Summer 8-2-2016

Definition of a Method for the Formulation of Problems to be Solved with High Performance Computing

Ramya Peruri

Follow this and additional works at: http://digitalcommons.kennesaw.edu/cs_etd

Part of the Computer Sciences Commons, Other Mathematics Commons, and the Science and Mathematics Education Commons

Recommended Citation

Definition of a Method for the Formulation of Problems to be Solved with High Performance Computing.

Master of Science in Computer Science

Thesis

By

Ramya Peruri
MSCS Student
Department of Computer Science
College of Computing and Software Engineering
Kennesaw State University, USA

Submitted in partial fulfillment of the Requirements for the degree of Master of Science in Computer Science

July 2016
Definition of a Method for the Formulation of Problems to be Solved with High Performance Computing.

This thesis is approved for recommendation to the Graduate Council.
DEDICATION

This thesis is dedicated to my family and my Husband.

I thank everyone who supported me throughout my journey.
ACKNOWLEDGEMENTS

I would like to thank my Advisor Dr. Jose Garrido for his support, encouragement and motivation through this entire process.

I am very thankful for this experience.

I would also like to thank my thesis committee members, Dr. Ken Hoganson, and Dr. Ying Xie for their insightful comments and valuable suggestions.

This research paper is made possible through the help and support from Everyone, including my professors, parents, my husband, family and friends.
LIST OF TABLES

Table 1: Position statistics per round for problem size 5 ..................................................48
Table 2: Position statistics per round for problem size 6 ..................................................49
Table 3: Position statistics per round for problem size 7 ..................................................50
Table 4: Execution times for shared-memory size 5 .........................................................54
Table 5: Execution times for shared-memory size 6 .........................................................55
Table 6: Execution times for shared-memory size 7 .........................................................55
Table 7: Execution times for Message Passing size 5 ......................................................57
Table 8: Execution times for Message Passing size 6 ......................................................57
Table 9: Execution times for Message Passing size 7 ......................................................57
Table 10: Performance in Multiprocessor 1 ..................................................................59
Table 11: Performance in Multiprocessor 2 ..................................................................59
Table 12: Speedups for SM-size 6 ..............................................................................59
Table 13: Speedups for MP-size 6 ..............................................................................59
Table 14: Speedups for SM-size 7 ..............................................................................59
Table 15: Speedups for MP-size 7 ..............................................................................59
Table 16: Performance with Load Imbalance .................................................................67
Table 17: Performance without Load Imbalance ..............................................................68
LIST OF FIGURES

Figure 1: Parallel Computing.........................................................................................9
Figure 2: Inter-process communication...........................................................................21
Figure 3: An illustration of a shared memory system of three processors......................23
Figure 4: Message passing model..................................................................................33
Figure 5: Shared memory machine with one memory bank............................................41
Figure 6: Memory with “dance hall” organisation........................................................41
Figure 7: Machine with processors tightly coupled with memory banks.......................41
Figure 8: Steps to show Peg game..................................................................................44
Figure 9: Axes of symmetry.........................................................................................45
Figure 10: Four data structures used in PBFS.................................................................46
Figure 11: Shared Memory size -5..................................................................................55
Figure 12: Shared Memory size -6..................................................................................56
Figure 13: Shared Memory size -7..................................................................................56
Figure 14: Message passing size -5................................................................................58
Figure 15: Message passing size -6................................................................................58
Figure 16: Message passing size -7................................................................................58
Figure 17: Comparison of SM and MP-size 5.................................................................60
Figure 18: Comparison of SM and MP-size 6.................................................................60
Figure 19: Comparison of SM and MP-size 7.................................................................60
Figure 20: Graphical representation for performance with Load Imbalance...............68
Figure 21: Graphical representation for performance with No Load Imbalance............68
ABSTRACT

Computational power made available by current technology has been continuously increasing, however today’s problems are larger and more complex and demand even more computational power. Interest in computational problems has also been increasing and is an important research area in computer science. These complex problems are solved with computational models that use an underlying mathematical model and are solved using computer resources, simulation, and are run with High Performance Computing. For such computations, parallel computing has been employed to achieve high performance. This thesis identifies families of problems that can best be solved using modelling and implementation techniques of parallel computing such as message passing and shared memory. Few case studies are considered to show when the shared memory model is suitable and when the message passing model would be suitable. The models of parallel computing are implemented and evaluated using some algorithms and simulations. This thesis mainly focuses on showing the more suitable model of computing for the various scenarios in attaining High Performance Computing.
# TABLE OF CONTENTS

Chapter 1: Motivation, Problem Statement and Contribution ................................................................. 9  
1.1 Background ........................................................................................................................................ 9  
1.2 Motivation ........................................................................................................................................ 11  
1.3 Problem Statement ............................................................................................................................ 14  
1.4 Research Method ............................................................................................................................. 14  
1.5 Contribution ..................................................................................................................................... 15  

Chapter 2: Literature Review .................................................................................................................... 17  
2.1 Overview .......................................................................................................................................... 17  
2.2 Shared memory and Message passing Techniques .............................................................................. 19  

Chapter 3: Shared Memory ....................................................................................................................... 25  
3.1 Overview .......................................................................................................................................... 25  
3.2 Shared memory model ....................................................................................................................... 28  
3.3 Family of Problems solved by shared memory.................................................................................... 30  

Chapter 4: Message Passing .................................................................................................................... 33  
4.1 Overview .......................................................................................................................................... 33  
4.2 MPI standards used for high performance and portability ............................................................... 36  
4.3 Support provided by MPI .................................................................................................................. 37  
4.4 Message passing programming model .............................................................................................. 38  
4.5 Family of Problems solved by Message passing .............................................................................. 39  
4.6 Shared Memory and Message Passing ............................................................................................. 40  

Chapter 5: Experimental Studies and Evaluation ..................................................................................... 43  
A. Case Study 1- peg puzzle search problem ........................................................................................... 43  
5.1 Overview .......................................................................................................................................... 43  
5.2 Implementation with PBFS .............................................................................................................. 46  
5.3 Implementation Details ..................................................................................................................... 50  
5.4 Evaluation ...................................................................................................................................... 54  
5.5 Analysis .......................................................................................................................................... 60  
5.6 Conclusion ..................................................................................................................................... 61  
5.7 Summary ....................................................................................................................................... 62  
B. Case study 2- Solving Matrix Inversion problem ................................................................................ 64  
5.8 Hardware Specifications ................................................................................................................... 65  
5.9 Evaluation ...................................................................................................................................... 67  
5.10 Analysis ....................................................................................................................................... 69  

Chapter 6: Conclusions and Future works ............................................................................................. 70  
6.1 Conclusion ...................................................................................................................................... 70  
6.2 Future Work ................................................................................................................................... 71  

Appendix A: Reference Code ................................................................................................................ 72  
References ..................................................................................................................................................... 78
CHAPTER 1

Motivation, Problem Statement and Contribution

1.1 Background

Parallel computing [1] is considered as a computation that performs various number of calculations at the same time. The principle on which it is operated is that, problems of large size can be further divided into the smaller ones on which they are solved at the same time. The concept of parallelism has been employed in the High Performance Computing for many years but interest on it has recently grown due to the lack of frequency scaling. In recent years, as there was a concern over consumption of power, parallel computing is considered as one of the most dominant method in terms of in architecture of computers that takes the form of multi core processors [1, 2].

![Figure 1: Parallel Computing](image)

Parallel computing is applied with the concept of concurrency computing and these two terms are frequently used together and also integrated, although both the concepts are distinct. Parallelism is possible without the presence of concurrency and concurrency can be present without the parallelism. In parallel computing, as shown in figure 1, a task is broken into several tasks that are similar and then processed independently and results are then added up on completion of processing [1].

Parallel computers are generally classified as per the level at which parallelism is supported by the hardware. The computers with multicore and multi-processor have many number of processing elements working together in the single machine whereas other grids and clusters make use of the multiple computers for working on the same task. Often, parallel computers
having specialized architectures are used on side along with the traditional processors for specific tasks to be accelerated. Among the subtasks, communication and synchronization [2] are considered to be the typical obstacles for attaining of the parallel program performance.

In the present world, many complicated and events are not related to each other are taking place at the same time but still within a temporal sequence. When compared to the serial computing, parallel computing is more suitable for the purposes of modelling, simulating and to understand the complex phenomena of the real world. They can be well used for solving of the larger problems of more complexity. Concurrency is provided by the parallel form of computers.

Historically, parallel computing is considered to be “the high end of computing” and is also used for the modelling and simulation of hard problems in several areas of science and engineering. It is the process of sending and receiving messages or transferring of the data from part or place (generally notated as sender) to another part or place (generally notated as receiver). [3]

Parallel computing is considered to be the discipline of computer science that deals with the architecture of system and software methods that are related to the execution of applications concurrently. The interest of research and application of parallelism and concurrency has been active for several decades with main focus of high performance computing. But now, it is been emerging as the paradigm of computing as the industry of semiconductor started shifting to multi core processors.

In terms of highest level of abstraction, the two terms “message passing” and “shared memory” are referred as the models of communication that are applied by programmers of systems or applications. Though many semantics of both the models are explored, they have deficiencies of one kind or the other.

In the evolution of parallel computing [1], the world of software has been very active. It has been harder for the parallel programs to be written than the sequential programs. As the program here is divided into many number of subtasks, it becomes harder to write due to synchronization and communication that needs to take place between those subtasks. Several number of interfaces of application programming are converged to a single standard called MPI for the purpose of MPPs and clusters. For the multiprocessor of shared memory, two standards
of pthreads [4] and OpenMP [5] have been emerged. Along with these, several other competing models of parallel programming and languages have also emerged. These models and languages may provide a better solution to the problems of parallel programming than compared to the above standards [1, 2, 3].

As processors of multicore bring the parallel computing to the customers of mainstream, the main challenge of computing at present is to change the industry of software to parallel programming. The history of parallel software indicates that there cannot be only single technology that could make parallel software universal. For making it ubiquitous, it may be required wide collaborations over the industries so as to create the technology family that work together to get the power of parallel computing for the upcoming applications. The entire industry gets affected for the changes that are needed from consumers to manufactures of hardware as well as from the complete development of software infrastructure to the developers of applications who depend on it.

The capabilities such as graphics of photorealistic, perception of computation and machine learning mainly depends on the parallel algorithms. Making these capabilities enable, new generation of experiences get advanced by extending the scope of what people can accomplish in their digital lifestyles and work place. We can say that the future of the parallel computing is bright, but as the new opportunities emerge, there will come new challenges also.

1.2 Motivation:
Since early 1980’s, one of the main research areas in computational problems is on finding and utilizing the parallelism to solve these complex problems. Because most of the problems are large and complex, it is difficult to solve them on a single processor. The main motivation behind this quest to achieve high computational speed. An indication of the importance of all this has been the publication of “Grand Challenge Problems” [6] which were policy terms of US that set as goals in late 1980’s for the funding of high performance computing and communications research as a response to a 10-year project. [6].

The role of parallelism in increasing the speeds of the computing has been identified for many number of decades. Its role in providing multiplicity of the data paths and extended access for the elements to be stored has been very important in the applications of commercial field. The
performance with scalability and lesser cost of parallel platforms has been effective in various number of applications.[6]

The development of computer processors has changed dramatically in the past decade. In the past, processors were developed with only one core that could execute a single stream of instructions. Multi-tasking in these single-core processors was achieved using concurrent processing, where context-switching [7] between different running streams of instructions, or threads, was used. The technology of single-core processors was advanced over the years by decreasing the transistors' size, and increasing the clock frequency.

In early 2000s, a ceiling in frequency scaling was reached because of the heat dissipation barrier that prevented processors from having faster clocks. Multicore processors started to appear where a single processor has two or more processing cores that can run independent threads in true parallel fashion. Today, most CPUs in desktop computers, office workstations, servers, and even smart phones have multicore processors. Desktop computers nowadays have processors with up to 6 cores, while larger computer systems, such as servers, may have processors with up to 15 cores.

In addition to the use of multicore processors, multiprocessor computers have been used for a long time. Most servers today have two processor sockets, and each has a multicore processor. A computer cluster is a multi-machine system where a number of similar but independent computers are interconnected using a high-speed network to act as one powerful system. These type of computers are commonly used in data centers and research facilities.

Today's computer systems are multiprocessor systems. Utilizing such systems require writing software that can run efficiently in parallel over multiple processors and machines. While parallel programming has been used for a long time, the recent development of multiprocessor technology has resulted in a growing interest in parallel computing.

Shared memory is an efficient means for transferring the data between the processors. It can used as one or more memory blocks that may be simultaneously accessed by several processes so as to enable communication and thus avoid redundant copies. Message passing requires synchronization and communication among the set of processes through exchange of data by sending and receiving messages.
Shared memory and message passing approaches each have advantages and disadvantages over each other. When transferring a large amount of data, message passing may exhibit higher performance than shared memory and it is even more efficient or convenient when communication patterns are well known. Parallelism finds applications in very diverse application domains for different motivating reasons. These range from improved application performance to cost considerations.

Simulations and modelling [8] of science drive the need for having greater power of computing. Processors of single core satisfying the required resource cannot be made for the needed simulations. It would be difficult in terms of cost as well as with limitations of power and heat for making the processors that has clock with faster speed. For this, parallel computing can be taken as the solution which involves dividing up the work among the numerous linked systems. Both Parallel computing and HPC are intimately related to each other. Higher performance needs more number of processor cores.

By understanding various models of parallel programming, it allows to also understand the usage of resources of HPC more effectively. It is also useful to better understand and critique the work that makes use of HPC in our area of research. By knowing the different types of hardware of HPC, we can understand why certain things are good on one resource than the other. It allows us for choosing the suitable resource for our application. It also allows us to understand the ways for parallelising of the serial application. Appreciation for the parts will be given which are considered to be important for performance.

Other research [10] efforts have been concentrating on architecture and other hardware features and however, little work has been done on understanding the scope and extent of solving problems by making use of these techniques. There has been even research on integrating of both the systems of message passing and shared memory. The present society is completely leading the way to message passing leaving the shared memory apart. But, shared memory also possess some advantages and it could also have some problems get solved. So far, there are many researches done on communication models but no research is made on identifying the problems that could be solved with the shared memory along with the model of message passing. This could be solved by doing research on both the shared memory and message passing benefits in possible scenarios. So this resulted in the motivation of our thesis. Part of this research is to identify which set of problems can be better solved with each of these two
techniques. Considering a scenario, we would like to evolve which memory model will be best suitable in that particular case. With that, we would evaluate the method for high performance computing.

1.3 Problem Statement
Complex computational problems can be solved using the concepts of parallelism in high performance computing. Both implementation techniques – shared memory and message passing [11] of parallel computing are useful in implementing parallel computing in solving complex problems. Part of this work will evaluate and define which family of problems are well suited for which type of techniques in parallel computing. A more general aspect of the goals of this research is to implement a solution to solve a set of problems using parallel computing.

The research question is: “How can we identify a “family of problems” that are better adapted for the Shared-memory paradigm of parallel computing and the ones that are more adapted for Message-passing paradigm of parallel computing”. In other words, which subset of problems are better modelled with shared memory and which subset of problems are better modelled with message passing? We would also like to compare a case study of a problem using shared memory and another case study using message passing at the specification and implementation level. We then evaluate one performance metric (the speed up) of each communication model of parallel computing with the help of simulations and algorithms thereby giving a precise model for attaining high performance through each of the memory model of shared memory and message passing.

1.4 Research Method
The Scientific method [12] is a body of techniques for investigating phenomena and integrating previous knowledge. It is a method of inquiry commonly based on empirical or measurable evidence subject to specific reasoning principles. However, in this project we are going to apply a more pragmatic approach to the research.

- Revisit the idea of why some problems need to be solved using parallel computing.
- Collect data about case studies from the current literature that apply parallel computing.

The high level approach here is to find out from the design description what model is used.
Then in more detail, find out algorithmic description and hardware description in the design documentation for every case study.

- Classify these problems as to which approach to apply message passing or shared memory.
- Analyze this classification of problems and start identifying which class is better adaptable to each parallel computing approach. { Message Passing and Shared Memory}
- Select two or more case studies and propose the best parallel computing approach.
- Validate the previous steps by implementing the algorithms in a program and discuss results.

1.5 Contribution

The objective of this thesis is to conduct an in-depth comparison of various types of communication models of parallel computing. The work is to evaluate which subset of problems are better for solving by shared memory and which subset of problems are better for solving by message passing. A case study is considered and checked for which type of memory is suitable in that particular case. Later, it is evaluated by considering programs and performance is defined in each case of shared memory and message passing. On evaluation, we can get a defined method for the formulation of high performance computing.

The work addresses the stated problem statement by performing the following tasks:

Conduct Literature Search

- Literature search is conducted on the parallel computing mechanisms.
- In depth analysis of search is made on the communication models of shared memory and message passing methods of parallelism.
- Literature search on high performance computing is performed.

Reviewing the problems solved

- Shared memory model of parallel computing is reviewed.
- All the problems that are dealt with shared memory are analyzed and its advantages are discussed.
- Reviewing of message passing model of parallelism is conducted.
- All the problems that are dealt with message passing memory are analyzed and its advantages are discussed.

Analysing both the communication models of parallel computing
• On reviewing, analysis is made on both the communication models by comparing and differentiating.

Experimenting through Case studies
• Case studies are considered and solved to check which memory model of parallel computing would be best suitable in that particular scenario.

Evaluation and Formulation
• On implementation of the case studies, they are evaluated with the help of some programs to check the performances.
• On evaluating, they are represented using the representation graphs evolved with the help of simulations.
• Method for the formulation of high performance computing is derived.

Dissemination of Results
• Disseminate the results for ease of access and understanding.
• Prepare and submit one or more papers for publication at related venues
CHAPTER 2

Literature Review

2.1 Overview

Early research of multiprocessors was divided into two separate approaches based on primitives of hardware communication those adapting shared memory and those adapting message passing [13]. Supporters of shared memory pointed to the fact that a programming model of shared address space is easy and its location-independent communication semantics reduces burden in locating shared data explicitly and direct towards inter process communication. On the other hand, supporters of message passing stated that hardware of shared memory was difficult and not scalable to build whereas implementation of message passing is much simpler and a minimum communication primitive is provided. In this early state of affairs, researchers often ignored the fact that each approach of parallelism has advantages and disadvantages.

At another level, research has emerged on integration of message passing and shared memory models of communication among parallel processors [14, 15]. Further, such a system of integration was implemented efficiently by integrating the hardware of primitive message passing and shared memory with a run time software.

Further, shared memory is amenable for the dynamic optimization [16] with the help of an operating system. Mainly, in techniques like data migration, caching and data replication may be transparently employed for the enhancement of locality and thus latency of access can be reduced. Though shared memory communication might be more efficient in some cases, it could give a negative effect on operations of synchronization. Message passing provides with interrupt-with-data which is necessary for several operating systems in which patterns of communications can be known in advance. The presence of higher end points costs in this when compared to shared memory.

Computer architecture vendors already made their own decisions on where to include between shared memory and message passing and have implemented their decisions in 'clusters on a chip' architectures [17]. These are multi-tiered architectures where the communication/sharing
mechanism that designers have chosen depends on the performance of current technologies. So the break point may change as technology evolves. The trend has been to implement more with message-passing and less with shared-memory, and building large virtual memories.

The parallelizability of algorithms depends heavily on the data-dependency between processes. When a process requires data from other processes during runtime, inter-process communication and synchronization must take place, which are considered an overhead and therefore have an impact on the parallel algorithm performance. Algorithms that require minimal or no communication between processes are highly parallelizable, as each sub task is run independently of others. On the other hand, algorithms that have a lot of inter-process data dependency are usually more difficult to parallelize.

In serial programs, the two main metrics for measuring performance are time and memory usage. An algorithm's cost can be determined by the amount of time it takes to run, and the amount of memory it requires. Both of these metrics are usually represented as a function of the input size. In parallel programs, communication is considered as a third cost metric, since it is an overhead that has an impact on the overall performance.

Most numerical applications, such as simulators and solvers, have some form of data-dependency, and therefore require inter-process communication at a certain point. For example, a parallel simulator usually requires inter-process communication to exchange data that is needed to perform the computations in the next time-step. This means that there is a synchronization point at the end of each time-step. Fortunately, most of these data dependencies are spatially-local. That is, only a small subset of data, such as the outer data point of spatially discretized sub-domains, needs to be communicated. Hence, the communication overhead, while still present, is relatively small.

Hypercube of the Intel iPSC [Intel Personal Supercomputer] is one of the first commercial highly-parallel computers. For some time, it has been known that the structure of hypercube has many features which make it a useful architecture for parallel computation. Intel 80386 processor is present on each node of the cube. It has 4 megabytes of memory, and hardware for connection to use the backplane of message passing. The hardware of iPSC/2 message passing is encapsulated in a set of machine dependent classes which are used by the machine independent message passing system to send and receive hypercube messages. On the other
hand, the Transputer [20] which was produced by INMOS—a Semiconductor company designed a physical implementation of parallel programming method with features of integrated memory and links of serial communication that are intended for Parallel computing. More research on parallel computer systems continued with machines built from transputers. Message passing system was used on the systems of Transputer. This conveys the importance of parallel processing techniques in various fields leading to more research using those techniques.

From the above discussion, one obvious conclusion is that the communication models of message passing and shared memory each have domains of application for which they are suited. So defining subset of problems that best suits each of the parallel processing techniques could resolve many problems. Up to now from my current research no similar research project has been found but I would continue to search. This motivation lead us to propose research for thesis in finding which subset of problems can best be solved by shared memory and which subset of problems can best be solved using message passing and thereby to define a method for high performance computing by evaluating and implementing case studies and then by analysing the results.

In the further sections, shared memory and message passing are described in detail and thereby analysing the problems solved by both the communication models of parallel computing is done.

2.2 Shared memory and Message passing Techniques

Workload Balancing
Developing a parallel algorithm usually requires breaking a large problem into smaller, but similar sub-problems. Each sub-problem uses a small subset of the data, known as the working set, to perform certain computation. Usually, these computations are done in parallel over multiple processors, where each processor is assigned one or more of these sub-problems.

To achieve the best performance, the amount of computation done by each processor should be similar. This results in better workload-balancing [21], and therefore higher performance speedup. Unbalanced workload, where one or more processors are spending more time performing computations than other processors, impacts the performance of the application, as the processors that do less work will become idle at each synchronization point, waiting for the other processors to finish their computation, before they can start the next task. This results in
poor resource utilization, and lower speedup.

Workload balancing does not depend only on the size of the working set data, but also on the type of processing and computation that needs to be performed on that data. In many cases, it is easy to split the input data into smaller working sets of the same size. However, the amount of computation that each processor will perform on these working sets is not always possible to control or even predict. Certain algorithms perform a fixed amount of computation on the input data, which results in high workload balancing. On the other hand, the amount of processing other algorithms perform depends completely on the input data, and therefore, it is highly likely to have unbalanced workload among processors. This presents a challenge when designing parallel applications as the performance could be significantly impacted by this imbalance.

**Data Dependency and Inter-process Communication**

The parallelizability of algorithms depends heavily on the data-dependency [22] between processes. When a process requires data from other processes during runtime, inter-process communication and synchronization must take place, which are considered an overhead and therefore have an impact on the parallel algorithm performance. Algorithms that require minimal or no communication between processes are highly parallelizable, as each sub task is run independently of others. On the other hand, algorithms that have a lot of inter-process data dependency are usually more difficult to parallelize.

In serial programs, the two main metrics for measuring performance are time and memory usage. An algorithm's cost can be determined by the amount of time it takes to run, and the amount of memory it requires. Both of these metrics are usually represented as a function of the input size.

In parallel programs, communication is considered as a third cost metric, since it is an overhead that has an impact on the overall performance. Inter-process communication is considered as a set of programming interfaces which allows a programmer in coordinating activities that are among various processes of program which could run concurrently in an operating system as shown in Figure 2.
Most numerical applications, such as simulators and solvers, have some form of data-dependency, and therefore require inter-process communication at a certain point. For example, a parallel simulator usually requires inter-process communication to exchange data that is needed to perform the computations in the next time-step. This means that there is a synchronization point at the end of each time-step. Fortunately, most of these data dependencies are spatially-local. That is, only a small subset of data, such as the outer data point of spatially discretized sub-domains, needs to be communicated. Hence, the communication overhead, while still present, is relatively small.

**Parallel Programming Libraries**

There are several libraries and frameworks for building multithreading and multiprocessing parallel applications. All modern operating systems provide Application-Programming Interface (API) [24] for creating and managing threads and processes additionally, a number of third-party libraries and standards are designed to provide more functionality and hide some of the system complexities.

**Operating System API**

The most direct way of writing parallel programs is to use the operating system's API for multithreading or multiprocessing. Operating systems provide means to create, control, synchronize and terminate processes and threads. In Unix-based operating systems, the POSIX [25] standard interface for threads and processes allows the programmer to make system calls to create processes and threads that are executed immediately in parallel to the main process or thread. It also provides a set of synchronization tools, such as mutexes, locks, and semaphores. The API for multithreading in POSIX is called pthreads. Windows operating system also
provides similar set of APIs and system calls for managing threads and processes. While the system calls are different under Windows system, the functions these APIs provides are essentially the same.

There are many third-party libraries that are written to hide the system-dependent APIs, and provide a standard way of creating and managing threads and libraries. This allows programmers to write portable code that can run on different systems. Boost library, for example, provides portable C++ multithreading functions that can be used to create and manage threads. Similarly, other multithreading libraries provide more advanced features, such as the use of worker-thread pools and job queues.

**Parallel Programming Models**

There are several ways to write parallel programs. Today's processors are capable of performing several levels of parallelism. Some of them, such as data- and instruction-level parallelism, are built-in into the CPU architecture, and are performed transparently without the need for any specially programming. Task-level parallelism, on the other hand, requires actual parallel programming, where the developer has to write multiprocessor or multithread application, and handle the needed data distribution, communication, and synchronization.

**Shared memory.**

In this model, programmers view their programs as a collection of processes accessing local variables and a central pool of shared variables. Each process accesses the shared data by asynchronously reading from or writing to shared variables. As more than one process may access the same shared variables at the same time, mechanisms to resolve mutual exclusion problems need to be provided, such as locks or semaphores. This model is adequate in multiprocessor computers with uniform access to main memory. An example of technology to support this model is multithreading, where light versions of processes called threads run simultaneously while having shared and private memory regions. Multithreading implementations (e.g. POSIX threads) are currently available in most modern operating systems. In a shared memory architecture, there is one huge bank of memory that is shared among all the processors as shown in Figure 3.
Thread-Level Parallelism

Parallel programming usually refers to task-level parallelism, where the programmer has to write code that is executed concurrently and in parallel on one or more processors. A process is the unit task that most operating systems deal with as an executing program. Within a process, one or more flows of instructions, or threads, are executed. Multiple processes and threads can be executed concurrently on a single processor.

Thread-level parallelism [27] is the use of multiple threads of a single process to perform a certain task. Threads within a process share the same address space that belong to that process, and therefore can directly read and write data to that shared memory. Most processors today can execute multiple threads at the same time by using multiple cores. Additionally, a single core can execute two threads at the same time using thread-level speculation, also known as hyper-threading. In parallel applications that use multithreading, synchronization is crucial to avoid race-conditions, where two or more threads try to write to the same memory address, resulting in inconsistent data.

OpenMP

OpenMP [28] is a programming interface for shared-memory systems. Unlike operating system APIs, OpenMP provides a simple way to create threads through the use of programming directives in the code. OpenMP simplifies converting serial programs to parallel ones, as only few changes need to be made. Parallelization using OpenMP is done by marking sections of the code using C++ preprocessor directives. These directives are also used to specify the shared and private variables in the code.

At run time, OpenMP creates a number of threads to perform these parallel tasks. OpenMP
provides a simple and flexibly way to create multithreaded applications, and hides the complexity of dealing with the operating system low-level APIs. However, there are a number of cons for using OpenMP, such as its limited scalability and the lack of error handling.

**Message-Passing Interface (MPI)**

Message-Passing Interface (MPI) [29] is a popular standard for parallel programming. It has become the de facto standard for writing multiprocessing applications. MPI, as its name suggest, provides means for inter-process communication through message-passing, which are done within a single system through memory, or between multiple systems through the network. This flexibility makes MPI suitable for distributed-memory systems.

A parallel application written using MPI can run on a single or multiple systems. MPI provides means for communicating data between processes, as well as tools for synchronization and process-management. Each processes is assigned an MPI process number, known as rank that identifies it. Communication between processes could be performed point-to-point, where one processes sends data to another process, or collectively, where all the processes exchange data at the same time such that each process has the same data. Processes could also be grouped into subsets, called worlds, such that processes within one world could perform collective communications together.

MPI also hides the underlying communication and synchronization layers. The medium of communication, whether through the main memory or the interconnecting network, is automatically picked by MPI based on where the communicating processes reside. This provides better performance when running multiple processes in the same system.
CHAPTER 3

Shared Memory

3.1 Overview

In the field of computer science, Shared memory [30] is considered as a memory that can be accessed by multiple number of programs so as to enable communication between them as well as for the redundant copies to be avoided. For passing of data between the programs, shared memory can be considered to be so efficient. Based on the need, programs can run on a single processor as well as on separate multi processors.

A single program that uses memory for communication inside it, say, among its multiple threads is also referred to as shared memory. In terms of hardware, shared memory is considered as the large block of random access memory (RAM) which can be used by various different central processing units in a computer system of multiprocessors. The systems of shared memory may use uniform memory access, non-uniform access, architecture of cache-only memory.

A system of shared memory is easily programmable as a single view of data is shared by all the processors and as the memory accesses to the same location, the communication between the processors can become faster. The systems of Shared memory gets an issue as fast accessing of memory is needed by many of the processors which leads to two complications.

Degradation of access time: Contention takes place when many of the processors try to access the same location of memory. Scaling is not so well in the systems of shared memory.

Data coherence is lacked: When the information in the cache that is used by the processors gets updated, it should also be reflected to the other processors otherwise they will be working with the data that is incoherent. Such protocols of cache coherence, on working yields access of high performance for the multi processors. Sometimes, they can get overloaded and results as bottleneck for the performance.

Some other defects can be

Coarse granularity: when huge amount of data is to be shared, cost is incurred so as to transfer the entire data from one processor to the other. Block transfer in which single message is used for sending all the data is considered to be the most efficient mechanism. For this purpose,
more network and bandwidth is needed due to the fixed overhead that is associated with each
transaction of the shared memory.

Patterns of communication: caches of coherency on a multiprocessors perform well when data
is read-only or if it is accessible on a single processor multiple times. Writing frequently on
various processors leads to bad performance as they make caching less effective and causes
extra overhead.

Polling interface is one of the deficiency of shared memory model. Though it can have
advantage in making communication more effective, it also leads to a negative impact in case
of operations of synchronization. On this concern, many of the architects of the multiprocessor
have adopted the basic communication model of shared memory along with the additional
mechanisms of shared memory. Other defect of shared memory in case of real operation of
communication is that a complete round trip of network is required whereas communication of
data through one way is not possible.

In the architecture of heterogeneous systems, the memory management unit of the CPU and
GPU’s input-output memory management unit needed to share the same address space. In
terms of computer software, shared memory is considered as a method of inter-process
communication. IPC is defined as the process of data getting exchanged between the programs
that run at the same instance. An area is created in the RAM by one process from which other
processes could access. Memory is conserved by the shared memory by directing the accesses
for which would be copies of a data for a single instance by making use of mappings of virtual
memory. This is often used in the shared libraries.

As shared memory can be accessed by the both the processors just like the regular working
memory, this is considered as a very fast mode of communication. On the other side, scalability
is less in it. As an example, running of the communicating processes must be on the same
machine and care has to be taken for avoiding of the issues if the processes that are sharing
memory are working on the different CPUs and if the architecture is not cache coherent.

Shared memory provides support on Unix-like systems with a standardized API of POSIX
shared memory. On windows, the function CreateSharedMemory is used for the purpose of
creating memory. Some of the libraries of C++ enables a portable as well as object oriented
access for the functionality of shared memory. Besides C and C++, there is also support of
native programming languages for the shared memory. For example, API for the creation of shared memory is provided in PHP similar to that of the POSIX functions.

The main fundamental feature of the shared memory is that, all the communication is done internally by making use of stores and loads to a global address space. For the information to be communicated, operations of load and store are issued by the processor whereas the system underlying is responsible for taking the decision whether data is cached or not so as to locate the data in an event which is not cached. The second important feature of shared memory is that synchronization is different from that of the communication. In order to detect when the data has been produced and when it has been consumed, special mechanisms of synchronization have to be employed along with the operations of load and store.

Advantages of shared memory: In the research community of multi processors, it is important to support an address space of shared memory or communication model of shared memory. The primary reason for this could be that programmers are sheltered by the shared memory from communication details of interprocessor. The independency of location semantics of the shared memory allow programmers for focusing on the issues of parallelism and correctness and ignoring the issues of where the data is and how the data could be accessed. This allows quick construction of the algorithms that makes communication internally with the help of the data structures. That is, the communication model of shared memory provides one of the easiest method to extend the programming paradigm of the uniprocessor to the multiprocessors.

Techniques like caching, migration and replication of data can be used to improve the locality and thereby could reduce the access latency which is so powerful as it means that layout of data can be adjusted dynamically towards optimal even if the programmer avoids it. Of course with the help of any algorithm online, reordering dynamically are unlikely to achieve. This indicates that the nature of implicitness of the programming model of shared memory is both its strength as well as weakness. When the needed parallelism is identified by the compiler or the programmer, then rapid context switching can be made use so as to overlap the latency. All of these techniques give an aid to the communication model of shared memory in its implementation by maintaining the same semantics of the location-independent for the programmer.
Shared memory is compatible with hardware of SMP. At the time of execution, if the patterns of communication are typical and if they vary dynamically, shared memory provides the ease for programming. It has the ability for developing of applications making use of the familiar model of SMP giving attention for critical accesses of performance. Communication overhead is lower because of the communication taking place implicitly, and mapping of the memory in order to provide protection in hardware. It enables controlled caching of hardware so as to decrease the remote communication by caching the all shared as well as private data.

One main advantage of the connection network of shared memory is that it goes hand to hand with the bus connected parallelism in many of the networks of UNIX. Shared memory is very compatible with the operating systems of UNIX. For implementing in the software, shared memory is very easy making it an advantage. The basic advantage of shared memory model is that the programming for IPC becomes simple in the sense that we simply write and read to an address pointer available in the process address space. We need not use system calls like write and read. The updates to the kernel resident shared memory object is done by the kernel asynchronously. It saves lot of time compared to write and read because in write and read lot of switching should take place between user mode to kernel mode vice versa.

### 3.2 Shared memory model

The parallel systems that have the ability to support abstraction of shared memory are becoming accepted so widely in several areas of computing. For the purpose of writing such programs correctly and efficiently, systems need a formal specification of semantics of memory which is termed as a memory consistency model. The shared memory or address space of single abstraction provides many advantages over the model of message passing or the memory that is private by presenting a transition that is more natural from uniprocessors and by making easy the programming tasks that are difficult such as partitioning and distribution of load dynamically. For this reason, parallel systems that give a support for the shared memory are attaining wide acceptance in computing of both technical and commercial.

For passing of data between the programs, shared memory can be considered to be so efficient. Based on the need, programs can run on a single processor as well as on separate multi processors. A single program that uses memory for communication inside it, say, among its multiple threads is also referred to as shared memory. In terms of hardware, shared memory is considered as the large block of random access memory (RAM) which can be used by various
different central processing units in a computer system of multiprocessors. The systems of shared memory may use uniform memory access, non-uniform access, architecture of cache-only memory.

The connection network of shared memory goes hand in hand with the bus-connected parallelism of many networks of UNIX. It also gets moulded well for operating system of UNIX. The model of shared memory in certain applications can be useful for creating of the multiple processes that run on a single machine. In such case, the parallelism is not over the processors but instead the processes being time shared on the single machine of UNIX. Another advantage of the model of shared memory is that it is easy or the implementation in software.

The model of shared memory is not easily scaled up to many processors as the cost of a single large memory and the system of connection needed for it is avoided. For this purpose all the parallel machines that are built massively work based on the paradigm of distributed memory network. Consequently, machines of shared memory haven’t gained the very high computing speeds of supercomputers. Another remark for this model is that the need for separate provisions of synchronization of data at the time of programming that includes the necessity for locking and unlocking of the data.

For a server, it should be initiated before the client does. The following tasks have to be performed by the server.

- Ask for the shared memory with a key of memory and then it has to be memorized the ID of the returned shared memory. It can be performed by using the system call of shmget().
- This shared memory have to be attached for the address space of server by using the system call of shmat().
- If needed, shared memory have to be initialized.
- Something has to be done and should wait for the completion of the client.
- The shared memory have to be detached with the system call of shmdt().
- Shared memory have to be removed with the system call of shmct1()
- At the side of client, the procedure is almost the same.
- Ask for a shared memory with the same key of memory and the returned ID of the shared memory have to be memorized.
- This shared memory have to be attached to the address space of the client.
• Memory have to be made used.
• All the segments of shared memory are to be detached if necessary.
• Exit.

3.3 Family of Problems solved by shared memory
Matrix problems-
Several fundamental problems of matrix are solved using fast and highly scalable system of shared memory. Some of them include
• matrix multiplication
• Matrix chain product
• Computing the powers
• Computing of inverse problems
• The characteristic polynomial
• The determinant
• The rank
• The matrix of krylov
• Factorization of a matrices
• Solving of equations of linear systems.

Parallelisation of Matrix Multiplication problem
In this research [31], problem is solved using shared memory. Parallel implementation generally depends on the architecture of the shared memory. This can be done by making use of an API package OpenMP. The library itself does the task of reducing the overhead of the programmer. The primary advantage of this library is that it is easily used for the parallelizing of a program. Dynamic allocation of memory will be used in this so as to avoid failure to the memory during initialization of matrix for the purpose of testing as well as for efficiency of the code. Here each processor is assigned with a single task that is independent at a time. The general algorithm used here is
Algorithmic Steps: 1. set number of threads by using following command in Linux terminal. $ export OMP_NUM_THREADS = 4
• Current processor is set as Master. (theoretically processor-0/P0
A parallel region is created and the used variables are declared as private or shared based on requirement and scope. 

```
#pragma omp parallel shared(var_1, var_2, chunk)
private(thread_id, iterations)
```

Matrices should be initialized as A, B and C in Master in parallel fashion as given below. 

```
#pragma omp for schedule (static, chunk)
for (i=0; i<ROW; i++)
{
    for (j=0; j<Column; j++)
    {
    }
}
```

Matrices have to be multiplied in parallel.

The time in parallel algorithm was calculated using the function of `opm_get_wtime()`.

In this analysis, it was showed that shared memory systems performs better when there is less communication between the processors. Here the time complexity was found to be $O(m^3)$ in the best case. The results obtained in solving this problem showed that program has almost 90% when used 4 and 8 number of processors. When only two processors are used, the efficiency was up to 80 percent. It was concluded in this problem that shared memory could be efficiently used having more than eight number of cores [31].

**Problems solved with load imbalance**

The research [32] shows that the applications having load imbalance inherently are often executed more efficiently on using the model of shared memory rather than using the model of message passing. The results were shown that as the communication was cheap in the earlier generations of multiprocessors, it favoured the model of shared memory. As there is an increase in the cost of communication in terms of computation, there is more favour towards the model of message passing.

As the model of message passing allocates a space of address for each process, the operating system gets involved in the process management. In such model, processes are so expensive are therefore used so sparingly. When looked at the shared memory, as the threads in it share a single space of address, implementation of threads can be done in the user-space and the overhead of process management could be negligible.
For quantifying the relative importance of load imbalance and communication, shared memory and message passing implementations of transitive closure are been executed on four shared memory multiprocessors. For this purpose, 2 bus based cache coherent machines are used along with the two large scale distributed machines of shared memory.

In their results, it was shown that the shared memory implementations performs slightly better than the message passing on the processor of Butterfly. The machines they used have enormous bandwidth for the purpose of communication and slow processors for computation. It is said that as the results produced a bit of load imbalance, the advantages of model of shared memory were dominated on those systems.

In the other results, the implementation of message passing performed better than the implementation of shared memory. Here the locality benefits that are associated with the message passing has made it better. So the problem was analysed as both the models of programming has advantages in offering the performance. In certain situations like where the communication is cheaper with the importance in load imbalance, the shared memory is considered to be the best choice. In case where the communication is more expensive or where the load imbalance is unlikely, the model of message passing is considered better.

In the earlier research of those models, it was also seen that when two models are implemented on the olden microprocessors like Butterfly, symmetry, multimax, there is not much difference in the performance of both the models. However when implemented on the modified new processors like RISC and Iris in which processors are very fast and highly scalable, and where the medium of communication is less, the shared memory doesn’t have the substantial advantages. [32]
CHAPTER 4
Message Passing

4.1 Overview
In the field of computer science, message passing [33] sends the process a message and depends on the process as well as on the supporting infrastructure for the selection and invoking of the actual code to work. Message passing is completely different from the conventional programming in which name invokes a process, subroutine or function. Message passing is considered as a key to certain models of concurrency and programming through object orientation.

In the world of modern computer software, the concept of message passing is being used universally. The usage is in a way for the instances that build up a program for working with each other and also as a way for the objects and systems that are running on computers that are different for interaction. Implementation of message passing can be made by various mechanisms even with the channels.

![Message passing model](image)

Message passing is a model defined as an explicit communication. It means all communication maintains explicit messages from the processors having data to those that needed data. Such messages accomplish any of the communication among the processors. Both the techniques of synchronization and communication are considered to be unified with message passing. Generating of events that are remote and asynchronous is taken to be the integral part of any communication model of message passing.
The only message passing library that is considered as a standard is MPI. All platforms of HPC support it virtually. Practically, it can be said to be replaced all the earlier libraries of message passing. There is no need for modifying the source code when an application is ported to a different platform which has the support of MPI standard. The message passing model is considerably simpler to understand — you just have a stream of messages coming in that you want to process — and it maps much more nicely to what is actually possible with networks so it obviously can scale up massively.

MPI is considered to be the standard implementation of the model of message passing in parallel computing. There will be many processes in the parallel computation with each one working on some local data. Local variables are present for every process and no process has any mechanism to access directly another memory. Data sharing between the processes takes place by message passing with the help of sending and receiving the data explicitly between the processes.

Model of message passing can be implemented on various platforms from multiprocessors of shared memory to the networks of workstations and even machines of single processors. It also allows more control on the location of data and flow within a parallel application than the model of shared memory. By this, programs can achieve higher performance using message passing explicitly. Indeed, primary reason why message passing is still in the world of parallel programming is due to the reason of high performance.

**Advantages of Message passing:** Being interrupt driven by nature, it is considered as one of the biggest semantic advantages of the communication model of message passing. Both the data and synchronization are combined into a single unit by the messages. Interrupt-with-data is provided by message passing that is desirable for a number of activities of operating system in which patterns of communication are known in advance explicitly. Further, large data structures manipulation like memory pages is mediated with this communication model. Though message passing needs explicit management of locality of data and communication, it is not considered to be the disadvantage in an operating system because most of the operating systems can manage their own data structures anyway explicitly.

Along with the functions of operating systems, other applications are amenable for a communication model of message passing. Such applications have a component of large
synchronization and are referred as “data-driven”. Simulation by event driven and systems of solutions of sparse matrices can be considered as example. Communication models of message passing are taken to be natural for decomposition of client-server style mainly when communication must occur across the domains of protection. The last decade has seen the emergence of various number of interfaces of message passing supporting semantics of polling such as CMMD, p4 and MPI [34].

In message passing, explicit communication puts the focus on aspects of costly parallel computation, sometimes leading to the improved structure in the program of multiprocessor. Synchronization is generally associated with sending of the messages by reducing the possibility of errors that are introduced by the incorrect synchronization. Message passing is easy for using sender initiated communication which may have some advantages in performance.

In addition to the requirement of the management of the data and communication explicitly, there is an intrinsic disadvantage for the paradigm of message passing which cannot be eliminated by the design of good interface. The cost incurred on assembling and disassembling of the messages which is generally called as the marshalling cost is intrinsic to message passing because by nature, messages are considered to be transient and not associated with data structures of computation. At the message source, data is collected from the data structures based on the memory and are then copied into the network. At the destination, it occurs in reverse process namely data will be copied from the network into the data structures based on memory, for which data is remained associated with data structures based on memory at all the times even at the time of communication.

The important note can be that the cost of marshalling for the message passing can be considered as a less issue when compared with the patterns of implementation and communication of the model of shared memory. In such cases, the cost of marshalling in model of message passing is matched directly with the copying costs of the shared memory.

In the field of computer science, interprocess communication refers to the mechanisms that are provided by an operating system allowing its processes for managing of the shared data. Generally, application can make use of the IPC that are categorized as clients and servers in which client requests the data and where it is responded by the server. Methods that are used
for achieving of this IPC are divided into several categories that are based on the requirements of the software such as performance and requirements of the modularity and circumstances of system such as bandwidth and latency of the network.

Communication of the processes can be through the shared areas of memory. Inter process message passing is much more useful for the transfer of the information. It can be made used for just the synchronisation and can coexist with the communication of shared memory. Message passing can be referred as a means of communication between the different threads within a process, different processes that are running on the same node, different processes that are running on the nodes which are different from each other. When messages are passed between the two processors, it is then represented as inter-process communication, or IPC. Message passing can be used as an approach of process oriented for the synchronization than the approaches of data-oriented that are used in enabling mutual exclusion for the shared resources. The two dimensions mentioned are:

- Synchronous vs. asynchronous
- Symmetric or asymmetric process/thread naming

4.2 MPI standards used for high performance and portability

The interface of message passing is a specification for a message passing standard library which was defined by the forum of MPI – a broadly based group of vendors of parallel computers, writers of library and specialists of applications. Development of several MPI implementations have been done. The model of message passing in the parallel computation has been emerged as an expressive, efficient and well understood paradigm of the parallel programming. The proliferation of the library of the message passing designs from both vendors and users were correct for a certain period of time, but later general semantics for message passing had been reached that an attempt at standardization may be undertaken usefully.

The target for developing of the portable implementation of MPI began at the same time as definition of MPI process itself. It included all systems capable of supporting the model of message passing. MPICH is available freely with complete implementation of the specification of MPI that is designed to be both portable as well as efficient. The CH in MPICH stands for “Chameleon” which is considered to be the symbol of adaptability to one’s environment and
thus of portability. Chameleons are so fast and from the initial point, secondary goal was to give up as little efficiency as possible for the portability.

P4 is a third generation library of parallel programming that includes the components of both the shared memory and message passing which are portable for many of the parallel environments of computing that include networks of heterogeneous. P4 contributed much of the code for the networks of TCP/IP and multiprocessors of the shared memory. Chameleon is a package with high performance and provides portability on all the parallel computers. Implementation of it is done as a thin layer over the vendor systems of message passing for the performance and systems that available for portability. ZipCode is a system of portability for writing up the libraries with scalability. It has contributed several concepts to the design of MPI standard. It also has huge collective operations.

4.3 Support provided by MPI

It is an application programme interface of the message passing which includes protocol and specifications with semantics for how its features should react at the time of implementation. MPI includes message passing of point-to-point and operations that are global which are specified to a user specified group of processes. MPI also provides abstractions for the processes at two levels. First, processes are named as per the rank of the group in which communication takes place. Secondly, virtual topologies enables for the graph or Cartesian naming of the processes that help to relate the semantics of an applications to the semantics of message passing in an efficient and convenient way.

Three additional classes of services are provided by the MPI. They are environmental inquiry, information of basic timing for the measurement of performance of the application and a profiling interface for monitoring of the external performance. MPI does a heterogeneous conversion of data a transparent part of its services by attaining the specification of data type for all the operations of communication. Both the data types of built in and user defined are provided.

The functionality of MPI is accomplished with the opaque objects which includes the well-defined constructors and destructors enabling MPI to have a look and feel of object based. Opaque objects include groups which are the fundamental containers for processes and communicators that contain groups and are used as arguments for the calls of communication
and request objects for the operations of asynchronous. Data types of built in and user defined allow for the communication of heterogeneous and description of elegant semantics in operations of send/receive as well as in the collective operations.

MPI provides the support for both the SPMD and MPMD [35] modes of parallel programming. MPI also support computations of inter-application with the help of operations of inter-communicator that provide support for the communication between the groups rather than to a single group. MPI enables an application programming interface (API) which is a thread safe that can be used in environments of multi-threaded as the implementations mature and provide support thread safety themselves.

4.4 Message passing programming model

The most widely used approach for the parallel computing is considered to be the message passing. In the model of message passing, a computation comprises one or more processes that communicate by calling routines of the library for sending and receiving of the messages. Communication here is cooperative in which data is sent by calling of a routine and the data is not received until the process of destination makes a call to the routine for receiving of the data.

There are two great strengths for the model of message passing. The most important for the users is that the programs that are written using message passing are highly portable. Virtually any number of computers can be made used for the execution of the parallel program that is written using the message passing. The model of message passing doesn’t require any special support of hardware for executing efficiently. Secondly, the strength is that the programmer is provided with the explicit control over the memory location in a parallel program by the message passing. As the access of memory and placement generally determine the performance, such ability for managing the location of memory can allow the programmer to achieve high performance.

Several models of message passing and libraries have been developed among which most of them support the basic mechanisms that are same. For the communication of point-to-point, an operation of send is used for initiating of the transfer of data between the 2 programs executing concurrently and then an operation of receive is used for the extraction of data from the systems into the memory space of applications. Addition to this, collective operations like broadcast
and reductions are provided; these develop common global operations which involve many processors.

Variations in the interface of message passing can make a varying impact on the programs performance written to that interface. The major factors that has the influence on performance are bandwidth and latency of actual message passing and the ability for overlapping of the communication with computation. In many of the modern parallel computers, latency is dominated by the setup time of the message rather than actual flight of time through the network of communication. So, the overhead of the software for initializing of the message buffers and interfacing with the hardware of communication can be important. The bandwidth which is achieved by a particular implementation of message passing is always dominated by the number of times the data is communicating should be copied while transferring the data among the components of the applications. The interfaces that are designed poorly can lead to the extra copies decreasing the overall performance of an application.

4.5 Family of Problems solved by Message passing

Satisfiability problem solved using techniques of message passing

The propositional satisfiability problem (SAT) [36] is one of the oldest and most researched problem in the computational physics. The main objective of this problem was to solve and find an assignment for a set of Boolean variables in such a way that logic statement about those variables is ‘true’ which is otherwise called satisfied. It was investigated with the help of application of Bayesian inference for finding of the most probable assignment for the variables in a way that the logic is satisfied. Algorithms used here for solving are either complete or incomplete type of problems. The most common approach used here was ‘backtracking’. It worked by assigning each value to the proposition.

\[(P \land \neg Q) \land (Q \land \neg R) \land (\neg P \land \neg R)\]

In this, techniques of message passing are used with a graph based treatment in which all the logics and propositions are arranged in a graphic network manner. They were represented as vertices and edges through which the information is communicated. On implementing the sum product algorithm through network of graph, it was evaluated on the model of message passing where the results obtained are with high efficiency and scalable.
Discrete location problems solved using Message passing algorithms

The Problem of Uncapacitated Facility Location [37] is one of the mainly considered discrete problem. Max Product algorithm is used here which has the features of linear programming of graphical models. The iterations in this algorithm are almost similar to that of problems in the linear programming algorithms. Here in this research, the advantages of algorithms of non-metric instances is demonstrated. MPLP here is considered as the LP relaxation based algorithm of message passing in graphical models. The obtained solution here is very optimal. The message updates in the MPLP are block coordinate that are augmented with the dummy variables.

Inference and optimization problems

Several algorithms of message passing are used for solving of the inference problems [38] and the optimization problems. In the problems of inference, some noisy or ambiguous measurements are taken as input are inferring is done from those inputs. “Channel coding which is the fundamental problem of information theory. Additional bits gets appended to the encoder when it gets transmitted. The task of the decoder here is probabilistic inference in which it tries to calculate from all the possible bits of noise. This transmission of the messages and manipulation of bits in the code is achieved using the techniques of message passing like sending and receiving messages with the help of the algorithm of message passing used here in such problems.

Computer vision is taken as an example in which problems of probabilistic inference are most prevalent. In this scenario, one obtains images or videos from one or more cameras and in need of inferring something that is being captured. In terms of computer, images of photography are considered as the objects with matrices of 2-dimensional intensities of color where the scene is a three dimensional. The inference system of computer vision makes use of the statistical model which is the part of algorithm.

4.6 Shared Memory and Message Passing

Both the shared memory and message passing are the two mechanisms of Interprocess communication. Memory can be shared by the multiple processors in 2 ways. In first way, all processors can make use of the single bank of memory by sending messages over the network as shown in the figure 5. Such an organisation may not be scalable as the single bank of shared memory may result in a bottleneck as the count of processors becomes large. Secondly,
multiple memory banks can be accessed by all the processors in a “dance hall” fashion shown in the figure 6. [14, 15, 16].

Figure 5: Shared memory machine with one memory bank

Figure 6: Memory with “dance hall” organisation

Machines with such an organisation can resolve the problem of bottleneck, but can be improved with the tightly coupling of the memory organization of processor which are generally called as the machines of distributed shared memory and perform well when compared to those of dance hall as the wires between the banks of memory and processors are shorter. This is the reason why machines of distributed shared memory are low in cost and a processor given needs so less time for accessing its bank of memory which is tightly coupled as shown in figure 7.

Figure 7: Machine with processors tightly coupled with memory banks.

The process of caching helps to reduce the accessing time of memory. For the implementation of the shared memory, if tightly coupled organization is used, and every processor having a cache, then every reference of memory must perform certain check like: if address A’s data is
cached, then it should be loaded from the cache. If not, if A is an address of local, then data should be loaded from the remote memory and should perform the actions of cache coherency.

Some machines of message passing organize the processors, banks of memory, and the network as shown in the figure. Rather than having a single space of address for all the processors, each one of it will have its own space of address for its local bank of memory. The programming model of Shared memory can be implemented using such architecture of message passing.

As the machines of shared memory provides programmers with a single address space while machines of message passing gives multiple address spaces, shared memory machines are conceptually easier for programming. Machine of shared memory having many banks, makes programming easier as it frees the writer for managing the banks of memory and also it hides the message passing from the programmer. If the message passing is used by the programmer, then the programmer will be in charge for orchestrating all the events of communication with the help of explicit sends and receives. This may become harder when the communication is difficult.

For the machines of shared memory, various library functions provide the synchronization. Data transfer is attained by loads and stores of memory. Both data transfer and synchronization could be combined together on using message passing. When one processor sends data to another message, the processor that receives it can process the message by atomically executing a block of code.
CHAPTER 5
Experimental Studies and Evaluation

A. Case Study 1- Peg puzzle search problem

5.1 Overview
The peg board game [39] which has 15 holes is considered to be the modern game mostly played in Europe since 17th century. It was also known to be peg solitaire or the Crackle Barrel puzzle. The Peg game is chosen as it is easy for understanding and to solve. Addition to this, many parallel programming issues are involved to solve this problem. Load distribution, data management and synchronization are the issues considered in solving this puzzle. We consider our application to be fine-grained as the time spent for accessing of shared memory or events of communication is less. It can be said as irregular due to the data structures used in it evolved dynamically as the application executed.

The peg game of size 5 is a board having a shape of triangle containing 15 holes each of which may have a peg. At the beginning of the game, all the holes except hole in the middle of the third row will have a peg. The movement in the game is to take off the peg by moving one peg over the other. The solution for this would be leaving of single peg on the board in a sequence of moves. The final goal of this Peg Puzzle Search Problem is to count the possible no of distinct solutions. Exhaustive search has to be made so as to find all the solutions.

The general working phenomenon of this game is that the game begins with one hole and all other slots filled with peg like pins as in configuration (a) with solid circles as pegs and empty circle representing a hole. The goal of the puzzle is to make a sequence of moves with each peg being removed so that only one peg will be remained at the last. Removal of peg can be made only by jumping it with the other peg. So, a move can be legally made if only there is a peg flanked by another peg and a hole which is arranged in a line. For example in figure 8, configuration (b) shown below is obtained from configuration (a) by one jump, and then (c) is obtained from (b) by making a second jump.
The ultimate goal of this peg puzzle search problem is to count the number of distinct solutions. Because we must find all solutions, solving this search problem involves an exhaustive search. Pegs which are placed in the holes are represented by each node in the search tree which are obtained in the beginning through sequence of moves. Such placing of pegs is called to be position. Bit vector is used to represent this position by using a bit representing the presence of a peg in a hole. Node A to B makes an edge giving the position indicated as B that is derived by moving one place from node A. Node B position is said to be the position’s extension indicated as node A. So here from the search tree, the root represents the beginning position and the leaves are indicated as positions such that no other moves are made from it. For initial position with P pegs, moves of P-1 is needed to get the solution for puzzle, as exactly 1 peg is removed for each move. So the solution could be a path root->leaf having depth of P-1 in a tree. Leaves of depth P-1 are said to be valid ones and that which are lower than depth of P-1 are invalid leaves.

Transposition table stores the positions so as not to explore extension positions that are already explored. In the transposition table, on finding the position extension, exploration of sub tree which is produced from this extension is not needed again and instead the extension and position are joined in the transposition table. As joining occurs in the search tree, it is referred to as directed acyclic graph (dag) [40]. Hash table is used to implement the transposition table. Same as the element of an array, a set of positions are held by the slot of the transposition table under a hash function. If the same slot is mapped by two positions, then they are combined together as a chain in slot.

So as to reduce the no. of explored positions, symmetry of the triangle puzzle board is also exploited in the implementation. As the peg game is of the shape of equilateral triangle, it has
three axes of symmetry through which 6 of positions could be noted in every node of the search dag with the help of appropriate reflections as shown in Figure 9.

![Figure 9: Axes of symmetry](image)

When exploiting symmetry, there is a trade-off of time space. More computation is needed in order to do the reflections if symmetry is exploited by the search procedure but still computation and space are saved by decreasing the search dag’s size and by avoiding the positions that are already explored. Along with the usage of transposition table and exploiting the symmetry, some other optimizations like compression, pre-loading of transposition table are also performed at the time of tree search using some algorithms.

**PBFS Vs PDFS**

When both the search algorithms of Parallel Depth first search (PDFS) and Parallel breadth-first search (PBFS) are compared, PBFS executes faster and less memory is used than PDFS in general [41]. PBFS is considered to be more appropriate than PDFS as an exhaustive search to find all the solutions is performed inherently. If the search problem is for getting any kind of solution, then suitable algorithm would be the PDFS. The result is the difference in the performance of how the entire no. of solutions are determined. The counter of solution is kept with every node storing the paths count reaching from root to this node. The root counter of search dag in PBFS is set to 1 and then every node in the next level adds up values of parent counters. On adding up all the values from the valid leave counters, we get the no. of solutions for the game of peg.

In contrast, in PDFS, as several processors will concurrently operate on different levels of dag, solutions to get counted in top down manner is not possible in PDFS as that of in PBFS. In general, procedure of search has two options: counter values can be pushed down with complete subdags similar to that of the solutions counted in the PBFS, or solutions can be counted in bottom-up fashion after the formation of entire dag. Option 1 search procedure here
makes the portions of the dag to traverse many times. Second one is very costly as it involves many number of joins.

Execution is faster in PBFS than in PDFS because the search dag is traversed only once in PBFS whereas traversing in PDFS is done twice- once at the time of constructing dag and then to count the solutions. Addition to this, detecting of formation of dag completely in PDFS leads to extra overhead.

Memory usage is less in PBFS than that of PDFS. As solutions in PBFS are counted in top down fashion, it is enough for it to remember only the positions of dag at two successive levels at most. On the other hand, as solutions are counted in bottom-up fashion in PDFS, all the positions of the dag are needed to be remembered. In this way, PBFS use less memory for storing of the positions in the table of transposition than PDFS and also progressively less amount of memory will be required than PDFS as the size of the problem increases.

5.2 Implementation with PBFS

Four data structures are used in PBFS shown in figure 10: a transposition table, for the extension of current levels in the search dag, positions are held in the current queue. For the extension of positions at next level, next queue is used and the other data structure is Pool of positions called to be work queues. For the purpose of storing positions, pool of positions which is a bank of memory is used so as to put the counters that are linked with positions. The other data structures use pointers for referring the positions of pool.

![Figure 10: Four data structures used in PBFS.](image)

Data structures of this kind are manipulated by PBFS in certain rounds that is equal to the total levels in an entire search dag. Current queue is placed with the initial position before the first round begins. Then positions are removed from the current queue by the processors and for each position, all positions that are available are generated. If the transposition table don’t have
the extension in it, then the extension is sited in both the table as well as next queue along with the counter of solution that is set to the value of the position counter through which extension was generated. If in case, transposition table already have the extension in it, it is combined with position present in table through ascending position’s solution counter by extending value counter and not placing in the next queue. After extending all the positions that are present in the current queue, the round gets ended. The next queue positions are put in the current queue along with the clearing of transposition table. Later the next round gets started. On completion of the last round, the total sum of values of the position’s solution counters present in the current queue gives the result of number of solutions.

Below is the source code for Peg game with “N” indicating the size of the peg puzzle. By running this code it provides all the possible movements of the peg with corner peg as empty. In the recursive way it searches every node in all possible ways by travelling through each node for finding all the solution from the current configuration node. The main theme of this algorithm is to illustrate the concepts of the functions that are recursive.

```plaintext
Peg := proc( N )
    local peg, i, j, r, A;
    peg := proc( A, N, path, sol )
        local i, j, p;
        p := [op(path),A];
        for i from 1 to N do
            for j from 1 to N-i+1 do
                if A[i,j] then
                    if i+2 <= N-j+1 and A[i+1,j] and not A[i+2,j] then
                        peg( subsocp([i,j]=false,[i+1,j]=false,[i+2,j]=true,A), N, p, sol );
                    end if;
                    if i-2 > 0 and A[i-1,j] and not A[i-2,j] then
                        peg( subsocp([i,j]=false,[i-1,j]=false,[i-2,j]=true,A), N, p, sol );
                    end if;
                    if i-2 > 0 and j-2 <= N-i+3 and A[i-1,j+1] and not A[i-2,j+2] then
                        peg( subsocp([i,j]=false,[i-1,j+1]=false,[i-2,j+2]=true,A), N, p, sol );
                    end if;
                    if j-2 > 0 and i-2 <= N-j+3 and A[i+1,j-1] and not A[i+2,j-2] then
```
This is the primary algorithm for the Nth size peg game, where parameter N is the size of the peg game and number of all possible searches by using recursive way is the output of the program which further helps as input for calculation of time taken to execute the program with shared memory and message passing as the implementation techniques.

<table>
<thead>
<tr>
<th>Round Number</th>
<th>Number of Positions Explored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>134</td>
</tr>
<tr>
<td>6</td>
<td>319</td>
</tr>
<tr>
<td>7</td>
<td>524</td>
</tr>
<tr>
<td>8</td>
<td>595</td>
</tr>
<tr>
<td>9</td>
<td>462</td>
</tr>
<tr>
<td>10</td>
<td>265</td>
</tr>
<tr>
<td>11</td>
<td>81</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>(including initial position) 2463</td>
</tr>
</tbody>
</table>

Table 1: Position statistics per round for problem size 5
<table>
<thead>
<tr>
<th>Round Number</th>
<th>Number of Positions Explored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>166</td>
</tr>
<tr>
<td>5</td>
<td>805</td>
</tr>
<tr>
<td>6</td>
<td>3255</td>
</tr>
<tr>
<td>7</td>
<td>10842</td>
</tr>
<tr>
<td>8</td>
<td>29037</td>
</tr>
<tr>
<td>9</td>
<td>62813</td>
</tr>
<tr>
<td>10</td>
<td>108651</td>
</tr>
<tr>
<td>11</td>
<td>144681</td>
</tr>
<tr>
<td>12</td>
<td>143275</td>
</tr>
<tr>
<td>13</td>
<td>102976</td>
</tr>
<tr>
<td>14</td>
<td>53927</td>
</tr>
<tr>
<td>15</td>
<td>20684</td>
</tr>
<tr>
<td>16</td>
<td>5893</td>
</tr>
<tr>
<td>17</td>
<td>1180</td>
</tr>
<tr>
<td>18</td>
<td>120</td>
</tr>
<tr>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>(including initial position)  688349</td>
</tr>
</tbody>
</table>

Table 2: Position statistics per round for problem size 6
Table 3: Position statistics per round for problem size 7

<table>
<thead>
<tr>
<th>Round Number</th>
<th>Number of Positions Explored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>234</td>
</tr>
<tr>
<td>5</td>
<td>1362</td>
</tr>
<tr>
<td>6</td>
<td>7320</td>
</tr>
<tr>
<td>7</td>
<td>32694</td>
</tr>
<tr>
<td>8</td>
<td>124336</td>
</tr>
<tr>
<td>9</td>
<td>395100</td>
</tr>
<tr>
<td>10</td>
<td>1057990</td>
</tr>
<tr>
<td>11</td>
<td>2372103</td>
</tr>
<tr>
<td>12</td>
<td>4435444</td>
</tr>
<tr>
<td>13</td>
<td>6881092</td>
</tr>
<tr>
<td>14</td>
<td>8829772</td>
</tr>
<tr>
<td>15</td>
<td>9336725</td>
</tr>
<tr>
<td>16</td>
<td>8102692</td>
</tr>
<tr>
<td>17</td>
<td>5755731</td>
</tr>
<tr>
<td>18</td>
<td>3356026</td>
</tr>
<tr>
<td>19</td>
<td>1602445</td>
</tr>
<tr>
<td>20</td>
<td>618748</td>
</tr>
<tr>
<td>21</td>
<td>191239</td>
</tr>
<tr>
<td>22</td>
<td>47219</td>
</tr>
<tr>
<td>23</td>
<td>8620</td>
</tr>
<tr>
<td>24</td>
<td>1061</td>
</tr>
<tr>
<td>25</td>
<td>144</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>(including initial position) 53158132</strong></td>
</tr>
</tbody>
</table>

Table 1, 2, 3 shows the number of positions explored per each round for problem sizes 5, 6 and 7 respectively.

### 5.3 Implementation details

*Processor:* As the part of hardware, a workstation is used with the high speed processor of Intel. The details of processor could be noted as:
- Processor number: Processor Name: Intel® Xeon® Processor E5-2699 v4
- Cache: 55 MB SmartCache
- Bus Speed: 9.6 GT/s
Instruction Set: 64-bit
No of Cores: 22
No of Threads: 44
Processor Base Frequency: 2.2 GHz
Max Turbo Frequency: 3.6 GHz

Operating system: Then, an operating system of Linux from Ubuntu is downloaded onto the Desktop. Linux is immune for the malwares and no software of anti-malware is needed as well as offers both options with free of cost. Linux runs greater on the hardware that is older and less powerful. It doesn’t need any restrictions of insane license. Linux is so configurable with more flexibility. Its software can be easily installed and removed from secure sources. It provides with a great hardware support. It can be said as a world with having many softwares for free. Linux will be maintained by a community of first rate developers that is open and global. All the software of Linux is available on the internet with no scope of losing it.

Shared memory interface:
Then a software Simulator of Smart [42] is used for creating the environment of shared memory. Events of system-level such as process switching and migration of task will have a major effect on the performance of the systems of a computer. Smart software is a new environment of simulation which has a capability for emulating the effect of several mechanisms that are provided by different other simulators. A very user friendly environment is provided by the Smart simulator which allows the control of different parameters and mechanisms of the system like type of cache coherency protocols, organization of cache, scheduling policies of threads and processes. The environment of smart can be used for monitoring, analysing and measuring different events of system. The Smart simulator which was presented here supports the simulation of shared memory architectures.

The simulation is divided as 2 main components in which the layer of front end acts like a generator engine of an event and the layer of back end that can make changes to these events to reflect the changes in the machine state as per the events and mechanisms of system level. The simulator of Smart can control the policies of scheduling threads and processes. It also provides a state of art for scheduling of the primitives so that the user can build easily on the top of its own schedulers of task.
The Smart simulator provides a tool of convenient and configurable for both monitoring and debugging events of communication model of shared memory. It also provides with wide number of range measurements for performance of both system level and single processor level that includes utilization of a system, performance of the cache. Smart is also used for visualizing the sharing pattern of systems of multi cache as per the state of coherency and the process to which they belong to. It also provides the user with the easy way for writing and adding of the new protocol of cache coherency to the environment of smart environment.

The parameters that are provided by the simulator of Smart are cache size, cache block size, slices of time that threads are scheduling each moment of execution, amount of processors in the systems, width of the local bus, parameters of time like hit time of the cache, access time of the memory and transfer time, etc.

In this software simulator, we have chosen parameters of number of processors and the simulation time for the game to be solved is noted.

- Create a shared memory segment. Throws if already created:

Using boost::interprocess;

```cpp
    shared_memory_object shm_obj
        (create_only
        ,"shared_memory"
        ,read_write
        );
```

On getting access to the shared memory on the simulator, the PBFS code of the peg game is run in it. Before that, parameters of the simulator are set. Here, we have considered different number of processors for running the code in it. So the Parameter of number of processors is considered. For the evaluation of experiment with the time taken for execution, the parameter of Execution time is considered. It is calculated to see how much time is being taken for the execution to solve the peg game on changing the number of processors.

The execution time for amount of processors taken is noted for the problem size of 5. Then the experiment is extended to the problem size of 6, and 7 and is then showed with the same parameters of execution time and amount of processors.
Message passing interface:

MPI-SIM [43] is used for running the program with the environment of message passing. Execution of models of MPI-SIM in a parallel manner are synchronized by making use of a set of conservative protocols that are asynchronous. MPI-SIM reduces the overheads of synchronization by exploiting the characteristics of communication of the program that it simulates.

MPI which is a library of message passing offers a host of point-point and collective interprocess communication functions for a set of single threaded processes that are executing parallel. It makes use of a non- blocking functions of MPI. The core functions include MPI_Issend, a non-blocking synchronous send, MPI_Ibsend, a non-blocking buffered send, MPI_Irecv non-blocking receive and MPI_Wait.

The two protocols that are used in simulating are the synchronous or quantum protocol and protocols that are asynchronous. Some protocols used in this library are SimOs and Lapse. Environment of MPI-SIM includes the setup of parameters like cache size, cache block size, slices of time that threads are scheduling each moment of execution, amount of processors in the systems, width of the local bus, parameters of time like hit time of the cache, access time of the memory and transfer time, etc. In this software simulator, we have chosen parameters of number of processors and the time taken by the processor to run the program given to the software.

On getting access to the message passing memory on the simulator, the PBFS code of the peg game is run in it. Before that, parameters of the simulator are set. Here, we have considered different number of processors for running the code in it. So the Parameter of number of processors is considered. For the evaluation of experiment with the time taken for execution, the parameter of Execution time is considered. It is calculated to see how much time is being taken for the execution to solve the peg game on changing the number of processors.

The values of time taken for the execution for amount of processors taken is noted for the problem size of 5. Then the experiment is extended to the problem size of 6 and is then showed with the same parameters of execution time and amount of processors.
5.4 Evaluation:
In this way, the puzzle of sizes 5, 6 and 7 are solved using the PBFS code on both the communication models of shared memory and message passing. Execution time for every amount of processors is collected in both the cases of shared memory and message passing for different sizes of game. The points obtained are valuated using the graphs using Simulink and are then analysed when shared memory will be better than message passing and when message passing memory could be better than shared memory.

The table 4 below shows the values when the program of game size 5 is run on a shared memory model. It could be observed that, time taken for the game to be executed when used one processor is 0.262 seconds. As the number of processors are increased from 1 to 4, it is seen that the execution time reduced to 0.121 seconds. As the processors count is increased to 32, time taken for the execution of the peg game is 0.028 seconds which results in the best performance.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.15</td>
</tr>
<tr>
<td>2</td>
<td>0.098</td>
</tr>
<tr>
<td>4</td>
<td>0.058</td>
</tr>
<tr>
<td>8</td>
<td>0.034</td>
</tr>
<tr>
<td>16</td>
<td>0.025</td>
</tr>
<tr>
<td>32</td>
<td>0.007</td>
</tr>
</tbody>
</table>

Table 4: Execution times for shared-memory size 5

The table 5 below provides the values of time taken for execution of a peg game against the amount of processors for the game size of 6. The time taken here for execution on using one processor is 3.28 sec. The execution time gradually gets decreased as that of the size 5 on increasing number of processors. The execution time for sizes 5 and 6 vary drastically as the number of positions explored will be more in case of size 6 game. This is analysed in the further sections.
The table 6 below provides the values of time taken for execution of a peg game against the amount of processors for the game size of 7. The time taken here for execution on using one processor is 9.22 sec. The execution time gradually gets decreased as that of the size 5 and 6 on increasing number of processors. The execution time for sizes 5, 6 and 7 vary drastically as the number of positions explored will be more in case of size 6 and 7 game. This is analysed in the further sections.

By analysing the above values of table, below is the graphical representation (figure 11, 12, 13) of execution time for various number of processors for shared memory implementation for size 5, 6 and 7 of peg game.

![Graph](image-url)  
**Figure 11**: Shared Memory size -5
By implementing the game of peg in the model of message passing, the values of execution time are evaluated by varying the number of processors using both the sizes of 5 and 6 of the game. The table 7 below shows the values when the program of game size 5 is run on a message passing model. It could be observed that, time taken for the game to be executed when used one processor is 0.168 seconds. As the number of processors are increased from 1 to 4, it is seen that the execution time reduced to 0.048 seconds. As the processors count is increased to 32, time taken for the execution of the peg game is 0.018 seconds which results in the best
performance. The table 8 and 9 below provides the values of time taken for execution of a peg game against the amount of processors for the game size of 6 and 7.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.168</td>
</tr>
<tr>
<td>2</td>
<td>0.116</td>
</tr>
<tr>
<td>4</td>
<td>0.071</td>
</tr>
<tr>
<td>8</td>
<td>0.048</td>
</tr>
<tr>
<td>16</td>
<td>0.037</td>
</tr>
<tr>
<td>32</td>
<td>0.018</td>
</tr>
</tbody>
</table>

Table 7: Execution times for Message Passing size 5

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.97</td>
</tr>
<tr>
<td>2</td>
<td>1.28</td>
</tr>
<tr>
<td>4</td>
<td>0.71</td>
</tr>
<tr>
<td>8</td>
<td>0.42</td>
</tr>
<tr>
<td>16</td>
<td>0.21</td>
</tr>
<tr>
<td>32</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Table 8: Execution times for Message Passing size 6

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.89</td>
</tr>
<tr>
<td>2</td>
<td>6.21</td>
</tr>
<tr>
<td>4</td>
<td>4.11</td>
</tr>
<tr>
<td>8</td>
<td>2.97</td>
</tr>
<tr>
<td>16</td>
<td>1.68</td>
</tr>
<tr>
<td>32</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Table 9: Execution times for Message Passing size 7

By analysing the above values of table, below are the graphical representations (Figure 14, 15, 16) of execution time for various number of processors for message passing implementation for size 5, 6 and 7 of peg game.
Figure 14: Message passing size -5

Figure 15: Message passing size -6

Figure 16: Message passing size -7
Performance with metrics of speedups.

Speedups are calculated using execution times. Execution time taken by one processor is considered as a standard reference for calculating speedups. Speedups are calculated for all number of processors in each of the concurrency model for problem sizes 5, 6, and 7. For example, Speedup for 2 processors of shared memory size 5.

<table>
<thead>
<tr>
<th>No of Processors</th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.35</td>
<td>1.44</td>
</tr>
<tr>
<td>4</td>
<td>2.938</td>
<td>2.35</td>
</tr>
<tr>
<td>8</td>
<td>4.411</td>
<td>5.5</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>4.54</td>
</tr>
<tr>
<td>32</td>
<td>21.42</td>
<td>9.33</td>
</tr>
</tbody>
</table>

Table 10: Speedups for SM- size 5

<table>
<thead>
<tr>
<th>No of Processors</th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.08</td>
<td>2.32</td>
</tr>
<tr>
<td>4</td>
<td>1.70</td>
<td>4.285</td>
</tr>
<tr>
<td>8</td>
<td>3.83</td>
<td>7.07</td>
</tr>
<tr>
<td>16</td>
<td>5.68</td>
<td>11.14</td>
</tr>
<tr>
<td>32</td>
<td>14.26</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 11: Speedups for MP- size 5

<table>
<thead>
<tr>
<th>No of Processors</th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.25</td>
<td>1.27</td>
</tr>
<tr>
<td>4</td>
<td>1.72</td>
<td>1.91</td>
</tr>
<tr>
<td>8</td>
<td>2.34</td>
<td>2.65</td>
</tr>
<tr>
<td>16</td>
<td>3.09</td>
<td>4.86</td>
</tr>
<tr>
<td>32</td>
<td>4.04</td>
<td>8.13</td>
</tr>
</tbody>
</table>

Table 12: Speedups for SM- size 6

Table 13: Speedups for MP- size 6

Table 14: Speedups for SM- size 7

Table 15: Speedups for MP- size 7
5.5 Analysis

Shared memory vs Message passing

From the below figure 17, it can be seen that the execution time taken by the game of size 5 on running in the shared memory is lesser than the execution time of the game run on the message passing. That is by comparing both the implementations of shared memory and message passing for game size 5, we observed that shared memory could perform better than message passing. But, shared memory takes more time for executing in case of higher size game. Similarly from the figures and tables above, we could observe that the execution time of the program for size 6 and 7 is greater for the shared memory than the message passing. Message passing is better than shared memory in this case.

![Figure 17: Comparison of SM and MP-size 5](image1.png)

![Figure 18: Comparison of SM and MP-size 6](image2.png)

![Figure 19: Comparison of SM and MP-size 7](image3.png)
5.6 Conclusion

**Shared Memory Performing Better Than Message Passing**

On comparing message passing and shared memory implementations, shared memory could perform better than message passing at the problem size 5 as previously noted. The first reason for it could be the fewer possible moves in size 5 than in 6 size as there are fewer holes and less initial pegs present on the board. In the problem size of 6, the given position will be associated with more possible moves than that of a given position in size of 5, the rate at which extensions are added to the next queue will be more in size 6 problem than compared to size 5 in the implementation of shared memory.

Secondly, the joined positions are more in problem of size 6 than in problem of size 5. As positions are filled in the transposition table, more positions gets joined. As a result, adding of the positions to the next queue gets slowed down. So as the problem size 6 has more number of joined positions, processor fetches a position from the current queue if no more extensions get produced from a given position. So this fetching of positions from the current queue will be higher in problem of size 6 than that of 5. As the rate of higher requests indicates the greater contention, contention for the work queues will be greater in the problem size of 6 than that of size 5. For problem size 6, the rates of request are high as the possible moves and joined positions per round are greater than of size 5.

In problem size 5, the lower contention for the work queues results in less time spent attaining locks and lower demand on access of memory. This may be a factor for why shared memory is better than message passing in case of size 5. As less amount of time is spent on acquiring a locks, other overheads are considered more important. The requests of the data access are processed using the hardware in parallel with threads and this makes the execution time to get affected when memory access has a less demand. In message passing, data access requests are interrupted by the threads on the processors making the thread to run in sequence manner. Shared memory offers data access with lower overhead and performs better than the message passing for the applications that exhibit less contention.

As the contention for the work queues is higher, the demand will also be more for the accessing of the memory. As a result, the parallelism will be reduced in the shared memory. Above figures 18 and 19 shows the shared memory performing worse than message passing. The
overheads in the implementation of message passing do not have an effect as the parallelism of sending messages also increases.

**When Message Passing Can Perform Better Than Shared Memory**

In few cases, message passing performs well over shared memory. Firstly, shared data’s size is more than the cache line, performance of shared memory may be bad as data transfer needs many actions of coherence that needs more network bandwidth and latency increase in the transfer of data. Sending data with a single message could be more efficient. Shared memory could be still worse on using prefetching as it may yield to overhead while performing prefetching. The main reason why the shared memory is still bad than the message passing even with prefetching is because when the size of the shared data is more than a cache line then the cache coherence’s overhead increases the need for network bandwidth the latency of the data transfer is increased. Apart from this, if consuming of data is made after it is obtained and then not used, the used time for performing of the cache coherence is not spent well.

As a second point, shared memory performs badly when communication patterns are regular and already known. The accessing of shared memory needs 2 messages: one message from the processor that is requesting to the shared memory bank and the other from the memory bank sending back to the processor that is requesting. On the other hand, message passing is one way where a message is sent from one point to the other. Because of the one way, message passing gets a lock more effectively and data is transferred and lock is released.

As a summary, it can be said as shared memory performs lesser than message passing at conditions like when shared data’s size is bigger than that of a cache line and when the patterns of communications are regular and are known and on combining data transfer and synchronization. When the size of the shared data is small and the communication patterns are not regular, shared memory could perform better than the message passing.

**5.7 Summary:**

In the Peg game case study, the time used is simulation time obtained from the simulator by varying the number of processors for solving different problem sizes 5, 6, 7 of Peg game in the simulating environment using both concurrency models. Performance metrics of speedup and time are obtained from the simulator by using the number of processors as an input parameter. Analysis of speedup for both concurrency models is analysed done in detail using the concept
of contention. To show that a shared-memory version can outperform the message-passing implementation under low contention, we extended our evaluation by obtaining the percentage difference in contention values provided by the simulator when solving different problem sizes of Peg game. From the results, we observe that the percentage at which contention changed for each problem size and variation in speedup for both concurrency models, shared memory is shown to be better than message passing using problem size 5 with respect to the speedup, and this is due to the low contention.

Contention is defined as a situation in which two different processes try to read data from the same block of memory at the same time. In loosely coupled multiprocessor systems, the contention ratio is the ratio of the potential maximum demand to the actual communication bandwidth. The higher the contention ratio, the greater the number of processes that may be trying to use the actual bandwidth at any one time and, therefore, the lower the effective bandwidth offered.

In the algorithm of PBFS, as the transposition table fills up with positions, more positions in the Peg game are joined, and the rate at which positions are added to the next queue slows down. This is because problem size 6 has a higher percentage of joined positions per round than problem size 5 and because a process (when running the Peg game) fetches a new position from the current queue. If no more extensions can be generated from a given position, the rate of fetching positions from the current queue is higher in problem size 6 than in problem size 5. This leads to overhead on work queues (current and next queues collectively) in problem size 5. As the size increases, a fetching process tries reading data from same block of memory at the same time causing contention.
B. Case study 2- Solving Transitive Closure of a Matrix

In this case study, we have tried to solve a matrix problem of transitive closure. This kind of problem is taken so as to see which concurrency model could be the best in solving such kind of family of problems. The general problem of transitive closure of n by n matrix is chosen that inherently has the property of load imbalance and is solved using an algorithm that solves the problem of transitive closure of a matrix with both concurrency models of shared memory and message passing. Results are evaluated for both the concurrency models of shared memory and message passing using the simulating software of RSim [44].

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reachability matrix is called as transitive closure of a matrix.

For example, consider the graph below.

![Graph Diagram]

Transitive closure of above graphs is

```
1 1 1 1  
1 1 1 1  
1 1 1 1  
0 0 0 1  
```

The graph is given in the form of adjacency matrix say ‘graph[V][V]’ where graph[i][j] is 1 if there is an edge from vertex i to vertex j or i is equal to j, otherwise graph[i][j] is 0.

We can calculate the distance matrix dist. [V][V] if dist[i][j] is infinite, then j is not reachable from i, otherwise j is reachable and value of dist[i][j] will be less than V. Instead of directly using Floyd Warshall, we can optimize it in terms of space and time, for this particular problem. Following are the optimizations:
1) Instead of integer resultant matrix (dist[V][V]), we can create a boolean reach-ability matrix reach[V][V] (we save space). The value reach[i][j] will be 1 if j is reachable from i, otherwise 0.

2) Instead of using arithmetic operations, we can use logical operations. For arithmetic operation ‘+’, logical and ‘&&’ is used, and for min, logical or ‘||’ is used. (We save time by a constant factor. Time complexity is same though).

5.8 Hardware specifications:

Processor: As the part of hardware, a workstation is used with the high speed processor of Intel. The details of processor could be noted as:
Processor number: Processor Name: Intel® Xeon® Processor E5-2699 v4
Cache: 55 MB SmartCache
Bus Speed: 9.6 GT/s
Instruction Set: 64-bit
No of Cores: 22
No of Threads: 44
Processor Base Frequency: 2.2 GHz
Max Turbo Frequency: 3.6 GHz

Operating system: Then, an operating system of Linux from Ubuntu is downloaded onto the Desktop. Linux is immune for the malwares and no software of anti-malware is needed as well as offers both options with free of cost. Linux runs greater on the hardware that is older and less powerful. It doesn’t need any restrictions of insane license. Linux is so configurable with more flexibility. Its software can be easily installed and removed from secure sources. It provides with a great hardware support. It can be said as a world with having many softwares for free. Linux will be maintained by a community of first rate developers that is open and global. All the software of Linux is available on the internet with no scope of losing it.

RSIM simulates multiprocessors that exhibits instruction level parallelism. It is an execution driven and a system of aggressive memory and has a coherence protocol in the multiprocessor along with the contention at all the resources.
The simulation of the processors with RSim includes the features like multiple instruction issue, out of order scheduling and register renaming, prediction of static and dynamic branch, Non-blocking loads and stores. The parameters chosen here are no of processors and execution time for each of the memory environment. Memory simulation features: Two-level cache hierarchy, Multiported and pipelined L1 cache, pipelined L2 cache multiple outstanding cache requests, Memory interleaving Software-controlled non-binding prefetching.

The main part of the algorithm used here is written below.

```plaintext
for i = 1 to N do
    for all j = 1 to N do
        if (M(j,i)) then
            for k = 1 to N do
                if (M(i,k)) M(j,k) := true

for k = 2 to N do
    for all i = k to N do
        for j = k-1 to N+1 do
            A[i][j] = A[i][j] -
            A[k-1][j] * A[i][k-1]/A[k-1][k-1]

for k = 2 to N do {
    receive_pivot(pivot,k)
    for i = k to N step P do
        for j = k-1 to N+1 do
            A[i][j] = A[i][j] -
            pivot[j] * A[i][k-1]/pivot[k-1]
    if (I_have_row(k+1))
        broadcast_pivot(A[k+1])
}
```

External threading software libraries (e.g., Windows* threads, Pthreads*, and Java* threads) do not have any means to automatically schedule a set of independent tasks to threads. When needed, such capability must be programmed into the application. Static scheduling of tasks is a straightforward exercise. In the algorithm of transitive closure of matrix of size n, to explore the role of load imbalance, we repeatedly executed the transitive closure application using different inputs to vary the amount of load imbalance. We used as input to the matrix of transitive closure a graph of 1000 nodes. We varied N between 100 and 1000. By statically
scheduling tasks, load imbalance is created in the program, and simulation time is outputted for solving that matrix problem in both the concurrency models, speedup is calculated from these timings. In this scenario, we then conclude that shared memory outperformed message passing with respect to speedup. Shared memory took less time than message passing in solving the transitive closure of n size matrix which has load imbalance in it. The tasks that are needed to compute transitive closure of the matrix of n size are scheduled in the algorithm statically. [45]

Solving without load imbalance, dynamic partitioning is used in the algorithm so as to remove load imbalance in the application of transitive closure problem. Dynamic partitioning has tasks allocated dynamically to the threads at the time of runtime efficiently. [46]

5.9 Evaluation:

Below are the evaluated tables and graphs showing the variation in execution times by changing number of processors on each of the concurrency model of shared memory and message passing with and without the property of load imbalance in the algorithm that is solving a matrix problem with transitive closure.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.345</td>
<td>1.634</td>
</tr>
<tr>
<td>2</td>
<td>0.942</td>
<td>1.212</td>
</tr>
<tr>
<td>4</td>
<td>0.624</td>
<td>0.842</td>
</tr>
<tr>
<td>8</td>
<td>0.439</td>
<td>0.612</td>
</tr>
<tr>
<td>16</td>
<td>0.341</td>
<td>0.415</td>
</tr>
<tr>
<td>32</td>
<td>0.214</td>
<td>0.315</td>
</tr>
</tbody>
</table>

Table 16: Performance with Load Imbalance
Figure 20: Graphical representation for performance with Load Imbalance

Table 17: Performance without Load Imbalance

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.454</td>
<td>1.945</td>
</tr>
<tr>
<td>2</td>
<td>1.645</td>
<td>1.132</td>
</tr>
<tr>
<td>4</td>
<td>1.009</td>
<td>0.786</td>
</tr>
<tr>
<td>8</td>
<td>0.784</td>
<td>0.452</td>
</tr>
<tr>
<td>16</td>
<td>0.465</td>
<td>0.298</td>
</tr>
<tr>
<td>32</td>
<td>0.296</td>
<td>0.161</td>
</tr>
</tbody>
</table>

Figure 21: Graphical representation for performance with No Load Imbalance
5.10 Analysis:
From the figures above, the concurrency model of shared memory implementation performs slightly better than the message-passing implementation when there is a load imbalance in the application chosen. Load imbalance on the Matrix problem—the advantages of the shared memory model dominate. The exact opposite occurs when the application without load imbalance is evaluated on both the concurrency models of shared memory and message passing. Message passing outperforms the shared memory with respect to the performance metric of time and speed up. Under conditions of extreme load imbalance, the shared-memory model is preferable.

Although Message Passing and Shared Memory models of parallelism have been used for over 35 years, more recently many software libraries and concurrency models have been developed, such as Pthreads, Open-MP (etc.), multi-threading, and more recently Hyper-threading [47]. Hyper Threading is considered as the proprietary of Intel that is used for improving of the parallelisation of computations. In addition to requiring simultaneous multithreading (SMT) [48] support in the operating system, hyper-threading can be properly utilized only with an operating system specifically optimized for it. Hyper-Threading can improve the performance of some MPI applications, but not all. Depending on the cluster configuration and, most importantly, the nature of the application running on the cluster, performance gains can vary or even be negative. The next step is to use performance tools to understand what areas contribute to performance gains and what areas contribute to performance degradation.
CHAPTER- 6
Conclusions and Future works

6.1 Conclusion

We identified some family of problems that are solved using modelling techniques of parallel computing such as message passing and shared memory. We also presented parallel shared memory and message passing implementations through simulating environment of two concurrency models of shared memory and message passing and algorithm of Parallel Breadth First search is run for solving Peg puzzle that needs parallelism in finding number of distinct positions for each sizes of game. First, when using shared memory, performing cache coherence actions and decoupling synchronization and data transfer can make a shared-memory implementation less efficient than the message-passing implementation drawn with the metrics of speed up for different number of processors. Second, shared memory offers low-overhead data access and can perform better than message passing for applications that exhibit low contention. We learned from this case study that under certain circumstances and for this particular application, a shared-memory implementation sometimes performs better than the message-passing one. Under other circumstances, the message-passing implementation performs better.

As a second case study, transitive closure of Matrix problem is solved using two concurrency model interfaces of shared memory and message passing through simulation software. Many software libraries and concurrency models have been developed, such as Pthreads, Open-MP (etc.), multi-threading, and more recently Hyper-threading. We show that applications having load imbalance are more adaptable for the concurrency model of shared memory than the message passing. The application showed better performance in shared memory than message passing when load imbalance is included in it. The opposite occurred when load imbalance is removed from the application or algorithm that is used in solving the transitive closure of matrix problem. Shared memory is preferable on multiprocessors where communication is relatively cheap. In these machines, the load balancing advantages of shared memory tend to outweigh the locality advantages of message passing. Even in cases where there is no load imbalance, if communication is sufficiently cheap, message passing is only marginally better than shared memory. As the cost of communication increases in shared-memory multiprocessors, message passing is becoming an increasingly attractive alternative to shared
memory. The locality advantages of message passing increase with a rise in the cost of communication, while the load balancing advantages of shared memory are not influenced by a change in the relative cost of communication.

Problems that have less communication overhead, better use of bandwidth, lower latency, and small scaled machines are better with shared memory. Problems focusing on non-local operations, large scale machines, less load imbalance can be good in message passing. Depending on the scenario, communication model can be chosen that gives High performance of computing.

6.2 Future work

To develop newer more complete and advanced methods that will help to apply parallel computing in solving large and complex problems.
Appendix A: Reference Code

PEG Puzzle

Peg := proc(N)
    local peg, i, j, r, A;
    peg := proc(A, N, path, sol)
        local i, j, p;
        p := [op(path), A];
        for i from 1 to N do
            for j from 1 to N-i+1 do
                if A[i,j] then
                    if i+2 <= N-j+1 and A[i+1,j] and not A[i+2,j] then
                        peg(subsop([i,j]=false,[i+1,j]=false,[i+2,j]=true,A), N, p, sol);
                    end if;
                    if i-2 > 0 and A[i-1,j] and not A[i-2,j] then
                        peg(subsop([i,j]=false,[i-1,j]=false,[i-2,j]=true,A), N, p, sol);
                    end if;
                    if i-2 > 0 and j+2 <= N-i+3 and A[i-1,j+1] and not A[i-2,j+2] then
                        peg(subsop([i,j]=false,[i-1,j+1]=false,[i-2,j+2]=true,A), N, p, sol);
                    end if;
                    if j-2 > 0 and i+2 <= N-j+3 and A[i+1,j-1] and not A[i+2,j-2] then
                        peg(subsop([i,j]=false,[i+1,j-1]=false,[i+2,j-2]=true,A), N, p, sol);
                    end if;
                    if j+2 <= N-i+1 and A[i,j+1] and not A[i,j+2] then
                        peg(subsop([i,j]=false,[i,j+1]=false,[i,j+2]=true,A), N, p, sol);
                    end if;
                    if j-2 > 0 and A[i,j-1] and not A[i,j-2] then
                        peg(subsop([i,j]=false,[i,j-1]=false,[i,j-2]=true,A), N, p, sol);
                    end if;
                end if;
            end do;
        end do;
    end proc;
    if numelems(p) = N*(N+1)/2-1 then
        sol(numelems(sol)+1) := p;
    end if;
end proc
end if;
end proc:
A := [seq([seq(true,j=1..N-i+1)],i=1..N)]; #fill all holes
A[1,1] := false; #pick first spot as empty
r := Array(1..0);
peg(A,N,[],r);
r;
end proc:

PegPlot := proc( moves, N )
local pegframe, i;
pegframe := proc( A, N )
local i, j, p1, p2;
#plot board
p1 := plots:-pointplot( [ 
seq(seq([[i-N)/2+j,N-i],j=1..N-i+1),i=1..N) ],symbol=circle,symbolsize=31,axes=none);
#plot pegs
p2 := plots:-pointplot( [ 
seq(seq(('if(A[i,j],[[(i-N)/2+j,N-i],NULL],j=1..N-i+1),i=1..N) ],symbol=solidcircle,symbolsize=30,color=blue,axes=none);
plots[display]([p2,p1]);
end proc;
plots:-display([seq(pegframe(moves[i],N),i=1..numelems(moves))],insequence=true);
end proc:
> PegPlot(R[1],5)

ParallelPeg := proc( N )
local peg, i, j, r, holes, nholes, me, numnodes, A;
peg := proc( A, N, path, sol, lev )
local i, j, p, k1, k2, h, split;
p := [op(path),A];
if lev = 6 then
  me := Grid:-MyNode();
  numnodes := Grid:-NumNodes();
  nholes := numelems(holes);
split := holes[(me)*floor(nholes/numnodes) + 1

.. if (me=numnodes-1,holes,(me+1)*floor(nholes/numnodes))];

#printf("Node %d level %d split: %La\n",Grid:-MyNode(),lev,split); >

else

split := holes;

end if;

for h in split do

i,j := op(h);

if A[i,j] then

if i+2 <= N-j+1 and A[i+1,j] and not A[i+2,j] then

peg( subsop([i,j]=false,[i+1,j]=false,[i+2,j]=true,A), N, p, sol, lev+1 );

end if;

if i-2 > 0 and A[i-1,j] and not A[i-2,j] then

peg( subsop([i,j]=false,[i-1,j]=false,[i-2,j]=true,A), N, p, sol, lev+1 );

end if;

if i-2 > 0 and j+2 <= N-i+3 and A[i-1,j+1] and not A[i-2,j+2] then

peg( subsop([i,j]=false,[i-1,j+1]=false,[i-2,j+2]=true,A), N, p, sol, lev+1 );

end if;

if j-2 > 0 and i+2 <= N-j+3 and A[i+1,j-1] and not A[i+2,j-2] then

peg( subsop([i,j]=false,[i+1,j-1]=false,[i+2,j-2]=true,A), N, p, sol, lev+1 );

end if;

end if;

end do;

if numelems(p) = N*(N+1)/2-1 then

sol(numelems(sol)+1) := p;

end if;

NULL;

end proc:
printf("Node %d called with N = %d\n",Grid:-MyNode(),N);
holes := [seq(seq([i,j],j=1..N-i+1),i=1..N)];
A := [seq([seq(true,j=1..N-i+1)],i=1..N)]; # fill all holes
A[1,1] := false; # pick first spot as empty
r := Array(1..0);
peg(A,N,[],r,1);
printf("Node %d found %d solutions\n",Grid:-MyNode(),numelems(r));
if Grid:-MyNode() = 0 then
    r := ArrayTools:-Concatenate(2,r,seq(Grid:-Receive(),i=1..Grid:-NumNodes()-1));
else
    Grid:-Send(0,r);
end if;
r;
end proc:

Transitive closure of Matrix problem
Program for transitive closure using Floyd Warshall Algorithm
import java.util.*;
import java.lang.*;
import java.io.*;

class GraphClosure
{
    final static int V = 4; // Number of vertices in a graph

    // Prints transitive closure of graph[][] using Floyd Warshall algorithm
    void transitiveClosure(int graph[][])
    {
        /* reach[][] will be the output matrix that will finally have the shortest distances between every pair of vertices */
        int reach[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph matrix. Or we can say the initial values of shortest distances are based on shortest paths considering no intermediate vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                reach[i][j] = graph[i][j];

        /* Add all vertices one by one to the set of intermediate vertices.

        75
--- Before start of a iteration, we have reachability values for all pairs of vertices such that the reachability values consider only the vertices in set \{0, 1, 2, \ldots, k-1\} as intermediate vertices.

--- After the end of a iteration, vertex no. k is added to the set of intermediate vertices and the set becomes \{0, 1, 2, \ldots, k\} */

for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on a path from i to j.
            // then make sure that the value of reach[i][j] is 1
            reach[i][j] = (reach[i][j] != 0) ||
                         ((reach[i][k] != 0) && (reach[k][j] != 0)) ? 1.0;
        }
    }
}

// Print the shortest distance matrix
printSolution(reach);

/\* A utility function to print solution */
void printSolution(int reach[][])
{
    System.out.println("Following matrix is transitive closure" +
                      " of the given graph");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            System.out.print(reach[i][j] + " ");
        }
        System.out.println();
    }
}

// Driver program to test above function
public static void main(String[] args)
{
    /* Let us create the following weighted graph */
    10
    (0)------>(3)
    |       /|
    5 |     1
    |      /|
    (1)---->(2)
    3

    /* Let us create the following weighted graph */
10
(0)-------->(3)
  |       \ |
  |        | 1
\ |       |
(1)-------->(2)
  3     */

int graph[][] = new int[][]{ {1, 1, 0, 1},
                           {0, 1, 1, 0},
                           {0, 0, 1, 1},
                           {0, 0, 0, 1} }
;

// Print the solution
GraphClosure g = new GraphClosure();
g.transitiveClosure(graph);
}
References


[5] Zhiyi Yu; Kaidi You; Ruijin Xiao; Heng Quan; Peng Ou; Yan Ying; Haofan Yang; Ming'e Jing; Xiaoyang Zeng, "An 800MHz 320mW 16-core processor with message-passing and shared-memory inter-core communication mechanisms," in Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International.


[27] Mehdi Alipour1, Hojjat Taghdisi, EFFECT OF THREAD LEVEL PARALLELISM ON THE PERFORMANCE, 2012, University, Qazvin 34185-1416 Iran.


[31] Virginia Vassilevska Williams, Stanford University, (2004) Multiplying matrices in O(n^2.373) time


[33] Nayeem Islam, Robert E. McGrath and Roy H. Campbell (2001) Parallel Distributed Application Performance and Message Passing, University of Illinois at Urbana-Champaign, Department of Computer science. IL.


[48] Deborah T. Marr; Frank Binns; David L. Hill; Glenn Hinton; David A. Koufaty; J. Alan Miller; Michael Upton (2006-12-12). "Hyper-Threading Technology Architecture and Microarchitecture".