TRINITY COLLEGE
DUBLIN

ERASMUS MUNDUS MASTER OF MECHANICAL ENGINEERING

MEMOIRE- THESIS

AUTHOR

Shekhar Tyagi

TITLE

Study of Parallel Implementation of Computational Codes

June 2007

1INSTITUT NATIONAL DES SCIENCES APPLIQUEES DE LYON - FRANCE

SCOLA TECHNICA SUPERIOR D’ENGINYERIA INDUSTRIAL DE BARCELONA DE LA UNIVERSITAT POLITECNICA DE CATALUNYA - ESPAGNE

THE COLLEGE OF THE HOLY AND UNDIVIDED TRINITY OF QUEEN ELIZABETH DUBLIN IRLANDE

CONTENTS

ACKNOWLEDGEMENTS		I
ABSTRACT		II
CHAPTER 1	Introduction	1
CHAPTER 2	Literature Review	10
CHAPTER 3	Designing Parallel Programs	26
CHAPTER 4	Parallel Programs	34
CHAPTER 5	Results	43
CHAPTER 6	Conclusion and Future work	51
REFERENCES		
APPENDICES		

Acknowledgement

First and foremost I would like to thank my supervisors Henry Rice and Fabrice Morestin who not only helped me in my project but also helped me getting accustomed to the European way of life and culture. I would like to mention the role played by the Erasmus Mundus Master program, which provided me the opportunity to study in Trinity College and INSA Lyon. I would also like to thank Bharath Gopalaswamy for his help and encouragement. Last but not the least I am very grateful to the other Erasmus Mundus students who were a constant source of support and inspiration for me in the past two years.

Abstract

The use of computers to solve scientific problems is abundant and almost a need of the day. As the complexity of the problems increased, the need to get fast and accurate results has also taken an altogether different meaning. The area where one needs to use all the resources at disposal to compute the results efficiently and quickly is known as HPC (High Performance Computing). Evolution of computers has given the idea of using the entire RAM at disposal while computing the results; hence, parallelization becomes highly desirable.

The goal of this project is to parallelize distributed memory systems using WINDOWS platform, analyzing some small(computationally) parallelized problems and if possible then trying to implement them for the software ABAQUS(FEM tool). Moreover the parallel program developed can be used for numerical solution to Helmholtz equation.

The basic equation governing acoustics is the Helmholtz equation; analytical complications have driven researchers to develop Numerical methods to solve this equation. Numerical methods in turn have posed computational difficulties even with the use of modern day computers, parallelization of solution of Helmholtz equation is the result of these problems. The standard algorithms are available for the Numerical methods and they only need to be programmed on a platform which can support parallel programming. By the implementation of suitable parallel techniques a considerable amount of efficiency with respect to time in obtaining the solutions can be achieved.

Chapter 1: Introduction

1.1 Introduction 2

1.2 Amdahl's Law 4

1.2.1 Description 5

1.3 Helmholtz Equation 7

1.4 Numerical Methods for Helmholtz Equation 7

Chapter 1

1.1 Introduction

Numerical simulation and other computationally intensive problems are often successfully tackled using parallel computing. Frequently these problems are too large to solve on a single system or the time needed to complete them makes single-cpu calculation unpractical. The need for more computer power will always be there. Scientists often state that

“**If you have the computer power, we will use it”**

With more computational power available, the scientists are able to perform their work faster or increase the accuracy of their computations. The ability to perform calculations in parallel opens doors to a whole new way of doing computation. Complex mathematical problems can often be broken down into smaller sub-problems. These sub-problems can often be parallelised with respect to either memory or computation.

Successful parallelisation is usually measured by the problem “speedup”. This quantity indicates how much faster a given problem is solved on multiple processors, compared to the solution time on one processor. More often than not, this speedup is only based on the computationally intensive part of the code, and phases as program start-up or data loading and saving elude the test. Also, when the ratio of computation to the input data is high enough, I/O (Input-Output) time is negligible in the total execution time.

Multiple processors are used to tackle a single computational task. The general principle involved is the division of single task into many independent operations. These independent operations will be performed by different processors in parallel, at the same time. The necessary condition for the program algorithm to work is the division of single task into mutually contiguous and independent operations. Although mathematical algorithm may contain independent operations, the program may not be completely free from the dependence of operations. This is quite a common situation, because most programs are written for sequential processing hence data exchange becomes a must between these so called independent operations. The goal of parallel processing is reduction in elapsed execution time. Processor time is attributed to processor cycles spent in execution of the instruction stream for that program. Parallel programs might include additional instructions to facilitate parallel execution. The total processor time, spent on all processors should increase as a result, see *Fig.* 1.1.

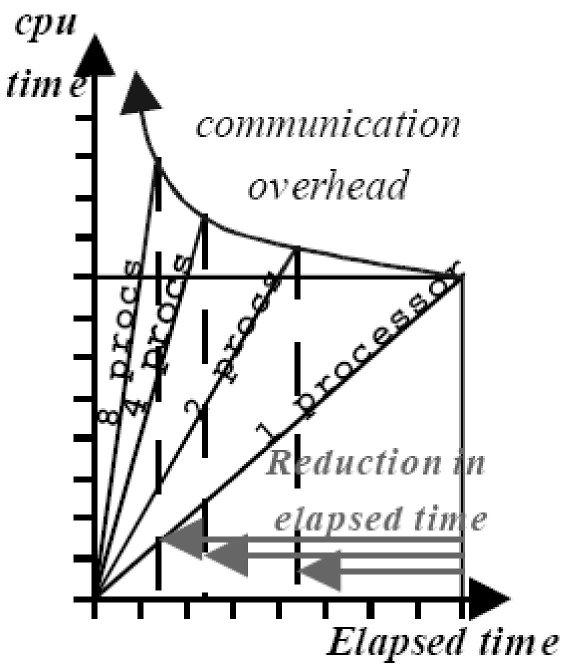


Figure 1.1 Elapsed time compared to CPU time[8]

The overhead due to parallel processing can stall the improvement in elapsed execution time. In some cases the overhead is such that it takes longer to run program on several processors. The overhead might come from algorithmic complexities, such as the necessity to exchange data between the computational threads or synchronise the threads.

The analysis of any parallel program is generally done by using scalability. Scalability is **defined as the ratio between the performance on one processor and n processors, or the respective execution times**. Sometimes efficiency is also used to analyse the results achieved by a certain parallelization. Efficiency is the ratio of speed up to the number of processors used for certain solution.

$S = \frac{T_1}{T_n}$ **(Eqn.1.1)**

Where

S: Speed up

T_n **:** Time of execution on n processors

T_1 : Time of execution on 1 processor

n: Number of processors

$E = \frac{S}{n}$ **(Eqn. 1.2)**

In terms of the speedup and efficiency, parallel programs might behave according to the different models presented in *Fig.1.*2 Ideal speedup is improvement of performance proportional to the number of processors n.

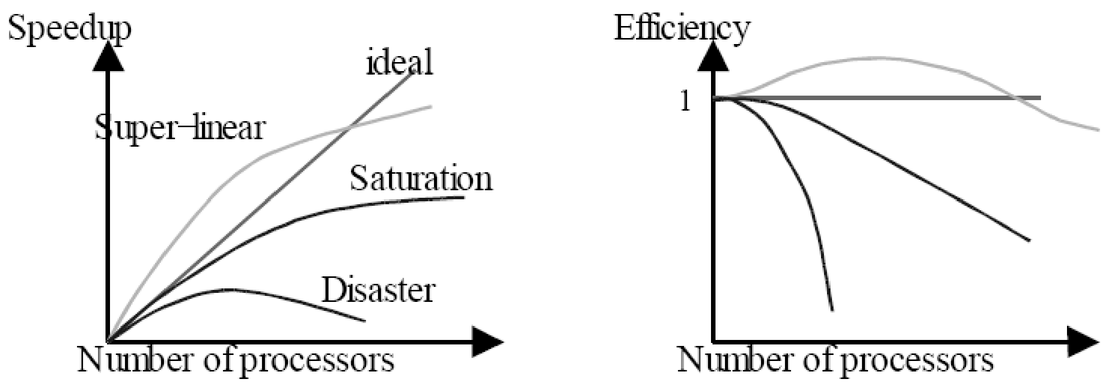


Figure 1.2 Speedup and Efficiency for parallel programs[8]

Saturation in performance increase is a common pattern when the overhead of parallel computation is comparable to the speedup gained from distributing the work load to processors. Sometimes when the number of processes are too large a decrease in performance can be seen, it happens due to the synchronisation required to complete the work. Moreover, sometimes there is not enough work to be distributed but the algorithm of the program assigns some memory to those processes and hence the elapsed time increases. The most sought after speedup is the linear one but it is not the case always, infact there is a cap on the speedup achievable for any serial program. The Amdahl's law explained in the next section explains as to why any program can be made faster only to a certain extent.

1.2 Amdahl’s law[1]

Amdahl's law, named after computer architect Gene Amdahl, is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors. The generalized Amdahl's law is: $S \leq \frac{1}{1 - f_p}$ **(Eqn. 1.3)**

- S: Maximum possible scalability
- f_p : Value of parallel fraction in a program

Another way of presenting the Amdahl's law is:

$$S \leq \frac{1}{\sum_{k=0}^{n} \frac{P_k}{U_k}}$$ **(Eqn. 1.4)**

Where,

- S: Scalability
- P_k : is a percentage of the instructions that can be improved (or slowed)
- U_k : is the speed-up multiplier (where 1 is no speed-up and no slowing)
- k : represents a label for each different percentage and speed-up
- n : is the number of different speed-up/slow-downs resulting from the system change

1.2.1 Description

Amdahl's law is a formula that computes the expected speedup of parallelized implementations of an algorithm relative to the non-parallelized algorithm. For example, if a parallelized implementation of an algorithm can run 12% of the algorithm's operations arbitrarily fast (while the remaining 88% of the operations are not parallelizable); Amdahl's law states that the maximum speedup of the parallelized version is $\frac{1}{1-0.12} = 1.136$ times faster than the non-parallelized implementation. More technically, the law is concerned with the speedup achievable from an improvement to a computation that affects a proportion *P* of that computation where the improvement has a speedup of *S*. (For example, if an improvement can speed up 30% of the computation, *P* will be 0.3; if the improvement makes the portion affected twice as fast, *S* will be 2). Amdahl's law states that the overall speedup of applying the improvement will be

$$\frac{1}{(1-P)+\frac{P}{S}}$$ **(Eqn. 1.5)**

P: is a percentage of the instructions that can be improved (or slowed)

S: is the speed-up multiplier (where 1 is no speed-up and no slowing)

To see how this formula was derived, assume that the running time of the old computation was 1, for some unit of time. The running time of the new computation will be the length of time the unimproved fraction takes, (which is $1 - P$), plus the length of time the improved fraction takes. The length of time for the improved part of the computation is the length of the improved part's former running time divided by the speedup, making the length of time of the improved part P/S. The final speedup is computed by dividing the old running time by the new running time, which is what the above formula does. Here's another example. We are given a task which is split up into four parts: P1 = .11 or 11%, P2 = .18 or 18%, P3 = .23 or 23%, P4 = .48 or 48%, which add up to 100%. Then we say P1 is not sped up, so S1 = 1 or 100%, P2 is sped up 5x, so S2 = 5 or 500%, P3 is sped up 20x, so S3 = 20 or 2000%, and P4 is sped up 1.6x, so S4 = 1.6 or 160%. By using the formula $\frac{P_1}{S_1}+\frac{P_2}{S_2}+\frac{P_3}{S_3}+\frac{P_4}{S_4}$ **(Eqn. 1.6)**

We find the running time is $\frac{0.11}{1}+\frac{0.18}{5}+\frac{0.23}{20}+\frac{0.48}{1.6}=0.4575$ or a little less than ½ the original running time, which we know is 1. Therefore the overall speed boost is $\frac{1}{.4575}=2.186$,a little more than double the original speed using the formula $\frac{1}{\frac{P_1}{S_1}+\frac{P_2}{S_2}+\frac{P_3}{S_3}+\frac{P_4}{S_4}}$

Clearly, the 20x and 5x speedup don't have much effect on the overall speed boost and running time when over half of the task is only sped up 1x, (i.e. not sped up), or 1.6x.

1.3 Helmholtz Equation

Sound essentially is pressure fluctuations in any medium, satisfying some constraints any medium can be solved for the pressure field by the equation written below

$$\nabla^2 p(\mathbf{x},t) = \frac{1}{c^2}\frac{\partial^2 p(\mathbf{x},t)}{\partial t^2} \qquad \textbf{(Eqn.1.7)}$$

$p(\mathbf{x},t)$ is the acoustic pressure at position **x** at time t,

c is the velocity of sound in the medium of propagation.

Equation (1.7) is valid for the propagation of the acoustic waves in a constant density medium where no bulk flow is present.

Most of the acoustic sources are harmonic with respect to time, this makes the field harmonic in the time domain i.e. $p(\mathbf{x},t)$ can be presented in

$$p(\mathbf{x},t) = p(\mathbf{x})\,e^{-j\omega t} \qquad \textbf{(Eqn.1.8)}$$

$p(\mathbf{x})$ is the spatial pressure field

$\omega = 2\pi f$ is the frequency of the fluctuation

Substituting *equation (1.8)* in to *(1.7)* we get the equation written below

$$\nabla^2 p(\mathbf{x}) + k^2 p(\mathbf{x}) = 0 \qquad \textbf{(Eqn.1.9)}$$

$k = \dfrac{\omega}{c}$ is known as the wave number. *Equation (1.7)* is known as the Helmholtz equation, it is a second order linear differential equation which can be solved for simple systems. The boundary conditions for the system become important as their knowledge can be used to solve the Helmholtz equation to get the spatial pressure field. More often than not the complexity of the geometry forces the analytical problem to be very difficult and hard to solve.

1.4 Numerical Methods for Helmholtz equation

In most of the practical engineering problems analytical solution to Helmholtz equation is rendered very difficult due to the factors mentioned in the last section. Many numerical methods have been developed over the years to counter this problem; numerical methods tend to transform the Helmholtz equation to a set of different linear algebraic equations of the form

$$\mathbf{Ax} = \mathbf{b} \qquad \textbf{(Eqn.1.10)}$$

Where **A** is a square matrix, **b** is the forcing vector obtained by the boundary value data and **x** is the solution of the acoustic domain. Once **A** and **b** are determined by the geometry and boundary conditions, the unique solution **x**, which is a close approximation to the exact solution can be obtained. The order of the matrix and vectors depends upon the various parameters used in any particular numerical method. Some of the conventional numerical methods are:

- Finite Difference method
- Finite Element method
- Boundary Element method

The effectiveness of the numerical methods can be gauged by the fact that a differential equation is transferred to a linear algebraic one. The pressure field is transferred to discrete pressure points in the domain; generally **A** and **b** are related to the stiffness and source of domain respectively. The first look at *equation (1.10)* may give the impression of it to be very simple to solve, on the contrary its computational challenges are immense and widespread. The order of the matrix and vector of a domain may run into a large number of unknowns, the problem is also compounded by the fact that the matrix also contains complex values. All over the world mathematicians [2] have worked hard to solve the linear system of equations in the most easy and effective ways. There are standard iterative algorithms [2] available to solve this linear system of equations, these algorithms fall in the category of Numerical methods, some of these standard iterative methods are mentioned below:

- GMRES (Generalized minimal residual method)
- Jacobi method
- BiCG (Bi-conjugate gradient method)
- BiCGSTAB (Biconjugate Gradient Stabilization method)
- MINRES (Minimal Residual method)

The parallel solution of Helmholtz equation has its roots in the availability and programmability of the above mentioned standard algorithms. One can program to run these numerical methods on various processors giving the solution in a fast and effective manner. The rate at which an iterative method converges depends greatly on the spectrum of the coefficient matrix A. Hence, iterative methods usually involve a second matrix that transforms the coefficient matrix into one with a more favorable spectrum. The transformation matrix is called preconditioner. A good preconditioner improves the convergence of the iterative method, sufficiently to overcome the cost of constructing and applying the preconditioner. Indeed, without a preconditioner the iterative method may even fail to converge. In the next section an overview of all the major iterative methods has been given besides explaining two methods.

Chapter 2: Literature Review

2.1 Parallel computing 10

2.2 Parallel computing systems 11

2.3 Parallel programming models 15

2.3.1 Shared memory model 15

2.3.2 Threads model 16

2.3.3 Message passing model 18

2.3.4 Data parallel model 19

2.3.5 Hybrid 20

2.4 Iterative Methods for Helmholtz equation 20

2.4.1 Overview of methods 20

2.4.2 GMRES 21

2.4.3 BiCG 23

Chapter 2 Literature Review

2.1 Parallel Computing

Parallel computing can be defined as simultaneous usage of multiple processors for solving the same problem so that the result can be achieved faster. The concept of parallelization comes from the belief that any problem can be divided into smaller tasks which can be solved independent of each other but with some coordination. Generally, any program works by dividing itself into a group of instructions which are executed serially in an order (see *Fig. 2.1)*.

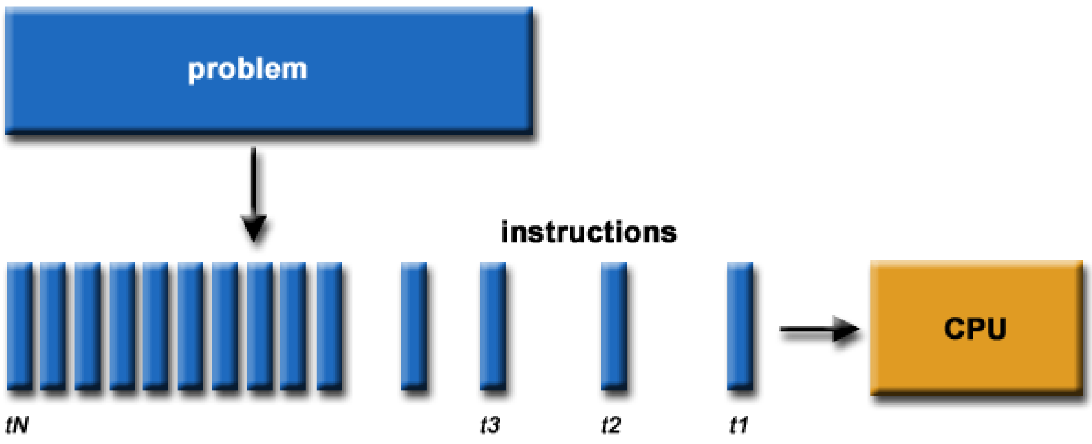


Figure 2.1 Serial computing[3]

The only and most important drawback with serial computing is the inability to process more than one instruction at a time. This results in time losses when the computational limits of a single processor are stretched to its limits. In simplest sense a parallelized program does what a serial one is not capable of doing. A problem is broken into various discrete parts that can be solved concurrently; these parts are made to form a series of instructions which are then carried out at the same time on different processors. The computer resources utilized can be of the three types mentioned below.

- A single computer with multiple processors
- An arbitrary number of computers connected by a network
- Any combination of the above two

A simple schematic representation of the parallel computing is shown in the *Fig 2.2*

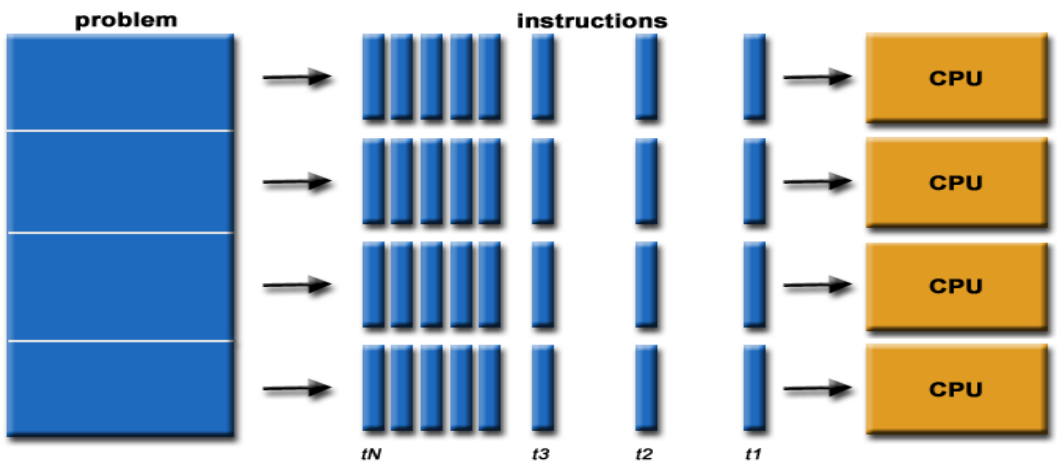


Figure 2.2 Parallel computing[3]

Any computational problem needs to have the following properties to be made parallel

- Broken apart into coherent pieces of work
- Multiple program instructions can be executed by the division
- Solved in less time with multiple computer resources than with single resource

Generally, parallel computing has been considered to be the high end of computing fed by the need of numerical simulations for highly complex problems. Weather forecasting, nuclear reactions and space technology are just some of the fields which led to pioneering research and development in parallel computing.

2.2 Parallel computing systems

A parallel computing systems consists of resources where more than one processor is at disposal for computational needs. There are many different types of parallel systems. They are distinguished by the kind of interconnection between processors and memory. One of the most accepted terminologies of parallel architecture classifies these systems according to

- whether all processors execute the same instructions at the same time (*single instruction/multiple data* --SIMD) or
- each processor executes different instructions (*multiple instruction/multiple data* –MIMD)

One other way of dividing the parallel systems is based on the memory architecture. Shared memory parallel systems (See *Fig.2.3)* have multiple processors sharing the same global address space. Changes affected by one processor in the memory space are visible to other processors as well. The shared memory parallel systems can further be divided into two main classes based upon memory access times.

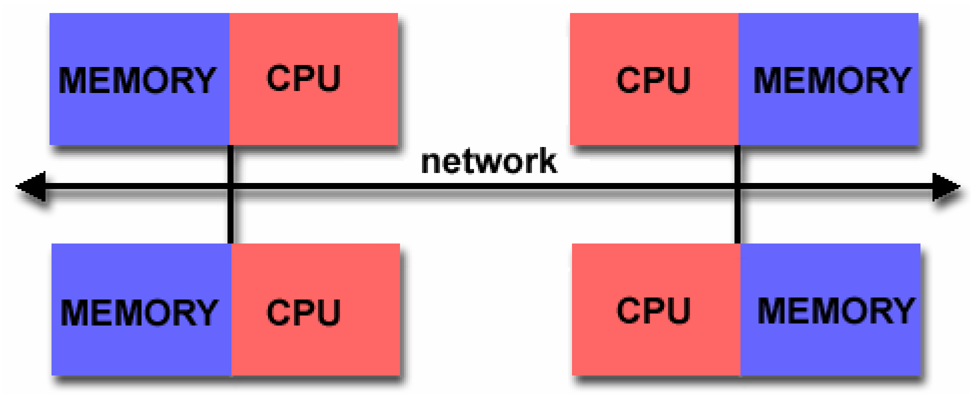


Figure 2.3 Shared Memory systems[3]

❖ UMA(Uniform Memory Access):It consists mainly of a combination of Symmetric multiprocessor machines (SMPs), all the processors are identical. Moreover, the access time to the shared memory is equal for all the processors. Sometimes these types of systems are known as Cache- Coherent UMA. It is to say that one change made my any of the processor on the memory is easily visible to the rest of the other processors.

❖ NUMA(Non-uniform Memory Access):Non-uniform as the word suggests is referred to a system where memory access is not equal for all the processors. Normally these are made by connecting various SMPs (Symmetric multiprocessor machines), for this reason only the communication over two SMPs (Symmetric multiprocessor machines) is slower than communication inside SMPs (Symmetric multiprocessor machines). Shared memory systems have their advantages as well as disadvantages. The programming is easy and user-friendly and the data sharing is fast and uniform due to proximity of the memory. The primary disadvantages are the lack of scalability between memory and CPUs and the responsibility of synchronization for correct access to memory space lies with the programmer. Moreover it becomes difficult to make more and more SMPs with ever increasing CPUs.

Distributed memory systems (*Fig.2.4*) also run on multiple processors but every processor can only access its own local memory space, no global address space exists between them. The architectural differences between shared-memory multiprocessors and distributed-memory multiprocessors have implications on how each is programmed. With a shared-memory multiprocessor, different processors can access the same variables. This makes referencing data stored in memory similar to traditional single-processor programs, but adds the complexity of shared data integrity. A distributed-memory system introduces a different problem: how to distribute a computational task to multiple processors with distinct memory spaces and reassemble the results from each processor into one solution.

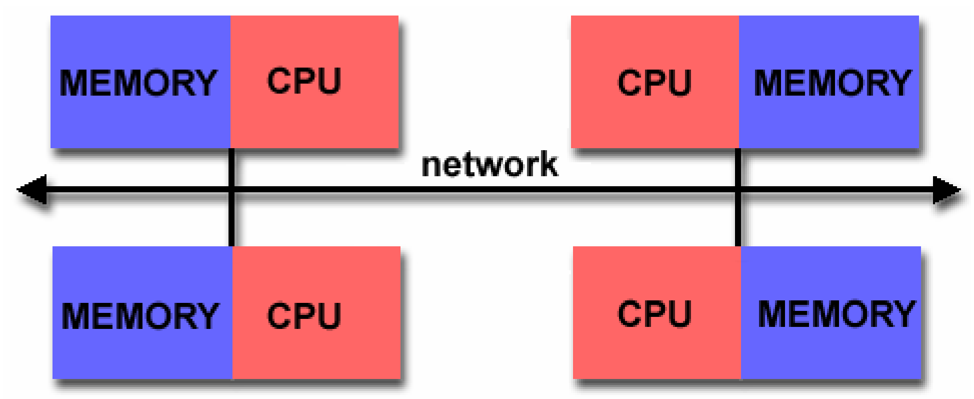


Figure 2.4 Distributed Memory systems[3]

When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility. The various advantages and disadvantages associated with Distributed memory systems are mentioned as follows. The most significant advantage with distributed systems is the direct proportionality between number of processors and memory availability. A x fold increase in processor will deliver a x fold increase in the memory available. Sometimes the cheap and cost-effectiveness of distributed systems outweighs the benefits enjoyed with Shared memory systems. The processors which are not being used for any purpose can be utilized by the parallel program. Memory access in a distributed environment is quite fast vis a vis the Shared memory systems. It is also very easy to develop large clusters of computers which are inexpensive and customizable. There are lot of flip-sides too associated with distributed environment, the programming becomes highly complex as the programmer has to keep track of all the communications between the processors. Moreover the access to the memory is no uniform in nature. It is an individual memory system which is providing the access hence it would be different from one processor to another.

These days the ideal way of parallel programming would be to incorporate a system utilizing both shared as well as distributed memory systems. Infact, most of the largest and fastest computers in the world employ both these architectures

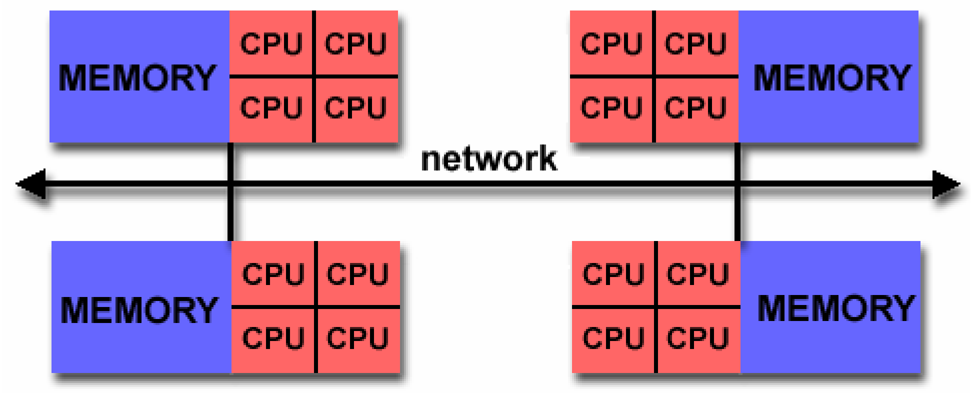


Figure 2.5 Latest parallel systems[3]

The shared memory component is usually a cache coherent SMP machine. Processors on a single SMP can refer to its memory as global, while distributed system is the networking of various SMPs (Symmetric Multiprocessor). The network communications are a requirement for data tranfer between the SMPs. This type of system incorporates both the advantages as well as inconveniences of all parallel architectures available.

2.3 Parallel programming models

There are various methods for doing the programming on parallel systems, as is clear from abovementioned texts that most significant part is to ensure the communication. There are several parallel programming models in common use:

- Message passing
- Data parallel
- Shared Memory
- Threads
- Hybrid

Parallel programming models exist as an abstraction above hardware and memory architectures. Although it might not seem apparent, these models are not specific to a particular type of machine or memory architecture. In fact, any of these models can (theoretically) be implemented on any combination of hardware. Which model to use is often a combination of what is available and personal choice? There is no "best" model, although there certainly are better implementations of some models over others. The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

2.3.1 Shared Memory Model

In this type of programming model since the global memory is common to all the processors the data is asynchronously read and written. Programmers use different mechanisms to control access to the shared memory. Program development can often be simplified. An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality. Moreover since the data sharing is done through memory address, the need for explicit communication is substituted.

2.3.2 Threads Model[3]

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

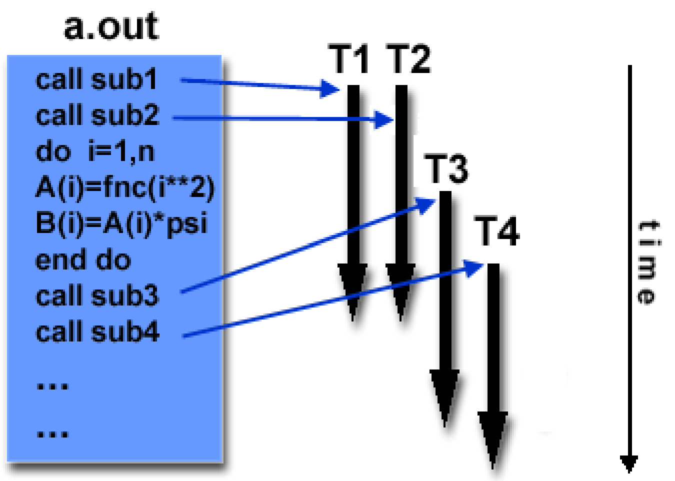


Figure 2.6 Thread based parallel model[3]

The main program **`a.out`** is scheduled to run by the native operating system. `a.out` loads and acquires all of the necessary system and user resources to run. `a.out` performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently. Each thread has local data, but also, shares the entire resources of `a.out`. This saves the overhead associated with replicating a program's resources for each thread. Each thread also benefits from a global memory view because it shares the memory space of `a.out`. A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads. Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to insure that more than one thread is not updating the same global address at any time. Threads can come and go, but `a.out` remains present to provide the necessary shared resources until the application has completed. Threads are commonly associated with shared memory architectures and operating systems.

Implementations:

From a programming perspective, threads implementations commonly comprise:

- A library of subroutines that are called from within parallel source code
- A set of compiler directives embedded in either serial or parallel source code

In both cases, the programmer is responsible for determining all parallelism.

Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP[3]***.

- **POSIX Threads**
  - Library based, requires parallel coding
  - Specified by the IEEE POSIX 1003.1c standard (1995).
  - C Language only
  - Commonly referred to as **Pthreads**.
  - Most hardware vendors now offer **Pthreads** in addition to their proprietary threads implementations.
  - Very explicit parallelism requires significant programmer attention to detail.
- **OpenMP[5]**
  - Compiler directive based can use serial code
  - Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP FORTRAN API was released October 28, 1997. The C/C++ API was released in late 1998.
  - Portable / multi-platform, including Unix and Windows NT platforms
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for "incremental parallelism"

2.3.3 Message Passing Model

The message passing model is essentially based on the division of problems into various number of tasks. Normally, each task works independently until data exchange is required. Communication forms the back-bone of this type of method, various types of send and receive messages are implemented to get the desired results. The tasks are executed on various local processes, the processes can be on different processors or the same processor. The name message passing is derived from the sharing of data that is done for complete execution of the job. Every process has its own local variables and by no means can any process access the memory of another one. The onus is on the programmer to divide the tasks intelligently and equally to utilize all the resources at disposal. Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation *Fig. 2.7*.

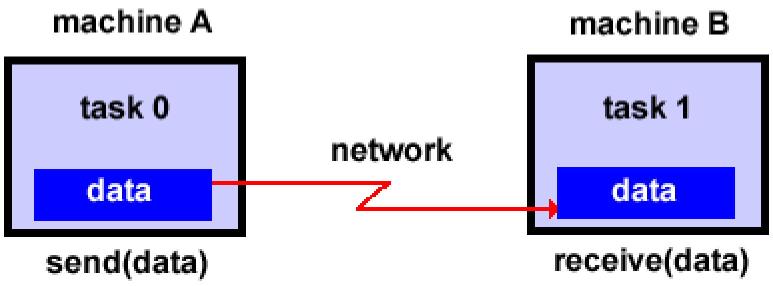


Figure 2.7 Message passing model[3]

Implementations:

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are embedded in source code. The programmer is responsible for determining all parallelism.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard inter-

face for message passing implementations.

- Part 1 of the **Message Passing Interface (MPI)[2]** was released in 1994. Part 2 (MPI-2) was released in 1996. Both MPI specifications are available on the web at www.mcs.anl.gov/Projects/mpi/standard.html.
- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. The latest implementation being MPICH2 [7].

2.3.4 Data Parallel Model

Data parallel model refers to the division of data into various groups. The divided data is sent to different tasks which then work on it. Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube *Fig 2.8*. A set of tasks work collectively on the same data structure, however, each task works on a different partition of the data structure. Generally, each of the task consists of same operation but implemented on different data sets. Distributed memory systems use this model by storing only a part of the data in the local memory. On shared memory architectures, all tasks may have access to the data structure through global memory.

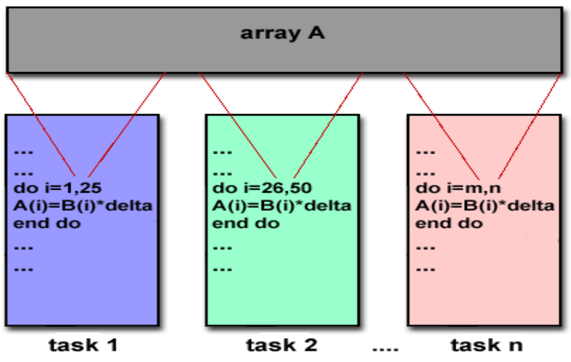


Figure 2.8 Data parallel model[3]

Implementations:

- Distributed memory implementations of this model usually have the compiler convert the program into standard code with calls to a message passing library (MPI usually) to distribute the data to all the processes. All message passing is done invisibly to the programmer.

2.3.5 Hybrid In this model, any two or more parallel programming models are combined. Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP). This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines. Another common example of a hybrid model is combining data parallel with message passing.

2.4 Iterative Methods for Helmholtz equation

2.4.1 Overview of the Methods

In this section some of the above mentioned numerical methods for the system of linear equations will be explained with a description on the class of matrices for which they are most suitable.

- The Jacobi method is based on the solving for every variable locally with respect to the other variables; single iteration of the method corresponds to solving for every variable once. The resulting method is easy to understand and implement, but convergence is slow.
- The MINRES method is used to generate a sequence of orthogonal vectors, these vectors are the residuals of iterates, it's an effective method when the coefficient matrix is symmetric but positively indefinite. The storage space required with this method is optimized to a great extent.
- The Generalized Minimal Residual method computes a sequence of orthogonal vectors (like MINRES), and combines these through a least-squares solve and update. However, unlike MINRES it requires storing the whole sequence, so that a large amount of storage is needed. For this reason, restarted versions of this method are used. In the restarted versions, computation and storage costs are limited by specifying a fixed number of vectors to be generated. This method is useful for general nonsymmetrical matrices.
- The Biconjugate Gradient method generates two sequences of vectors, one based on a system with the original coefficient matrix A, and one on A^T. Instead of orthogonalizing each sequence, they are made mutually orthogonal. This method uses limited storage and is very useful when the matrix is nonsymmetrical and nonsingular; however, convergence may be irregular, and the possibility that the method might break down. BiCG requires a multiplication with the coefficient matrix and with its transpose in all the iterations.
- The Biconjugate Gradient Stabilized method is a variant of BiCG but using different updates for the A^T- sequence in order to obtain smoother convergence.

The iterative algorithms generally employed for solving the Helmholtz equation would have to satisfy certain conditions for the coefficient matrix. The coefficient matrix formed after using Numerical methods is sparse, nonsymmetric and positive definite. The iterative methods suitable for this system of linear equations obtained by the WEM are GMRES and Big STAB, the GMRES method is explained as follows

2.4.2 GMRES

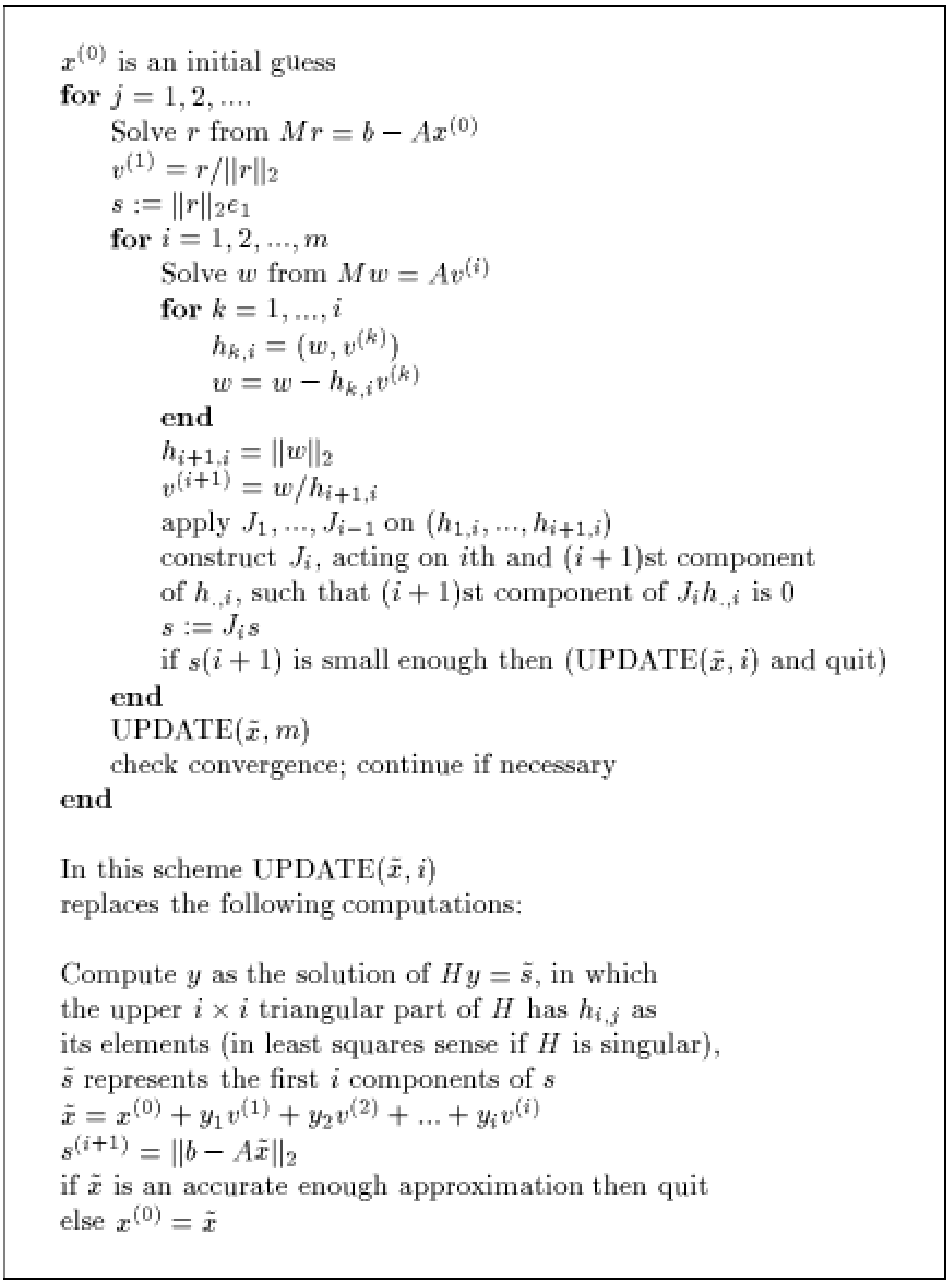
The Generalized Minimal Residual method is an extension of MINRES (which is only applicable to symmetric systems) to the unsymmetrical systems. It generates a sequence of orthogonal vectors, but in the absence of symmetry this can no longer be done with short recurrences; instead, all previously computed vectors in the orthogonal sequence have to be retained. For this reason, "restarted" versions of the method are used.

Theory

The Generalized Minimal Residual method is designed to solve nonsymmetrical linear systems. The most popular form of GMRES uses restarts to control storage requirements.

If no restarts are used, GMRES will converge in no more than n steps (assuming exact arithmetic). Of course this is of no practical value when n is large; moreover, the storage and computational requirements in the absence of restarts are prohibitive. Indeed, the crucial element for successful application of GMRES revolves around the decision of when to restart; that is, the choice of m. There are examples for which the method stagnates and convergence takes place only at the n^{th} step, for these kinds of systems any choice of m less than n fails to converge.

The major drawback of GMRES is that the amount of work and storage required per iteration rises linearly with the iteration count. In case one is fortunate enough to obtain extremely fast convergence, the cost will rapidly become prohibitive. The restarted version to some extent has been able to overcome this problem, after a chosen number of iterations the accumulated data are cleared and the intermediate results are used as the initial data for the next m iterations. The procedure is repeated until convergence is achieved. The difficulty is on choosing an appropriate value for m. If m is "too small" GMRES may be slow to converge, or fail to converge entirely. A value of m that is larger than necessary involves excessive work (and uses more storage). Unfortunately, there are no definite rules governing the choice of m choosing when to restart is a matter of experience. The pseudo code for GMRES method is explained with the use of a preconditioner M in the *Fig 2.10*



```
$x^{(0)}$ is an initial guess
for $j = 1, 2, ....$
    Solve $r$ from $Mr = b - Ax^{(0)}$
    $v^{(1)} = r/\|r\|_2$
    $s := \|r\|_2 e_1$
    for $i = 1, 2, ..., m$
        Solve $w$ from $Mw = Av^{(i)}$
        for $k = 1, ..., i$
            $h_{k,i} = (w, v^{(k)})$
            $w = w - h_{k,i} v^{(k)}$
        end
        $h_{i+1,i} = \|w\|_2$
        $v^{(i+1)} = w/h_{i+1,i}$
        apply $J_1, ..., J_{i-1}$ on $(h_{1,i}, ..., h_{i+1,i})$
        construct $J_i$, acting on $i$th and $(i+1)$st component
        of $h_{.,i}$, such that $(i+1)$st component of $J_i h_{.,i}$ is 0
        $s := J_i s$
        if $s(i+1)$ is small enough then (UPDATE($\tilde{x}, i$) and quit)
    end
    UPDATE($\tilde{x}, m$)
    check convergence; continue if necessary
end

In this scheme UPDATE($\tilde{x}, i$)
replaces the following computations:

Compute $y$ as the solution of $Hy = \tilde{s}$, in which
the upper $i \times i$ triangular part of $H$ has $h_{i,j}$ as
its elements (in least squares sense if $H$ is singular),
$\tilde{s}$ represents the first $i$ components of $s$
$\tilde{x} = x^{(0)} + y_1 v^{(1)} + y_2 v^{(2)} + ... + y_i v^{(i)}$
$s^{(i+1)} = \|b - A\tilde{x}\|_2$
if $\tilde{x}$ is an accurate enough approximation then quit
else $x^{(0)} = \tilde{x}$
```

Figure 2.10 GMRES pseudo code [2]

2.4.3 BiCG (Biconjugate Gradient method)

This method is a variant of the CGS(Conjugate Gradient) method which is not suitable for the unsymmetrical systems because the residual vectors cannot be made orthogonal with short recurrences. The GMRES method retains orthogonality of the residuals by using long recurrences, at the cost of a larger storage demand. The Biconjugate Gradient method takes another approach, replacing the orthogonal sequences, at the price of no longer providing a minimization

Theory

Few theoretical results are known about the convergence of BiCG. For symmetric positive definite systems the method delivers the same results as Conjugate Gradient (CG), but twice the cost per iteration. For nonsymmetrical matrices it has been shown that in phases of the process where there is significant reduction of the norm of the residual, the method is more or less comparable to full GMRES. In practice this is often confirmed, but it is also observed that the convergence behavior may be quite irregular, and the method may even breakdown. BiCG requires computing a matrix-vector product $Ap^{(k)}$ and a transpose product $A^Tp^{(k)}$. In some applications the latter product may be impossible to perform, for instance if the matrix is not formed explicitly and the regular product is only given in operation form, for instance as a function call evaluation.

In a parallel environment, the two matrix-vector products can theoretically be performed simultaneously; however, in a distributed –memory environment, there will be extra communication costs associated with one of the two matrix-vector products, depending upon the storage scheme for *A*. A duplicate copy of the matrix will alleviate this problem, at the cost of doubling the storage requirements for the matrix. Care must also be exercised in choosing the preconditioner, since similar problems arise during the two solves involving the preconditioning matrix.

It is difficult to make a fair comparison between GMRES and BiCG. GMRES really minimizes a residual, but at the cost of increasing work for keeping all residuals orthogonal and increasing demands for memory space. BiCG does not minimize a residual, but often its accuracy is comparable to GMRES, at the cost of twice the amount of matrix vector products per iteration step. However, the generation of the basis vectors is relatively cheap and the memory requirements are modest. Several variants of BiCG have been proposed that increase the effectiveness of this class of methods in certain circumstances. The pseudo code for this method is shown in the *Fig 2.11.*

```
Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$.
Choose $\tilde{r}^{(0)}$ (for example, $\tilde{r}^{(0)} = r^{(0)}$).
for $i = 1, 2, \ldots$
    solve $Mz^{(i-1)} = r^{(i-1)}$
    solve $M^T\tilde{z}^{(i-1)} = \tilde{r}^{(i-1)}$
    $\rho_{i-1} = z^{(i-1)^T}\tilde{r}^{(i-1)}$
    if $\rho_{i-1} = 0$, method fails
    if $i = 1$
        $p^{(i)} = z^{(i-1)}$
        $\tilde{p}^{(i)} = \tilde{z}^{(i-1)}$
    else
        $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
        $p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)}$
        $\tilde{p}^{(i)} = \tilde{z}^{(i-1)} + \beta_{i-1}\tilde{p}^{(i-1)}$
    endif
    $q^{(i)} = Ap^{(i)}$
    $\tilde{q}^{(i)} = A^T\tilde{p}^{(i)}$
    $\alpha_i = \rho_{i-1}/\tilde{p}^{(i)^T}q^{(i)}$
    $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
    $\tilde{r}^{(i)} = \tilde{r}^{(i-1)} - \alpha_i \tilde{q}^{(i)}$
    check convergence; continue if necessary
end
```

Figure 2.11 Pseudo code of Pre-Conditioned BiCG [5]

Chapter 3: Designing Parallel Programs

3.1 Problem Decomposition 26

3.1.1 Domain Decomposition 26

3.1.2 Functional Decomposition 27

3.2 Parallel Programming Issues 28

3.2.1 Key Factors 28

3.2.2 Load Balancing 30

3.2.3 Minimizing Communication 30

Chapter 3

Designing Parallel Programming

In order to improve the overall efficiency of any solver, it is imperative to solve larger, more memory intensive problems, or simply solve problems with greater speed than is possible on a serial computer. Parallel programming and parallel computers can be used to satisfy these needs. Using parallel programming methods on parallel computers gives access to greater memory and Central Processing Unit (CPU) resources not available on serial computers. Hence, one is able to solve large problems that may not have been possible otherwise, as well as solve problems more quickly. One of the basic methods of programming for parallel computing is the use of Message Passing Interface (MPI) libraries. These libraries manage transfer of data between instances of a parallel program running on multiple processors in a parallel computing architecture.

The topics to be discussed in this chapter are

- The difference between domain and functional decomposition.
- Some Parallel programming issues

3.1 Problem Decomposition

The first step in designing a parallel algorithm is to decompose the problem into smaller problems. Then, the smaller problems are assigned to processors to work on simultaneously. Roughly speaking, there are two kinds of decompositions.

1. Domain decomposition
2. Functional decomposition

These are discussed in the following two sections.

3.1.1 Domain Decomposition

In domain decomposition or "data parallelism", data is divided into pieces of approximately the same size and then mapped to different processors. Each processor then works only on the portion of the data that is assigned to it. Of course, the processes may need to communicate periodically in order to exchange data. Data parallelism provides the advantage of maintaining a single flow of control. A data parallel algorithm consists of a sequence of elementary instructions applied to the data: an instruction is initiated only if the previous instruction is ended. Single-Program-Multiple-Data (SPMD) follows this model where the code is identical on all processors. Such strategies are commonly employed in finite differencing algorithms where processors can operate independently on large portions of data, communicating only the much smaller shared border data in every iteration.

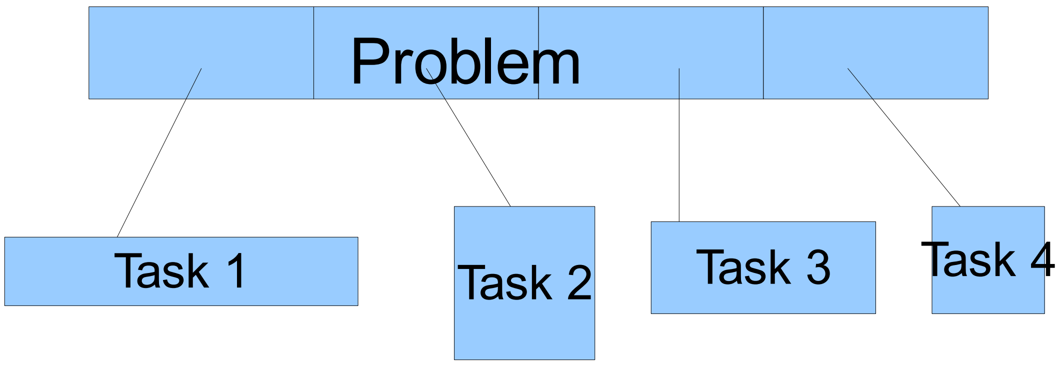


Figure 3.1 Domain decomposition

3.1.2 Functional decomposition

Frequently, the domain decomposition strategy turns out not to be the most efficient algorithm for a parallel program. This is the case when the pieces of data assigned to the different processes require different lengths of time to process. The performance of the code is then limited by the speed of the slowest process. The remaining idle processes do no useful work. In this case, functional decomposition (see *Fig. 3.2)* or "task parallelism" makes more sense than domain decomposition. In task parallelism, the problem is decomposed into a large number of smaller tasks and then, the tasks are assigned to the processors as they become available. Processors that finish quickly are simply assigned more work.

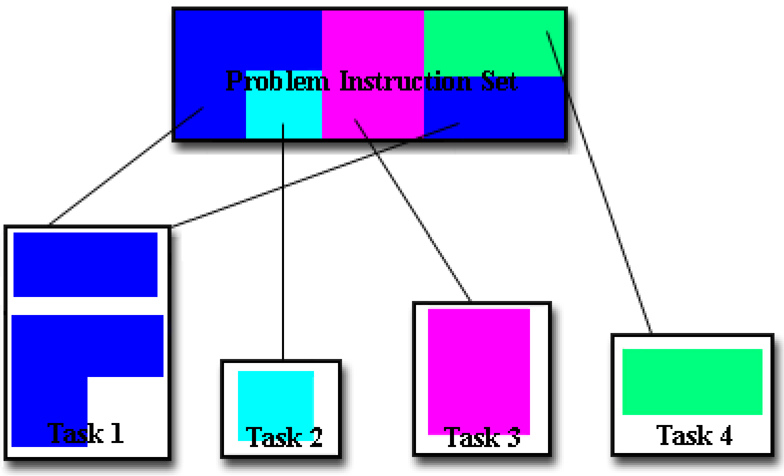


Figure 3.2 Functional decomposition[3]

Task parallelism is implemented in a client-server paradigm *see Fig 3.3*. The tasks are allocated to a group of slave processes by a master process that may also perform some of the tasks. The client-server paradigm can be implemented at virtually any level in a program. For example, if one simply wishes to run a program with multiple inputs, a parallel client-server implementation might just run multiple copies of the code serially with the server assigning the different inputs to each client process. As each processor finishes its task, it is assigned a new input. Alternately, task parallelism can be implemented at a deeper level within the code.

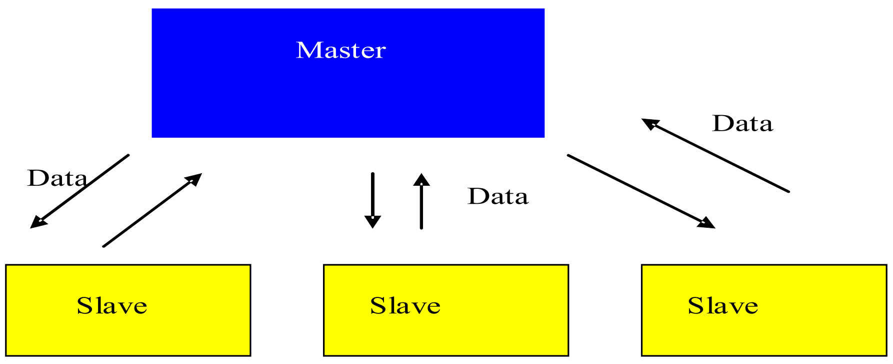


Figure 3.3 Client server paradigms

.

3.1 Parallel Programming Issues

Parallelization is not envisaged if communications is not possible; the extent of communication varies from problem to problem. The need for communications between tasks depends upon the problem:

Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward. Very little inter-task communication is required.

3.2.1 Key Factors[3]

There are a number of important factors to consider when designing your program's inter-task communications:

- **Communications cost**
  - Inter-task communication virtually always implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to

package and transmit data.
    - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
    - Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

- **Latency vs. Bandwidth**
    - ***Latency*** is the time it takes to send a minimal (0 byte) message from point A to point B, commonly expressed as microseconds.
    - ***Bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec.
    - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

- **Visibility of communications**
    - With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
    - With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even be able to know exactly how inter-task communications are being accomplished.
- **Synchronous vs. asynchronous communications**
    - Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
    - Synchronous communications are often referred to as ***blocking*** communications since other work must wait until the communications have completed.
    - Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
    - Asynchronous communications are often referred to as ***non-blocking*** communica-

tions since other work can be done while the communications are taking place.

  - Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

- **Scope of communications**
  - Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scoopings described below can be implemented synchronously or asynchronously.
  - ***Point-to-point*** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
  - ***Collective*** - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variations (there are more):

The main goal of writing a parallel program is to get better performance over the serial version. This thing in mind, there are several issues that you need to consider when designing your parallel code to obtain the best performance possible within the constraints of the problem being solved. These issues are

- load balancing
- minimizing communication

Each of these issues is discussed in the following sections.

3.2.2 Load Balancing

Load balancing is the task of equally dividing work among the available processes. This can be easy to do when the same operations are being performed by all the processes (on different pieces of data). It is not trivial when the processing time depends upon the data values being worked on. When there are large variations in processing time, you may be required to adopt a different method for solving the problem.

3.2.3 Minimizing Communication

Total execution time is a major concern in parallel programming because it is an essential component for comparing and improving all programs. Three components make up execution time:

1. Computation time
2. Idle time
3. Communication time

Computation time is the time spent performing computations on the data. Ideally, if one has N processors working on a problem then it should be finished in 1/Nth the time of the serial job. This would be the case if all the processors' time was spent in computation.
Idle time is the time a process spends waiting for the data from other processors. During this time, the processors do no useful work. An example of this is the ongoing problem of dealing with input and output in parallel programs. Many message passing libraries do not address parallel I/O, leaving all the work to one process while all other processes are in the idle state.
Finally, communication time is the time it takes for processes to send and receive messages. The cost of communication in the execution time can be measured in terms of latency and bandwidth. Latency is the time it takes to set up the envelope for communication, where bandwidth is the actual speed of transmission, or bits per unit time. Serial programs do not use interprocess communication. Therefore, one must minimize this use of time to get the best performance improvements.

Chapter 4: Parallel Programs

4.1 Distributed Parallel Programming 35

4.1.1 MPI Library 35

4.2 Matrix vector Multiplication 38

4.2.1 Serial Matrix vector multiplication 38

4.2.2 Row-wise Matrix vector multiplication 38

4.2.3 Block-wise Matrix vector multiplication 39

Chapter 4

Parallel Programs

From the previous sections it is quite clear that a Helmholtz equation is changed to a system of linear equations using numerical methods. The order of these equations is quite large and hence iterative methods should be used to solve them easily, moreover parallel programming gives us the chance to solve this big problem in fast manner. The programs were made in **C++** language while the library from which the parallel functions were imported was **MPI** (Message Passing Interface). A primary reason for the usefulness of this model is that it is extremely general. Essentially, any type of parallel computation can be cast in the message passing form. In addition, this model can be implemented on a wide variety of platforms, from shared-memory multiprocessors to networks of workstations and even single-processor machines. Generally allows more control over data location and flow within a parallel application than in, for example, the shared memory model. Thus programs can often achieve higher performance using explicit message passing. Indeed, performance is a primary reason why message passing is unlikely to ever disappear from the parallel programming world.

MPI stands for "Message Passing Interface". It is a library of functions (in C and C++) or subroutines (in FORTRAN) that you insert into source code to perform data communication between processes.

MPI is a standard for inter-process communication on distributed-memory multiprocessor. The standard has been developed by a committee of vendors, government labs, and universities [4]. Implementation of the standard is usually left up to the designers of the systems on which MPI runs, but a public domain implementation, MPICH, is available [7].

The execution model of a program written with MPI is quite different from one written with OpenMP. When an MPI program starts, the program spawns into the number of processes as specified by the user. Each process runs and communicates with other instances of the program, possibly running on the same processor or different processors. The greatest computational speedup will occur when processes are distributed among processors. Basic communication consists of sending and receiving data from one process to another, unlike OpenMP's thread communication via shared variables. This communication takes place over a high-speed network which connects the processors in the distributed-memory system.

A data packet sent with MPI requires several pieces of information: the sending process, the receiving process, the starting address in memory of the data to be sent, the number of data items being sent, a message identifier, and the group of processes that can receive the message. All of these items are able to be set by the programmer. For example, one can define a group of processes, and then send a message only to that group. Some collective communication routines do not require all of items. For example, a routine which allows one process to communicate with all other processes in a group when called by each of those processes would not require the specification of a receiving process since every process in the group should be a receiver.

In the simplest MPI programs, a master process sends off work to worker processes. Those processes receive the data, perform tasks on it, and send the results back to the master process which combines the results. More complex coordination schemes are possible with MPI, but they introduce new challenges. *Fig 4.1* shows the execution model of a basic MPI program.

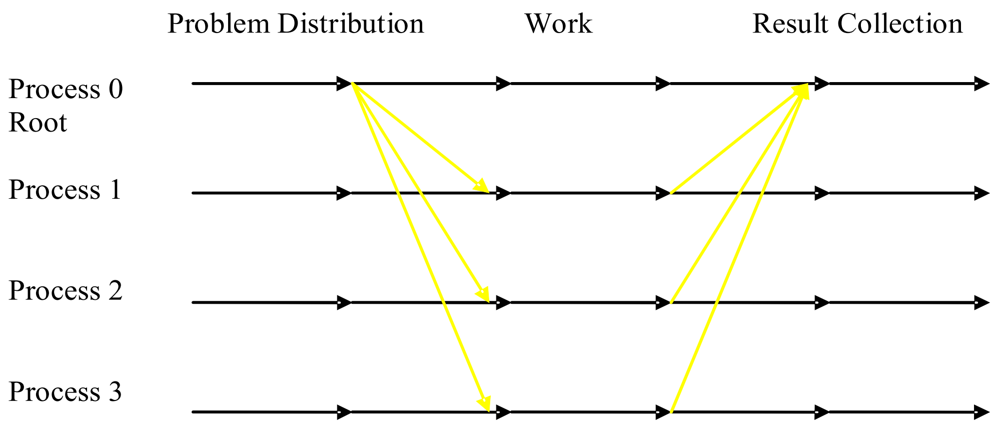


Figure 4.1 MPI execution model

The primary goals addressed by MPI are:

- Provide source code portability, MPI programs should compile and run as-is on any platform.
- Allow efficient implementations across a range of architectures

MPI also offers

- A great deal of functionality, including a number of different types of communication, special routines for common "collective" operations, and the ability to handle user-defined data types and topologies.

Some things that are outside the scope of MPI are

- The precise mechanism for launching an MPI program. In general this is platform-dependent and language used dependent.
- Dynamic process management that is, changing the number of processes while the code is running.

One should use MPI when one needs any one of the following

- Write a portable parallel code
- Achieve high performance in parallel programming, e.g. when writing parallel libraries.

Clearly there are cases when MPI is not required

- Can achieve sufficient performance and portability using a data-parallel or shared-memory approach.
- Parallelism is not required at all; using it when the requirement is almost zilch would make the code complex and hard for the users to understand.

4.1 Distributed memory programming

4.1.1 MPI Library

One of the biggest challenges in programming a distributed-memory multiprocessor is implementing efficient inter-process communication. Communication is not limited to the simple master-worker relationship shown in *Fig 4.1*. It may very well be the case that a process requires data or computed results from any other process during execution. It may also be the case that each process requires the same data sent from a single process, or that all processes require data from all the other processes. Ensuring process synchronization in these cases adds a level of complexity to programs developed on distributed-memory multiprocessors. Making communication efficient, that is, minimizing the overhead involved in message passing, adds further complexity. MPI provides many communication routines to aid the programmer in developing inter-process communication. These routines include:

- Barriers - points within a program where each process waits until all other processes get there. When this occurs, each process resumes execution.
- Blocking sends and receives - message passing routines which cause a process to wait until a message is sent or received before continuing execution
- Non-blocking sends and receives - message passing routines which do not cause a process to wait until a message is sent or received before continuing execution
- Collective communications *Fig. 4.2* - messages sent in one of three ways: one process sends other processes in a process group a message, all processes in a group send messages to all processes in the same group, and all processes in a group send a message to one process

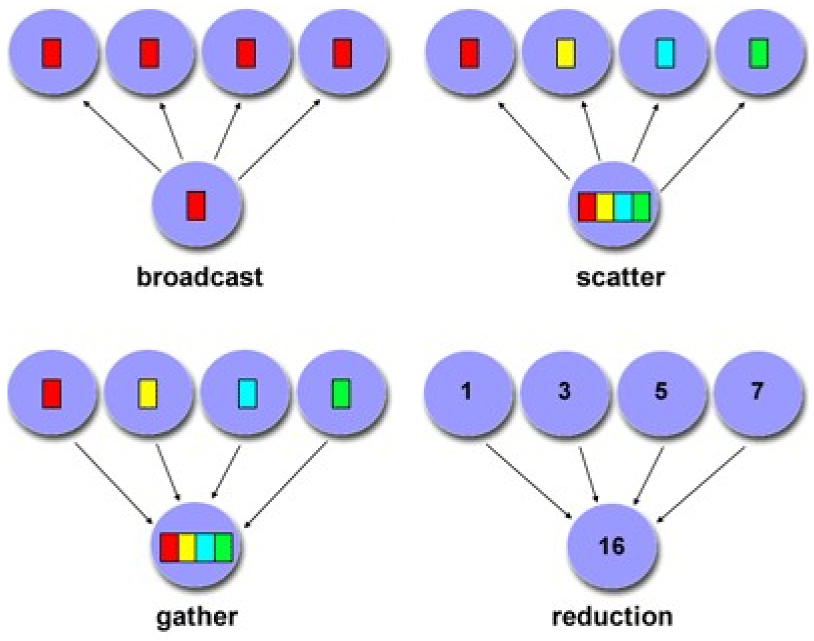


Figure 4.2 Collective communications[3]

MPI Routine	Use
MPI_Barrier	Cause all processes in a group to block until all other processes reach this routine
MPI_Send	Send a process data; block until received
MPI_Recv	Receive data from another process; block until sent
MPI_Isend	Send a message; do not block
MPI_Irecv	Receive a message; do not block
MPI_Probe	See if a message is waiting; block until message is detected
MPI_Iprobe	See if a message is waiting; does not block
MPI_Bcast	Send data to all processors in a group
MPI_Reduce	Collect a variable from all processors in a group with a combination operation
MPI_Allreduce	All processes receive the reduction variable after it has been combined
MPI_Gather	Collects data from all processes in a group into an array. Data is of same size
MPI_Allgather	All processes receive collected data from all other processes in a group. Data is of same size
MPI_Scatter	Distribute data to all processes in a group. Data is of same size
MPI_Gatherv	Collects data from all processes in a group into an array. Data may have different sizes
MPI_Scatterv	Distribute data to all processes in a group. Data may have different sizes
MPI_Alltoall	Distribute data to all processors in a group from all processes in the group. Data is of same size
MPI_Alltoallv	Distribute data to all processors in a group from all processes in the group. Data may have different sizes

Table 4.1 Common MPI subroutines [6]

The primary goal of this project was to study the parallelization on WINDOWS based distributed memory computers. After due consideration to the available resources and time frame, the MPICH[7] version of the open source parallel library was chosen for this task. The approach to programming has been explained in *Chapter 2* and *3.* A list of commonly used subroutines (FORTRAN) or functions(C and C++) are presented in the *table 4.1*. The language chosen for programming was C++ and the compiler used is Microsoft Visual C++. The easiest way to start with parallelizing a program was to take up a matrix vector multiplication. The method of programming and issues related to it are tackled in the next section.

4.2 Matrix vector multiplication

As we have seen earlier (*Chapter 1*) that solving Helmholtz equation (*Equation 1.7)* requires us to do a matrix vector multiplication hence the solution of this becomes an imperative issue. The study of parallelization effects are done by parallelizing this multiplication using two different algorithms. The elapsed time by a serial program is compared with times taken by the different algorithms. The results invariable follow the Amdahl's law mentioned in *Chapter 1.*

4.2.1 Serial matrix vector multiplication

Serial matrix vector multiplication refers to the solution done using single processor on a computer. The complete code for this can be found in *Appendix C*. The code follows a simple rule of forming a buffer space for answer vector and then goes on tackling the matrix row-wise. Each row is multiplied to the vector one at a time and the answer is stored in the buffer formed initially to store the answer

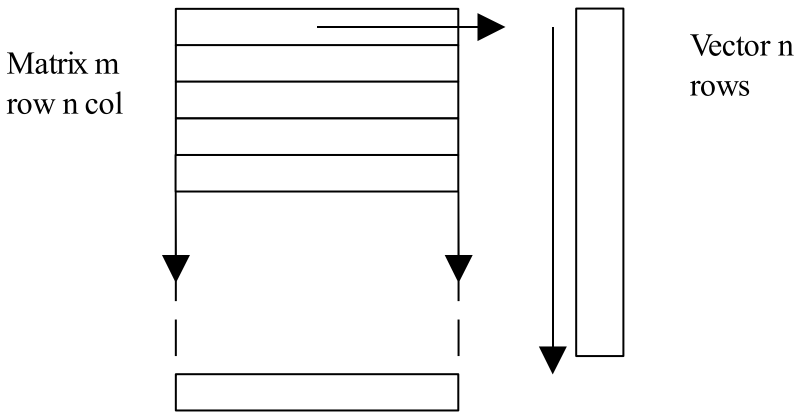


Figure 4.3 Serial Matrix vector multiplications

4.2.2 Row-wise parallel matrix vector multiplication

The code to matrix vector multiplication using this type of algorithm is available in *Appendix B*. The only difference between this program and the serial one is division of program into mutually independent tasks in the parallelized version. The strategy still remains the same of multiplying each row to the vector, though in this case it is done at the same time but independently. This algorithm mainly utilizes domain decomposition; the initial matrix is divided into data sets row-wise. Now clearly this will divide the whole problem into **m** number of parallel tasks, where **m** is the number of rows in the matrix. The number of processes will decide the number of simultaneous task completion done by this method. Obviously there is overhead associated with this algorithm also, each task has to be sent to the different process and memory requirements also shoot up as the buffer space is dynamic and has to be intimated to other processes. The vector too is sent to each and every process, the I/O is not parallel and hence initially some of the processes are left idle.

4.2.3 Block-wise parallel matrix vector multiplication

The code for this algorithm is presented in the *Appendix A.* One of the disadvantages that the previous algorithm suffers is the unequal load distribution. The load is divided on the availability of the processes, i.e. when one of the processes has completed one of the row vector multiplication then and only then will the master process check for the availability of tasks. In this algorithm the tasks are equally formed at the start or domain decomposition is done. The task formation in itself is dependent on the number of processes.

In order to have a good load balance it is necessary to divide the matrix into blocks of equal size. The size should be the size of the matrix divided by the number of processes. Because the processes have to multiply these pieces with a vector again, the blocks should be contiguous.

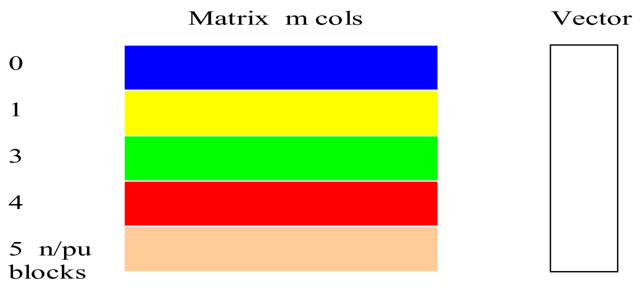


Figure 4.4 Block-wise matrix vector multiplication

As can be seen from *Fig. 4.3* the matrix has been divided into 5 blocks of contiguous nature. Each block with the vector is then sent to the corresponding process where multiplication is done and the result is sent back to the master process. The master process hence divides as well as assembles the solution for this problem. It is quite clear that the communication overhead will increase with the increasing number of processes.

Chapter 5: Results

5.1 Introduction 41

5.2 Program Results 41

5.2.1 Serial Matrix vector 41

5.2.2 Row-wise matrix vector multiplication 42

5.2.3 Block-wise matrix vector multiplication 45

Chapter 5

5.1 Introduction

The three codes made to test the parallelism have been attached in the *Appendices*. The codes were made and run on a twin processor WINDOWS based computers with configuration of 2MB RAM and 3 GHz processor speed. The two computers were installed with MPICH[6] and a parallel program was made on one of the computer. The execution of program requires the executable to be copied on all the systems being used for the calculation purposes. MPICH is readily available on the internet for free download with the installation instructions necessary for WINDOWS based platforms. The use of MPI users manual cannot be neglected to say the least, it provides with a sound knowledge of various functions/subroutines to go about the parallel program.

5.2 Program results

All the three programs are essentially matrix vector multiplication but done using different algorithms and hence quite easily the time of execution changes with each of the program. The whole objective is to reduce the time it takes on a serial execution and then analyse the extent to which it can be parallelized.

5.2.1 Serial matrix vector

The serial matrix vector multiplication is easily the easiest and most common approach to do the program. A single processor was needed to execute the program provided in *Appendix C*, a matrix of the size 1000$\times$ 1000 was multiplied with an appropriate vector. The average time taken for this exercise comes out to be almost 170 seconds. The *table 5.1* shows the various times taken for the serial matrix vector multiplication.

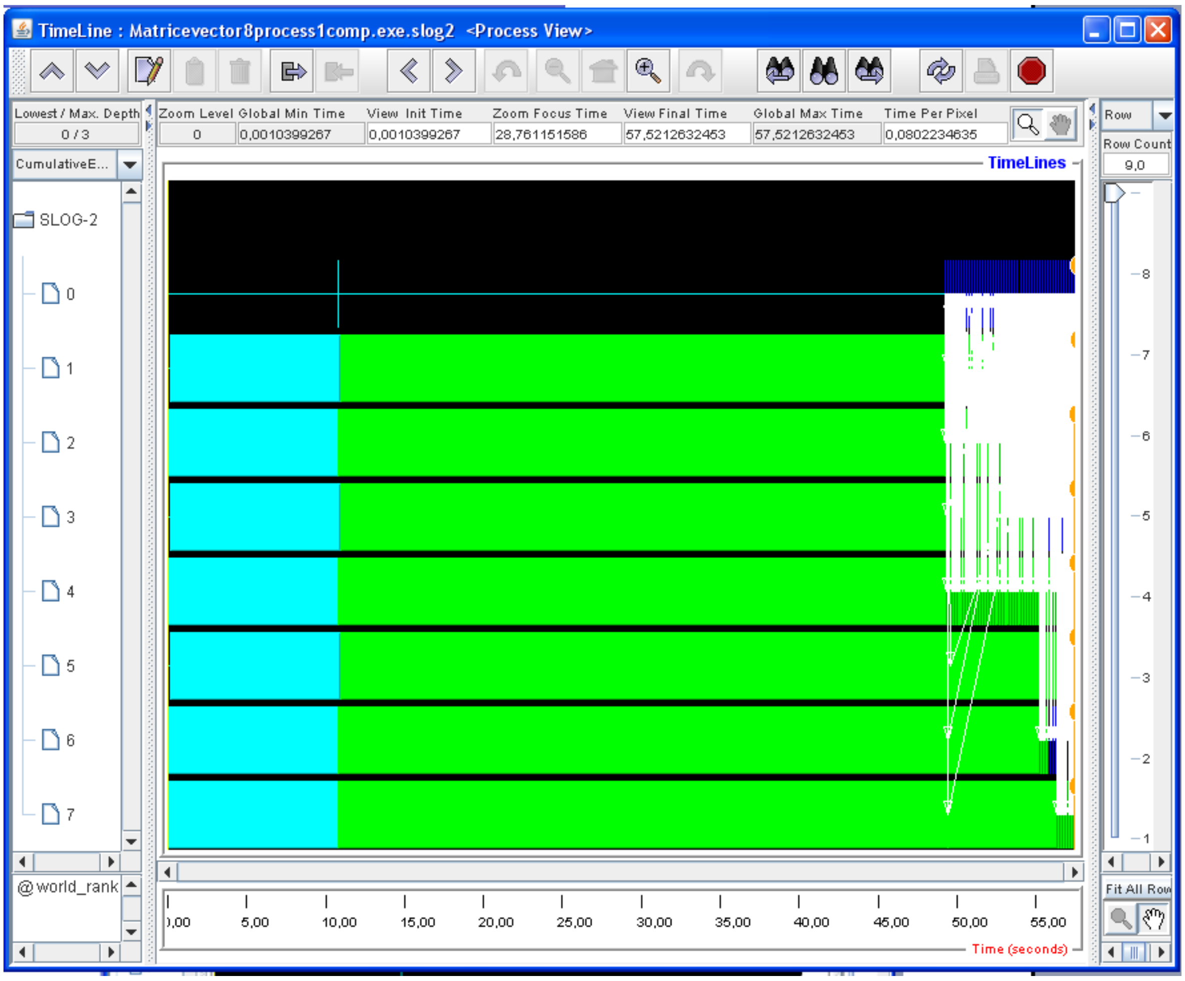
No. of processes	No. of processor	Time taken (seconds)	Average time(seconds)
1	1	167,25	170,15
1	1	169,34	
1	1	171,23	
1	1	172,99	
1	1	170,23	
1	1	169,87	

Table 5.1 Serial matrix vector multiplication

As can be seen from the *table 5.1* the program was run on a single processor machine with the program accounting for one single task. The use of MPICH gives us the flexibility of using more than one processor and formation of more than one task for the same problem. The results of parallelization will be found in the next sections.

5.2.2 Row-wise matrix vector multiplication

The algorithm of row-wise matrix vector multiplication was explained in *Chapter 4*. This program was executed on a twin processor computer and then in combination with a four processor distributed memory system. The time taken for the execution of this program with various combinations is provided in the *table* 5.2. The use of MPICH libraries also enables user to use the log file made to look through the various communications done during the execution of the program. The use of log file tells the user about the work done by the various processes during the timeline history. A sample log file picture will look like *Fig. 5.1*


TimeLine : Matricevector8process1comp.exe.slog2 <Process View>

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 3	0	0,0010399267	0,0010399267	28,761151586	57,5212632453	57,5212632453	0,0802234635

Row Count: 9,0

Figure 5.1 Sample Process view

The log file is known as Process view and gives us the good knowledge regarding load balancing and communication overheads if any are present in the algorithm.

No. of processes	No. of processor	Time taken(sec)	Average time(sec)	Speed up
2	2	30,12	30,23	5,62
		30,5		
		30,25		
		30,03		
		30,24		
3	2	31,17	31,82	5,34
		31,59		
		33,15		
		31,55		
		31,65		
4	2	32,03	31,97	5,32
		31,58		
		31,67		
		32,46		
		32,11		
6	2	61,01	53,57	3,17
		59,67		
		57,54		
		32,02		
		57,6		
8	2	60,96	59,86	2,84
		59,22		
		61,97		
		57,52		
		59,62		
3	3	30,35	29,81	5,7
		29,51		
		29,74		
		29,84		
		29,6		
4	4	29,32	29,8	5,71
		30		
		29,96		
		29,89		
		29,81		
6	4	32,47	33,04	5,15
		32		
		35,11		
		33,1		
		32,52		

Table 5.2 Row-wise matrix vector parallel multiplication

The speed up Vs Number of processes chart was made for this particular program and shown in *Fig. 5.3*

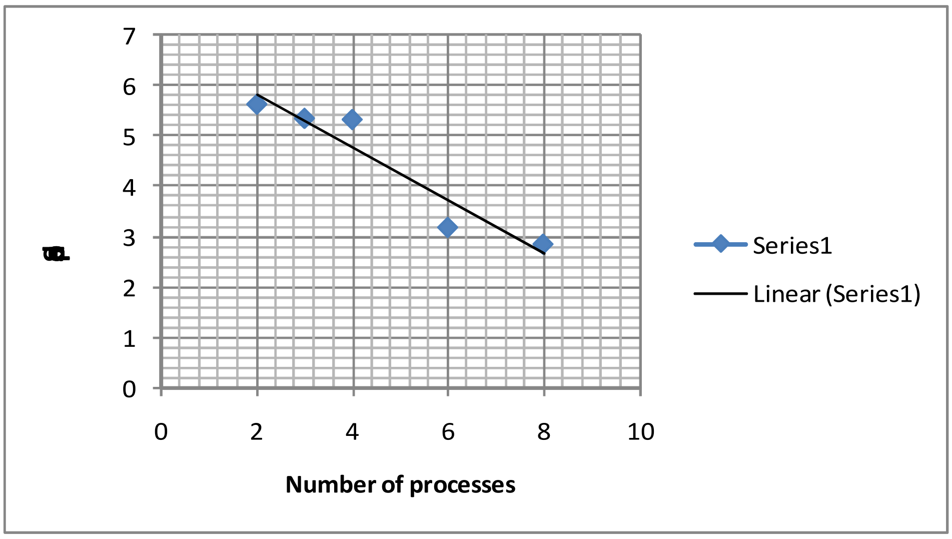
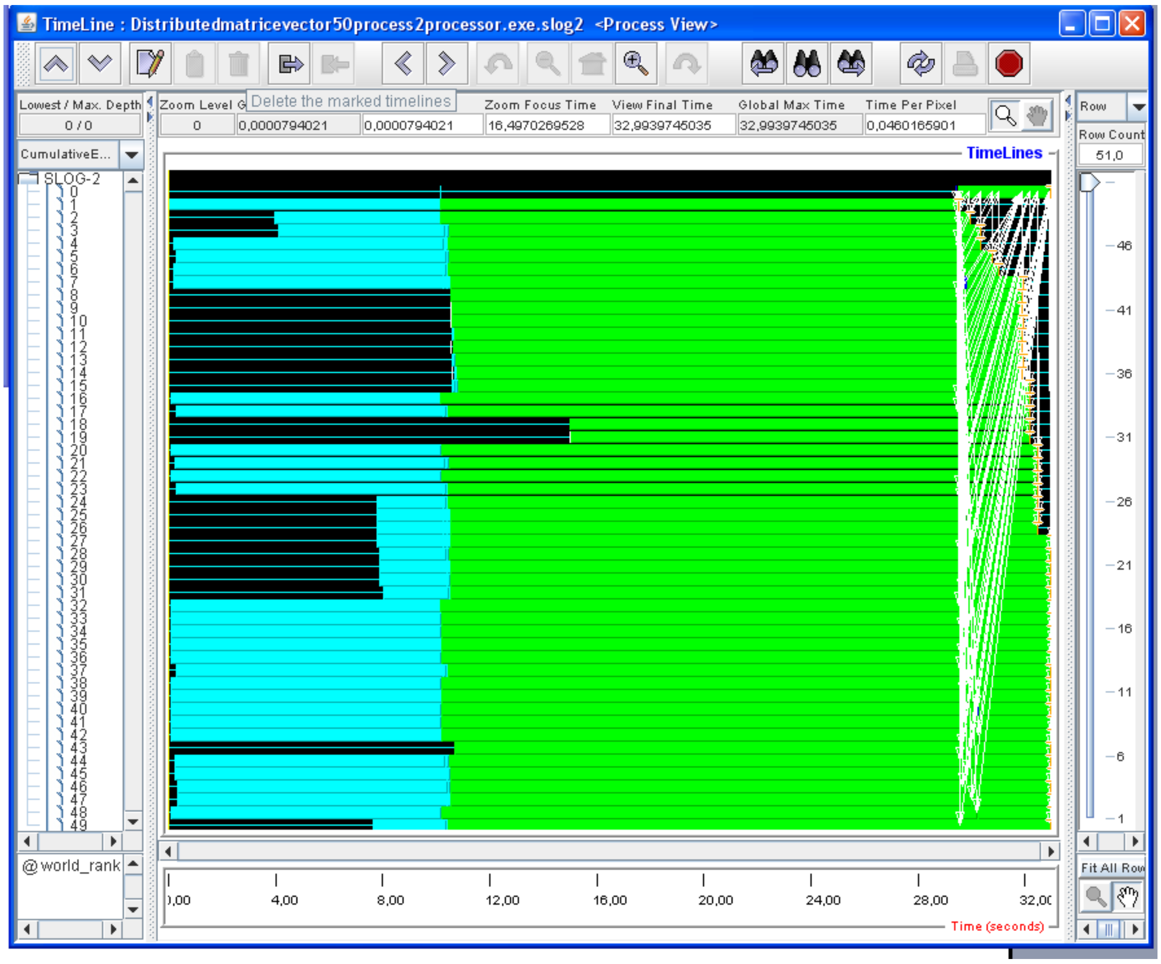


Figure 5.3 Speed up Vs Number of processes

The chart was made only for a twin processor computer, the use of MPICH gives the flexibility of launching as many number of processes as one wishes to do. It is quite clear from the above chart that increasing number of processes innumerably will not increase the speed up. The reason for this can be the communications overhead which are associated with increasing number of processes. This algorithm is a clear example of cost paid towards communications (*Chapter 3)*. Increasing the number of processes might work if the initial problem size increases, i.e. if the size of matrix is increased further then more speed up will be achieved but the overall trend of decreasing speed up will remain the same.

5.2.3 Block wise matrix vector multiplication

The algorithm to this type of program has been explained in *Chapter 4*. The load balancing in this type of algorithm is more dynamic in nature as the load depends on the number of processes used by the user. The domain decomposition is clearer if we view the Process view diagram for this type of algorithm in *Fig 5.4*



TimeLine : Distributedmatricevector50process2processor.exe.slog2 <Process View>

Lowest / Max. Depth 0 / 0
Zoom Level 0
0,0000794021
0,0000794021
Zoom Focus Time 16,4970269528
View Final Time 32,9939745035
Global Max Time 32,9939745035
Time Per Pixel 0,0460165901
Row Count 51,0

Figure 5.4 Process view block-wise matrix vector multiplication

As can be seen clearly from the figure that some of the processes are completed before the others and while some are left idle for a substantial amount of time. The time taken for the various combinations of this algorithm are shown in the *table 5.3*

No. of Process	No. of processors	Time taken(sec)	Average time (sec)	Speed up
		30,23		
		31,01		
		30,25		
		30,2		
33	2	31,55	30,65	5,55
		30,2		
		30,13		
		30,65		
		30,14		
33	4	30,54	30,33	5,6
		57,82		
		54,88		
		54,94		
		54,91		
40	2	54,9	55,49	3,06
		30,64		
		30,67		
		30,12		
		30,55		
40	4	30,66	30,53	5,57
		59,28		
		56		
		56,64		
		33,9		
50	2	32,96	47,76	3,56
		30,69		
		30,38		
		30,87		
		30,12		
50	4	30,2	30,45	5,58

Table 5.3 Block wise matrix vector multiplication

An interesting observation regarding this algorithm was the time taken if the processes were less than 33. The program timed-out after 600 seconds i.e. it could not finish itself off after the stipulated 600 seconds. Moreover, if the number of processes was increased to more than 75 the result was the same. This observation could not be explained as to why the program would time itself out if the number of processes is less than a certain value. The MPICH library does not provide any useful explanation regarding this observation, moreover when the speed up was plotted with the number of processes it was found to remain almost the same *see Fig. 5.5*.

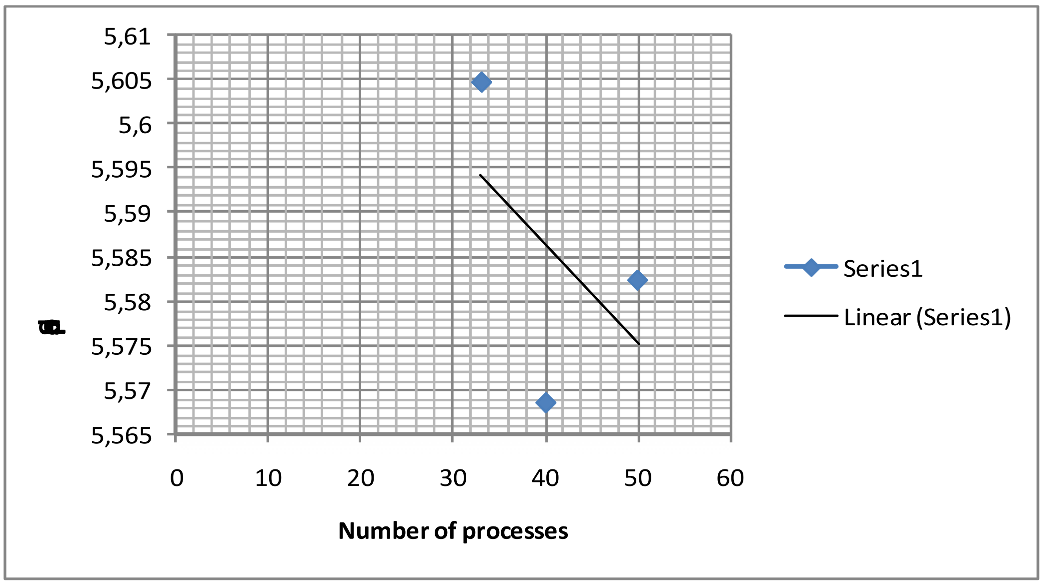


Figure 5.5 Speed up VS Number of processes

It is clear from the above *Fig.5.5*; speed up is independent of the number of processes. Since the domain decomposition is done based upon the number of processes the speed up remains constant, the portion of program parallelized is neutralized by the communication overhead associated with it. The true picture of load balancing could also be checked if more number of processors was used to execute the program. This analysis is somewhat incomplete without increasing number of processors; doing so might also explain the time-out which occurred when the number of processes was increased to the order of 75.

Chapter 6: Conclusions and Future Work

6.1 Conclusion 49

6.2 Future Work 49

Chapter 6

Conclusions and Future work

6.1 Conclusion

This report provides an overview of the parallelization techniques which can be used for the computers based on WINDOWS platforms. It is possible to use MPICH open source library to parallelize programs on distributed memory environment, though the use of MPI to parallelize ABAQUS on WINDOWS is currently not supported. The use of ABAQUS for solving jobs on multiple processors requires LINUX or some vendor based parallel architecture. Although, ABAQUS can not be parallelized using WINDOWS platform, the problem is essentially of hardware. In the future with new developments in MPICH it might become possible to use it implicitly in ABAQUS, though as of now parallelization of any code requires deep analysis for getting efficient and positive results. At this moment of time any problem which might have any probability of parallelization can be solved on distributed memory computer systems using MPICH. The report also discusses the scope of the usage of domain decomposition techniques for improving the usage of parallel computation techniques.

The non availability of more than two computers rendered it impossible to test this program on increasing number of processors. The study of parallelization is almost incomplete without analysis of the program on increasing number of processors.

6.2 Future Work

In order to demonstrate the applicability of the techniques, scalability of the usage of the techniques must be tested. Suitable domain decomposition algorithms must also be tested in order to achieve maximum efficiency. The only program which was parallelized was that of matrix-vector multiplication, there are innumerable possibilities of checking this parallel system. There is a lot of scope of tackling more complex problems and testing the solution on increasing number of processors. The matrix vector multiplication can be further modified for sparse matrices and used for the GMRES or BiGSTAB iterative methods to solve the Helmholtz equation. The effect of increasing number of processors on the elapsed time was not taken up due to the shortage of time. One should join as many processors as possible to completely study the parallelization.

REFERENCES

[1] Amdahl's Law. http://en.wikipedia.org/wiki/Amdahl%27s_law

[2]Barrett,R. et al.: *Templates for the solution of Linear Systems: Building blocks for iterative methods*. SIAM, 1994

[3]Introduction to parallel computing. http://www.llnl.gov/computing/tutorials/parallel_comp/

[4]MPI - The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi/standard.html

[5]OpenMP C and C++ Application Program Interface.http://www.openmp.org/drupal/.

[6] Gropp,W. et al.: *Using Mpi* - 2nd Edition. MIT Press, 1999

[7] MPICH2 – Implementation of Message Passing Interface http://www-unix.mcs.anl.gov/mpi/mpich/ (December 13, 2006).

[8] Introduction to Parallel Programming. http://onyx2ced.frascati.enea.it/grafica/Corsi/SGIHPC/5_Parallel_intro.pdf

Appendix A

The parallel implementation of a Matrix by vector multiplication is shown below. This computation (Multiplication of vector by matrix) is basic to almost all of the Numerical methods discussed in the report. The Message passing library used is **MPICH version 2**, the programs are made in C++ with a **Visual C++** compiler used to compile it. The computers on which the programs were executed, consisted of 2 processors each on a **WINDOWS** Local Area Network(LAN). The implementation of this program should be done only after reading the configuration of MPICH2 with **WINDOWS** manual.

```
/*
*********************************************************************
Example
Objective : Matrice-Vector Multiplication by dividing the Matrix into blocks to send to other processes
Input : User provides the values of matrice and vector with two different files
Output : Process 0 prints the result of Matrice-Vector
Multiplication in a file
Necessary Condition : The executable needs to be copied to each of the Computers where the program runs preferably
in the folder "C://Program Files/MPICH2/bin
*********************************************************************
*/
// Header files
#include "mpi.h"
#include "mpe.h"
#include <iostream>
#include <fstream>
#include <stdio.h>
using namespace std;
// Start of the Program
int main(int argc, char *argv[])
{
// MPI Initialisation
MPI::Init(argc,argv);
MPI_Status stat;
// Common Declarations
int NumofRows,NumofCols,x;
int index, irow, VectorSize,root=0,namelen;
int MatrixFileStatus = 1, VectorFileStatus = 1;
float *c;
float *Vector;float *ansbuff;
float *Matrix; float *ans; float t=1.0;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int rank = MPI::COMM_WORLD.Get_rank(); // Returns a particular rank of a processor
int size = MPI::COMM_WORLD.Get_size(); // Returns the Number of processors
MPI_Get_processor_name(processor_name,&namelen);
// Main processor starts the work
if(rank==0)
{
if(size==1) // Number of Processors need to be more than one
{
cout<<"The number of processes should be more than 1"<< endl;
MPI_Finalize();
exit(-1);
}
// Reading data from a file called Matrix.txt
ifstream inmatrixdata;
inmatrixdata.open("C://Program Files/MPICH2/bin/Matrix.txt"); // opens the file, location of the file must be
mentioned
if(!inmatrixdata)
{ // File couldn't be opened
MatrixFileStatus = 0;
cerr << "Error: Matrix file could not be opened" << endl;
exit(1);
}
while ( !inmatrixdata.eof() ) // sets EOF(End of File) flag if no value found
{
inmatrixdata >> NumofRows>> NumofCols; // Read the Number of Rows and Columns
Matrix = new float[NumofRows*NumofCols]; // Creation of matrixarray
/* Allocate memory and read Matrix from file */
for(irow=0 ;irow<NumofRows*NumofCols; irow++)
inmatrixdata>> Matrix[irow];
}
inmatrixdata.close();
x= NumofRows/(size-1); // x will divide the Matrix in blocks horizontally
cout << "End-of-Matrixfile reached.." << endl;
/* Read vector from Vector file */
ifstream invectordata; // invectordata is like cin
invectordata.open("C://Program Files/MPICH2/bin/Vector.txt"); // opens the file, location of the file must be
mentioned
if(!invectordata)
{ // File couldn't be opened
VectorFileStatus = 0;
cerr << "Error: Vector file could not be opened" << endl;
exit(1);
}
```

```
while ( !invectordata.eof() ) // Keep reading until end-of-file
{
invectordata >> VectorSize; // sets EOF flag if no value found
/*Allocate memory and read Vector from file*/
Vector = new float[VectorSize];
if(VectorFileStatus != 0)
{
for(index = 0; index<VectorSize; index++)
invectordata>>Vector[index];
}
}
invectordata.close();
cout << "End-of-Vectorfile reached.." << endl;
// Checking whether the dimensions of Matrix and Vector are appropriate
if (VectorSize !=NumofCols)
{
cout<<"The number of Columns in Matrix are not equal to number of Rows in Vector"<<endl;
exit(1);
}
}
// Broadcasting the values of x,VectorSize and columns to other processes
// This is done by all the processors hence we have to come out of Master process
if(size !=0)
{
MPI_Bcast(&x,1,MPI_INT,root,MPI_COMM_WORLD);
MPI_Bcast(&NumofCols,1,MPI_INT,root,MPI_COMM_WORLD);
MPI_Bcast(&VectorSize,1,MPI_INT,root,MPI_COMM_WORLD);
MatrixFileStatus=2;
MPI_Bcast (&MatrixFileStatus, 1, MPI_INT, root, MPI_COMM_WORLD);
}
// We again enter the master process
if(rank ==0)
{
cout << "Hello World! I am process " << rank << " I am the Master " << endl;
cout<< " Number of rows read "<<NumofRows<<" "<< " Number of columns "<< NumofCols<< endl;
c=new float[NumofRows];
// Print out the matrix
for(int i=0;i<NumofRows;i++)
{
for(int j=0;j<NumofCols;j++)
cout << Matrix[i*NumofCols+j] << " ";
cout << "\n";
```

```
}
// Forming a buffer to store the various divisions of matrix
ansbuff= new float[x*NumofCols];
ans=new float[x]; // Answer matrix received from the slave processes
int numsent=0;int numrcvd=0;
for (int i=0;i<(size-1);i++)
{
// Formation of ansbuff array to send each divided block of matrix
for (int j=0;j<x*NumofCols;j++)
ansbuff[j] = Matrix[i*x*NumofCols+j];
MPI_Send(ansbuff,x*NumofCols, MPI_FLOAT, i+1, i, MPI_COMM_WORLD);
MPI_Send(Vector,VectorSize,MPI_FLOAT,i+1,VectorSize*x*NumofCols,MPI_COMM_WORLD);
numsent++;
cout<<"The number of block sent is "<< numsent<<endl;
}
}
// Now decide which process does what
// The slave processes
if (rank != 0 && MatrixFileStatus==2)
{
Vector=new float[VectorSize];
ansbuff= new float[x*NumofCols];
MPI_Recv(Vector,VectorSize,MPI_FLOAT,0,VectorSize*x*NumofCols,MPI_COMM_WORLD,&stat);// Recovers the Vector
MPI_Recv(ansbuff,x*NumofCols,MPI_FLOAT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&stat);// Recovers the buffer formed
while (stat.MPI_TAG !=(VectorSize+1))
{
if (stat.MPI_TAG !=(VectorSize+1))
{
cout <<" Tag received is "<<stat.MPI_TAG<<endl;
int row=stat.MPI_TAG;
ans=new float[x];
for(int i=0;i<x;i++)
{
float a=0;
for (int j=0;j<NumofCols;j++)
a=a+ ansbuff[i*NumofCols+j]*Vector[j];
ans[i]=a;
}
MPI_Send(ans,x,MPI_FLOAT,0,row,MPI_COMM_WORLD); // Sending the Answer formed on the process
}
MPI_Recv(ansbuff,x*NumofCols,MPI_FLOAT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&stat); // Recovering the message to
terminate or not
```

```
}
}
if (rank==0)
{ // Checking for the answer
for(int i=0;i<(size-1);i++)
{
MPI_Recv(ans,x,MPI_FLOAT,i+1,i,MPI_COMM_WORLD,&stat);
int sender=stat.MPI_SOURCE; // Checking for the process which sent the answer
// Answer matrix
for (int j=0;j<x;j++)
c[i*x+j]=ans[j];
cout <<"The process from which the answer is received is "<< sender << endl;
MPI_Send(ansbuff,x*NumofCols,MPI_FLOAT,sender,VectorSize+1,MPI_COMM_WORLD);
}
int y=NumofRows-x*(size-1); // Finding rows for which the answer has not been calculated
if (y!=0)
{
for (int i=0;i<y;i++)
{
float a=0;
for (int j=0;j<NumofCols;j++)
a=a+ Matrix[NumofCols*((size-1)*x+i)+j]*Vector[j]; // Finding the Answer for the Rows which were
not sent
c[x*(size-1)+i]=a;
}
}
fstream outdata("C://Program Files/MPICH2/bin/Output.txt",ios::out); // Forming the Output file
outdata<<"Test Write to file"<<endl;
outdata<<"The Answer Vector is "<<endl;
for (int i=0;i<NumofRows;i++)
outdata<<c[i]<<endl;
outdata.close();
}
MPI::Finalize(); // Finishing the MPI operations, no MPI routine will work after calling MPI::Finalize
return 0;
}
```

Appendix B

This program was written in **C++** using **MPICH2** library and follows the algorithm of sending each row to a processor to multiply with the vector.

```
/*
********************************************************************
Example
Objective : Parallel Matrice-Vector Multiplication by sending each row to a slave process
Input : User provides the values of matrice and vector with a file
Output : Process 0 prints the result of Matrice-Vector
Multiplication into a file
Necessary Condition : The executable needs to be copied to all the computers used, preferably in the folder
C://Program Files/MPICH2/bin
********************************************************************
*/
// Header Files
#include "mpi.h"
#include "mpe.h"
#include <iostream>
#include <fstream>
#include <stdio.h>
using namespace std;
// Main program starts here
int main(int argc, char *argv[])
{
// Initialisation of MPI(Message Passing Interface)
MPI::Init(argc,argv);
MPI_Status stat;
// Common Declarations
int NumofRows,NumofCols;
int index, irow, VectorSize,root=0,namelen;
int MatrixFileStatus = 1, VectorFileStatus = 1;
float *c;
float *Vector;float *ansbuff;
float *Matrix; float ans; float t=1.0;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int rank = MPI::COMM_WORLD.Get_rank();
int size = MPI::COMM_WORLD.Get_size();
MPI_Get_processor_name(processor_name,&namelen);
// Main processes start the work
if(rank==0)
{
```

```
// OK This is the master routine
// Reading data from a file
ifstream inmatrixdata; // indata is like cin
inmatrixdata.open("C://Program Files/MPICH2/bin/Matrix.txt"); // opens the file, location of the file must be
mentioned
if(!inmatrixdata)
{ // File couldn't be opened
MatrixFileStatus = 0;
cerr << "Error: Matrix file could not be opened" << endl;
exit(1);
}
while ( !inmatrixdata.eof() )
{ // Keep reading until end-of-file
inmatrixdata >> NumofRows>> NumofCols; // sets EOF flag if no value found
// Creation of matrixarray
Matrix = new float[NumofRows*NumofCols];
/* ...Allocate memory and read Matrix from file .......*/
for(irow=0 ;irow<NumofRows*NumofCols; irow++)
inmatrixdata>> Matrix[irow];
}
inmatrixdata.close();
cout << "End-of-Matrixfile reached.." << endl;
/* Read vector from Vector file */
ifstream invectordata; // invectordata is like cin
invectordata.open("C://Program Files/MPICH2/bin/Vector.txt"); // opens the file, location of the file must be
mentioned
if(!invectordata)
{ // File couldn't be opened
VectorFileStatus = 0;
cerr << "Error: Vector file could not be opened" << endl;
exit(1);
}
while ( !invectordata.eof() )
{ // Keep reading until end-of-file
invectordata >> VectorSize; // sets EOF flag if no value found
/* ...Allocate memory and read Vector from file .......*/
Vector = new float[VectorSize];
if(VectorFileStatus != 0)
{
for(index = 0; index<VectorSize; index++)
invectordata>>Vector[index];
}
}
invectordata.close();
cout << "End-of-Vectorfile reached.." << endl;
// The number of Processors needs to be more than 1
if(size==1)
{
cout<<"The number of processes should be more than 1"<< endl;
MPI_Finalize();
exit(-1);
}
// Checking the feasibility of Matrix-Vector multiplication
if (VectorSize !=NumofCols)
{
cout<<" The dimensions of Matrix and Vector don't match up for Multiplication"<< endl;
}
}
// Coming out of process 0 to broadcast the values of NumofRows, VectorSize and Columns to other processes
if(size !=0)
{
MPI_Bcast(&NumofRows,1,MPI_INT,root,MPI_COMM_WORLD);
MPI_Bcast(&NumofCols,1,MPI_INT,root,MPI_COMM_WORLD);
MPI_Bcast(&VectorSize,1,MPI_INT,root,MPI_COMM_WORLD);
MatrixFileStatus=2;
MPI_Bcast (&MatrixFileStatus, 1, MPI_INT, root, MPI_COMM_WORLD);
}
// Again entering the Master process 0
if(rank ==0)
{
cout << "Hello World! I am process " << rank << " I am the Master " << endl;
cout<< " Number of rows read "<<NumofRows<<" "<< " Number of columns "<< NumofCols<< endl;
c=new float[NumofRows];
// Print out the matrix
for(int i=0;i<NumofRows;i++)
{
for(int j=0;j<NumofCols;j++)
cout << Matrix[i*NumofCols+j] << " ";
cout << "\n";
}
// Forming a buffer to store the rows
ansbuff= new float[NumofCols];
int numsent=0;int numrcvd=0;
// Find whether number of processors is smaller than no. of NumofRows
```

```
int l= ((size-1) <= NumofRows ? (size-1) : NumofRows);
for (int i=0;i<l;i++)
{
// send a row to each slave process; tag with row number
for (int j=0;j<NumofCols;j++)
// Formation of ansbuff array to send each row of matrix
ansbuff[j] = Matrix[i*NumofCols+j];
MPI_Send(ansbuff,NumofCols, MPI_FLOAT, i+1, i, MPI_COMM_WORLD);
MPI_Send(Vector,VectorSize,MPI_FLOAT,i+1,VectorSize*NumofRows*NumofCols,MPI_COMM_WORLD);
numsent++;
}
// Checking for the answer
for(int i=0;i<NumofRows;i++)
{
MPI_Recv(&ans,1,MPI_FLOAT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&stat);
// Checking for the process which sent the answer
int sender=stat.MPI_SOURCE;
// Checking the tag of the answer
int anstype=stat.MPI_TAG;
// Answer matrix
c[anstype]=ans;
cout <<"The process from which the answer is received is "<< sender << endl;
cout <<"The answer from above tag is "<< ans<< endl ;
// Checking whether the whole of Matrice(row wise) is sent or not
if (numsent < NumofRows)
{
for(int j = 0;j<NumofCols;j++)
ansbuff[j] = Matrix[numsent*NumofCols+j];
MPI_Send(ansbuff,NumofCols, MPI_FLOAT,sender, numsent, MPI_COMM_WORLD);
MPI_Send(Vector,VectorSize,MPI_FLOAT,sender,VectorSize*NumofRows*NumofCols,MPI_COMM_WORLD);
numsent++;
}
else // Sending the information to stop the multiplication to other processors
{ MPI_Send(ansbuff,NumofCols,MPI_FLOAT,sender,NumofRows+1,MPI_COMM_WORLD);
MPI_Send(Vector,VectorSize,MPI_FLOAT,sender,NumofRows+1,MPI_COMM_WORLD);
}
}
}
// Now decide which process does what
// The slave processes
if (rank != 0 && MatrixFileStatus==2 && rank <= NumofRows)
{
```

```
Vector=new float[NumofRows];
ansbuff= new float[NumofCols];
MPI_Recv(Vector,VectorSize,MPI_FLOAT,0,VectorSize*NumofRows*NumofCols,MPI_COMM_WORLD,&stat);// Recovers
the Vector
MPI_Recv(ansbuff,NumofCols,MPI_FLOAT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&stat);// Recovers the buffer of NumofRows
formed
while (stat.MPI_TAG !=(NumofRows+1))
{
if (stat.MPI_TAG !=(NumofRows*VectorSize*NumofCols))
{
cout <<" Tag received is "<<stat.MPI_TAG<<endl;
int row=stat.MPI_TAG;
ans=0;
for(int i=0;i<NumofCols;i++)
{ float x= Vector[i];
float y= ansbuff[i];
ans=ans + x*y;
}
cout <<" answer is "<<ans <<" on process "<<rank<<endl;
MPI_Send(&ans,1,MPI_FLOAT,0,row,MPI_COMM_WORLD);// Sending the Answer formed to Master process
}
MPI_Recv(ansbuff,NumofCols,MPI_FLOAT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&stat);// Recover the information
whether to stop or not
}
}
// Forming the output file
if (rank==0)
{
fstream outdata("C://Program Files/MPICH2/bin/Output.txt",ios::out);
outdata<<"Test Write to file"<<endl;
outdata<<"The Answer Vector is "<<endl;
for (int i=0;i<NumofRows;i++)
outdata<<c[i]<<endl;
outdata.close();
}
MPI::Finalize(); // MPI finalized, no MPI routine should be called after finalization
return 0;
}
```

Appendix C

```
/**********************************************************************
Example
Objective : Serial Matrice-Vector Multiplication
Input : User provides the values of matrice and vector with a file
Output : Process 0 prints the result of Matrice-Vector
Multiplication into a file
Necessary Condition : No constraints till now
********************************************************************
*/
// Header Files
#include "mpi.h"
#include "mpe.h"
#include <iostream>
#include <fstream>
#include <stdio.h>
using namespace std;
// Main program starts here
int main(int argc, char *argv[])
{
// Initialisation of MPI(Message Passing Interface)
MPI::Init(argc,argv);
// Common Declarations
int NumofRows,NumofCols;
int index, irow, VectorSize,namelen;
int MatrixFileStatus = 1, VectorFileStatus = 1;
double starttime=0,endtime;
float *Vector;float *ansbuff;
float *Matrix;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int rank = MPI::COMM_WORLD.Get_rank();
int size = MPI::COMM_WORLD.Get_size();
MPI_Get_processor_name(processor_name,&namelen);
starttime = MPI::Wtime();
// Start the work by reading data from a file
ifstream inmatrixdata; // indata is like cin
inmatrixdata.open("C://Program Files/MPICH2/bin/Matrix.txt"); // opens the file, location of the file must be
mentioned
if(!inmatrixdata)
{ // File couldn't be opened
MatrixFileStatus = 0;
cerr << "Error: Matrix file could not be opened" << endl;
exit(1);
}
while ( !inmatrixdata.eof() )
{ // Keep reading until end-of-file
inmatrixdata >> NumofRows>> NumofCols; // sets EOF flag if no value found
// Creation of matrixarray
Matrix = new float[NumofRows*NumofCols];
/* ...Allocate memory and read Matrix from file .......*/
for(irow=0 ;irow<NumofRows*NumofCols; irow++)
inmatrixdata>> Matrix[irow];
}
inmatrixdata.close();
cout << "End-of-Matrixfile reached.." << endl;
/* Read vector from Vector file */
ifstream invectordata; // invectordata is like cin
invectordata.open("C://Program Files/MPICH2/bin/Vector.txt"); // opens the file, location of the file must be
mentioned
if(!invectordata)
{ // File couldn't be opened
VectorFileStatus = 0;
cerr << "Error: Vector file could not be opened" << endl;
exit(1);
}
while ( !invectordata.eof() )
{ // Keep reading until end-of-file
invectordata >> VectorSize; // sets EOF flag if no value found
/* ...Allocate memory and read Vector from file .......*/
Vector = new float[VectorSize];
if(VectorFileStatus != 0)
{
for(index = 0; index<VectorSize; index++)
invectordata>>Vector[index];
}
}
invectordata.close();
cout << "End-of-Vectorfile reached.." << endl;
// Checking the feasibility of Matrix-Vector multiplication
if (VectorSize !=NumofCols)
{
cout<<" The dimensions of Matrix and Vector don't match up for Multiplication"<< endl;
}
```

```cpp
cout << "Hello World! I am process " << rank <<endl;
cout<< " Number of rows read "<<NumofRows<<" "<< " Number of columns "<< NumofCols<< endl;
// Print out the matrix
for(int i=0;i<NumofRows;i++)
{
for(int j=0;j<NumofCols;j++)
cout << Matrix[i*NumofCols+j] << " ";
cout << "\n";
}
// Forming a buffer to store the answer
ansbuff= new float[NumofRows];
for (int i=0;i<NumofRows;i++)
{
float a=0;
for (int k=0;k<NumofCols;k++)
a=a+ Matrix[i*NumofCols+k]*Vector[k];
ansbuff[i] = a;
}
// Forming the output file
fstream outdata("C://Program Files/MPICH2/bin/Output.txt",ios::out);
outdata<<"Test Write to file"<<endl;
outdata<<"The Answer Vector is "<<endl;
for (int i=0;i<NumofRows;i++)
outdata<<ansbuff[i]<<endl;
endtime=MPI_Wtime();
double wallclocktime=endtime-starttime;
outdata<<"The wall clock time is "<<wallclocktime<<endl;
outdata.close();
MPI::Finalize(); // MPI finalized, no MPI routine should be called after finalization
return 0;
}
```