points, odd points and a common maximum value computation task. Each of the above two tasks consumes 32% of the total time taken for the development and the tolerance function takes 36% of the total development time. The total source code of the algorithm is about 201 lines. The total time taken for the development time of this algorithm in Cilk++ is about 22 man-hours.

The following observations are noted by analyzing the source code.

- Code development in Pthreads consists of more activities like the work load partitioning, worker creation and management, task mapping, communications and synchronizations. The number of lines of code that is linked with these parallel activities is considerably higher than any other model. Further the development time for writing the code is higher. For the Pthreads Communications activity which includes the establishment of shared writes is hard and consumes more time, the next

thing that consumes more development time is the synchronization activity and then follows the worker creation and management and partitioning activity e.t.c. This is the reason for Pthreads having large variations in the development time between different activities.

- For OpenMP the activities that had to be developed while parallelizing the applications include the specification of the work sharing directives, parallel loop iterations, communications and synchronizations among the threads. This less number of LOC in OpenMP is because of the advantage of the directives in reducing the code length. The longer development time than tasking models is because of the obscurity due to using the directives for structured communications among the threads where the threads need to be synchronized while on the other hand only one thread is allowed to write into a shared variable and also since the structure of the applications in OpenMP is different.
- TBB is the second model that has great impact on the Source code length. The development of the applications in TBB consists of only the activities like specification of the tasks that consists of parallel loop iterations and communications e.t.c. Individual activities in TBB doesn't need any synchronization. The activity that consumes most of the development time for data parallel applications is the parallelization of the loop iterations. The next activity that consumes much development time is the communications routines for guarding the shared writes. But since the advantage with task models is that to parallelize the data parallel applications the developer only has to partition the loop iterations and therefore the effort can be more or less equally distributed among the tasks of the application. And there is not much variation in the development times of the individual activities.
- Cilk++ is the best model amongst all the models that is capable of decreasing the code length during the development of parallel applications. As in TBB the development of the parallel applications consists of the activities like specifying the tasks with parallel loop iterations, communications and synchronizations. The last two activities are easy to develop and consume less time than any other model. It is easy to distribute the development effort among the activities in Cilk++ during the development. Most of the development time of the applications is consumed for the development of parallel tasks and then for communication routines and at last for the synchronizations among the tasks. As in TBB the effort can be equally distributed among the tasks and due to this there are no large variations in the development times between the different tasks in Cilk++.

From the above analysis the development times for the source code by considering all the activities constitutes 100% of the development time. This includes other tasks like the thread routines(thread id,), the time calculation routines, the time taken for establishing the function calls for these activities and the time taken for testing and debugging, time for taking the measurements and so-on. This is because the time is distributed for model dependent/independent functions routines. It can be seen from the above as one progresses from the development of the applications in threading models to the development of the applications in task based models; the number of model specific parallel activities that the developer has to implement decreases on the other hand the time spent in developing the function or activity increases gradually even though the total development time is less. As a result the actual development time of an activity in a particular programming model is also small; as one progresses from thread based models to task based models.

7.2 Overall Development Activity

The total number of model specific routines used and the frequency of those routines for each model are discussed in this section. The development times taken for the development of the different algorithms in different parallel programming models are shown below in figure 7.1.

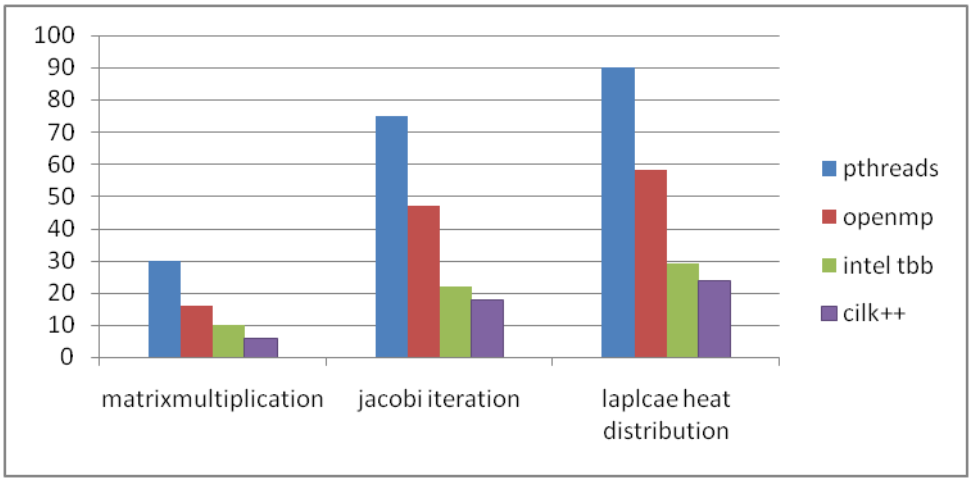

Figure 7.1: Development times of dwarfs in different Programming models.

7.2.1 Pthreads

The number of model specific routines and their frequency of occurrence encountered during the development of the dwarfs are depicted in the table 7.4. Pthreads has more flexibility and the number of library routines provided to the developer is considerably high than any other models. The library routines exposed by Pthreads are at very low level on the other hand uniting several of them in a proper way achieve a parallel feature. For example for sequential consistency synchronizing the threads at some predefined points is done and this is achieved using the mutual exclusion locks and global data variables and conditional variables. Writing parallel applications in this way lead to lengthy source code and also the frequency of calls to those routines is also considerably high. This type of development of the parallel applications has its effect on increasing the defects and also the readability is very low. Novice programmers who had knowledge with using threads can easily migrate to Pthreads.

Table 7.4: Pthreads library routines and their frequency during development.

Library routine	Frequency
Pthread_t	1
Pthread_attr_t	1
Pthread_mutex_t	1
Pthread_cond_t	1
Pthread_attr_init()	1
Pthread_cond_init()	9
Pthread_mutex_init()	9
Pthread_create()	1
Pthread_join()	1
Pthread_mutex_lock()	9
Pthread_mutex_unlock()	9
Pthread_mutex_trylock()	0
Pthread_cond_wait()	9
Pthread_cond_broadcast()	9
Pthread_cond_destroy()	9
Pthread_mutex_destroy()	9

7.2.2 OpenMP

OpenMP is a directive model with almost the flexibility compatible with Pthreads, on contrast decreasing the amount of the source code associated with using the directives. The frequency of the calls in OpenMP is also considerably high as in Pthreads. Therefore the structure of the applications developed in OpenMP have a slightly different structure, but also facilitates the use of low level routines as in Pthreads under such cases the possibility of defects slightly increases also they are difficult to identify but easy to rectify. The directives and library routines used in the development of the applications in OpenMP are shown below in table 7.5.

Table 7.5: OpenMP library routines and their frequency during development.

Library routine	Frequency
#pragma omp parallel	1
#pragma omp for	8
#pragma omp atomic	1
#pragma omp critical	2
#pragma omp barrier	17
#pragma omp single	9
Private	3
Shared	8
Static	8
Dynamic	8
Guided	3
Omp_get_thread_num()	3
Omp_set_num_threads()	3
Omp_get_wtime()	6

7.2.3 TBB

The structure of the parallel applications developed in TBB is completely different. TBB projects the specification of parallel features from the perspective of tasks rather than from the perspective of threads. This had a great effect on decreasing the flexibility of the developer on the other hand hiding several of those low level features also facilitates easy development and also the number of library routines use for the development is low, but the frequency of the function calls are high. This has the effect of producing always the same kinds of errors. As a consequence they are easy to identify and debug. The number of model specific routines used in TBB is shown and also their frequency is shown in table 7.6. TBB stands at high level of abstraction of all the programming models and because of the complex structure of TBB in specifying the tasks, sometimes the scope of the errors even though limited are little hard to rectify from a novice programmers perspective.

Table 7.6: TBB library routines and their frequency during development.

Library routine	Frequency
Task_scheduler_init()	3
Init.initialize()	3
Terminate()	3
Parallel_for()	7
Void operator()	7
typedef mutex mut_ex :: lock	2
mut_ex::scoped_lock mylock()	3
tick_count :: now	6
task_scheduler_init:: deferred	3
Simple_partitioner	7
Auto_partitioner	7
Affinity_partitioner	7

7.2.4 Cilk++

Cilk++ follows the simplest structure for parallelizing the applications. As compared to all the other models the number of library routines required is very low i.e., about 3 functions. This will improve the ease of development of the applications by the user as compared to the other models, also at the same time defines the boundaries of the application. On the other hand Cilk++ decreases the flexibility associated with user. Also those parallel routines are easy to use and less number of source lines of code is involved in writing these routines. On the other hand the frequency of the calls however is more because Cilk++ is a tasking model like TBB and follows the same parallelization points as TBB. The model specific routines used in TBB are shown and also their frequency is shown in table 7.7.

Table 7.7: Cilk++ library routines and their frequency during development.

Library routine	Frequency
Cilk_spawn	7
Cilk_sync	8
Cilk_for()	7
Cilkview	3
Cilkview::start()	3
Cilkview:: stop()	3
#pragma cilk_grainsize	7

7.3 Overall Defect Studies

This section presents the defects that are encountered during the development of the dwarfs in each programming model. Table 7.8 summarizes the basic types of errors and the possibility of those errors in a programming model is shown.

7.3.1 Pthreads defects study

Pthreads is a programming model leaving more flexibility to the developer by allowing the developer to achieve the parallel features by using the very low level functional routines as a consequence the possibility for the occurrence of errors in Pthreads is high also their severity and frequency is observed to be high in Pthreads. It is observed during the study that the testing and debugging takes almost more than half of the time spent in writing the code. Figure 7.2 depicts the time taken for rectifying the defects in Pthreads for different applications in man - hours.

7.3.1.1 Data Errors

These are the errors that arise due to the incorrect decisions in making the data as either private or global. Pthreads development uses pointers as a method to parallelize the application. Therefore the developer must know about what are the variables that are used as shared or local variables. Incorrect decisions about the usage of the variables as shared or global has a great impact in committing the errors in the code when modified by the threads the following are the data errors identified during the development activity. The pointer problems can be casting the data value to a different type, Improper Justification of the private and shared data, false sharing.

7.3.1.2 Pointer Errors

Pthreads development uses pointers as a method to parallelize the application. Therefore the developer must be conscious about the usage of the pointers in the parallel applications. Failure to do so leads to hard to find errors during the development of the applications. Incorrect decisions about the usage of the pointer variables as shared or global leads to

confusion and therefore lead to wrong computations when code is modified by all the threads. The following are the pointer errors identified during the development activity. They can be casting the data value to a different type, passing the parameters to the function of different data type e.t.c.

7.3.1.3 Partitioning Bugs

These are errors due to improper work allocation amount the threads. A parallel program is said to be parallelized 100% then the work shared among them must be equal in any case. Incorrect work allocation leads to more execution times and consequently less speedup achieved. This is due to more work done /undone or work done wrong. The developer must ensure the correct work allocation among the threads; in general the work allocated must be equal for whatever is the number of threads used in the program.

7.3.1.4 Worker management

Worker creation and management in Pthreads are difficult in Pthreads, this is because the creation and termination of the threads includes the calling the library routines with pointers as parameters. Also Pthreads only passes the parameters to the thread function as of type (void*). As in the case of a novice programmer the developer must be sure in passing the correct parameters. The following are the defects identified during the development.

7.3.1.5 Synchronization and Communication Bugs

Communication bugs are mainly due to the incorrect usage of the conditional variables in the program, which leads to the condition failure. In such cases an abnormal halt of the program is encountered. Wrong invocation of conditional variables (or) Errors in the coding of condition variables has huge drawbacks on the performance of the algorithm like the threads reporting communication failures or reporting wrong conditions. In Some cases threads may hold on the conditions for infinite time, or a thread may never reach the condition. These are mainly due to specifying wrong conditions (or) the wrong boundary values. Global communications are very hard to establish in Pthreads. this is because the developer must keep track of the which threads to be communicated using which locks and conditions when there are more shared writes on the global variables in the program. Communication bugs can be improper conditional waits, improper conditions achieved, or conditions never achieved. More threads waiting for one or more conditions at the same time. Synchronization bugs on the other hand also includes the errors due to specifying incorrect locking on the data items, threads holding one or more locks, threads trying to acquire lock held by others. Applying more locks on the same shared variable.

7.3.2 OpenMP Defects Study

OpenMP relieves the pain of rectifying the defects by the developer. OpenMP exposes the environment required for parallelizing the applications, the programmer only have to specify the directives and routines at the parallel points identified during the code design. An incorrect directive doesn't introduces new defects in OpenMP rather the directive is skipped and the region is executed sequentially. If the functional routine is specified incorrectly an error message is displayed about the error. Figure 7.2 depicts the time taken for rectifying the defects in OpenMP for different algorithms in man-hours.

7.3.2.1 Work sharing defects

These are similar to the partitioning defects in Pthreads, and occur due to the incorrect sharing of the workload among the threads. This can be due to the improper specification of the chunk size or improper looping variables for the omp for work sharing construct.

7.3.2.2 Nested directives

These defects arise due to the improper use of the directives. Typically nested directives i.e., improper use of directives inside other directives leads to these kinds of errors are also easy to identify and rectify.

7.3.2.3 Work scheduling Defects

OpenMP provides three different kinds of scheduling approaches. Improper specification of the chunk sizes for sharing the workload leads to data races also the work scheduling defects. This type of errors also increases the overhead in scheduling the tasks.

7.3.2.4 Data races

These defects usually occur due to the improper use of the critical directives or the use of the atomic directives for complex instructions. Data races are hard to identify and rectify this is because the behavior f the application with data races when executed is not reproducible or unpredictable. Also this depends on the work load and the relative timing of the threads involved.

7.3.2.5 Data sharing defects

These errors occur due to the improper decisions taken for the usage of the shared/private data variables by the developer while developing the application. Deciding which data values to be used as private and which are used as shared data is complex in OpenMP. These errors even though are easy to identify which is done by the underlying compiler, but are hard to rectify.

7.3.2.6 Synchronization defects

Improper use of the barrier functions for synchronizing the threads leads to these kinds of errors. The result of these errors is the abnormal termination of the program. These types of defects are usually more common in local communication based problems.

7.3.3 TBB Defect Study

7.3.3.1 Task management Defects

These defects occur due to the invalid usage of the task_scheduler () routines by specifying the wrong parameters for the initialization and/or termination of the threads and tasks. These effects are easy to detect and rectify. Figure 7.2 depicts the time taken for rectifying the defects in TBB in man - hours.

7.3.3.2 Work sharing Defects

These defects are similar to those in OpenMP and occur due to the invalid specification of the Parallel_for () call or the body of the parallel_for () construct. Improper specification of the partitioner function also leads to these kinds of errors.

7.3.3.3 Data races

These defects usually occur due to the improper use of the locks on shared data items. Data races are hard to identify and rectify. However TBB simplifies the rectification of the data races by supporting the usage of tools like Intel's thread checker and VTune Performance analyzer.

7.3.4 Cilk++ Defects study

Of all the programming models evaluated Cilk++ has the least possibility of causing errors. There is only possible kind of errors identified during the development they are Data races. Figure 7.2 depicts the time taken for rectifying the defects in Cilk++ in man - hours.

7.3.4.1 Data races

In Cilk++ are easy to identify the underlying runtime system which is ported with the Cilk++ compiler facilitates the identification of defects can be easily rectified by the Developer.

Table 7.8: Types of defects Occurred in Programming models during development.

Type of defects	Pthreads	OpenMP	TBB	Cilk++
Worker management defects	Yes	No	No	No
Partitioning bugs	Yes	Yes	No	No
Dependency in directives	No	Yes	No	No
Task mapping	Yes	No	No	No
Task management Defects	No	No	Yes	No
Loops with Data Dependencies	No	Yes	Yes	Yes
Synchronization defects	Yes	Yes	No	No
Data races	Yes	Yes	Yes	Yes
Data bugs	Yes	Yes	Yes	Yes
Communication Defects	Yes	No	No	No
Pointer defects	Yes	No	No	No

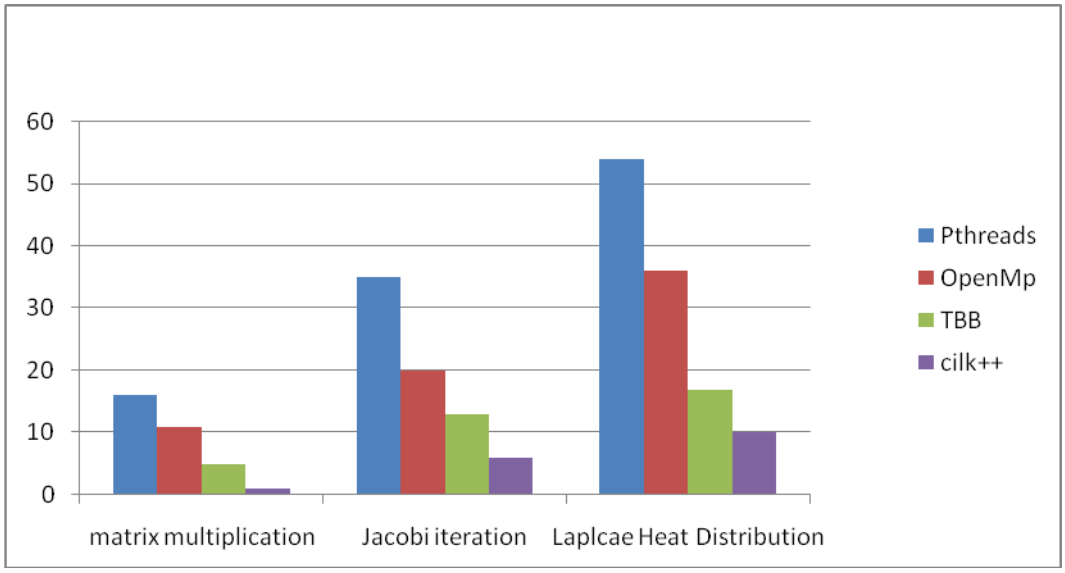

Figure 7.2: Time taken for rectifying the Defects during Development.

As can be seen from the figure 7.2 and table 7.8 the scope of defects as one progresses from thread based models to task based models decreases and as a consequence the total number of defects decreases. Further the overall time taken to rectify those defects also decreases. The extremely high times for rectifying the defects in thread based models is because of the frequency of encountering those errors mentioned above and the time taken to rectify those errors during development is more as compared to directive models which are in turn more than that of the task based models.

7.4 Summary

This chapter the results collected from the empirical study. These results are used to analyze and discuss and therefore to compare the parallel programming models.

CHAPTER 9: RESULTS - PERFORMANCE

8.1 Execution times

The execution times obtained for the algorithms developed in all the models are explained below. The execution times are taken for 1 node, 2 nodes, 4 nodes and 8 nodes.

8.1.1 Matrix Multiplication

Table 8.1: Execution times of Matrix - Matrix multiplication.

Programming model	Type of partition	Execution times			
		1 – node	2- nodes	4-nodes	5 – nodes
Pthreads	Normal	17.090	10.64898	6.65007	3.195
OpenMP	Static	17.613773	10.641508	6.009883	3.214531
	Dynamic	16.560187	10.644078	5.984909	3.315854
	Guided	17.312715	11.091114	5.642544	3.236784
TBB	Normal	18.497208	10.595537	6.280703	3.258302
	Simple	17.411227	9.096853	5.812421	3.076922
	Auto	17.880539	9.560571	6.033887	3.209946
	Affinity	17.174236	9.353453	5.576552	3.176292
Cilk++	Normal	17.202000	9.686000	5.537000	3.362000
	Dynamic	17.575001	10.758000	5.917000	2.860000

The execution times of the matrix multiplication are gathered by multiplying two matrices of size 1024 and are depicted in table 8.1. All the input values are initialized to the random values. Of all the models the Cilk++ model is observed to have better execution times and the Cilk++ which is 2.860000 on 8-nodes and the size of the chunk partition is chosen arbitrarily by the Cilk++ compiler. The next model with better execution times is the TBB with 3.076922 on 8-nodes which is observed to be obtained from the simple partitioning strategy of the application. The next model is Pthreads with 3.195 seconds on 8-nodes. The last one is the OpenMP with an execution time of 3.214531 on 8-nodes obtained for the static scheduling strategy. The outputs of all the algorithms are observed to be 100% accurate compared to the serial versions of the application.

8.1.2 Jacobi iteration

Table 8.2: Execution times of Jacobi Iteration.

Programming model	Type of partition	Execution times			
		1 – node	2- nodes	4-nodes	5 – nodes
Pthreads	Without Conditional variables	18.65160	9.378	6.445	6.378
	With conditional variables	18.65163	10.537	6.491	6.420
OpenMP	Static	12.528361	8.441140	7.180260	6.925106
	Dynamic	12.527674	8.450400	7.181433	6.924912
	Guided	12.530998	10.268568	7.174528	6.947828
TBB	Normal	12.402403	7.287078	6.403194	6.346713
	Simple	12-603139	9.481981	6.413131	6.352903
	Auto	12.402198	7.296328	6.411719	6.358586

	Affinity	12.404731	9.413560	6.388827	6.340177
Cilk++	Normal	12.402000	7.279000	6.384000	6.337000
	Dynamic	12.403000	7.277000	6.387000	6.340000

The execution times obtained for the Jacobi iteration are shown in the table 8. 2.
The execution times of the Jacobi iteration are gathered by computing the Jacobi iteration on the matrices of size 8000. And of all the models the Cilk++ model is observed to have better execution times which is 6.337000 on 8-nodes and the size of the chunk partition is specified by the programmer. The next model with better execution times is the TBB with 6.346713 on 8-nodes which is observed to be obtained from the simple partitioning strategy of the application. The next model is Pthreads with 6.378 seconds on 8-nodes. The last one is the OpenMP with an execution time of 6.924912 on 8-nodes obtained for the dynamic scheduling strategy. The outputs of all the algorithms are observed to be 100% accurate compared to the serial versions of the application.

8.1.3 Laplace Heat Distribution

Table 8.3: Execution times of Laplace Heat Distribution

Programming model	Type of partition	Execution times			
		1 – node	2- nodes	4-nodes	5 – nodes
Pthreads	Without conditional variables	83.65202	48.667	26.368	18.425
	With conditional variables	83.65200	48.65169	26.403	18.371
OpenMP	Static	118.448790	67.438849	36.256885	26.653553
	Dynamic	118.433673	69.798527	39.435519	29.585771
	Guided	118.420378	62.853548	39.452928	29.596448
TBB	Normal	118.348188	67.546543	38.703501	28.574435
	Simple	118.610353	62.816612	39.088978	29.276902
	Auto	118.352908	62.855214	40.090532	29.446836
	Affinity	116.175936	61.778997	35.638079	27.617736
Cilk++	Normal	118.292000	62.516998	38.962002	28.531000
	Dynamic	118.296997	62.546001	39.362000	29.04900

The execution times for the Laplace heat distribution for all the models are shown in the table 8.3. The execution times of the Laplace heat distribution are gathered by computing the algorithm on the matrices of size 2048 with a tolerance value of 0.02 and a relaxation factor of 0.5. And of all the models the Pthreads model is observed to have better execution times which is 18.371 on 8-nodes and the size of the chunk partition is specified by the programmer. The next model with better execution times is the OpenMP with 26.585771 on 8-nodes which is observed to be obtained from the dynamic scheduling strategy of the application. The next model is TBB with 27.617736 seconds on 8-nodes obtained for the affinity partitioning strategy. The last one is the Cilk++ with an execution time of 28.531000 on 8-nodes obtained for the partitioning strategy specified by the programmer. The outputs of the algorithm for the Cilk++, TBB and OpenMP are observed to be 100% accurate compared to the serial versions of the application. Whereas for the Pthreads version a deviation of is observed for the algorithm and this is mainly due to the variation of the floating point computations between models.

8.2 Speedup

8.2.1 Matrix-Matrix Multiplication

Table 8.4 depicts the speedup achieved by all the programming models for the matrix multiplication. Of all the models Cilk++ achieved high speedup which is 6.145105 on 8-nodes and that is for dynamic partitioning strategy done by the underlying runtime system. The second model to achieve high speedup is the TBB which is 5.676947. The next models that achieved speedup are OpenMP for the static partitioning strategy and Pthreads with 5.479422 and 5.211978 on 8-nodes respectively. All models for Matrix-Matrix multiplication achieved almost linear speedup.

Table 8.4: Speedup of matrix-Matrix Multiplication.

Programming model	Type of partition	Speedup		
		2-nodes	4-nodes	8-nodes
Pthreads	Normal	1.563743	2.504074	5.211978
OpenMP	Static	1.655195	2.930801	5.479422
TBB	Simple	1.745754	2.945085	5.676947
Cilk++	Dynamic	1.633668	2.699692	6.145105

The speedup for the Matrix-Matrix multiplication is depicted in the figure 8.1

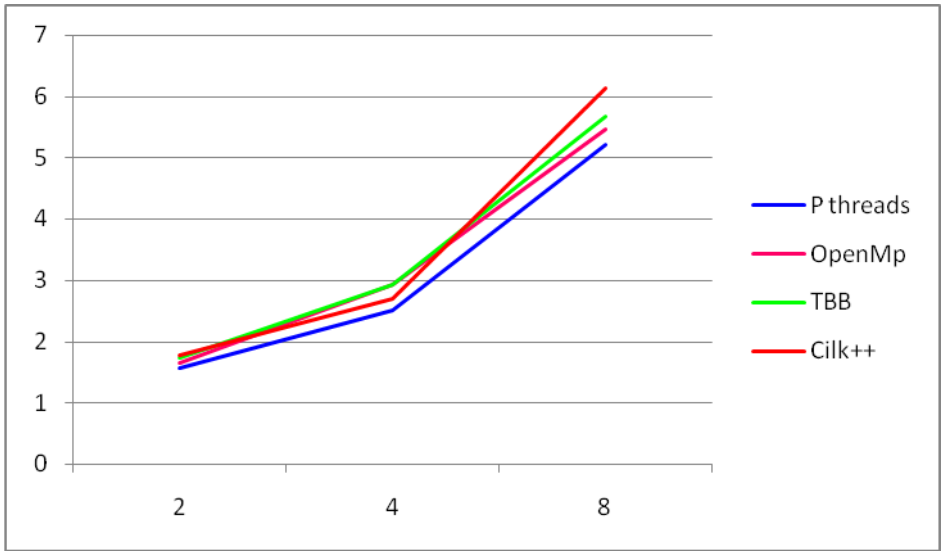

Figure 8.1: Speedup of Matrix-Matrix Multiplication.

8.2.2 Jacobi iteration

Table 8.5 depicts the speedup achieved by all the programming models for the Jacobi iteration algorithm. In general it's hard to achieve speedup for problems based on the global communications and the speedup achieved and the growth of the speedup in most of the cases is less than linear this is due to the overheads involved due to communications between threads. As the number of threads increases the overhead associated with the communications also increases. The speedup for the Jacobi iteration is depicted in the figure 8.2. Of all the models Pthreads achieved high speedup which is 2.924365 on 8-nodes for the version with conditional variables. The second model to achieve high speedup is the TBB for affinity partitioning strategy which is 1.983839. The next models that achieved speedup are Cilk++ for normal partitioning strategy and OpenMP for static partitioning strategy with 1.956768 and 1.809121 on 8-nodes respectively.

Table 8.5: Speedup of Jacobi Iteration.

Programming model	Type of partition	Speedup		
		2-nodes	4-nodes	8-nodes
Pthreads	Without conditional variables	1.988867	2.893964	2.924365
OpenMP	Static	1.484202	1.744833	1.809121
TBB	Affinity	1.329167	1.965208	1.983839
Cilk++	Normal	1.703805	1.942669	1.956768

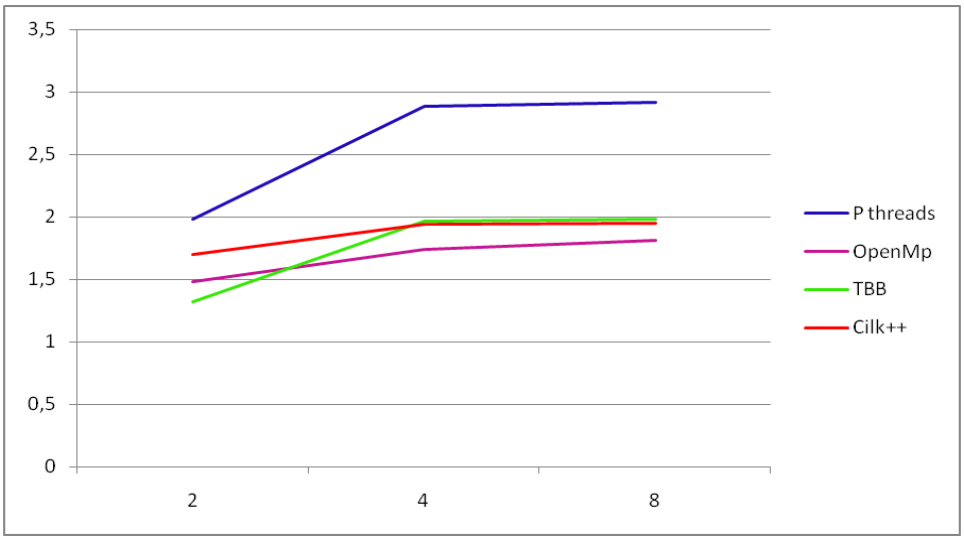

Figure 8.2: Speedup of Jacobi Iteration.

8.2.3 Laplace Heat Distribution

Table 8.6 depicts the speedup achieved by all the programming models for Laplace heat distribution. Problems based on local communications however incurs less overhead also the speedup achieved is greater than those with global communications. The speedup achieved in most of the cases is slightly less than or equal to the linear because of the less overheads involved due to communications between threads. As the number of threads increases the overhead associated with the communications also increases by very less amount. The speedup for the Laplace heat distribution is depicted in the figure 8.3. Of all the models Pthreads achieved high speedup which is 4.547540 on 8-nodes for the version with no condition variables. The second model to achieve high speedup is the OpenMP which is 4.444015 for static partitioning strategy. The next models that achieved speedup are TBB for the affinity partitioner strategy and Cilk++ for the normal partitioning strategy with 4.206569 and 4.146086 on 8-nodes respectively.

Table 8.6: Speedup of Laplace Heat Distribution.

Programming model	Type of partition	Speedup		
		2-nodes	4-nodes	8-nodes
Pthreads	With conditional variables	1.719405	3.168276	4.547540
OpenMP	Static	1.756388	3.266932	4.444015
TBB	Affinity	1.880508	3.259882	4.206569
Cilk++	Normal	1.892157	3.0362	4.146086

For Pthreads an error in tolerance of 0.00014 is observed compared to the other programming models.

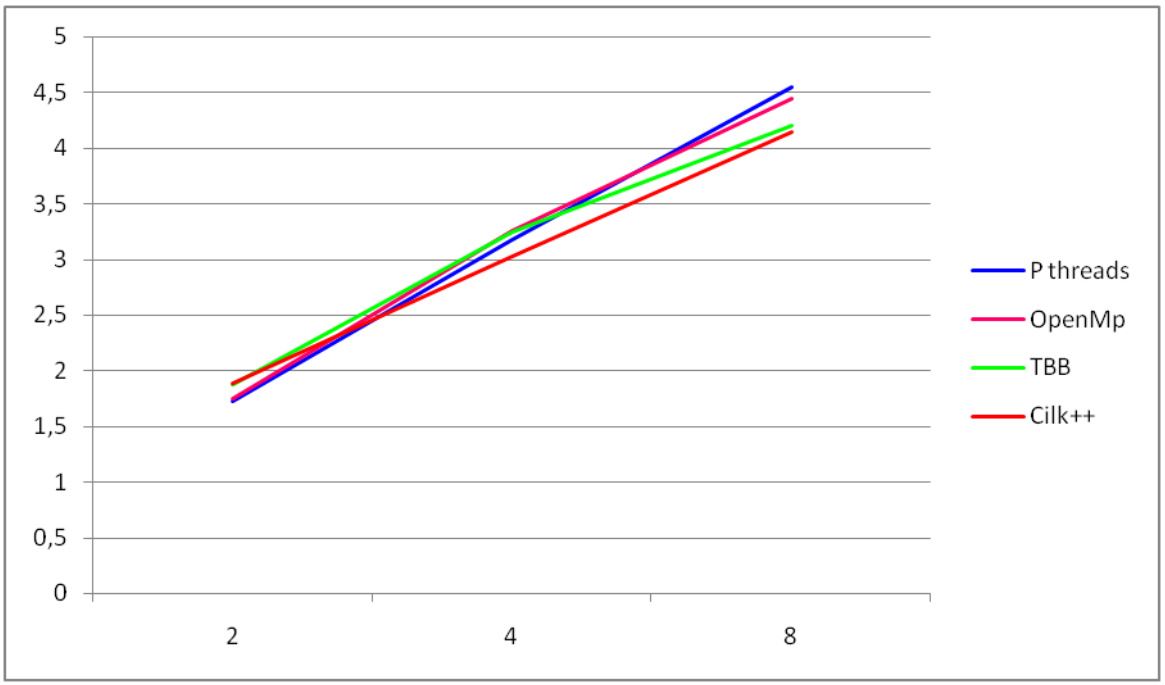

Figure 8.3: Speedup of Laplace Heat Distribution.

8.3 Summary

This chapter presents the performance details gathered from the results obtained due to the experiments. The performance factors presented here are the execution times and speedup of the applications achieved under each parallel programming models.

CHAPTER 9: ANALYSIS AND DISCUSSIONS

This section presents the results obtained from the empirical study. This section presents the answers to the research questions posed in the proposal of this study. This section presents the speedup results obtained during the empirical study and compares them with the results of the development time on the other scale. This type of comparison is useful to answer the challenges faced by the industries today.

9.1 Matrix multiplication

9.1.1 Performance

The results for the speedup of the matrix multiplication algorithm are shown in the graph depicted in figure 8.1. It is clear that Cilk++ parallel programming model ports high growth of the relative speedup for matrix multiplication on 2, 4 and 8 processors. The speedup is 1.63 on 2-nodes, 2.69 on 4-nodes and 6.14 on 8-nodes. The next model with better speedups is TBB with 1.74 on 2-nodes which is approximately 1.06X greater than the speedup of the cilk++ model on 2-nodes, and the growth continues to 4-nodes with approximately 1.09X greater than that of the cilk++ speedup on 4-nodes. The speedup on 8-nodes drops for TBB and is approximately 0 .92X the speedup supported by Cilk++. The next model is Pthreads with approximately 0.95X the speedup supported by the Cilk++ model observed on 2-nodes and 0.92X the speedup of Cilk++ on 4-processors and 0.84X the speedup of Cilk++ model on 8-nodes. The last model is OpenMP with approximately 1.01X the speedup of Cilk++ model on 2-nodes. The speedup supported on 4-nodes is approximately 1.08X the speedup supported by the Cilk++ model and on 8 – nodes the speedup supported is approximately 0.89X the speedup of the Cilk++ model.

9.1.2 Development time

On contrast the development times of the matrix multiplication is depicted in the figure 7.1. The development time of the matrix multiplication algorithm in Cilk++ takes only 6 hours to develop the full working application where as for the TBB the development time is 10 hours which is approximately 1.66X the development time of the algorithm in Cilk++. Whereas for Pthreads on the extreme has the highest development time which is 30 hours and is approximately 5X the development time relative to the development of the same algorithm in Cilk++. For the last programming model OpenMP the development time of the algorithm takes 16 hours which is approximately 2.6X the development time of the algorithm in Cilk++ model.

9.1.3 Conclusion

For matrix multiplication the TBB programming model is best from both views of speedup and development time with the speedup of 6.14 on 8 – nodes and consumes a development time of 6 hours for the successful working of the algorithm. The next models are TBB, OpenMP and Pthreads respectively.

9.2 Jacobi Iteration

9.2.1 Performance

From the graph depicted in figure 8.2 it is clear that Pthreads parallel programming model ports high growth of the relative speedup for Jacobi Iteration on 2, 4 and 8 processors. The speedup observed is 1.98 on 2 processors and 2.89 on 4-processors and 2.92 on 8-nodes.

The second good speedup is supported by TBB model which is approximately 1.32 which is approximately 0.69X of the speedup relative to the speedup supported by the Pthreads on 2 – nodes. And the growth of the speedup continues up to 4 - nodes where the speedup supported by TBB is approximately 1.96 which is approximately 0.67X of the speedup supported by the Pthreads on 4-processors. The speedup growth continues and on 8-processors for the TBB model the speedup observed is 0.67X the speedup supported by the Pthreads model. The next model that achieves good performance for the Jacobi iteration algorithm is Cilk++. The speedup achieved by Cilk++ on 2-processors is only 1.70 which is only 0.85X of the speedup supported by the Pthreads model on 2-processors. A fall in the growth of the relative speedup on 4 –processors for Cilk++ is observed where the speedup achieved is only 1.94 which is only 0.67X of the speedup relative to the speedup supported by the Pthreads. At last on 8-processors the speedup achieved by Cilk++ is 1.95 which is approximately 0.66X of the speedup relative to the speedup supported by the Pthreads on same number of nodes. For OpenMP finally the speedup up achieved on 2-nodes is 1.48 which is only 0.74X the speedup achieved by the Pthreads on 2-nodes. On 4-processors the speedup achieved is approximately 1.74 which is 0.59X of the speedup obtained by the Pthreads on 4-processors and on 8-processors the speedup achieved by the OpenMP model is 1.81 which is approximately 0.61X the speedup supported by the Pthreads model on the same number of nodes.

9.2.2 Development Time

On the other hand the development times of the Jacobi iteration is depicted in the figure 7.1. The development time of the Jacobi iteration algorithm in Pthreads as that of the development time of matrix multiplication in the same model takes the highest development time which is 75 hours to develop the full working application where as for the TBB the development time is 22 hours which is approximately 0.29Xthe development time relative to the development time of the algorithm in Pthreads. And for the Cilk++ on the extreme has the least development time which is 18 hours and is approximately 0.24Xthe development time relative to the development of the same algorithm in Pthreads. For the last programming model OpenMP the development time of the algorithm takes 46 hours which is approximately 0.61 times relative to the development time of the algorithm in Pthreads.

9.2.3 Conclusion

For Jacobi iteration which involves global communication between the threads the Pthreads programming model is best compared to others. But on the other hand Pthreads also consumes more development time and has the scope of introducing more errors leave most of the details to be ensured by the programmer. The speedup achieved by the Pthreads model is approximately 2.92 on 8-nodes and the development time is 75 hours. The next better models are mode TBB, Cilk++ and OpenMP respectively.

9.3 Laplace Heat Distribution

9.3.1 Performance

Clearly from the graph depicted in figure 8.3, it is clear that Pthreads parallel programming model again ports high growth of the relative speedup for Laplace heat distribution on 2, 4 and 8 processors. The speedup observed is 1.72 on 2 processors, 3.16 on 4-processors and 4.55 on 8-nodes. The second good speedup is supported by OpenMP model which is approximately 1.75 which is approximately 1.01X the speedup supported by the Pthreads on 2 – processors. And the growth of the speedup continues up to 4- processors and is better than Pthreads as well where the speedup supported by OpenMP is approximately 3.26 which is approximately 1.03X of the speedup supported by the Pthreads on 4-processors. But the

speedup on 8-processors faces a drop off for the OpenMP model which is 4.44 and is only 0.97X of the speedup relative to the speedup supported by the Pthreads model. The next model that achieves good performance for the Laplace heat distribution algorithm is TBB. The speedup achieved by TBB on 2-processors is only 1.88 which is only 1.09X of the speedup relative to the speedup supported by the Pthreads on 2-processors and is better in this case. A continuation in the growth of the relative speedup on 4 –processors for TBB is observed where the speedup achieved is only 3.25 which is only 1.02X of the speedup relative to the speedup supported by the Pthreads and in this case is more than Pthreads and also equally compatible with OpenMP. At last on 8-processors the speedup achieved by TBB is 4.20 which is approximately 0.92X of the speedup supported by the Pthreads on same number of nodes where a sudden fall of the speedup is observed for this model. For Cilk++ finally the speedup up achieved on 2-nodes is 1.89 which is only 1.09X of the speedup relative to the speedup achieved by the Pthreads on 2-nodes and also simply best than other models in this case. on 4-processors a drop in the growth is observed where the speedup achieved is approximately 3.03 which is 0.95X of the speedup relative to the speedup obtained by the Pthreads on 4-processors and on 8-processors the speedup achieved by the Cilk++ model is 4.14 which is approximately 0.91X of the speedup relative to the speedup supported by the Pthreads model on the same number of nodes.

9.3.2 Development Time

The development times of the Laplace heat distribution is depicted in the figure 7.1. The development time of the algorithm follows the same trend as in the development time of Jacobi iteration algorithm. Where the Pthreads model encounters the highest development time of about 90 man hours and the model at the second place related to the development time is the OpenMP model which is about 58 hours to develop the full working application which is approximately 0.64X of the Pthreads development. Whereas for the TBB the development time is 29 hours which is approximately 0.32Xthe development time relative to the development time of the algorithm in Pthreads. Whereas for Cilk++ on the extreme has the least development time which is 24 hours and is approximately 0.26Xthe development time relative to the development of the same algorithm in Pthreads.

9.3.3 Conclusion

For Laplace heat distribution which involves local communication between the threads the Pthreads programming model is best compared to others. The next good model for this is OpenMP. The tasking models stand after these models. For Laplace heat distribution algorithm also the Pthreads model consumes the more development time with encountering more errors during development. Also the type of errors is similar to that of the Jacobi iteration algorithm. The speedup achieved by the Pthreads model is 4.54 with a development time of 90 hours.

9.4 Validity Discussion

One important factor observed at the end of the study is the threat known as “the learning effect” as mentioned in the section 6.9.2. In this study the development of the dwarfs in programming models proceed with Pthreads model, OpenMP model then TBB and Cilk++. If a change of preconditions is done during the development process then the development times may alter i.e., the development of the applications in the programming models proceed with Pthreads model, TBB then OpenMP model and Cilk++. Under such conditions TBB may have more development times than OpenMP; this effect of change in the orders of evaluation is due to the skill of the developer characteristics as mentioned before. This effect may occur also for other ordering of the models that are chosen for the development.

One development issue related to TBB that negotiates with this argument is that the development of data parallel applications in TBB incurs the same kind of structure or patterns for all kind of applications depicting the parallel iterations or computations done by the multiple threads. As a result, the whole application involves only the development of the parallel iterations to parallelize the application. Thus for the programmer the development of the application is the same for all kinds and incurs less and same library routines. Therefore, the development times of the dwarfs are less. This effect can be validated by using another similar study with the same conditions but with different orders of evaluation. Then the final results obtained can be used to verify the result. Therefore, it could be an extension to this work as a future work to compare this study with the results obtained by an expert programmer in his study for the validation of the results.

9.5 Summary

This chapter analyzes and discusses about the results obtained from the empirical study. The results are presented in detail in the chapters – 7 and 8 and the analysis is done for each and every dwarf separately in terms of the speedup and development time.

SUMMARY

This report presents the empirical study about the comparison of shared memory based parallel programs. The study includes four parallel programming models tested for the independent parameters speedup and development using three dwarfs. The following answers are identified for the research questions posed in the Chapter – 2.

RQ1. What is the speedup achieved by the application under the chosen programming models?

The speedup achieved by a parallel application varies for different programming models. For the models chosen in this thesis only considering from the speedup perspective; for problems without communication the task based models out performs than the threading models which are rated on the other extreme and the directive models can be rated in between them. Sections 9.1.3, 9.2.3 and 9.3.3 assist this answer.

RQ2. What is the development time of the application under each parallel programming model?

Developing the parallel code and making it successful as mentioned before depends on the number of factors like the model specific library routines used, the number of defects and the time taken to rectify those defects. From the development time perspective by considering all the above factors obtained from the experimental results; the task based models facilitates easy development and consumes considerably less time than other models by sacrificing the flexibility like the one associated with the threading models which stand on the other extreme of the development time scale. Threading models have more development times because of the complexity associated with the models when parallelizing the applications and the hard times that developers face in rectifying the defects and so-on. On the other hand the directive models stand in between and also the time taken for rectifying the defects is also moderate.

RQ3. Which model is best when compared to the relative speedup and the total development time of the application?

From the experimental results to compare the parallel programming models based on the trade-off between development time and Speedup; the directive models are the best compared to other models by supporting a balance between these two factors. If development time is not a problem and the performance is important; then the threading models are preferred next on the other hand with a sacrifice of very less performance then comes the task based models.

From this study it is clear that threading model Pthreads model is identified as a dominant programming model by supporting high speedups for two of the three different dwarfs but on the other hand the tasking models are dominant in the development time and reducing the number of errors by supporting high growth in speedup for the applications without any communication and less growth in self-relative speedup for the applications involving communications. The degrade of the performance by the tasking models for the problems based on communications is because task based models are designed and bounded to execute the tasks in parallel without out any interruptions or preemptions during their computations. Introducing the communications violates the purpose and there by resulting in less performance. The directive model OpenMP is moderate in both aspects and stands in between these models. In general the directive models and tasking models offer better speedup than any other models for the task based problems which are based on the divide and conquer strategy. But for the data parallelism the speedup growth however achieved is

low (i.e. they are less scalable for data parallel applications) are equally compatible in execution times with threading models. Also the development times are considerably low for data parallel applications this is because of the ease of development supported by those models by introducing less number of functional routines required to parallelize the applications.

This thesis is concerned about the comparison of the shared memory based parallel programming models in terms of the speedup. This type of work acts as a hand in guide that the programmers can consider during the development of the applications under the shared memory based parallel programming models. We suggest that this work can be extended in two different ways: one is from the developer's perspective and the other is a cross-referential study about the parallel programming models. The former can be done by using a similar study like this by a different programmer and comparing this study with the new study. The latter can be done by including multiple data points in the same programming model or by using a different set of parallel programming models for the study.

REFERENCES

[1] Alan H. Karp and Horace P. Flatt. “Measuring Parallel Processor Performance”, Communications of the ACM, vol. 33, Issue 5, pp. 539-543, ACM, New York, NY, USA, may, 1990.

[2] A. L.Wolf and D. S. Rosenblum.” A study in software process data capture and analysis”, In Proceedings of the Second International Conference on Software Process, vol. 24 , Issue 8, pp. 115–124, February 1993.

[3] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, ”Introduction to Parallel Computing”, Wesley Publishers Date: January 16, 2003.

[4] Andrew Funk, John R. Gilbert, David Mizell and Viral Shah. "Modeling Programmer Work flows with Timed Markov Models," CTWatch Quarterly, vol. 2, Number 4B, November, 2006.

[5] Angela C. Sodan. ” Message-Passing and Shared-Data Programming Models Wish vs. Reality”, Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications, pp.131-139, May 15-18, 2005.

[6] Barbara Chapman, Gabrielle Jost. “Using OpenMP: Portable shared memory parallel programming”, The MIT Press, Cambridge, Massachusetts, England, pp. 1-378, Oct-2007.

[7] Barry Wilkinson and Michael Allen. “Parallel programming", Pearson prentice-hall, 2005.

[8] Bil Lewis and Daniel J. Berg. “Pthreads primer: A Guide to multi-threaded programming”, SunSoft Press, USA, 1996.

[9] Charles E. Leirson. “How to Survive the Multi-core Software Revolution (at least Survive the Hype)”,In Cilk Arts, pp. 1-38, 2008.

[10] Cory W. Quammen,. “Introduction to Programming Shared-Memory and Distributed-Memory Parallel Computers”, ACM Crossroads, vol. 12, No. 4, pp. 3–9, 2005.

[11] C. Wohlin, P. Runeson, O.M. Host, M. C. Regnell and A. Wesslen. “Experimentation in Software Engineering: An Introduction”, Kulwer Academic Publishers, Boston/Dordrecht/London, 2000.

[12] Dewayne E. Perry, Nancy A. Staudenmayerx, and Lawrence G. Votta Jr., “Understanding and Improving Time Usage in Software”, Technical report, vol. 4, pp. 1-25, AT&T Bell Laboratories, USA, 1996.

[13] D. Helmbold, and C. McDowell. “Modeling speedup(n) greater than n”, Proc. 1989 International Conference on Parallel Processing, volume 1 , Issue 2, pp. 219-225, 1989.

[14] James Reinders. “Intel Threading Building Blocks”, O’reilly, November, 2007.

[15] Jeff Carver, Sima Asgari, Vic Basili, Lorin Hochstein, Jeff Hollingsworth, Forrest Shull, and Marv Zelkowitz. "Studying Code Development for High Performance Computing: The HPCS Program", Workshop on Software Engineering for High Performance Computing Applications (SE-HPCS ’04) , 2004.

[16] John Grosh. “Supercomputing and Systems engineering”, in proceedings of the second international workshop on Software engineering for high performance computing system applications, pp. 7-7, 2005.

[17] John L. Gustafson. “Fixed Time, Tiered Memory, and Superliner Speedup”, Distributed Memory Computing Conference proceedings of the fifth Distributed memory conference DMCC5, vol. 2, pp. 1255-1260, October, 1990.

[18] John L. Gustafson. “Re-evaluating Amdahl’s law”, Communications of the ACM, vol. 31, Issue 5, pp. 1-3, May, 1988.

[19] J.W Creswell.” Research Design: Qualitative, Quantitative and Mixed Approaches”, Sage Publications Ltd., 2002.

[20] Karl-Filip Fax´en, Christer Bengtsson, Mats Brorsson, Håkan Grahn, Erik agersten, Bengt Jonsson, Christoph Kessler, Björn Lisper, Per Stenström, and Bertil Svensson.”Multi-core computing—the state of the art”, Technical report, SICS, December, 2006.

[21] Kriste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. “The Landscape of Parallel Computing Research: A View from Berkeley”, Technical report, EECS Department, University of California at Berkeley, UCB/EECS-2006-183, December, 2006.

[22] Lorin Hochstein, Forrest Shull, and Lynn B. Reid. “The role of MPI in development time: a case study”, Proceedings of the 2008 ACM/IEEE conference on Supercomputing, pp. 1-10, Austin, Texas, USA, 2008.

[23] Lorin Hochstein, and Victor R. Basili, "The ASC-Alliance Projects: A Case Study of Large-Scale Parallel Scientific Code Development”, Computer, vol. 41, no. 3, pp. 50-58, Mar. 2008, doi:10.1109/MC.2008.101.

[24] Lorin Hochstein, Taiga Nakamura, Victor R. Basili, Sima Asgari, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, Forrest Shull, Jeffrey Carver, Martin Voelp, Nico Zazworka, and Philip Johnson. “Experiments to Understand HPC Time to Development”, CTWatch Quarterly, vol. 2, No. 4A, November 2006.

[25] Michael A. Bauer. “High Performance computing: The software Challenges”, In Proceedings of the 2007 international workshop on Parallel Symbolic Computation, pp. 11-12, 2007.

[26] M. L. Barton and G.R. Withers. “Computing Performance as a function of the speed, quantity, and cost of the processors”, Conference on high Performance Networking and Computing, Proceedings of the 1989 ACM / IEEE conference on Supercomputing, pp. 759-764, ACM New York, NY, USA, 1989.

[27] Ralph Duncan. "A Survey of Parallel Computer Architectures," Computer, vol. 23, issue 2, pp. 5-16, Feb. 1990, doi:10.1109/2.44900.

[28] Satraj Sahni, and Venkat Thanvantri, ”Parallel Computing: Performance metrics and models”, Technical report 96-008, pp. 1-42, Univ. of Florida, Dept. Computer & Information Sci. & Engr., 1996.

[29] S. Krishna Prasad. “Uses and Abuses of Amdahl’s law”, Journal of computing sciences in colleges, Volume 17, Issue 2, pp. 288-293, USA, December , 2001.

[30] Victor Basili, Sima Asgari, Jeff Carver, Lorin Hochstein, Jeffrey K. Hollingsworth, Forrest Shull and Marvin V. Zelkowitz.” A Pilot Study to Evaluate Development Effort for High Performance Computing”, Technical report, Report no A310844, University of Maryland, USA, 2006.

[31] Vipin Kumar and Anshul Gupta. ”Analyzing the scalability of parallel algorithms and architectures”, Journal of parallel and Distributed computing, Volume 22, issue 3, pp. 379-391, 1994.

[32] Xian-He Sun and Lionel M. Ni. “Another View on Parallel Speedup”, Proceedings of the 1990 ACM/IEEE conference on Supercomputing, pp. 1-10, USA, 1990.

[33] Xian-He Sun. “Scalable problems and memory bound speedup”, Journal of Parallel and Distributed Computing, Volume 19, Issue 1,pp. 27-37, Academic Press, Inc. Orlando, Fl, USA,1993.

[34] “Cilk++ Programmers Guide”, Reference manual, Cilk Arts, pp. 1-132.

[35] Internet: http://en.wikipedia.org/wiki/Parallel_computing, last visited: 4th Nov 2009.

[36] Forrest Shull, Victor R. Basili, Jeff Carver, Lorin Hochstein. "Empirical study design in the area of High-Performance Computing (HPC)". 4th International Symposium on Empirical Software Engineering (ISESE '05). November 2005.