Parallel Algorithms on Configurable Hybrid UPC/MPI Clusters

K.Ganeshamoorthy

May 2010

Parallel Algorithms on Configurable Hybrid UPC/MPI Clusters

A thesis submitted for the Degree of Master of Philosophy

K.Ganeshamoorthy
University of Colombo School of Computing
May 2010

Declaration

The Thesis is my original work and has not been submitted previously for a Degree at this or any other University/Institute. To the best of my knowledge it does not contain any material published or written by another person, except as acknowledge in the text.

Author's name ······················· Date ·········
Signature ······························

This is to certify that this thesis is based on the work of Mr./Mrs./Ms ··············· ································ under my/our supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:
Supervisor 1 Name:·························· Date:·········
Signature:·······················

Supervisor 2 Name:·························· Date:·········
Signature:·······················

To my loving mother, Sinnathangam . . .

Table of Contents

Table of Contents **iv**

List of Tables **vii**

List of Figures **viii**

Abbreviations **ix**

Abstract **x**

Acknowledgment **xii**

1 Introduction **1**

1.1 Background 1
1.2 Motivation 3
1.3 Aims and Objectives 3
1.4 Summary of Main Results 4
1.5 Thesis Outline 6

2 Cluster Computing **7**

2.1 Background 7
2.2 Distributed Memory (DM) 8
2.2.1 Message Passing Interface (MPI) 9
2.2.2 Parallel Virtual Machine (PVM) 9
2.2.3 Theoretical Chemistry Group Message Passing System (TCGMSG) 10
2.2.4 AVS/Express Parallel Edition 10
2.3 Distributed Shared Memory (DSM) 11
2.3.1 TCP-Linda 11
2.3.2 Unified Parallel C (UPC) 12
2.3.3 Intel® Cluster OpenMP* (CLOMP) 12
2.3.4 CORSO 13
2.4 DM Vs. DSM - Comparative Performance on a HPC Cluster 14
2.4.1 Performance Comparison of Treadmarks, PVM, and MPI 14
2.4.2 Performance Comparison of UPC, CORSO, and MPI 15
2.4.3 Performance Comparison of CLOMP and OpenMP 17
2.4.4 Performance and Flexibility of MPI and UPC 18

3 Extracting Parallelism from a Hierarchy of Processing Levels in a Cluster **20**

3.1 Mixed Mode Approach for HPC Cluster Programming. 20
3.1.1 Mixed mode MPI/OpenMP algorithms 21
3.1.2 Mixed mode MPI/pthreads and MPI/cache algorithms 23

4 Shallow Water Model on a Cluster 25
4.1 Shallow Water Equations . . . 25
4.2 Related Work . . . 25
4.2.1 Parallel Simulations of Tsunami Effect . . . 25
4.2.2 Parallel implementation of a highly nonlinear Boussinesq equation model through domain decomposition . . . 27
4.2.3 Implicit Parallel FEM Analysis of Shallow Water Equations . . . 28
4.3 Linear Long Wave Theory . . . 29
4.4 Algorithm Design . . . 31
4.5 Implementations . . . 34
4.5.1 Distributed Memory Implementation . . . 36
4.5.2 Distributed Shared Memory Implementation . . . 36
4.5.3 Results . . . 36
4.5.4 Analysis of Results . . . 37

5 Artificial Neural Network Model on a Cluster 39
5.1 Parallelization of Feed-Forward ANNs . . . 43
5.2 Related Work . . . 45
5.2.1 Mapping of a Multilayered ANN onto a Network of Workstations . . . 45
5.2.2 Substituting Parallel Matrix Multiplication in the Training Phase of Back-propagation ANNs . . . 46
5.2.3 Parallel ANN Training Algorithms for Finance Applications . . . 47
5.2.4 Massively Parallel ANNs . . . 48
5.2.5 Factors Affecting the Training of a Distributed Back-propagation algorithm 48
5.3 Design of a Parallel Algorithm . . . 49
5.3.1 Batch update back-propagation with parallel matrix multiplication (NHP with BBP) . . . 51
5.3.2 Hybrid partition with batch update back-propagation and matrix multiplication (HP with BBP) . . . 55
5.3.3 Hybrid partitioning approach with incremental update back-propagation (HP with IBP) . . . 60
5.4 Implementations . . . 61
5.4.1 Distributed Memory Implementation . . . 62
5.4.2 Distributed Shared Memory Implementation . . . 63
5.4.3 Results . . . 63
5.4.4 Analysis of Results . . . 66

6 A Configurable DM/DSM Cluster 68
6.1 Distributed Shared Memory Sub Clusters Interconnected by MPI . . . 68
6.2 Algorithm Design for Shallow Water Model . . . 69
6.2.1 Algorithm Design for Configuration-1 . . . 71
6.2.2 Algorithm Design for Configuration-2 . . . 73
6.3 Implementation . . . 76

7 Performance of the Shallow Water Model on the Configurable Cluster 77
7.1 Review of MPI, UPC, and UPC/MPI for Configuration-1 and for Configuration-2 . . . 77
7.2 Evaluation . . . 78
7.2.1 Extracting the capabilities of the memory hierarchy in a cluster . . . 78
7.2.2 Programming flexibility of the language . . . 78
7.2.3 Meta expressiveness . . . 79

8 Conclusion and Future Work 80

References 82

Web References 89

A Theory of Shallow Water Models I
A.1 Introduction I
A.2 Boundary condition I
A.3 The integration: Mass continuity II
A.4 Equations in velocity form III
A.5 Equations in transport form III
A.6 Linear shallow water equations IV

B Matrix Multiplications V
B.1 Cache layer matrix multiplication algorithms VI
B.1.1 Simple three loops algorithm VI
B.1.2 Blocking C algorithm VI
B.1.3 Blocking A algorithm VI
B.1.4 Transform and blocking A algorithm VI
B.1.5 Recursive algorithm VI
B.1.6 Strassen's algorithm VI
B.2 Shared memory layer matrix multiplication VII
B.2.1 Overlapping matrix B VII
B.2.2 Non-overlapping algorithm VII
B.2.3 Blocking algorithm VII
B.2.4 Transform and blocking algorithm VII
B.3 Distributed memory layer matrix multiplication VII
B.3.1 Two dimensional broadcasting algorithm VII
B.3.2 Cannon's algorithm VIII

C Hardware Configurations IX
C.1 Multiprocessors IX
C.2 Symmetric Multi Processors (SMP) IX
C.3 Chip Multi Processors (CMPs) X

D Feed-Forward ANN XII
D.1 The Multi-layer ANN XIII
D.2 Back-propagation Algorithm XIII
D.3 Standard Incremental Update Back-propagation algorithm XV
D.4 Standard Parallel Batch Update Back-propagation Algorithm XV

E Running Times for the Shallow Water Model XVIII

List of Tables

4.1 Run time in seconds (wot - without thread; wt - with thread) 37

4.2 Running time factors. 37

5.1 Artificial neural network architectures 50

5.2 Mean training time in seconds for ANN_1 using MPI 64

5.3 Mean training time in seconds for ANN_2 using MPI 64

5.4 Mean training time in seconds for ANN_3 using MPI 65

5.5 Mean training time in seconds for ANN_1 using TCP-Linda 65

5.6 Mean training time in seconds for ANN_2 using TCP-Linda 65

5.7 Number of MPI functions called at training time 66

7.1 Average run time in seconds for the shallow water model with (4×3) grid scheme on Upplanka cluster. 77

E.1 Run time in seconds (wot - without thread; wt - with thread) were obtained from the Monolith high performance cluster XVIII

E.2 Run time in seconds (wot - without thread; wt - with thread) were obtained from the Monolith high performance cluster XVIII

E.3 Run time in seconds were obtained from the Upplanka high performance cluster . . XIX

List of Figures

3.1 Mixed mode programming model on HPC cluster Ref. (Wu et al., 2002). 21

4.1 Three possible ways of decomposing a rectangular domain. The numbers in the sub-domains represent the processor number. 31

4.2 (A) Overview of the unstructured grid and assignment of worker nodes to sub-domains. (B) Arrows indicate the inner-worker communication. 32

4.3 Linear shallow water wave profiles at three different time-steps (ts) calculated using 48 processors. 32

5.1 Vertical partition of an ANN (Sudhakar & Murthy, 1998). 42

5.2 Hybrid partition of an ANN (Suresh et al., 2005). 42

6.1 DSM sub clusters interconnected by MPI using two-dimensional toroidal topology. . 69

6.2 A 12-node cluster is configured as two DSM sub clusters interconnected by MPI (configuration-1). 70

6.3 A 12-node cluster is configured as four DSM sub clusters interconnected by MPI (configuration-2). 71

B.1 Pictorial representation of cache level matrix multiplication algorithms (Wu et al., 2002). V

B.2 Pictorial representation of shared memory matrix multiplication algorithms (Wu et al., 2002). V

C.1 Communication between threads on a distributed memory computer (Buchau et al., 2008). X

C.2 Communication between threads on a shared memory computer (Buchau et al., 2008). X

C.3 The architecture of the CMP system with up to 16 processors. The core organizations for the cache-based and streaming models are shown on left and right side respectively (Leverich et al., 2007). XI

D.1 The perceptron. XII

D.2 A four-layer feed-forward ANN. XII

Abbreviations

1. ANN - Artificial Neural Network
2. BBP - Batch Back-Propagation
3. CLOMP- Cluster OpenMP
4. CMP - Chip Multi Processors
5. DM - Distributed Memory
6. DSM - Distributed Shared Memory
7. HP - Hybrid Partition
8. HPC - High Performance Computing
9. IBP - Incremental update Back-Propagation
10. MLP - Multi-Layer Perceptron
11. MP - Message Passing
12. NHP - Non Hybrid Partition
13. PVM - Parallel Virtual Machine
14. SMP - Symmetric Multi Processors
15. TCGMSG - Theoretical Chemistry Group Message passing System
16. UPC - Unified Parallel C
17. VP - Vertical Partition
18. VSM - Virtual Shared Memory

Abstract

High performance computing (HPC) applications have increasingly tended to use "off the shelf" commodity clusters of workstations as their executing platform, in the last decade due to their generic nature and low cost. The typical cluster of nodes, as well known, is a distributed memory structure, whose programming paradigm of message passing is well established.

This thesis presents a study of the impact of hybrid memory architectures composed of distributed memory and distributed shared memory, within a multi-processor cluster on the design of parallel algorithms. The thesis proposes a novel configurable virtual cluster arrangement interconnected by message passing links as a programming metaphor for parallel algorithm design. The study uses solution of parallel numerical algorithms and parallel back-propagation neural network algorithms as case studies. The parallel implementation of the shallow water equations to model a Tsunami is emphasized.

In the tsunami model, a rectangular array of data is partitioned into sub-domains, namely a four by three grid scheme and an eight by six grid scheme which are then used for the parallel implementation. In the preliminary study of hybrid memory architectures, four versions of the parallel algorithm for each grid scheme are used: distributed memory without threads, distributed memory with threads, distributed shared memory without threads, and distributed shared memory with threads. Experiments are realized using the Message Passing Interface (MPI) library, C/Linda, and the Linux pthreads. Subject to the availability of memory, the distributed shared memory version without threads performs best, but as the task is scaled up, the threaded version becomes efficient in both distributed memory and distributed shared memory implementations.

In the case of parallel algorithms for neural network training, the neural network is partitioned into sub neural networks by applying a hybrid partitioning scheme. Therefore, each partitioned network is evaluated as a matrix multiplication. Three different sizes of neural networks were used and exchange rate prediction was used as the reference problem. Parallel implementations for each of the distributed memory and distributed shared memory scenarios were obtained. The partitioned, matrix multiplication had the fastest execution time, and the MPI implementation was always faster than the TCP-Linda equivalent.

The configurable hybrid cluster scheme was implemented as two UPC/MPI sub cluster configurations where, in the first configuration, configuration-1, twelve computing nodes are partitioned into two equal sized DSM sub clusters, and in the second configuration, configuration-2, twelve computing nodes are partitioned into four DSM sub clusters. UPC is used for intra sub clusters while MPI is used for inter-sub-cluster communication. The shallow water model was implemented on each of these configurations.

The parallel algorithm for the uniform DSM implementation exhibits a moderately better performance than either of the two parallel algorithms for the UPC/MPI hybrid cluster configurations while these two parallel algorithms individually exhibit an overall better performance than the parallel algorithm for the uniform DM architecture, in terms of running time. Configuration-1 exhibits a moderately better performance than that for configuration-2. As the number of computing nodes per DSM sub cluster decreases, the overall performance approaches that of a DM.

A cluster with a large number of computing nodes when configured as a UPC/MPI hybrid sub cluster provides algorithmic timing values faster than that for the pure DM programming model. Moreover, the set of DSM sub clusters can be configured into arbitrary topologies such as a two dimensional toroidal mesh type topology, enabling a more flexible programming approach and the possibility of reusing well known algorithm designs for high performance applications.

Acknowledgment

The work described in this thesis was carried out while I was reading for my Master of Philosophy Degree at the Department of Computation and Intelligent Systems, University of Colombo School of Computing (UCSC).

I would especially like to thank Dr D N Ranasinghe, Senior Lecturer in Computer Science and Deputy Director of the UCSC, for his constant encouragement and support in the development of this research, particularly for the many stimulating and instructive discussions we had on Parallel Algorithm Developments.

I am also extremely grateful to Mr. Malik Silva, Lecturer in Computer Science at the UCSC for his assistance provided in various stages of this research.

I also wish to thank all my colleagues, both past and present, of the UCSC for their assistance during the progression of this research, with special emphasis to Dr Mahen Jayawardena, Lecturer in Computer Science at the UCSC and Mr Chaminda Ratnasinghe.

I also wish to thank the continuous support and encouragement provided my family members in making this research a success.

This research was performed using computational resources at the National Supercomputer Centre, Linköping University, Sweden, and the computational resources of University of Colombo School of Computing. Partial funding was provided by the National Science foundation, Sri Lanka (Grant No. RG/2005/FR/07) and by SPIDER, the Swedish programme of ICT for developing Regions, the latter especially for the setting up of the cluster infrastructure.

Chapter 1

Introduction

1.1 Background

Computer architects and application developers have attempted to extract the maximum performance out of emerging CPU designs and programming paradigms in the last 3-4 decades in order to run high performance applications. Processor architectures have improved dramatically over the years in line with Moore's Law, which predicts a doubling of power every 18 months [WWW 21]. With the current fabrication densities, it is now possible to have many thousands of cores per CPU. This has been one direction of advancement. The other has been to extract more computational power using the MIMD philosophy, i.e. commodity cluster computing (Hung & Adeli, 1994). Together, these two approaches provide a formidable computing power in the modern era. This trend makes parallelization of software not optional but indispensable (Buchau et ai., 2008; Leverich et al., 2007; Cheng et al., 2006). However the weak point in the whole picture has been the absence of proper tools and programming paradigms to extract the emerging power of parallel architectures.

Typically a modern computer consists of one or more multicore CPU's with shared memory. This means that all CPU's access the common memory. The software is designed in such a way that a single process is split into multiple threads. Ideally, one thread per processor should be assigned. Multithreading is supported by most compilers with the well-known OpenMP [WWW 08] standard. OpenMP is independent of platform and operating system. A very important advantage of OpenMP is that load distribution is done dynamically during runtime. OpenMP is a widely used shared memory programming approach (Smith & Bull, 2001; Blikberg, 2003; Wong & Rendell, 2007).

Parallel computing on a commodity computing cluster has been gaining more attention in recent years due to its cost effectively compared to conventional supercomputers. Furthermore together with high speed general purpose switched networks, powerful clusters are narrowing the performance gap between clusters and supercomputers (Sudhakar & Murthy, 1998; Suresh

et al., 2005).

As no physical memory is shared between CPU's in a HPC cluster, all communication between processes in such a system must be performed by sending and receiving messages over the network. Currently, the prevailing programming model for parallel computing on HPC cluster is message passing. Libraries such as MPI [WWW 01], (Werstein et al., 2003; Smith & Bull, 2001; Lotrič & Dobnikar, 2005; Bova et al., 2000), PVM [WWW 02], (Werstein et al., 2003; Araiijo et al., 2003; Mattson, 1995), TCGMSG [WWW 03], (Mattson, 1995), and AVS/Express [WWW 04] have been developed at different research institutions as solutions. MPI is very popular and allows the software developer full control over the execution of parallel program.

The latest popular distributed shared memory (DSM) systems (Shan et al., 2001; Werstein et al., 2003, Gansterer & Zottl, 2005; Wong & Rendell, 2007), [WWW 05], also known as virtual shared memory (VSM) systems [WWW 06], (Mattson, 1995) or partitioned global address space (PGAS) systems [WWW 05], [WWW 14], (Husbands & Yelick, 2007), provide a shared memory abstraction on top of message passing in a HPC cluster. This is a very useful abstractions for programmers who are familiar with shared memory programming. Libraries such as TCP-Linda® [WWW 06], (Mattson, 1995), Unified Parallel C (UPC) [WWW 14], (Husbands & Yelick, 2007), TreadMarks (Keleher et al., 1994), SCASH (Harada et al., 1999), Co-ORdinated Shared Objects (CORSO) [WWW 15] and Cluster OpenMP* for Intel® C++ Compiler [WWW 07] have been developed at different research institutions to address this model of programming. A parallel application programmer can write the parallel program as if it is executing in a shared memory multi-processor and access shared data with general *read* and *write* operations. The software developer leaves the distributed shared memory system to handle the underlying message passing. It is easier to program with the shared memory paradigm, especially when the algorithm is very complicated, because the programmer can concentrate more on the algorithm design rather than on explicitly moving data among processes (Shan et al., 2001; Werstein et al., 2003; Gansterer & Zottl, 2005; Keleher et al., 1994).

For an HPC cluster with multi-processor nodes parallelization is more complicated. Here, CPUs will share local memory on each node, while the overall memory is distributed among the computing nodes. The system requires a mixed mode programming model (Wu et al., 2002; Smith & Bull, 2001). While MPI and TCP-Linda can be used for communication between global nodes in the distributed memory environment (or in a distributed shared memory environment derived from the DM), inside each MPI process and within each TCP-Linda process POSIX threads can be used in order to extract further parallelism, resulting in a mixed

mode programming model.

Due to the evolving CPU architectures from single processor nodes to symmetric multiprocessors and multi-core CPU's in recent times, the processing hierarchy of a generic cluster has opened up new approaches for the development of novel parallel programming models.

A flexible programming model, which is scalable and can consolidate a wide hierarchy of shared and distributed memory architectures within a cluster with nodes having single to multi CPU's each with large number of possible cores, is sought after by researchers. Our work is an attempt in that direction.

1.2 Motivation

Computable tasks can be classified mainly in two different major views, computationally intensive problems and communicative intensive problems. Parallel programming approaches using a divide and conquer strategy are more efficient when the computation to communication ratio is high. This has to be considered in the master-slave computation prevalent in typical cluster computing. Among the programming models widely used on clusters, message passing is a model that arises directly from the architecture of distributed memory multiprocessors and networks of workstations (Smith & Bull, 2001; Sudhakar & Murthy, 1998; Suresh et al., 2005). A distributed shared memory programming model is more efficient when message passing between computing nodes is less, i.e. smaller number of shared variables are used to communicate between computing nodes [WWW 07]. Therefore, one of the motivations of this study is to analyze the relevance of the programming models to two classes of real world problems. The studies done so far (Werstein et al., 2003; Gansterer & Zottl, 2005) show that DSM implementations always have higher overhead than their DM counterpart irrespective of the type of task. We reexamine this aspect here. Another aspect we will examine here is the behavior of the mixed mode programming model under task and architecture scaling. This forms the hypothesis that there are alternative programming models, that could exploit the hierarchy of processing capabilities found in a cluster more effectively, even for a restricted class of computational tasks. Finally, the insights gained in the study might have interesting implications for emerging large scale multi-core based CPU programming as well.

1.3 Aims and Objectives

The objective here is to evaluate the effects of the programming model on scalability of the shallow water computational model and for the training of a back-propagation neural network.

The execution environment is a multinode cluster. Computing the wave propagation in the tsunami model, where the entire ocean is the solution domain, is challenging, both due to the huge amount of computation needed and due to the fact that different physics applies in different regions (Cai & Langtangen, 2007). Training of neural network using back-propagation is a computationally intensive task, requiring millions of floating point multiplications (Yoon et al., 1990; Sudhakar & Murthy, 1998; Suresh et al., 2005; Long & Gupta, 2008). Further, large amounts of memory is required for neural networks (Novokhodko & Valentine, 2001; Babii, 2007).

Further, as mentioned in section 1.2, we wish to study the impact of memory architectures associated with distributed memory and distributed shared memory in the extraction of multiple levels of parallelism, for the solution of numerical algorithms and for the training of neural network using back-propagation algorithms.

1.4 Summary of Main Results

In the tsunami model, two partition schemes, a four by three (4×3) grid scheme and a eight by six (8×6) grid scheme, have been applied to partition the data into sub-domains. The two grid schemes have been used for the parallel implementation of this model. We present four parallel algorithms under each grid scheme: (i) pure TCP-Linda; (ii) mixed mode TCP-Linda/pthreads; (iii) pure MPI; and (iv) mixed mode MPI/pthreads. The first two parallel algorithms belong to distributed shared memory approach and the last two algorithms belong to distributed memory approach.

In both threading and non-threading programming environments, the parallel algorithms for the distributed shared memory exhibit better performance than the algorithms for distributed memory, for both grid schemes. Though the distributed shared memory implicitly passes messages at the lower level, the replication management subsystem have been optimized in TCP-Linda compared to MPI [WWW 06], to yield better performance.

In the (4×3) grid scheme, in both scenarios for distributed memory and distributed shared memory, non-threading algorithms exhibit better performance than threading algorithms. In contrast to previous instance, in both algorithms for distributed and distributed shared memory, the threading algorithms exhibit better performance than non-threading algorithms.

Among the parallel variations not using threads in both memory architectures, the (4×3)

grid scheme shows better performance than (8×6) grid scheme. This is due to the ratio of computation time to communication overhead decreasing faster for domains with smaller size. However, when the task is scaled up, say up to (8×6) grid system, due to the smaller subdomain sizes aligning with the available memory in the node, the threaded versions become more efficient in both MPI and TCP-Linda implementations. In both threading and non-threading environments, the TCP-Linda version exhibits better performance than the MPI version. This result is significant as none of the literature reports a class of task that gives better timing performance under DSM than DM.

Three parallel back-propagation neural network training algorithms have been used to predict the exchange fluctuation rate as determined by demand and supply conditions in the foreign exchange market, under each of the memory systems, distributed memory and distributed shared memory. The first algorithm uses a hybrid partitioning approach and without using batch back-propagation. The second algorithm uses the batch back-propagation approach and fast parallel matrix multiplication for weight iterations. Finally, the third algorithm uses the hybrid partitioning approach and uses batch back-propagation approach within each sub neural network with fast matrix multiplication for weight iteration.

For all three neural network configurations, the parallel algorithms under the distributed memory architectures show vastly superior timing than distributed shared memory parallel algorithms. The reason being that according to the code, the neural network assigned to each TCP-Linda process is read and is written back from/to the distributed shared memory, for all training tasks. These read and write operations are implemented in TCP-Linda by *in* and *out* operations. The *in* and *out* operations take considerable time to act on the user defined data types, such as "struct". In the TCP-Linda program, the optimum number of processors have been dynamically assigned, even though the running time is very high. The third parallel algorithm is more efficient than the other two parallel algorithms in either of the memory architectures.

Since the communication time per training epoch for a given input pattern is higher than the time for computations, there is little advantage on the use of threads as a programming tool, in the neural network scenario.

1.5 Thesis Outline

The thesis is organized as follows: In Chapter 2, we elaborate on the cluster computing environments. Mostly focused on programming models such as message passing and distributed shared memory and on performance comparison among these two programming models. Mixed-mode algorithms which are used to exploit the parallel hierarchy, are reviewed in Chapter 3. In Chapter 4, shallow water equations to model the tsunami effect are described. Three parallel algorithms for the training of artificial neural networks using back-propagation on a cluster are analyzed in Chapter 5. In Chapter 6, a configurable hybrid DM/DSM cluster environment is explained and its implementation described. Analysis of results for the shallow water equations to model the tsunami effect on the proposed configuration is discussed in Chapter 7. Finally, Chapter 8 gives the conclusion of our research and future directions.

Chapter 2

Cluster Computing

2.1 Background

Cluster computing refers to approach in building up an execution environment consists of two or more computers via a high speed network in order to take advantage of the cumulative parallel processing power of those computers [WWW 17] (Shan et al., 2001; Aung et al., 2005). The recent advances in high speed networks and improved microprocessor performance are making clusters an appealing vehicle for cost effective parallel computing. Clusters built using commodity hardware and software components are playing a major role in redefining the concept of super computing (Aung et al., 2005).

Cluster computing is used as a relatively low-cost form of parallel processing for scientific and other large scale applications that lend themselves to parallel operations. Most important applications include computational simulations of weather, simulations of material physics, simulations of molecule dynamics, flow computing, etc. (Ganeshamoorthy et al., 2009; Ganeshamoorthy et al., 2007; Buchau et al., 2008; Thomas, 1994).

The trend in parallel computing is to move away from specialized platforms to cheaper, general purpose systems consisting of loosely coupled components built up from single or multiprocessor workstations or PCs. This approach has a number of advantages including that of being able to build a platform for a given budget, which is suitable for a large class of applications and workloads.

Cluster computing has become more popular in recent years as the power of commodity processors has approached that of workstation processors. The standardization of network hardware and software protocols has also improved the viability of low cost cluster computing. The complexity in the high performance applications and the need for ever more computational power has kept pace with one technological important in processors and network technologies, thereby making able to design clusters that can meet the computational demand.

However, not all problems are amenable to cluster solutions. As stated in Amdahl's Law (Hill & Marty, 2008) the task should have a high fraction of inherent parallelism to be of any use when parallelized. Depending on the communication costs of the algorithm used to solve the problem, employing a cluster may bring zero or negative benefits. On the other hand, there is a wide class of problems which can be cheaply and effectively solved using clusters (Ganeshamoorthy & Ranasinghe, 2008; Cai & Langtangen, 2007; Suresh, 2005).

Within the limitations of the cluster framework, there are other benefits of clustering. For example, clusters offer greater scalability than traditional symmetric multiprocessing (SMP) computers. With additional hardware, the performance of the cluster can increase substantially (Hope & Lam, 2008).

Generally, the target operating system used in a cluster computing machine is Linux being open and freely available. The Symmetric Multiprocessor (SMP) feature is also recognized by most Linux distributions.

Existing software tools generally take one of two major approaches to parallel program execution: message passing or virtual shared memory computing [WWW 06]. These two paradigms differ in many ways, but most importantly in their approaches to storing the data that is shared among the various components of a parallel program and to making the data available to the components that need it as the program runs.

Cluster computing systems can with the help of suitable middleware, provide additional services to users. We can use the queuing management software, Portable Batch System (PBS), to manage jobs submitted by different users [WWW 16]. This forms the basis of Grid Computing (Ouyang et al., 2008).

2.2 Distributed Memory (DM)

Message passing is a model that arises directly from the architecture of distributed memory multiprocessors and networks of workstations; the best known examples are MPI [1], a widely implemented standard message passing system, PVM [WWW 02], a library developed around the concept of a virtual machine that provides a single logical machine across the distributed memory cluster, TCGMSG [WWW 03], a toolkit for writing portable parallel programs using

a message passing model, and AVS/Express Parallel Edition [WWW 04], a framework for parallel computation on multi-processor and networked computers.

In this approach, each program datum belongs to some specific process, and it must be explicitly transmitted to any other processes that need it as the program progresses. Sending and receiving a single such message requires many steps by both the transmitting and receiving processes, and parallel programs built with message passing systems typically send many, many messages in the course of execution.

The above message passing libraries may require users to have a deep understanding in parallelizing their applications. Doing so will help improve the performance of running parallel applications on the cluster computing system. Unfortunately, training the users to achieve such a goal is costly.

2.2.1 Message Passing Interface (MPI)

There are many different versions of MPI libraries available with prominent examples being MPICH2 [WWW 18], LAM/MPI [WWW 19], and Open MPI [WWW 20]. As a result of this dominance, vendors of cluster network components often invest considerable time and effort in developing high performance MPI implementations that can make best use of their underlying hardware. MPI is very powerful and allows the software developer full control over the execution of program. Nevertheless, MPI has some disadvantages. The complete program must be parallelized and not only time-consuming parts. In practice, this is often ignored and some parts of the program are executed on all nodes with the same data. Then, conflicts in file access or the like must be avoided. The main disadvantage is that data exchange between the processes must be implemented by the software developer and load balance must be provided, too. This makes programming for message passing paradigm hard, especially for large applications with complex data structures (Werstein et al., 2003; Smith & Bull, 2001; Lotrič & Dobnikar, 2005).

2.2.2 Parallel Virtual Machine (PVM)

PVM was originally developed at Oak Ridge National Laboratory and the University of Tennessee. The PVM library is developed around the concept of a virtual machine that provides a single logical machine across the distributed memory cluster. It is freely distributed with Linux RedHat, providing a structure where a great variety of programs can be officially developed using the available hardware. PVM transparently handles all message routing, data conversion,

and task scheduling across a network of incompatible computer architectures. PVM enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. A virtual machine is associated with the user that starts it. The virtual machine exists as a daemon on each of the nodes in the cluster. Processes communicate via the daemon, which is responsible for handling communication between nodes. With PVM, resource management is dynamic. PVM includes a console program to manage the virtual machine. The console allows nodes to be added to or deleted from the virtual machine and provides process management utilities. PVM provides diversity of applications in parallel programming mainly because it supports the languages: C, C++, and Fortran. The planning of buffers and transmitters are idealized and optimized, providing a high degree of complexity in the structure for data transmission and reception in the PVM system. Owing to its ubiquitous nature (specifically the virtual machine concept) and also because of its simple but complete programming interface, the PVM system has gained widespread acceptance in the high performance scientific computing community [WWW 02], (Werstein et al., 2003; Araiijo et al., 2003; Mattson, 1995).

2.2.3 Theoretical Chemistry Group Message Passing System (TCGMSG)

TCGMSG is a toolkit for writing portable parallel programs using a message passing model. This message passing harness was written by Robert Harrison et al. of Battelle Pacific Northwest National Laboratory, USA (previously at Daresbury Laboratory and Argonne National Laboratory). Supported are a variety of common UNIX workstations, mini-super and super computers and heterogeneous networks of the same, along with true parallel computers such as the Cray T3D/E, IBM SP, Intel Paragon, the Kendall Square Research KSR-2, and several others. Applications port between all of these environments without modification to the parallel constructs [WWW 03], (Mattson, 1995).

2.2.4 AVS/Express Parallel Edition

AVS/Express Parallel Edition harnesses the power of multi-processor and networked computers for high-performance tasks by permitting distributed computation across shared memory or clustered processor architectures. AVS/Express Parallel Edition allows any combination of devices and platforms to contribute to the most demanding rendering and processing challenges. AVS/Express has been successfully deployed in a wide range of high performance implementations that demand the highest levels of processing power, reliability and flexibility [WWW 04].

2.3 Distributed Shared Memory (DSM)

The distributed shared memory also known as virtual shared memory (VSM) or partitioned global address space (PGAS) approach is built around a familiar paradigm for writing parallel programs; multiple processes interacting with and communicating by means of shared memory. It frees a programmer from having to deal explicitly with data exchange, and therefore, provides an easier method of programming. The DSM library is responsible for memory consistency maintenance and ensures each process has access to the most recent version of data. The best known products of this type are SCAI's TCP-Linda® and Paradise® systems [WWW 06], (Mattson, 1995), TreadMarks (Keleher et al., 1994), SCASH (Harada et al., 1999), UPC [WWW 05], [WWW 14], (Husbands & Yelick, 2007; Cantonnet et al., 2004), CORSO [WWW 15], and Cluster OpenMP* for Intel® C++ Compiler [WWW 07], (Buchau et al., 2008).

The DSM approach has number of noticeable benefits: (1) It is very simple to learn and to use, enabling existing programs to be parallelized rapidly and new ones to be developed easily; (2) It makes creating portable programs much less complicated, since architecture-specific low-level details are hidden from the user; (3) It enables advanced parallel execution features like dynamic load balancing to be implemented easily (Werstein et al., 2003; Gansterer & Zottl, 2005).

2.3.1 TCP-Linda

TCP-Linda, introduced in the mid of 1980's, was the first commercial product to implement DSM for large workstation clusters and supercomputers. TCP-Linda is a proven industry standard for parallel programming and is well known for its reliability and efficiency. TCP-Linda provides a single, logically shared memory to all of the processes in a parallel program.

Each process sees the same data space, and it can read or write shared data by using simple operations, *in* and *out*, respectively. No process ever has to worry about directly communicating with any other process; all such low level operations are handled by the system itself. The DSM works in this way regardless of whether it resides physically in a single memory, or it is actually distributed among the various processors participating in a program execution. TCP-Linda also differs from message passing in that it includes high-level coordination languages for parallel programming. It adds functionality to a standard programming language like C, C++, or FORTRAN for managing ensembles of independent processes, usually via a small set of simple operations which programmers use to implement parallelism [WWW 06], (Mattson, 1995).

2.3.2 Unified Parallel C (UPC)

UPC [WWW 14], (Gansterer & Zottl, 2005; Husbands & Yelick, 2007; Brightwell & Wen, 2004; Cantonnet et al., 2004; Patel & Gilbert, 2008) is a parallel extension of the C standard for distributed shared memory computers. UPC is an example of the Partitioned Global Address Space (PGAS) model that offers a global view of computation to the programmer. It supports high performance scientific applications. In the UPC programming model, one or more threads are working independently, and the number of threads is fixed either at compile time or run-time. Memory in UPC is divided into two spaces: (1) a private memory space and (2) a shared memory space. Every thread has a private memory that can only be accessed by the owning thread. The shared memory is logically partitioned and can be accessed by every thread. To improve the performance of memory accesses, UPC introduces the concept of thread affinity. With this feature, UPC optimizes memory-access performance between a thread and the per-thread address space where the thread has been bound.

In UPC, workload management is implicit, while workload partitioning and worker mapping can be either implicit or explicit. Implicit workload partitioning and task mapping are supported through an API called *upc_forall* which is similar to *for* iteration in C programming, except that the content of the iteration will be run in parallel. When this API is used, there is no need for additional programming effort for programmers to map the task to threads. The explicit approach in UPC for workload partitioning and worker mapping is similar to the one in MPI, where programmers have to specify on what will be run by each threads. In UPC, communication among threads adopt the Partitioned Global Address Space (PGAS) paradigm by making use of pointers. There are three types of pointer commonly used in UPC: (i) private pointer where the private pointers point to their own private address space, (ii) private pointer-to-share where the private pointers point to the shared address space, and (iii) shared pointer-to-share where the shared pointers from one address space point to the other shared address space.

2.3.3 Intel® Cluster OpenMP* (CLOMP)

In May 2006, Intel Corporation introduced the new Cluster OpenMP standard [WWW 07]. It is based on OpenMP and supports distributed memory cluster computers. In a first step, it appears to the software developer in the same way as classical OpenMP. One process is started and threads are created in the parallel section of the program. These threads are distributed among the computing nodes [WWW 07]. All threads see the same virtual shared memory. For

efficiency purposes, data must be declared sharable or not. If a thread writes to a memory page, all other threads on the other nodes are noticed about this action. If a thread reads a memory page, it is checked to be up-to-date and data is exchanged where required. To keep data transfer to a minimum, it is necessary to read and write data in a very ordered way.

Cluster OpenMP is used to distribute the threads among the computing nodes. If a node has multiple processors or multicore CPUs, classical OpenMP distributes the load of a Cluster OpenMP thread with classical threads among the processors of a node (Buchau et al., 2008; Wong & Rendell, 2007).

2.3.4 CORSO

CORSO is a representative of the shared object based (SOB) model and is a platform independent middleware. The SOB model is a subtype of the VSM model. A central idea of the space based concept is to have a very small set of commands for managing the objects in the space. Originally CORSO was developed at Vienna University of Technology, now a commercial product, produced by Tecco [WWW 15]. CORSO supports programming in C, C++, Java, and .NET. In a CORSO run-time environment, each host contains a Coordination Kernel, called CoKe. It communicates with other CoKe's within the network by unicast. Noticeable important features of CORSO are: (1) processes can be dynamically added to or removed from a distributed job during execution. Thus, this feature makes CORSO an attractive platform for dynamically changing grid computing environments, (2) CORSO distinguishes two types of communication objects: *Const* objects can only be written once, whereas *var* objects can be written an arbitrary number of times, (3) for caching, CORSO provides the *eager mode*, where each object replication is updated immediately when the original object was changed, and the *lazy mode*, where each object replication is only updated when it is accessed, (4) CORSO comprises two transaction models, *hard*-commit (in case of failure, the transaction aborts automatically without feedback) and *soft*-commit (the user is informed if a failure occurs) [WWW 15].

2.4 DM Vs. DSM - Comparative Performance on a HPC Cluster

Parallel computing provides a method to increase the performance of computationally intensive tasks by distributing their computation across multiple processors. Inter process communication can be provided through the libraries based on the message passing paradigm, in particular the MPI and PVM, or through the abstraction of a single memory address space available to all processes, known as DSM, in particular the TCP-Linda, UPC, CLOMP, CORSO, and Treadmarks.

Performance evaluation and comparison of representatives of two important parallel programming paradigms, the message passing model and the DSM model, have been studied by several authors.

2.4.1 Performance Comparison of Treadmarks, PVM, and MPI

TreadMarks DSM system has been compared with the message passing systems, parallel virtual machine (PVM), and message passing interface (MPI) by Werstein et al. (2003). Three main classes of parallel problems, namely, synchronous, loosely synchronous, and embarrassingly parallel have been selected for the case study. Training of back-propagation neural network, Mergesort and Mandelbrot set programs are chosen to represent the main class of parallel problems, in that order. The performance tests have been made on a cluster of 32 Intel Pentium III nodes.

For the MergeSort parallel algorithm, all three libraries show poor performance. The complexity of this problem is $n \log n$ which is similar to the complexity of communication. This causes communications between nodes to override the increased performance of additional nodes. When increasing the numbers to be sorted only the MPI shows any improvement as nodes are added. While PVM maintains a constant performance, the performance of the DSM degrades rapidly. This is due to virtual memory paging overhead. While the MPI program only shows a small increase in performance, the DSM program shows a major improvement when the amount of available memory is increased. But when the number of nodes increases, the DSM program still shows poor performance compared to the message passing programs. Profiling of the libraries shows that the DSM program generates significantly more network traffic than the message passing libraries.

For the Mandelbrot set parallel algorithm, all three libraries had shown similar performance. However MPI had the best overall performance. PVM also had a better speedup than DSM. The speedup scales almost linearly up to 24 nodes. After that point, additional nodes had resulted in decreasing performance. The near linear speedup reflects the embarrassingly parallel nature of this problem. The DSM program consistently generates approximately twice as much network traffic as the message passing programs. Thus it avoids the accelerating increase in data exchanged which is the main cause of its poor performance, especially with a high number of nodes, for the Mergesort and NN programs.

In the training of back-propagation neural network parallel algorithm, two data sets have been used to train the neural network. The first is the shuttle data set drawn from sensor data recorded during NASA space shuttle mission. The neural network is a three layer $9 \times 40 \times 1$ network. The second training data set is the forest cover data set. The neural network is required to determine the type of forest cover. The neural network is a three layer $54 \times 160 \times 1$ network.

With the shuttle data set, both message passing libraries show better performance. The speedup occurs almost linearly up to sixteen nodes before falling off to achieve a speedup of approximately 21 and 23 with 32 nodes for PVM and MPI, respectively. DSM performs well up to 8 nodes after which its speedup drops to less than one. With the forest data set, MPI shows the best performance with a speedup of 21 on 32 nodes. PVM performs similarly to MPI up to 24 nodes before slowing down a little on 32 nodes. DSM performs comparatively better obtaining a reasonable speedup with up to 16 nodes then reducing to a speedup of 3 with 32 nodes.

The slowdown of the DSM program is primarily due to the increase in network traffic which saturates the network. As the number of nodes is increased the volume of network traffic also increases. In message passing libraries, this increase is linear, but, in DSM, this leads to an exponential increase in the amount of data being transmitted for the neural network application. The reason for the reduced performance of the PVM version of the neural network compared to MPI version is due to the poor performance of the broadcast operation of PVM. The broadcast time for PVM is significantly higher than that for MPI.

2.4.2 Performance Comparison of UPC, CORSO, and MPI

Gansterer & Zottl (2005) have also studied the performance comparisons between the message passing (MP) paradigm and the DSM/PGAS paradigm. MPI and UPC/CORSO [8] have been

chosen for the implementation of the MP model and the DSM/PGAS model, respectively. UPC is a parallel extension of the ANSI C standard for PGAS model. In the UPC programming model, one or more threads are working independently, and the number of threads is fixed either at compile time or at runtime. CORSO is a representative of the SOB model. SOB (shared object based) model is a subtype of the DSM model. In this concept, objects are stored in a space. In CORSO, processes can be dynamically added to or returned from a distributed job during execution. Such dynamic allocation or deallocation cannot be implemented either in MPI or in UPC.

Three benchmark tests have been implemented in MPI, UPC, and CORSO: (i) two classical summation formulas for approximating π, (ii) a tree structured sequence of matrix multiplications, and (iii) the basic structure of the eigenvector accumulation. Because of the simple dependency structure (only two synchronization points) of π approximation, it is easy to parallelize and allows one to evaluate the overhead of managing shared objects in comparison to explicit message passing. In the parallel implementation of this problem, each processor computes its partial sum, and then finally all the partial sums are accumulated on one processor.

In the sequence of matrix multiplications, owing to its tree structure, it involves much more communication than benchmark(i) and is harder to parallelize. If the number of processors (N_P) allocated to benchmark(ii) is a power of 2, the binary tree is balanced, this leads to better utilization of the processors involved than for an unbalanced tree. Benchmark(iii) has the structure of a binary tree with matrix multiplications at each node. However, in contrast to benchmark(ii), the size of the node problems increase at each stage, this leads to a much lower computation per communication ratio. This makes it the hardest problem to parallelize.

For the benchmark(i), owing to the high degree of parallelism available, the speedup values of all three implementations are relatively high. However, DSM implementations show lesser performance than message passing paradigms. In benchmark(ii), for a balanced binary tree, the utilization of active processors is maximal, and therefore the speedup curve shows an oscillating pattern. Owing to the master-worker implementation, the speedup value and the execution time for one processor and for two processors are equal. For benchmark(iii), the speedup values are disappointingly low for all three implementations (in some cases, even a slowdown occurs). This reflects the difficulty of parallelizing benchmark(iii) - in particular, its low computation per communication ratio. Therefore, the implementations in the DSM/PGAS model for the all three benchmarks achieve somewhat lower performance than explicit message passing programs.

2.4.3 Performance Comparison of CLOMP and OpenMP

Intel Corporation's Cluster OpenMP (CLOMP) standard transforms a computer cluster with DM into a DSM system.

Buchau et al. (2008) have compared the CLOMP and OpenMP. CLOMP is used to parallelize a boundary element method based on the fast multipole method. Efficiency is studied for electrostatic field problems. The results of the novel CLOMP approach are compared with the classical OpenMP.

In this exercise, a system of linear equations is solved iteratively and a matrix vector product is computed in each iteration step. This product is accelerated by the fast multipole method. The number of operations, which must be evaluated, is relatively small. However, a relatively large amount of data must be interchanged.

Two numerical examples have been studied to analyze the efficiency of OpenMP and Cluster OpenMP in the context of a boundary element method (BEM) with the fast multipole method (FMM). The first example is an electrostatic field problem. A linear system of equations with 81688 unknowns is obtained. A classical BEM approach would result in memory requirements of 50GB. The FMM compresses the matrix to 500MB.

A second numerical example investigates the steady currents inside a printed circuit board (PCB). The PCB was discretized with second order quadrilateral elements and the resulting linear system of equations for the 86453 unknowns is solved. This would require 56GB of main memory with a classical BEM. The application of the fast multipole method reduces this amount to 900MB.

Two different computers have been used for the numerical solution of these two examples. One computer was equipped with 16 Intel Itanium 2 processors with a clock speed of 1.3 GHz. The other computer cluster consisted of eight computing nodes where each node had two AMD opteron processors with a clock speed of 2.2 GHz. To determine the speedup of parallelization, a serial reference program run was run.

Speedup related to matrix assembly and solution of linear system of equations in the case of OpenMP and Cluster OpenMP have been presented. The speedup for matrix assembly

was excellent. Even for 16 threads the speedup was 15.8. Unfortunately, the speedup for solution of the system of linear equations was much worse. Memory bandwidth was only one factor.

Since a DM computer causes communication overhead between the virtual threads, the efficiency of Cluster OpenMP was shown to be worse than the efficiency of OpenMP. Even in the case of matrix assembly, the speedup for 4 threads, which ran on 2 computing nodes, is only 3. One reason is that after each writing to memory, the other computing node has to be informed about this change in the DSM. The speedup for the solution of the linear system of equations is good in comparison to OpenMP. One reason, has been that the FMM algorithm was slightly modified in this scenario. Some fast operations were done in each thread. Hence, data was locally available and was not required to be transferred between the computing nodes.

Finally, the authors concluded that Cluster OpenMP is a promising approach to cluster computers. Particularly, as parallelization is easy to be carried out. Both OpenMP and Cluster OpenMP were recognized as good choices for an easy and efficient parallelization. Especially the combination of parallelization with a fast efficient method like that of BEM with the FMM was very attractive.

2.4.4 Performance and Flexibility of MPI and UPC

The choice of the parallel language has a major impact on the productivity of the HPC development process. A set of quantifiable criteria, defines a set of hypotheses based on commonly held beliefs, and evaluates them empirically with statistical tests, was selected (Patel & Gilbert, 2008). The comparison criteria are variables such as performance that are measurable with well-defined metrics such as program speedup and run time. Then, the hypotheses are statistically validated using the data from the human-subject experiments conducted as part of the DARPA High Productivity Computing Systems (HPCS) program.

This approach has been applied to empirically compare message passing (using MPI) and partitioned global address space (using UPC) programming models. Two key contributions have been made: (i) it presents a general approach to comparing parallel programming models using rigorous statistical hypothesis tests, and (ii) it presents novel hypotheses for comparing the performance and productivity of the MPI and UPC programming languages, validates them using empirical data, and analyzes the statistical results.

Data has been collected from the HPCS classroom productivity studies conducted at UCSB in 2006 and 2007, to validate the hypotheses. Parallel implementations with MPI and UPC for

power method have been developed. The power method is an iterative procedure that finds the largest absolute eigenvalue and the associated eigenvector of a matrix. The IMB MPI/C compiler and the Berkeley UPC compiler have been used for development.

Test results have shown to validate some of the commonly held beliefs about parallel programming models. MPI programs had lower run times compared to UPC, confirming the conventional wisdom. When both MPI and UPC programs achieved similar parallel speedup and scaling behavior on average, a higher variation in the performance of UPC programs has been observed. Analysis shows that this is because different design choices for parallelization in UPC being lead to significantly different performance results. Developing a parallel program for a problem in the PGAS model requires comparatively less effort. This is validated by the result that UPC programs require less code and thereby less effort compared to MPI.

In the analysis, several UPC solutions which had correctness and performance problems that typically do not manifest themselves at the scale of development tests were found (Patel & Gilbert, 2008). This leads them to believe that PGAS languages should be equipped with profiling capabilities that can optimize accesses to the global address space.

Chapter 3

Extracting Parallelism from a Hierarchy of Processing Levels in a Cluster

3.1 Mixed Mode Approach for HPC Cluster Programming.

Shared memory models have been more prominent in the conventional super computing domain where uniform memory access (UMA) and non-uniform memory access (NUMA) architectures have existed (Robertson & Rendell, 2003), but have recently emerged as symmetric multiprocessors (SMP), where a smaller number of CPUs also have access to a single memory space. Further, manufacturers have increasingly used clustering of SMP systems together to build powerful platforms. By utilizing a mixed mode programming model there is a possibility of taking advantage of the benefits of both shared memory and distributed memory models.

The majority of mixed mode applications involve a hierarchical model; MPI parallelization occurring at the inter-node level, and OpenMP/POSIX threads parallelization occurring at intra-node level. Whilst the majority of mixed mode programs implement this type of model (Smith & Bull, 2001), a number of authors have described non-hierarchical models (Smith & Bull, 2001; Bova et al., 2000). For example, message passing could be used within a code when it is relatively simple to implement and shared memory parallelism used where message passing is difficult.

Figure 3.1 shows the memory hierarchy of a typical HPC cluster. Mainly, a number of nodes are connected by a high-speed network where, within each SMP node there may be many processors; along with each processor's memory access being either to a high speed cache memory or the low speed main memory.

Message passing programming code written in MPI is obviously portable and should transfer

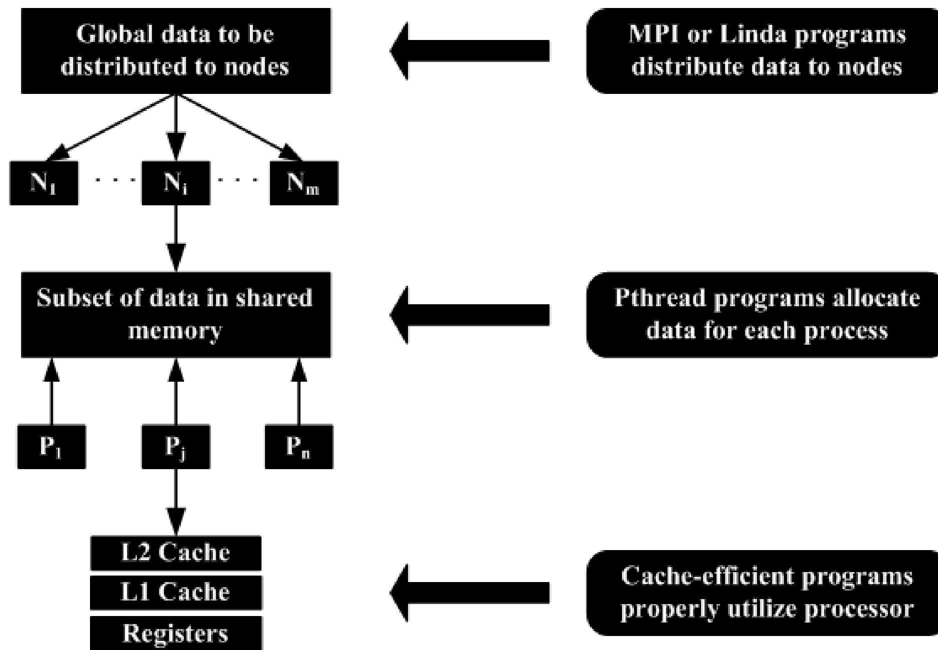

Figure 3.1: Mixed mode programming model on HPC cluster Ref. (Wu et al., 2002).

easily to clustered SMP systems. Inside each MPI process there may be three choices: (i) use POSIX threads for creating threads, one or more threads may belong to the MPI process mapped to the SMP node; (ii) use shared memory model such as OpenMP to provide a more efficient parallelization strategy within an SMP node; (iii) use MPI for local processes mapped to all processors of the each node. Finally, inside each process to use different algorithms that utilize the cache.

3.1.1 Mixed mode MPI/OpenMP algorithms

Smith & Bull (2001) have studied the benefits of developing mixed mode MPI/OpenMP applications on both single and clustered SMPs. The following various situations where a mixed mode code may be more efficient than a corresponding pure MPI implementation, whether on an SMP cluster or on a single SMP system, are discussed: codes which scale poorly with MPI (load balance and fine grain parallelism problems); replicated data; ease of implementation; restricted MPI process applications; poorly optimized intra-node MPI; poor scaling of the MPI implementation; and computational power balancing.

The code for *Game of life* problem has been chosen as a case study. This problem is a simple grid based one, which demonstrate complex behavior. It is a cellular automaton where the world is a 2D grid of cells which have two states, *alive* and *dead*. At each iteration the new state of the cell is entirely determined by the state of its eight nearest neighbors at the previous iteration.

The performance of the two OpenMP codes, OpenMP (SMPD) and OpenMP (Loop), and MPI code have been measured with two array sizes, 100×100 and 700×700, respectively, for $10,000$ iterations. The code was run on a Sun HPC 3500 system with exclusive access. This system had 8400 MHz UltraSparc II processors and 8 Giga bytes of memory running Solaris 2.7. Both the OpenMP codes had shown better performance than the MPI code on both problem sizes.

Thus developing a mixed mode MPI/OpenMP code may give better performance than the pure MPI code, and would therefore be more suited to an SMP cluster. Therefore three mixed mode versions of the code have been developed: (i) loop based OpenMP/MPI; (ii) 2D OpenMP/MPI; and (iii) SPMD OpenMP/MPI. The performance of three codes have been measured with the same array sizes for the same number of iterations. comparisons of these mixed code with the pure MPI implementation reveals a performance improvement have been obtained: the over all timings have reduced and the scaling of the code with increasing thread number is better.

Even when a pure OpenMP implementation gives better performance over a pure MPI implementation, this does not always mean that a mixed MPI/OpenMP code will give better performance than a pure MPI implementation. For example by implementing the mixed mode code with the MPI parallelization above the OpenMP parallelization, as is often the recommended case owing to the lack of a guaranteed thread-safe MPI implementation, extra synchronization is often introduced, which can reduce the performance. For this particular example the mixed mode code needed to be written with MPI and OpenMP at the same level, rather than using the more common hierarchical model. This creates issues with portability, relying on thread-safe MPI implementation, and adds to the code complexity, but increases performance.

Mixed mode programming style will not always be the most effective mechanism on SMP systems and cannot be regarded as the ideal programming model for all codes. Benefits may be obtained under the following MPI code environments: (i) poor scaling with MPI processes due to load imbalance or too fine a grain problem size; (ii) from memory limitations due to the use of a replicated data strategy; and (iii) from a restriction on the number of MPI processes combinations. Further more, if the system suffers from a poorly optimized or limited scaling MPI implementation, then a mixed mode code may increase the code performance.

3.1.2 Mixed mode MPI/pthreads and MPI/cache algorithms

Different types of matrix multiplication algorithms on system having multiple memory layers to show how performance is affected in mixed mode programming without a good cache algorithm, even when the work load is perfectly balanced, were used (Wu et al., 2002). The cache layer matrix multiplication algorithms such as *simple three loops* algorithm, *blocking C* algorithm, *blocking A* algorithm, *transform and blocking A* algorithm, *recursive* algorithm, and *Strassen's* algorithm have been used in their case study, which vary from those that have high cache misses to those that effectively use cache. *Overlapping matrix B*, *non-overlapping*, *blocking*, and *transform and blocking* matrix multiplication algorithms are chosen for the SM layer. Two dimensional *broadcasting* and *Cannon's* matrix multiplication algorithms are selected for the DM layer. SM layer algorithms and DM layer algorithms are implemented in Pthreads and in MPI, respectively. In the mixed mode programming model, MPI is used for the communication between global nodes and, within each MPI node Pthreads are used for creating one or more threads.

When the matrix size is power of 2, Strassen's algorithm shows the best performance. Otherwise, when the shapes of the matrices are not square, the cache misses caused by the block shell method and copying overhead reduces the performance of the following algorithms: transform and blocking A algorithm, recursive algorithm and Strassen's algorithm. The best performing algorithm is the blocking A algorithm.

With a good underlying cache algorithm, the performances of all SM algorithms are similar. The overall performance is not considerably affected when the number of threads are increased. However, when a cache sensitive SM algorithm is combined with a bad cache algorithm, there is a performance gain with an increased number of threads. Because, when the number of threads are increased the smaller block size per thread actually fits into cache thus reducing the cache misses.

When the underlying cache algorithm is bad, MPI algorithms shows better performance than Pthreads algorithms. The reason is that DM algorithms always split the data into smaller blocks while the SM algorithms works on a bulk data, and thus causing more cache misses. But, with good underlying cache algorithm, the choice of number of MPI tasks combined with the choice of number of threads does not seem to be so important. Without a good cache algorithm, run timings fluctuate. Different combinations of MPI tasks and number of threads exhibit different performance. The main dependence is on how the algorithms divide data and

thus make good use of cache as the chunk size decreases. For parallel algorithms with bad underlying cache placement algorithms utilized in a mixed programming mode, the saturation of the thread space beyond the total number of computing threads equal to the number of available processors should provide a modest performance enhancement.

Chapter 4

Shallow Water Model on a Cluster

Our aim here in this chapter is to study the impact of memory architectures associated with distributed memory and distributed shared memory in the extraction of multiple levels of parallelism, for the solution of numerical algorithms. We also evaluate the effects of the programming model on scalability of the algorithms for the shallow water application. Computing the wave propagation in the tsunami model, where the entire ocean is the solution domain, is challenging, both due to the huge amount of computation needed and due to the fact that different physics applies in different regions (Cai & Langtangen, 2007).

4.1 Shallow Water Equations

The shallow water equations have been widely used to study the Tsunami phenomenon (Dellar & Salmon, 2005; Williamson et al., 1991), as a model of the basic fluid dynamics of the ocean. Solving partial differential equations numerically for real-life problems is computationally demanding, therefore, utilizing super-computers/clusters efficiently, is important in order to achieve computational efficiency (Blikberg, 2003). Strategies to improve the accuracy and overall quality of model predictions have been and continue to be of great interest to numerical model developers but in addition to accuracy, the utility of a numerical model is greatly affected by the algorithm's efficiency (Sanders & Pau, 2003). Refer Appendix I for more details about shallow water equations.

4.2 Related Work

4.2.1 Parallel Simulations of Tsunami Effect

Hybrid tsunami simulators that allow different sub-domains to use different mathematical models, spatial discretizations, local meshes, and serial codes have been proposed (Cai & Langtangen, 2007). Boussinesq water wave equations given below are used for this purpose.

$$\frac{\partial \eta}{\partial t} + \nabla.(H + \alpha\eta)\nabla\phi + \epsilon H\left(\frac{1}{6}\frac{\partial \eta}{\partial t} - \frac{1}{3}\nabla H.\nabla\phi\right)\nabla H = 0, \tag{4.2.1}$$

$$\frac{\partial \phi}{\partial t} + \frac{\alpha}{2}\nabla\phi.\nabla\phi + \eta - \frac{\epsilon}{2}H\nabla.\left(H\nabla\frac{\partial \phi}{\partial t}\right) + \frac{\epsilon}{6}H^2\nabla^2\frac{\partial \phi}{\partial t} = 0, \tag{4.2.2}$$

where η and ϕ are primary unknowns denoting, respectively, the water surface elevation and velocity potential. The water depth H is assumed to be a function of the spatial coordinates x and y. In equations (4.2.1) and (4.2.2), the weak effect of dispersion and nonlinearity is controlled by the two dimensionless constants ϵ and α, respectively. The widely used linear shallow water equations can be derived by choosing $\epsilon = \alpha = 0$,

$$\frac{\partial \eta}{t} + \nabla.\left(H\nabla\phi\right) = 0, \tag{4.2.3}$$

$$\frac{\partial \phi}{\partial t} + \eta = 0. \tag{4.2.4}$$

The equations from (4.2.1) to (4.2.4) are used to model the tsunami. The equations (4.2.1) and (4.2.2) are resolved by unstructured meshes and finite element discretization, whereas structured meshes and finite differences are commonly used for equations (4.2.3) and (4.2.4). Such a parallelization strategy is most easily realized by using sub-domains, such that the entire spatial domain Ω is decomposed into a set of overlapping sub-domains $\{\Omega_s\}_{s=1}^{P}$. In generic setting, where a partial differential equation (PDE) is expressed as $Ł_{\Omega}(u) = f_{\Omega}$, the Schwarz algorithm consists of an iterative processes generating $u^0, u^1, \cdots\cdots, u^k$ as a series of approximate solutions. During Schwarz iteration k, each sub-domain first independently updates its local solution through

$$Ł_{\Omega_S}\left(U_s^k\right) = f_{\Omega_s}^{k-1}, \tag{4.2.5}$$

where $f_{\Omega_s}^{k-1}$ refers to a right-hand side which is restricted within Ω_s and depends on the latest global approximation u^{k-1}. Then, the new global solution u^k is composed by sewing together the sub-domain local solutions $u_1^k, u_2^k, u_3^k, \cdots\cdots, u_P^k$.

Equation (4.2.5) thus opens for the possibility of using different local solvers in different sub-domains. Taking the idea of additive Schwarz one step further, different mathematical models

in different sub-domains can be applied. Therefore, different serial codes may be deployed region wise. The hybrid parallel tsunami simulator has been implemented using object-oriented techniques and is based on an existing advanced C++ finite element solver named class Boussinesq applicable for unstructured meshes, and a legacy F77 finite difference code applicable for uniform meshes. The resulting hybrid parallel tsunami simulator thus has full flexibility and extensibility (Cai & Langtangen, 2007).

4.2.2 Parallel implementation of a highly nonlinear Boussinesq equation model through domain decomposition

Applications of the Boussinesq equations cover a broad spectrum of ocean and coasted problems of interest, from wind wave propagation in intermediate and shallow water depths to the study of tsunami wave propagation across large ocean basins. In general, implementations of the Boussinesq wave model to calculate free surface wave evolution in large basins are computationally intensive, requiring large amount of CPU time and memory. To facilitate such extensive computations, a parallel Boussinesq model has been developed by the Sitangang & Lynett (2005), using the domain decomposition technique in conjunction with MPI. The parallel Boussinesq model developed is based on its serial counterpart. The governing equations consist of the two-dimensional depth-integrated continuity equation:

$$\frac{\partial H}{\partial t} + \nabla.\left(H\mathbf{u}_x\right) -$$

$$\mu^2 \nabla.\left\{H\left[\left(\frac{1}{6}\left(\eta^2 - \eta h + h^2\right) - \frac{1}{2}z_x^2\right)\nabla S + \left(\frac{1}{2}\left(\eta - h\right) - z_x\right)\nabla T\right]\right\} = 0 \qquad (4.2.6)$$

and the horizontal momentum equation:

$$\frac{\partial \mathbf{u}_x}{\partial t} + \frac{1}{2}\nabla\left(\mathbf{u}_x.\mathbf{u}_x\right) + g\nabla\eta + \frac{\partial}{\partial t}\left\{\frac{1}{2}z_x^2\nabla S + z_x\nabla T - \nabla\left(\frac{1}{2}\eta^2 S + \eta T\right)\right\} +$$

$$\nabla\left\{\frac{\partial \eta}{\partial t}(T + \eta S) + (z_x - \eta)\left(\mathbf{u}_x.\nabla\right)T + \frac{1}{2}\left(z_x^2 - \eta^2\right)\left(\mathbf{u}_x.\nabla\right)S + \frac{1}{2}(T + \eta S)^2\right\} = 0 \qquad (4.2.7)$$

where $S = \nabla.\mathbf{u}_x,\ T = \nabla.\left(h\mathbf{u}_x\right) + \frac{\partial h}{\partial t},$ h is depth, η is free surface elevation, $H = h + \eta,\ \mathbf{u}_x$ is horizontal velocity vector, z_x is reference depth, and t is time. Equations (4.2.6) and (4.2.7)

differ from the equations given by Wei et al. (1995) in the inclusion of the time derivatives of the depth ($h_t - h_{tt}$) to account for temporal bottom profile changes that occur during landslide/earthquake, which is one of several possible sources of tsunami.

The parallel approach has three important aspects, domain decomposition, communication, and parallel solver of the tridiagonal system of the simultaneous linear equations. Three different domain sizes have been considered: 500×500, 1000×1000, and 2000×2000. The overall performance of the model is very good. The efficiency of the model decreases as the number of processors increases which is apparent in the case of $500 \times 500-$ and $1000 \times 1000-$ domains. The rate of the efficiency decrease is faster for smaller domain. This is due to the ratio of arithmetic operation time to communication time decreasing faster for domains with smaller number of nodes. The performance of the model improves as the number of grids increases; a favorable feature of a parallel model which is intended for simulation on ever-increasing domain sizes. Thus, this parallel model provides a good opportunity for large wave-resolving simulations in the nearshore, with global domains of many millions of grid points, covering $O(100km^2)$ and greater basins. Further, real-time simulation with Boussinesq equations have become a possibility.

4.2.3 Implicit Parallel FEM Analysis of Shallow Water Equations

Jianga et al. (2005) have solved the shallow water equations (SWEs) as the governing equation to model a river flow. SWEs are implemented on clustered workstations. For the parallel computation, the mesh is automatically partitioned using the geometric mesh partitioning method. The governing equations are then discretized implicitly to form a large sparse linear system, which is solved using a direct parallel generalized minimum residual algorithm (GMRES). The shallow water equations (4.2.8) and (4.2.9) used here are obtained by integrating the conservation of mass and momentum equations with assuming a hydrostatic pressure distribution in the vertical direction.

$$\frac{\partial \eta}{\partial t} + \frac{\partial q_j}{\partial x_j} = 0 \tag{4.2.8}$$

$$\frac{\partial q_i}{\partial t} + q_j \frac{\partial}{\partial x_j}\left(\frac{q_i}{H}\right) = -gH\frac{\partial \eta}{\partial x_i} + \frac{1}{\rho}\left(\tau_i^s - \tau_i^b\right) + \frac{\partial}{x_j}\left[v\left(\frac{\partial q_i}{\partial x_j} + \frac{\partial q_j}{\partial x_i}\right)\right], \quad i,j = 1,2 \tag{4.2.9}$$

where η is the water elevation, H is the water depth, $q_i = Hu_i$, u_i is the mean horizontal velocity, g is the gravitational acceleration, v is the eddy viscosity, ρ is the water density, τ_i^s is

the surface shear stress, and τ_i^b is the bottom shear stress.

The finite element method for triangular elements is used for the spatial discretization. Three kinds of finite element meshes are used to simulate the velocity field in the two numerical examples, flow around a circular cylinder and flow in the River. MPI has been used for the communication between nodes. The computed results agree well with the observed results. The good speedup and efficiency for the parallel computation show that the parallel computing technique is a good method to solve large-scale problems (Jianga et al., 2005).

4.3 Linear Long Wave Theory

There are many different numerical methods for computing shallow water equations on a sphere. Therefore, a standard test suite of seven problems for evaluating numerical methods for the shallow water equations in spherical geometry was proposed by Williamson et al. (1991), and accepted by the modeling community in order to compare newly proposed methods. The shallow water equations are widely used as a prototype to study phenomena like wave-vortex interactions that occur in more complicated models of large scale atmosphere/ocean dynamics (Dellar & Salmon, 2005).

Consider the sea to be a volume of incompressible water on a rotating sphere, with Coriolis force, fu. The horizontal coordinates are x and y, the vertical coordinate z, which is zero at the mean sea surface and positive upwards. The sea bed is located at $z = -H$, and the surface is located at $z = h$. The linear shallow water equations [WWW 09], [WWW 09], [WWW 11] consists of the continuity equation

$$\frac{\partial h}{\partial t} + \frac{\partial}{\partial x}[u(h+H)] + \frac{\partial}{\partial y}[v(h+H)] = 0 \tag{4.3.1}$$

and the conservation of horizontal momentum

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv = -g\frac{\partial h}{\partial x} \tag{4.3.2}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + fu = -g\frac{\partial h}{\partial y} \tag{4.3.3}$$

u and v are the velocity components in $x-$ and $y-$direction, h is the surface elevation and g

is the acceleration due to gravity.

Defining the equations in terms of the *discharge fluxes* $U = uH$, $V = vH$, leads to discretizations that always satisfy the conservation of mass. Then the conservation of horizontal momentum can be written as

$$\frac{\partial U}{\partial t} + \frac{\partial}{\partial x}\left(\frac{U^2}{h+H}\right) + \frac{\partial}{\partial y}\left(\frac{UV}{h+H}\right) - fV = -g(h+H)\frac{\partial h}{\partial x}, \tag{4.3.4}$$

$$\frac{\partial v}{\partial t} + \frac{\partial}{\partial x}\left(\frac{UV}{h+H}\right) + \frac{\partial}{\partial y}\left(\frac{V^2}{h+H}\right) + fU = -g(h+H)\frac{\partial h}{\partial y}. \tag{4.3.5}$$

The finite difference approximations using centered differences in space and a leap-frog time discretisaion, are based on a staggered grid corresponding to an Arakawa C-grid (Cai & Langtangen, 2007; Goto & Ogawa, 1992), with the continuity equation centered on the point $\left(x_i, y_j, t_{k+\frac{1}{2}}\right)$ and the equations of motion centered on the points $\left(x_{i+\frac{1}{2}}, y_j, t_k\right)$ and $\left(x_i, y_{j+\frac{1}{2}}, t_k\right)$, respectively. Writing $D \equiv h + H$, and using upwind differences for the convection terms to maintain stability it follows that

$$\frac{\partial}{\partial x}\left(\frac{U^2}{D}\right) \approx \frac{1}{\triangle x}\left(\lambda_{1,1}\frac{\left(U^{k-\frac{1}{2}}_{i,j+\frac{3}{2}}\right)^2}{D^{k-\frac{1}{2}}_{i,j+\frac{3}{2}}} + \lambda_{2,1}\frac{\left(U^{k-\frac{1}{2}}_{i,j+\frac{1}{2}}\right)^2}{D^{k-\frac{1}{2}}_{i,j+\frac{1}{2}}} + \lambda_{3,1}\frac{\left(U^{k-\frac{1}{2}}_{i,j-\frac{1}{2}}\right)^2}{D^{k-\frac{1}{2}}_{i,j-\frac{1}{2}}}\right), \tag{4.3.6}$$

where with up-winding,

$$U^{k-\frac{1}{2}}_{i,j+\frac{1}{2}}\begin{cases} \geq 0 & \lambda_{1,1} = 0, \quad \lambda_{2,1} = 1, \quad \lambda_{3,1} = -1 \\ < 0 & \lambda_{1,1} = 1, \quad \lambda_{2,1} = -1, \quad \lambda_{3,1} = 0 \end{cases}$$

and similarly for the other terms. The difference equations on a rectangular grid in terms of spherical polar coordinates. An accurate representation of tsunami running up the shore implies a grid spacing of no more than 100 meters in a region of about 4 km out from the shore. As this fine grid is not reasonable over the whole ocean a succession of overlapping grids is necessary near the coast. A data decomposition scheme is applied for the parallel solution of the shallow water equations. In data decomposition, we keep the sequential formulation of the problem, but distribute the data and operations among the processors. The scalability of several data decomposition algorithms for finite difference atmospheric and ocean models has

been analyzed by several authors (Skalin, 1996; Thomas et al., 1994).

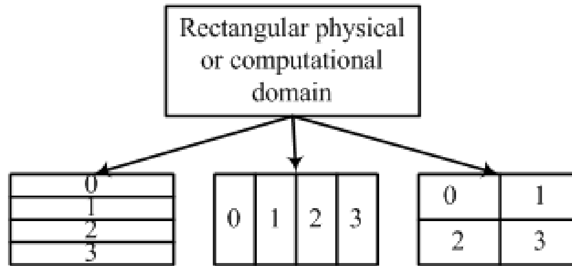

Figure 4.1: Three possible ways of decomposing a rectangular domain. The numbers in the sub-domains represent the processor number.

Several strategies exist within the data decomposition paradigm for dividing domains into sub-domains. In the two-dimensional grid, the computational domain is decomposed both in x and y coordinate directions. In many cases, the computation is proportional to the volume of a sub-domain and the communication is proportional to the surface area. In such cases, a logical strategy is to partition the domain in such away that it minimizes the surface area of each sub-domain relative to its volume. This keeps the computation-to-communication ratio high. In this study, two-dimensional decomposition is chosen and this involves assigning each sub-domain to a processor and solving the equations for that sub-domain on the respective processor. With the two-dimensional decomposition, no global information is required at any particular grid point and interprocessor communications are required only at the boundaries of the sub-domains. The inner-border of a sub-domain requires the outer border of the adjacent sub-domain during a time-step because of the spatial discretization [WWW 09], [WWW 10], [WWW 11].

4.4 Algorithm Design

In our work, the domain decomposition method is used to parallelize the tsunami model. In this method, the parallel algorithm is very similar to the serial algorithm with some additional routines added to facilitate the communication between processors. Using this method, all the processors involved in the parallel calculations basically perform the same computational operations. The only difference is in the data being processed in each processor.

The physical or computational domain used in our case study is rectangular in shape. In the domain decomposition method, the rectangular domain is divided into several smaller rectangular sub-domains, where the number of sub-domains is equal to the number of processors used. With 4 processors, for example, there are three possible ways of decomposing the domain into equal-area parts as depicted in Figure 4.1. The best decomposition depends not only on the

architecture of the system being used, but more importantly on the memory limitations on each node, especially in commodity clusters, as such, an assignment like two by two is recommended.

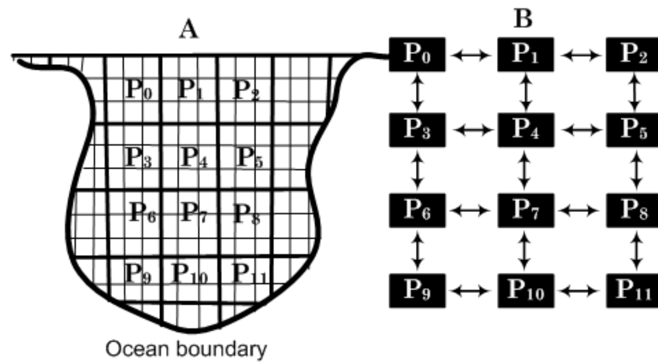

Figure 4.2: (A) Overview of the unstructured grid and assignment of worker nodes to sub-domains. (B) Arrows indicate the inner-worker communication.

An important aspect in decomposing the domain is the load balancing, i.e. all processors should have equal or almost equal amount of data to be processed. If the number of grid points is divisible by the number of processors, the grid points in each processor is simply the ratio of the number of grid points to processors. If it is not, the remainder is distributed across the first m processors, where m is the remainder. The load should be balanced in both x- and y-directions.

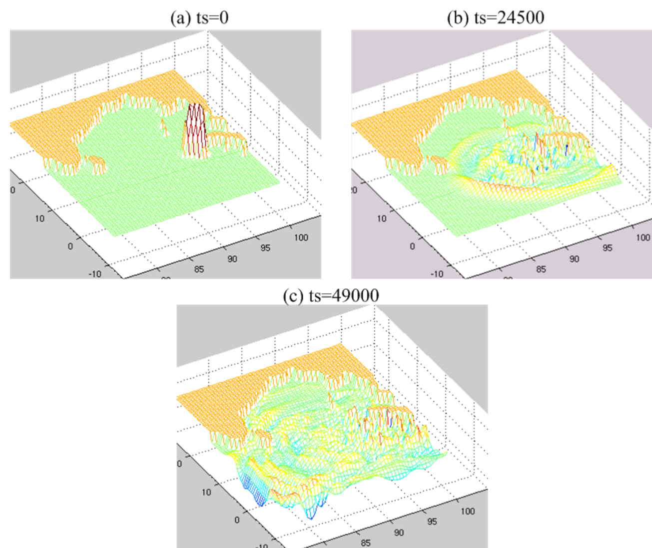

Figure 4.3: Linear shallow water wave profiles at three different time-steps (ts) calculated using 48 processors.

Two grid schemes, a 12-node (4×3) grid system and a 48-node (8×6) grid system, are used to parallelize the tsunami model with a domain size of 1226×900. Portions of the domains are assigned to each of the worker nodes, as illustrated in Figure 4.2 where each sub-domain is labeled with a processor (worker) number. Snapshots of the free surface evolution are shown in

Figure 4.3. Data for each sub-domain is stored with each processor including water depth, coordinates, and initial disturbance. This data is read by each of the workers in a pre-processing stage i.e., prior to initiating the time-integration loop. Since the model updates the solution explicitly using local data, each processor works independently of the others but requires data from neighboring workers to update solution along sub-domain boundaries. The exchange of data between processors occurs several times per time step. There are two key features to this exchange: First, only data along the boundaries between processors is exchanged. Second, each processor is only required to communicate with at most four other processors. The domain decomposition is performed with this second feature in mind to avoid communications between more than four other processors.

Communication occurs between two adjacent processors during message passing. In passing the data from one processor to another, an efficient and safe communication must be developed. To efficiently exchange the data between adjacent processors, the data is first stored in a contiguous memory prior to executing the sending processes. At the same time contiguous memories of the same size as used in the sending processes are created to receive the data from the sending processes. At this point, the data is ready for sending and receiving processes.

In this model, message passing occurs four times per time step, the MPI function, *MPI_Sendrecv*, and the TCP-Linda operations *in* and *out* are used to perform the data exchange between global nodes in the respective programming models. This corresponds to eight messages per time step per processor independent of the number of processors being used. Many more messages are sent during pre-processing, but these are ignored for run-time analysis purposes since time integration is by far the most time consuming element of the program. The generic parallel algorithm is outlined below. The original MPI FORTRAN version of the algorithm has been provided by (R.Wait, informal communication, March 10, 1996).

Algorithm 4.1: Generic parallel algorithm for the tsunami model

1. Decompose rectangular domain into load-balanced rectangular sub-domains, the width and the length of a rectangular sub-domain are approximately, $\left(\frac{W}{N_w}\right)$ and $\left(\frac{L}{N_l}\right)$, respectively, where W and L are the width and length of domain, and $N_P = N_w * N_l$ is the number of processors to be used in the $(N_w \times N_l)$ grid scheme.

2. Specify parallel language related parameters, such as locations, neighbors, sub-domain sizes, and file names, of processors. Location of kth processor is (r_k, c_k), where $r_k = \left(\left(\frac{k}{gdm1}\right) + 1\right)$ and $c_k = (k - (r_k - 1) * gdm1 + 1)$ with $gdm1$ and $gdm2$ being

dimension of the grid. In the four by three grid scheme, $gdm1 = 3$ and $gdm2 = 4$, and in the eight by six scheme, $gdm1 = 6$ and $gdm2 = 8$.

For $k = 0, 1, 2, \cdots\cdots, (gdm1 * gdm2 - 1)$, define $north_k$, $south_k$, $east_k$, and $west_k$ to be the neighbors located in the side of north, south, east and west of the kth processor, then:

$$\begin{aligned} north_k &= k - gdm1 \quad \text{if} \;\; r_k > 1 \\ south_k &= k + gdm1 \quad \text{if} \;\; r_k < gdm2 \\ east_k &= k + 1 \quad \text{if} \;\; c_k < gdm1 \\ west_k &= k - 1 \quad \text{if} \;\; c_k > 1 \end{aligned}$$

3. Input data and initial conditions:

   (a) Each processor, P_k, read the water depth, coordinates, and initial disturbance from the text file assigned in step 2.

   (b) Each processor, P_k, exchange sub-domain boundary data from $east_k$ to $west_k$, from $west_k$ to $east_k$, from $north_k$ to $south_k$, and from $south_k$ to $north_k$, in order.

4. Set parameters and coefficients used at the open sea boundary.

5. Repeat the following steps for the pre-defined time-steps:

   (a) Each processor, P_k, exchange sub-domain boundary data from $east_k$ to $west_k$.

   (b) Each processor, P_k, exchange sub-domain boundary data from $north_k$ to $south_k$.

   (c) Computation of the equation of continuity.

   (d) Setting of the open sea boundary condition.

   (e) Each processor, P_k, exchange sub-domain boundary data from $west_k$ to $east_k$.

   (f) Each processor, P_k, exchange sub-domain boundary data from $south_k$ to $north_k$.

6. Gather computed results among all processors.

7. Compute the output on processor, P_k, where both $south_k$ and $west_k$ are equal to null.

4.5 Implementations

In the domain decomposition method used, the rectangular computational domain is divided into several smaller rectangular sub-domains, where the number of sub-domains is equal to the number of processors used. Let W and L are the width and length of the computational domain, $N_P = (N_w * N_l)$ is the number of processors to be used in the N_w by N_l grid scheme,

and if the number of grid points is divisible by the number of processors. Then the width and length of a rectangular sub-domain are $\left(\frac{W}{N_w}\right)$ and $\left(\frac{L}{N_l}\right)$, respectively. Otherwise, the remainder is distributed across the first M_w and M_l processors in the N_w and N_l grid scheme, where M_w and M_l are the remainder of $\left(\frac{W}{N_w}\right)$ and $\left(\frac{L}{N_l}\right)$, respectively.

Since the model updates the solution explicitly using local data, each processor works independently of the others but requires data from neighboring workers to update solution along sub-domain boundaries as shown in Figure 4.2. Communication occurs between two adjacent processors using message passing. Therefore, we must explicitly program data transfers among processors.

Data is partitioned into sub domains, a four by three grid scheme and an eight by six grid scheme, used for the parallel implementation of this model. In each of the grid schemes, there are four parallel variations: (1) distributed memory without threads; (2) distributed memory with threads; (3) distributed shared memory without threads; and (4) distributed shared memory with threads. Implementations use the Message Passing Interface (MPI) library [WWW 01], the TCP-Linda [WWW 06] and the pthread library [WWW 12].

The mixed-mode programming model that uses thread programming on the shared memory layer and message passing programming on the distributed memory layer is a method commonly used to utilize the memory resources efficiently (Wu et al., 2002). In the distributed memory mixed-mode programming model, MPI is used for the data communication between the global nodes, and within each MPI process, pthreads are used. In the distributed shared memory mixed-mode approach, TCP-Linda is used for the data communication between the global nodes, and within each TCP-Linda process, pthreads are used.

In all, eight algorithmic variations were implemented on the high performance cluster, Monolith [WWW 13]. The Monolith cluster consists of over 396 nodes. Each node has two Intel Xeon processors at 2.2 GHz, 2 GBytes primary memory (ECC DDR), 512KB cache and a disk memory with 80 GBytes. In addition there is a common main disk storage of 7 TBytes. The operating system is Linux version 2.4.34-capl-smp. MPI (ScaMPI) on monolith runs on the SCI-network with a 2-rank ping-pong of around 250 MB/s and 4.5 micro seconds (large packet bandwidth and small packet latency respectively). TCP-Linda on monolith runs on TCP (which runs on a far from perfect 100 Mbps ethernet). Performance reach of MPI on TCP is around 10 MB/s and 60-100 microseconds.

4.5.1 Distributed Memory Implementation

MPI library is used to implement the Algorithm 4.1. Coordinates of each worker node at the assigned two dimensional grid scheme are computed according to their rank value. Rank of north, south, east, and west neighbors of each worker nodes, as depicted in Figure 4.2, are evaluated. Data for each sub-domain is read from a text file and is stored locally by each worker node, including water depth, coordinates, and initial disturbance. These computations and data assignments are done prior to initiating the time integration loop. Since the model updates solution explicitly using local data, each worker works independently of others but requires data from neighboring workers to update solution along sub-domain boundaries. The function *MPI_Sendrecv*, which is standard in MPI library, is used to exchange boundary data between neighboring workers. Twelve and forty eight MPI processes are created to simulate the parallel algorithm for the tsunami model under the four by three and eight by six grid schemes, respectively.

4.5.2 Distributed Shared Memory Implementation

TCP-Linda is used to implement the parallel algorithm. To achieve exchange of boundary data between neighboring workers, the *in* and *out* operations defined in TCP-Linda are used. The operation *eval* defined in TCP-Linda is used to create Linda processes at run time. This is the most important feature of Linda compared to MPI. Twelve Linda processes are created by calling *eval* at run time to simulate the parallel algorithm for tsunami model under the four by three grid scheme. Similarly, forty eight Linda processes are created to run the parallel algorithm under eight by six grid scheme.

4.5.3 Results

The MPI (ScaMPI) compiler and TCP-Linda are used to run the parallel algorithms for the tsunami model under DM and DSM environments, respectively, on the high performance computing cluster, Monolith [WWW 13]. The fundamental computations involved are matrix-vector multiplication, vector-vector multiplication, scaler multiplication and addition. We have used $49,000$ time steps to simulate the model. Twelve and forty eight computing nodes are assigned at run-time to simulate the model under four by three and eight by six grid schemes, respectively. Table 4.1 shows the timing values for the eight scenarios arising from the four parallel variations of the algorithm mentioned above. Table 4.2 shows the relative performance trade-offs within MPI and TCP-Linda implementations.

	MPI		TCP-Linda	
Nodes	wot	wt	wot	wt
12	9686.54	11053.54	2743.74	3297.78
48	11934.38	10273.73	3183.43	2845.34

Table 4.1: Run time in seconds (wot - without thread; wt - with thread)

	MPI	TCP-Linda
Thread overhead (12 node)	1.14	1.20
Increase in scalability overhead (12 to 48 node; without thread)	1.23	1.16
Decrease in scalability overhead (12 to 48 node; with threads)	0.93	0.86

Table 4.2: Running time factors.

4.5.4 Analysis of Results

Table 4.1 shows the average timing values for the eight scenarios arising from the four parallel variations of the algorithm mentioned above (Appendix E contains the timing values). Consider the four by three grid scheme, in both threading and non-threading programming environments. The parallel algorithms for the distributed shared memory exhibit better overall performance than the algorithms for distributed memory. Though the distributed shared memory implicitly passes messages, the replication management subsystem has been optimized in TCP-Linda compared to MPI [WWW 06], to yield better performance.

In the four by three grid scheme, in both scenarios for distributed and distributed shared memory, non-threading algorithms exhibit better performance than threading algorithms. One possible explanation for this is that each node keeps nine, two dimensional floating point type arrays of size equal to the sub domain size. Because of the memory limitations it is not possible to declare local two-dimensional arrays for threads, causing all threads in a node to concurrently use global arrays for their own computations.

Now consider the eight by six grid scheme with the identical input data size. Here too, for both threading and non-threading programming environments, the parallel algorithms for the distributed shared memory exhibit better overall performance than the algorithms for distributed memory. In contrast to previous instance, both algorithms for distributed and distributed shared memory, the threading algorithms exhibit better overall performance than non-threading algorithms. This is because that the sub domain size allocated to each node is half size of sub domain size of the smaller grid scheme of four by three size. The local two

dimensional floating point type array allocation for threads in a node is now possible compared to the four by three grid system. Therefore, since all threads of a node work independently, the threading parallel algorithm shows better overall performance than non-threading parallel algorithm.

Among the parallel variations not using threads in both memory architectures, the four by three grid scheme shows better performance than eight by six grid scheme. This is due to the ratio of computation time to communication overhead decreasing faster for domains with smaller size. However, when the task is scaled up, say up to eight by six grid system, due to the smaller sub-domain sizes aligning with the available memory in the node, the threaded versions become more efficient in both MPI and TCP-Linda implementations. In both threading and non-threading environments, the TCP-Linda version exhibits better performance than the MPI version. Also, independently in MPI and TCP-Linda scenarios, best timing is obtained for the smaller grid size of (4×3), without threads.

Typically, each node in the grid exchanges boundary data among each of their peer neighbors. Eight messages per node per time step and 49000 time steps gives 392000 messages involving large arrays of floating point numbers.

Table 4.2 shows the relative performance trade-offs within MPI and TCP-Linda implementations. Significantly, the 12 node thread overhead for TCP-Linda is higher than the equivalent MPI version owing to the effect of sharing data structures. The scalability overhead is much lower in TCP-Linda compared to MPI, implying the efficiency of TCP-Linda middleware for large scale problems. Where memory space permits both MPI and TCP-Linda to operate threaded versions have more or less the same overhead with respect to their best timings.

Chapter 5

Artificial Neural Network Model on a Cluster

Numerous advances have been made in developing intelligent systems, some inspired by biological neural networks. Artificial Neural Networks (ANNs) are based on the present understanding of the human biological nervous system (Long & Gupta, 2008). It represents a massively parallel system with hierarchical structured interconnection. This feature inspires the ANN researchers to design systems that are based on parallel processing as a alternative to the sequential processing architecture. ANNs have been shown to be powerful computational models that can effectively address forecasting stock processes, complex pattern classification, pattern recognition, face recognition, medical imaging, control, nonlinear function approximations, etc. (Sudhakar & Murthy, 1998; Suresh et al., 2005; Long & Gupta, 2008; Bazanov et al., 2002; Pethick et al., 2003; Jain et al., 1996). Although successful applications can be found in certain well-constrained environments, none is flexible enough to perform well outside its domain (Jain et al., 1996). ANNs provide exciting alternatives, and many applications could benefit from using them. Though ANNs have been studied since 1940s they are a relatively new method for modeling and forecasting financial data (Thulasiram et al., 2003). Time series analysis has direct implication in financial markets (Thulasiram et al., 2003; Chan et al., 2000; Chen et al., 2003; Skabar & Cloete, 2002). The high degree of complexity present in ANNs makes them computationally expensive to simulate (Novokhodko & Valentine, 2001; Long & Gupta, 2008). A class of ANN paradigms involve a learning phase in which the ANN is trained with a set of already known cases of a problem. The trained ANN is then used in a real environment to solve unknown instances of the problem. This is known as the recall phase (Jain et al., 1996). The learning phase usually takes a large amount of computing time for all but simple toy problems. For practical problems where the training data is large, training times of the order of days and weeks are not uncommon on serial machines (Long & Gupta, 2008; Sudhakar & Murthy, 1998; Suresh et al., 2005; Pethick et al., 2003; Sundararajan & Saratchandran, 1998). This has been the main stumbling block for ANNs use in real-world

applications and has also greatly impeded its wider acceptability. The problem of large training time can be overcome either by devising faster learning algorithms or by implementing the existing algorithms on parallel computing architectures (Yoon et al., 1990; Sudhakar & Murthy, 1998; Suresh et al., 2005). Improving the learning algorithm per se is an active area of research (Sheng Ma & Farmer, 1997), Osowski & Stodolski, 1996), but this case study focuses on the latter approach of parallel implementation. The good thing about this approach is that an improved fast learning algorithm can be further speeded up by parallel implementation.

Parallel architectures for implementing ANN can be subdivided into general purpose parallel computers and neurocomputers. Neurocomputers are designed as hardware boards and systems for high-speed ANN simulations (Atlas & Suzuki, 1989). Neurocomputers can be classified as general purpose or special purpose (Treleven, 1989). A general-purpose neurocomputer is programmable and is capable of supporting a large range of ANN models, whereas a special purpose neurocomputer implements one neural model on dedicated hardware. The latter benefits from higher speed than the former. Most neurocomputers are based on processing elements computing in parallel. A survey by Solheim (1995) lists about 80 different digital neural hardware projects. However at the time, only twenty of these proposed architectures have been implemented.

ANNs can be subdivided into three main groups according to the training approach (Hu et al., 2007; Heaton, 2005): (1) supervised, (2) unsupervised, and (3) various hybrid approaches. Supervised training is accomplished by giving the ANN a set of sample data along with the anticipated outputs from each of these samples. It is the most common form of ANN training. As supervised training proceeds, the ANN is taken through several iterations, or epochs, until the actual output of the ANN matches the anticipated output, with a reasonably small error. Each epoch is one pass through the training samples.

Unsupervised training is similar to supervised training except that no anticipated outputs are provided. This training usually occurs when the ANN is to classify the inputs into several groups. The training progress through many epochs, just as in supervised training. As training progresses the classification groups are ‘discovered’ by the ANN.

There are number of hybrid methods that combine several aspects of supervised and unsupervised training. One such method is called *reinforcement training* (Hu et al., 2007). In this method, the ANN is provided with sample data that does not contain anticipated outputs, as is done with unsupervised training. However, for each output, the ANN is told whether the

output was right or wrong given the input.

Multi-layer perceptron (MLP), Hopfield net, and Hamming net networks are based on supervised trained ANN (Jain et al., 1996). Among them MLP ANN is most popular for supervised learning (Pethick et al., 2003). It consists of several layers of computational elements. ANN research grew rapidly with the introduction of the back-propagation algorithm for training the feed-forward ANN. This training algorithm is used in the research presented in this case study, and the algorithm will be thoroughly explained in the following sections.

Parallel algorithms for some of ANN models and applications including a few on financial forecasting have been reported (Sudhakar & Murthy, 1998; Long & Gupta, 2008; Araiijo et al., 2003; Bazanov et al., 2002; Novokhodko & Valentine, 2001; Wang et al., 1989; Thulasiram et al., 2003). Most of this work has been focused on special purpose hardware implementations that provide a high degree of parallelism, on mapping ANNs onto conventional shared memory multiprocessor parallel computers (Pethick et al., 2003), or on mapping ANNs onto distributed memory multiprocessor clusters (Suri et al., 2002). Clusters have been widely used to make a flexible infrastructure for development of parallel algorithms (Lotrič & Dobnikar, 2005; Suri et al., 2002).

The MLP with the back-propagation learning algorithm is one of the most widely used training algorithms for ANNs (Long & Gupta, 2008; Novokhodko & Valentine, 2001; Pethick et al., 2003; Suresh et al., 2005; Marchesi et al., 1990). However, it makes extensive use of computational resources and training times of the order of days are not exceptional. One approach to solve this problem is to parallelize of the training algorithm (Araiijo et al., 2003; Hung & Adeli, 1994; Suresh et al., 2005; Seiffert, 2002). There are three categories of parallel implementation models for the ANNs: network based model, training set based model, and combination of network based and training set based model. In the network based model, the ANN is partitioned into number of sub ANNs and each sub ANN is mapped into a processor. This approach is more communication intensive, since the processors need to broadcast results for each layer (Novokhodko & Valentine, 2001).

In the training set based model, the training patterns are distributed among processors. Every processor processes independently and processes only the patterns that are assigned to it. When each individual ANN is trained, they either select the best one or produce a new ANN averaging gathered weights.

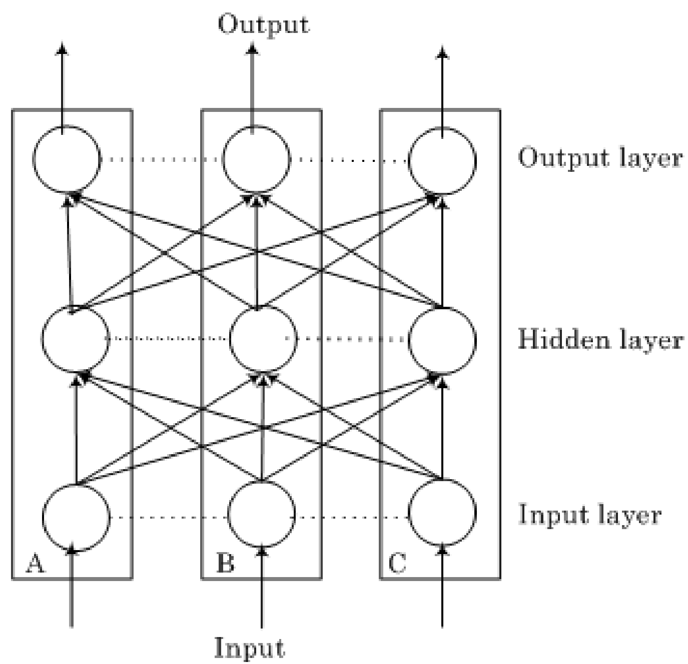

Figure 5.1: Vertical partition of an ANN (Sudhakar & Murthy, 1998).

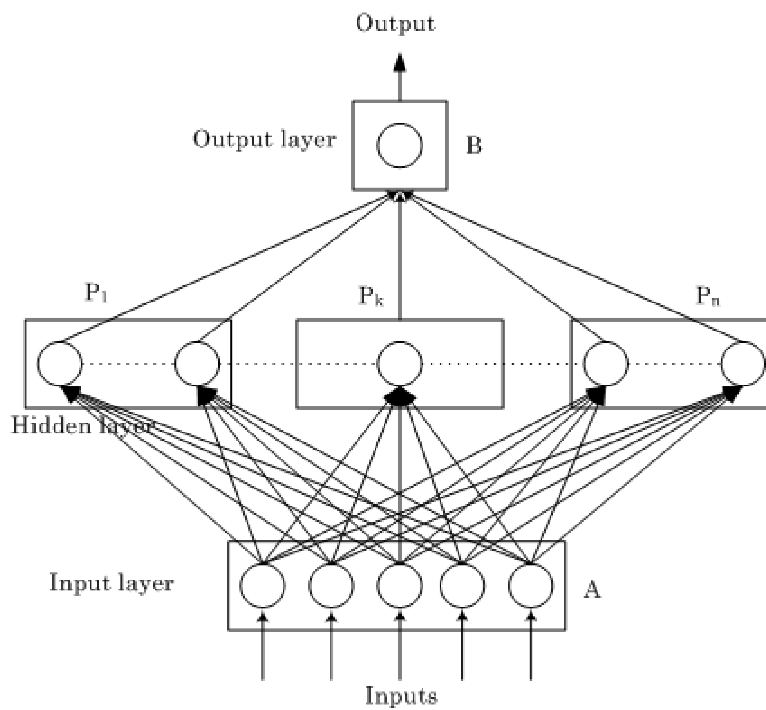

Figure 5.2: Hybrid partition of an ANN (Suresh et al., 2005).

In the network partitioning model, there are two methods, vertical partition (VP) (Sudhakar & Murthy, 1998) and hybrid partition (HP) (Suresh et al., 2005), for the parallel implementation of a ANN. In the vertical partitioning scheme as shown in Figure 5.1, each layer, $L_l,\ \ l = 1, 2, 3$, having N_l neurons, is divided into N_P partitions, where N_P is the number of processors to be allocated to simulate the ANN. Each partition has $\left(\frac{N_l}{N_P}\right)$ neurons that are assigned to a processor (Sudhakar & Murthy, 1998).

In the HP scheme as shown in Figure 5.2 , the hidden layer is partitioned using neuronal level parallelism and weight connections are partitioned on the basis of synaptic level parallelism (Suresh et al., 2005). In case of a homogeneous N_P-processor network, the hidden layer, having N_h neurons, is partitioned into $\left(\frac{N_h}{N_P}\right)$ neurons and the input and output neurons are common for all the processors. Each processor stores the weight connection between the neurons local to the processor. The HP scheme does not need recomputing the weights and require less communication cycle per pattern (Suresh et al., 2005). The data exchange among processors in the computing network is carried out by using grouped all-to-all broadcast.

In our case study, we use a hybrid partitioning scheme for a MLP ANN with a parallel back-propagation algorithm.

The important issue here is the mapping of the ANN model into a underlying network. Based on the type of workloads, the mapping scheme can be divided into dynamic mapping and static mapping (Suresh et al., 2005). In the dynamic mapping, the ANN partition will change with time due to other work loads in the underlying network. In the static mapping, ANN is partitioned and mapped onto the network and this partition remains until training process is completed. Since simple and computationally less intensive, the static partition has been used for our study. The HP scheme provides a way to formulate the static mapping problem as a way to schedule the load in the high performance cluster (Suresh et al., 2005).

5.1 Parallelization of Feed-Forward ANNs

Before the discussing of parallel implementation scheme, it is useful to define the basic terminology along with the different types of parallelism possible for back-propagation ANNs. Also, the commonly used measures for evaluating the performance of the parallel implementations are described:

Training set: consists of a number of training patterns, each given by an input vector and

the corresponding output vector.

ANN *size*: An ANN of N_I input units, N_H hidden units, and N_O output units is for short written $N_I \times N_H \times N_O$.

Iteration: Denotes one presentation of the whole training set.

Weight updating strategies: Three different approaches:

- *Learning by pattern*: update the weights after each training pattern has been presented.
- *Learning by block*: update the weights after a subset of the training patterns has been presented.
- *Learning by epoch*: update the weights after all patterns have been presented (i.e. One training iteration).

Weight update interval: The number of training patterns that is presented between weight updates is termed μ. For learning by pattern, $\mu = 1$, while for learning by epoch, $\mu = P$, where P is the number of training patterns in the training set.

The BP algorithm reveals four different kinds of parallelism (Thulasiram et al., 2003; Babii, 2007; Nordström & Svensson, 1992; Singer, 1990):

1. ***Training sessions parallelism***: Different processors run the algorithm with different initial training parameters. In this methods, the only advantage is the acceleration of the phase of determining the most suitable training parameters.

2. ***Training set parallelism***: Splits the training set across the processing elements. Each element has a local copy of the complete weight matrix (i.e. copy of the ANN) and accumulates weight change values for the given training patterns. Weights are updated using (learning by block/learning by epoch). This method is also named *data partitioning* or *job parallelism* (Sundararajan & Saratchandran, 1998).

3. ***Pipelining***: Pipelines the training patterns between the layers, i.e. compute hidden and output layer on different processors. While the output layer processor calculates output and error values for the present training pattern, the hidden layer processor processes the next training pattern. The forward and backward phase may also be parallelizes in a pipeline. Pipelining requires a delayed weight update or (learning by block/learning by epoch).

4. ***Node parallelism***: The neurons within a layer is computed in parallel. Further, the computation within each neuron may also run in parallel. In this method, the weights can

be updated using learning by pattern. This method is also known as *network partitioning* (Sundararajan & Saratchandran, 1998; Surech, 2005).

5.2 Related Work

5.2.1 Mapping of a Multilayered ANN onto a Network of Workstations

The important issue in parallel algorithm design is the mapping of the ANN architecture onto the HPC cluster. The optimal mapping depends on the type of computing network and workloads. Based on the type of workloads, the mapping scheme can be divided into static mapping and dynamic mapping. The mapping problem is generally a nonlinear mixed integer programming optimization problem with communication and memory constraints. The optimal static mapping of MLP ANN in multicomputers have been formulated as a mixed integer programming problem (Chu & Wah, 1992). The mapping can be solved using approximate linear heuristic optimization methods (Amawy & Kulasinghe, 1997; Ghosh & Hwang, 1989). An attempt to solve the static mapping problem using genetic algorithm is described in (Sundararajan & Saratchandran, 1998). The mapping problem using the approximate linear heuristic methods and genetic algorithm is computationally intensive (Suresh et al., 2005).

Two network partitioning schemes, vertical partition (VP) (Sudhakar & Murthy, 1998) and hybrid partition (HP) (Suresh et al., 2005), have been developed to map the ANN onto a network of workstations.

Sudhakar & Murthy (1998) have developed an efficient method for mapping a back-propagation learning algorithm for a fully connected multilayer ANNs onto a network of workstations (NOWs), where a per-pattern training regime to train the ANN has been used. A VP technique has been used to partition the ANN and each partition is mapped onto processors in NOWs. The experimental results obtained from the presented technique have been compared with an algorithm suggested by Yoon et al. (1990), employing a VP scheme for implementation on a distributed memory multiprocessor. The algorithm presented in Sudhakar & Murthy (1998) uses only two sets of communication, compared to the three sets of communication used in Yoon et al. (1990). The computation of the weights is saved by avoiding re-computing weights joining the neurons on the same processor. Further, the algorithm presented in Sudhakar & Murthy (1998) employs a grouped-broadcast strategy to broadcast all the values at a processor instead of the one-by-one and all-to-all broadcasting, to reduce the communication setup time. Experimental and analytical results of Sudhakar & Murthy(1998) show that it is

able to achieve better speedups than the algorithm of Yoon et al. (1990).

Suresh et al. (2005) have developed an efficient mapping scheme for the multilayer fully connected perceptron ANN trained using a back-propagation algorithm on NOWs. An HP scheme is used to partition the network and each partition is mapped onto processors in NOWs. The performance of the algorithm in Suresh et al. (2005) is compared with the algorithm developed in Sudhakar & Murthy (1998). The improvements of the HP scheme over the VP scheme (Sudhakar & Murthy, 1998) are: (i) HP scheme uses only one set of communication, when compared with two sets of communication used in VP scheme; (ii) recomputation of weights is avoided in HP scheme; (iii) the HP scheme can exploit the parallelism by using more number of processors than the VP scheme; and (iv) mapping the HP scheme on NOWs is simpler than the VP scheme. Another advantage of the HP scheme is that, a closed-form expression for optimal number of processors can be formulated. Analytical results for the benchmark problems and the experimental results for optical character recognition problem show that the HP method is efficient and computationally less intensive than the VP scheme Suresh et al. (2005).

5.2.2 Substituting Parallel Matrix Multiplication in the Training Phase of Back-propagation ANNs

Novokhodko & Valentine (2001) have proposed an approach for the parallel implementation of the batch back-propagation training for ANNs, using a universal parallel matrix multiplication routine implemented in MPI. This substitution uses the well established and widely used tools provided by MATLAB for the training of batch back-propagation ANNs. The NETTALK reference problem that has become a standard benchmark for ANN algorithms is used to test the performance of the ANN training algorithms. Two different training algorithms, the resilient back-propagation algorithm (TRAINRP) and the gradient descent with momentum and adaptive learning rate (TRAINGDX), have been used. Both algorithms are from MATLAB ANN toolbox. TRAIGNGDX gives better speed since it is less intelligent than TRAINRP. TRAINRP do some extra computations to reduce the number of epochs, which results in an added time spent for one epoch. Further, A speedup of 1.46 was achieved using 4 processors. A training speed of 12.43 MCPUS was reached on 4 Sun Ultra 5's. That is comparable with results given in Yasunaga & Yoshida (1998).

5.2.3 Parallel ANN Training Algorithms for Finance Applications

ANN training algorithms can be used for price forecasting and other finance applications (Chan et al., 2000; Chen et al., 2003; Skabar & Cloete, 2002). Thulasiram et al. (2003) have designed and developed four different parallel and multi-threaded back-propagation ANN training algorithms: neuron and training set parallelism on a DM system using MPI; loop-level (fine-grain) and coarse-grained parallelism in SM system using OpenMP. Theoretical results are derived for all four parallel algorithms and these results are compared with the experimental timing results. Performance results of the parallel algorithms on Beowulf cluster using MPI and on SMP using OpenMP have been presented. The training error for the parallel implementations in MPI, OpenMP, and the traditional autoregression model namely the ARMA model at various epochs for a price forecasting problem are presented.

Parallel algorithms that have been developed perform better than the ARMA model. Timing results for fine-grain algorithm for neuron parallelism in OpenMP and for neuron parallelism in MPI show that the performance is best for approximately 60 threads after which the execution time starts to increase. This is due to the load imbalance and the synchronization cost of entering the critical section. Also, there is an explicit barrier at the end of each loop, which is added by the system. But, there is a gradual decrease in execution time of the neuron parallelism in MPI, as the number of processors are increased for different epochs. However, the parallel implementation for neuron parallelism in MPI does not produce good performance with increasing number of processors. There is a significant difference in execution time of the training set parallelism for MPI, with even two processors compared to the neuron parallelism in MPI.

Compared with the neuron parallelism in MPI, the CGP in OpenMP shows better execution time. This is because there is less associated communication cost in the shared memory system as broadcast is eliminated by a common shared variable in critical section. Finally, the authors have concluded that the training set parallelism algorithm implemented on MPI provides the best results, and as one of the first attempts to parallelize the ANN for financial application, the results are encouraging in relation to the overall timings with traditional autoregression models (Thulasiram et al., 2003).

5.2.4 Massively Parallel ANNs

An object-oriented, massively-parallel ANN software package SPANN (Scalable Parallel ANN) has been developed in (Long & Gupta, 2008). SPANN runs on massively parallel computers and uses the back-propagation training algorithm. The software implements several C++ objects, such as Neuron, Layer, and ANN. MPI has been used to parallelize the C++ code. The software can define any number of layers and each layer can have any number of neurons. Neurons in each layer are distributed roughly equally among all the processors, and all the inputs are fed to each processor. Only the neurons on the edges of the domains were involved in communication, in order to reduce the communication costs and maintain scalability. The serial runs have been performed on an IBM P640 RS6000 Server. The parallel code has been tested using roughly the same number of weights as in the serial case. Results have been obtained from a 160-processor Beowulf computing cluster. The application has been used to identify character sets consisting of 48 characters and with increasing resolutions. Percentage of correct characters recognized against number of training iterations for different weights in a serial ANN has been presented. As is known, there seems to be an optimal value of weights for the ANN, which gives the most correct character recognitions for lesser number of iterations. This is due to the fact that the ANN is unable to learn due to under fitting when there are less than the required numbers of weights, and also, when the number of weights is much larger than the required, the ANN tends to remember rather than learn, due to over fitting.

The code correctly identified all the characters when adequate training was used in the ANN. The training of a problem with 2 billion neuron weights on an IBM BlueGene/L computer using 1000 dual PowerPC 440 processors required less than 30 minutes.Various comparisons in training time, forward propagation time, and error reduction have also be made. The software is scalable and permits the use of billions of weights. The approach also allows an ANN to be trained on a massively parallel computer and then to be used on small serial computers, in the recognition phase.

5.2.5 Factors Affecting the Training of a Distributed Back-propagation algorithm

Babii (2007) has proposed and developed an approach to make existing algorithms run faster by modifying them to run on some parallel architectures. An implementation with training set parallelism on a set of workstations in a LAN has been analyzed. The qualitative aspect of the analysis, in order to achieve a fair understanding of the factors determining the behavior of this parallel algorithm, has been the focussing point. Further, authors have been interested

in discovering and dealing with some of the specific circumstances that have to be considered when a parallelized ANN learning algorithm is to be implemented on a set of workstations in a LAN. Since training set parallelism is used, only one modification is necessary to the sequential algorithm: a scheme for performing the collection and summation of all component gradients and the distribution of updated weights. Summation of component gradients using a ring configuration of P processors in such a way that each processor after $(P-1)$ steps will hold the total gradient has been implemented. An advanced strategy, which minimizes the redundance is presented, where the processors do not keep a copy of the ANN. The weights are stored only in a single processor, the administrator, and these weights are circulated, one by one, to the other processors (the slaves). The administrator does not process training patterns. It controls only the weight circulation and updates them. The training patterns are evenly distributed between the other processors. This method is no more based on the classic method, where a pattern is propagated forward and backward, and a single gradient is calculated, before presenting the next pattern. Efficiency against batch size graphs have been presented. Efficiency of the algorithm is very low for the smallest batch size, since each processor in this case presents only one single pattern to the ANN between each weight update. As the size of the batch is increased execution time becomes more and more dominated by the time used for presenting patterns and calculating component gradients, since the number of weight updates is not increased. This way the time used for updating the weights becomes a smaller and smaller part of the total execution time, hence efficiency increases.

Efficiency against number of processors for a fixed problem size shows that, maximum speed up has not been reached even when using the maximum number of processors. As the number of processors are increased each processor handles a smaller part of the batch. This makes the processors run less efficiently. For the same batch size per processor, the efficiency decreases when the number of processors become larger. This is probably because of the time needed for collecting component gradients and distributing the weight updates.

5.3 Design of a Parallel Algorithm

The following assumptions are used in our case study for a feedforward ANN:

1. An ANN has only three layers the input, one hidden, and the output layer.
2. N_I neurons in the input layer, N_H neurons in the hidden layer, and N_O neurons in the output layer.
3. The total training patterns is denoted as N_R.

In our case study, we use the HP scheme shown in Figure 5.2, for a MLP ANN with a parallel back-propagation algorithm. The HP scheme does not need recomputing the weights and require less communication cycle per pattern (Suresh et al., 2005). The data exchange among processors in the computing ANN is carried out by using grouped all-to-all broadcast. Three parallel algorithms are used to simulate the ANN under each of the memory systems, DM and VSM.

The first algorithm, HP-with-IBP, uses hybrid partitioning approach with incremental update back-propagation. The second algorithm, NHP-with-BBP, uses the batch back-propagation approach with parallel matrix multiplication. Finally the third algorithm, HP-with-BBP, uses hybrid partitioning approach with batch back-propagation feature within each sub ANN with matrix multiplication. The latter is the variant designed by us.

ANN name	**ANN structure** $N_I \times N_H \times N_O$	**Total weights**
ANN_1	$5 \times 7 \times 1$	42
ANN_2	$36 \times 72 \times 1$	2664
ANN_3	$72 \times 144 \times 1$	10512

Table 5.1: Artificial neural network architectures

The two memory architectures under study are the distributed memory and the virtual shared memory. The details of the high performance cluster used are given in Section 5.4. The HP scheme is used to partition the ANN, and each partition is mapped to a single processor. In our case study, three, 3-layer ANNs as shown in Table 5.1, ANN_1, ANN_2, and ANN_3, are used. The ANN_1 contains 5 input nodes, 7 hidden nodes, and 1 output node. ANN_2 contains 36 input nodes, 72 hidden nodes, and 1 output node. ANN_3 contains 72 input nodes, 144 hidden nodes, and 1 output node. Number of processors ranging from 1 to 15 are used for the parallel implementation of this model. For each of the ANN models we present two parallel algorithms one for DM and one for VSM. Experiments are realized using the Message Passing Interface (MPI) [WWW 01] library and the TCP-Linda [WWW 06].

In each scenario, training takes place over a number of epochs until the average error for the patterns falls below a pre-defined value. Once convergence has been realized, the ANN is tested for its ability to correctly generalize as yet unseen input patterns.

5.3.1 Batch update back-propagation with parallel matrix multiplication (NHP with BBP)

There are two weight update approaches, batch update and incremental update. In batch mode the weights and biases of the ANN are updated only after the entire training set has been applied to the ANN. The gradients calculated at each training example are added together to determine the change in the weights and biases. In incremental update the ANN is updated after each training pattern. The first one is more suitable for a parallel implementation, since all patterns are presented to the same ANN (Novokhodko & Valentine, 2001; Sundararajan & Saratchandran, 1998; Sheng Ma & Farmer, 1997).

Consider the batch back-propagation training of multilayer perceptron (Novokhodko & Valentine, 2001): N_R is the number of input patterns, m_o is the number of input neurons, and m_i is the number of neurons in layer i.

$\mathbf{X}$ – input data matrix, m_0 by N_R.
$\mathbf{W}^i$ – weight matrix of layer i, m_i by m_{i-1}.
$\mathbf{Y}^i$ – output matrix of layer i, m_i by N_R. $\mathbf{Y}^0 = X$.
$\mathbf{A}^i = \mathbf{W}^i\mathbf{Y}^{i-1}$ – input matrix to the neurons in layer i, m_i by N_R.
$\mathbf{f}^i\left(\mathbf{A}^i\right)$ transfer function of layer i, m_i by N_R.
$\mathbf{Y}^i = \mathbf{f}^i\left(\mathbf{W}^i\mathbf{Y}^{i-1}\right)$, m_i by N_R.
$\mathbf{Y} = \mathbf{Y}^L$ – the output matrix of an ANN with L layers, m_L by N_R.
$\mathbf{Y(X)} = \mathbf{f}^L\left(\mathbf{W}^L\mathbf{f}^{L-1}\left(\mathbf{W}^{L-1}\mathbf{f}^{L-2}\left(\cdots\cdots\mathbf{f}^2\left(\mathbf{W}^2\mathbf{f}^1\left(\mathbf{W}^1\mathbf{X}\right)\right)\cdots\cdots\right)\right)\right).$
$\mathbf{D}$ – desired output for the input $\mathbf{X}$, m_i by N_R.
$\mathbf{E(X)=Y(X)\text{-}D}$ – error on the input X, m_i by N_R.

Thus, to parallelize the algorithm, we need to parallelize the matrix multiplication operation. Consider the matrix product, $\mathbf{W}^1\mathbf{X}$. Matrix $\mathbf{X}$ is the training set, it has as many columns as there are input patterns. Matrix $\mathbf{W}^1$ is the input weight matrix, it has as many rows as there are neurons in the first hidden layer. Let $N_P = N_{P_1} * N_{P_2}$ – number of processors, organized in a grid N_{P_1} by N_{P_2}. Parallelizing the matrix multiplication each processor will calculate a part of the result using a subset of training data and a subset of neurons' weights. Therefore, such a matrix product does both data parallelization and node parallelization.

Since three layers are used in our case study, the output matrix of our ANN is expressed as follows:

$$\mathbf{Y(X)} = \mathbf{f}^2\left(\mathbf{W}^2\mathbf{f}^1\left(\mathbf{W}^1\mathbf{X}\right)\right).$$

$$\mathbf{A}^o = \mathbf{f}\left(\mathbf{W}^o\mathbf{f}\left(\mathbf{A}^h\right)\right), \quad \text{where} \quad \mathbf{A}^h = \mathbf{W}^h\mathbf{X}.$$

$$\begin{pmatrix} a^h_{1,1} & . & a^h_{1,N_R} \\ a^h_{2,1} & . & a^h_{2,N_R} \\ . & . & . \\ . & . & . \\ a^h_{N_H,1} & . & a^h_{N_H,N_R} \end{pmatrix} = \begin{pmatrix} w^h_{1,1} & . & w^h_{1,N_I} \\ w^h_{2,1} & . & w^h_{2,N_I} \\ . & . & . \\ . & . & . \\ w^h_{N_H,1} & . & w^h_{N_H,N_I} \end{pmatrix} * \begin{pmatrix} x_{1,1} & . & x_{1,N_R} \\ x_{2,1} & . & x_{2,N_R} \\ . & . & . \\ . & . & . \\ x_{N_I,1} & . & x_{N_I,N_R} \end{pmatrix}$$

$$\begin{pmatrix} a^o_{1,1} & . & a^o_{1,N_R} \\ a^o_{2,1} & . & a^o_{2,N_R} \\ . & . & . \\ . & . & . \\ a^o_{N_O,1} & . & a^o_{N_O,N_R} \end{pmatrix} = \begin{pmatrix} w^o_{1,1} & . & w^o_{1,N_H} \\ w^o_{2,1} & . & w^o_{2,N_H} \\ . & . & . \\ . & . & . \\ w^o_{N_O,1} & . & w^o_{N_O,N_H} \end{pmatrix} * \begin{pmatrix} a^h_{1,1} & . & a^h_{1,N_R} \\ a^h_{2,1} & . & a^h_{2,N_R} \\ . & . & . \\ . & . & . \\ a^h_{N_H,1} & . & a^h_{N_H,N_R} \end{pmatrix}$$

Start with randomly chosen weight matrices $\mathbf{W}^h$ and $\mathbf{W}^o$.

While $MSE = \frac{1}{2N_R}\sum_{k=1}^{N_O}\sum_{p=1}^{N_R}(t(k,p) - O(k,p))^2$ is unsatisfactory and computational bounds are not exceeded, **do**

1. Compute the hidden layer activation:

   Compute the matrix multiplication, $\mathbf{A}^h = \mathbf{W}^h * \mathbf{X}$, in parallel.

   Where $\mathbf{W}^h = (w^h_{j,i})$, $1 \leq j \leq N_H$, $1 \leq i \leq N_I$, is the weight matrix between the hidden and the input layers, and $w^h_{j,i}$ is the weight between j^{th} hidden neuron and i^{th} input neuron.

   $\mathbf{X} = (x_{i,p})$, $1 \leq i \leq N_I$ and $1 \leq p \leq N_R$, is the input matrix. The pth column of $x_{i,p}$ is the pth input pattern.

   $\mathbf{A}^h = (a^h_{j,p})$ is the activation matrix, and $a^h_{j,p}$ is the activation value of the jth hidden neuron for the pth input pattern.

$\mathbf{B}^h = \left(b^h(j)\right) \quad ,1 \le j \le N_H$ is the bias vector for the hidden layer.

Add the bias value to each element of $\mathbf{A}^h$ and ten apply the activation function as follows:
$a^h(j,p) = f\left(a^h(j,p) + b^h(j)\right), \quad 1 \le j \le N_H$ and $1 \le p \le N_R$

$f(x) = \dfrac{1}{1+e^{-x}}$ is the sigmoid function which is used to compute the activation of each neuron in the hidden and output layers.

N_I and N_H are the number of neurons in the input and the hidden layers, respectively. N_R is the number of input patterns to be used to train the ANN.

2. Compute the output layer activation:
   Compute the matrix multiplication $\mathbf{A}^o = \mathbf{W}^o * \mathbf{A}^h$ in parallel.

   Where $\mathbf{W}^o = \left(w^o_{k,j}\right), \quad 1 \le k \le N_O$ and $1 \le j \le N_H$, is the weight matrix between the output and the hidden layers, and $w^o_{k,j}$ is the weight between k^{th} output neuron and j^{th} hidden neuron.

   $\mathbf{A}^o = (a^o_{k,p})$ is the activation matrix for output layer, and $a^o_{k,p}$ is the activation value of the kth output neuron for the pth input pattern.

   $\mathbf{B}^o = (b^o(k)) \quad ,1 \le k \le N_O$ is the bias vector for the output layer.

   Add the bias value to each element of $\mathbf{A}^o$ and then apply the activation function as follows:
   $a^o(k,p) = f\left(a^o(k,p) + b^o(k)\right), \quad 1 \le k \le N_O$ and $1 \le p \le N_R$. N_O is the number of neurons in the output layer.

3. Compute the output layer error:
   Do the following computation in parallel:
   $e^o(k,p) = a^o(k,p) * (1 - a^o(k,p)) * (a^o(k,p) - t(k,p)), \quad 1 \le k \le N_O$ and $1 \le p \le N_R$
   $e^o(k,p)$ is the error of the kth output neuron for the pth input pattern.

4. Compute the hidden layer error:

   Let $\mathbf{E}^o = (e^o(k,p))$ and $\mathbf{E}^h = \left(e^h(j,p)\right)$, $1 \le k \le N_O$, $1 \le p \le N_R$, $1 \le j \le N_H$.

   Compute the matrix multiplication, $\mathbf{W}^o\mathbf{E}^o = (\mathbf{W}^o)^T \times \mathbf{E}^o$, in parallel.

   Do the following computation in parallel:

   $e^h(j,p) = A^h(j,p) * (1 - A^h(j,p)) * W^oE^o(j,p)$.

   $e^h(j,p)$ is the error of the jth hidden neuron for the pth input pattern.

5. Compute the weight changes and then update weights at the output layer:

   Compute, $\mathbf{W}^o = \mathbf{W}^o + \Delta\mathbf{W}^o$

   where $\Delta\mathbf{W}^o$ is a matrix representing the weight changes for matrix $\mathbf{W}^o$ and is computed as follows:

   Compute the matrix multiplication, $\mathbf{W}^o_c = \mathbf{E}^o \times (\mathbf{A}^h)^T$, in parallel, and then compute,

   $\Delta\mathbf{W}^o_t = \alpha(\mathbf{W}^o_c) + \Theta\mathbf{W}^o_{t-1}$

   α is the learning rate and Θ is the momentum factor used to allow the previous weight change to influence the weight change in this time period, t.

6. Compute the weight changes and then update weights at the hidden layer:

   Compute, $\mathbf{W}^h = \mathbf{W}^h + \Delta\mathbf{W}^h$

   where $\Delta\mathbf{W}^h$ is a matrix representing the weight changes for matrix $\mathbf{W}^h$ and is computed as follows:

   Compute the matrix multiplication, $\mathbf{W}^h_c = \mathbf{E}^h \times \mathbf{X}^T$, in parallel, and then do the computation, $\Delta\mathbf{W}^h_t = \alpha\mathbf{W}^h_c + \Theta\mathbf{W}^h_{t-1}$

7. Compute the bias changes and then update biases at the output layer:

   $$b^o(k) = b^o(k) + \alpha\sum_{p=1}^{N_R} e^o(k,p), \quad 1 \le k \le N_O.$$

8. Compute the bias changes and then update biases at the hidden layer:

   $$b^h(j) = b^h(j) + \alpha\sum_{p=1}^{N_R} e^h(j,p), \quad 1 \le j \le N_H.$$

End-while

5.3.2 Hybrid partition with batch update back-propagation and matrix multiplication (HP with BBP)

The HP scheme is a combination of neuronal level as well as synaptic level parallelism (Suresh et al., 2005). In the case of a homogeneous N_P-processor ANN, the hidden layer is partitioned into $\left(\frac{N_H}{N_P}\right)$ neurons and the input neurons and the output neurons common for all the processors, as shown in Figure 5.2. The processor P_1 has the blocks of neurons, $(N_I,\ N_{H_1},\ N_O)$, the processor P_k has the blocks, $(N_I,\ N_{H_k},\ N_O)$, respectively. The weight connections between the neurons local to the processor will be stored.

The variant designed by us combines the hybrid partitioning approach and matrix multiplication within a each partitioned ANN enabling computation at two levels of the memory hierarchy: one at message passing level and one at a fine grain level.

Proposed Algorithm: Batch update back-propagation with hybrid partition with matrix multiplication.

1. Compute the number of hidden neurons per each processor:
   $N_{H_p} = \left(\frac{N_H}{N_P}\right)$, and R be the remainder of $\left(\frac{N_H}{N_P}\right)$, where N_P be the number processors to be used to train the ANN and $1 \leq p \leq N_P$.
   $N_{H_p} = N_{H_p} + 1,$ where $1 \leq p \leq R$.

2. All allocated processors create their own weight matrices, $\mathbf{W}_p^h$ and $\mathbf{W}_p^o$, and start with randomly chosen values. Where $\mathbf{W}_p^h = \left(w_p^h(j,i)\right)$ is the weight matrix between hidden and input layers and is handled by processor p, and $w_p^h(j,i)$ is the weight between $\left(\sum_{r=1}^{p-1} N_{H_r} + j\right)^{th}$ hidden neuron and i^{th} input neuron, with $1 \leq j \leq N_{H_p}$ and $1 \leq i \leq N_I$.

   $\mathbf{W}_p^o = (w_p^o(k,j))$ is the weight matrix between output and hidden layers and is handled by the processor, p, and $w_p^o(k,j)$ is the weight between k^{th} output neuron and $\left(\sum_{r=1}^{p-1} N_{H_r} + j\right)^{th}$ hidden neuron, with $1 \leq k \leq N_O$ and $1 \leq j \leq N_{H_p}$.

3. **While** $MSE = \frac{1}{2N_R}\sum_{k=1}^{N_O}\sum_{r=1}^{N_R}(t(k,r) - a^o(k,r))^2$ is unsatisfactory and computational bounds are not exceeded, **Do Concurrently**

   (a) Compute the hidden layer activation:

   Compute the matrix multiplication, $\mathbf{A}_p^h = \mathbf{W}_p^h * \mathbf{X}$, where,

$$\mathbf{A}_p^h = \begin{pmatrix} a_p^h(1,1) & . \; . \; . \; . & a_p^h(1,N_R) \\ a_p^h(2,1) & . \; . \; . \; . & a_p^h(2,N_R) \\ . & . \; . \; . \; . & . \\ . & . \; . \; . \; . & . \\ . & . \; . \; . \; . & . \\ a_p^h(N_{H_p},1) & . \; . \; . \; . & a_p^h(N_{H_p},N_R) \end{pmatrix}$$

$$\mathbf{W}_p^h = \begin{pmatrix} w_p^h(1,1) & . \; . \; . \; . & w_p^h(1,N_I) \\ w_p^h(2,1) & . \; . \; . \; . & w_p^h(2,N_I) \\ . & . \; . \; . \; . & . \\ . & . \; . \; . \; . & . \\ . & . \; . \; . \; . & . \\ w_p^h(N_{H_p},1) & . \; . \; . \; . & w_p^h(N_{H_p},N_I) \end{pmatrix}$$

$$\mathbf{X} = \begin{pmatrix} x(1,1) & . \; . \; . \; . & x(1,N_R) \\ x(2,1) & . \; . \; . \; . & x(2,N_R) \\ . & . \; . \; . \; . & . \\ . & . \; . \; . \; . & . \\ . & . \; . \; . \; . & . \\ x(N_I,1) & . \; . \; . \; . & x(N_I,N_R) \end{pmatrix}$$

   $\mathbf{X} = (x(i,r))$, where $1 \leq i \leq N_I$ and $1 \leq r \leq N_R$, is the input matrix. The rth column of $\mathbf{X}$ is the rth input pattern.

   $\mathbf{A}_p^h = (a_p^h(j,r))$ is the activation matrix, and $a_p^h(j,r)$ is the activation value of the $\left(\sum_{r=1}^{p-1} N_{H_r} + j\right)^{th}$ hidden neuron for the rth input pattern. Where $1 \leq j \leq N_{H_p}$ and $1 \leq r \leq N_R$.

Add the bias value and then apply the activation function to each element of $\mathbf{A}_p^h$.

$$a_p^h(j,r) = f\left(a_p^h(j,r) \;+\; b_p^h(j)\right).$$

$\mathbf{B}_p^h = \left(b_p^h(j)\right),$ where $1 \le j \le N_{H_p},$ is the bias vector for the hidden layer and is handled by the processor, p.

$f(x) = \dfrac{1}{1+e^{-x}}$ is the sigmoid function which is used to compute the activation of each neuron in the hidden and output layers.

N_I and N_{H_p} are the number of neurons in the input and the hidden layers, respectively. N_R is the number of input patterns to be used to train the ANN.

(b) Compute the output layer activation:

Compute the matrix multiplication $\mathbf{A}_p^o = \mathbf{W}_p^o * \mathbf{A}_p^h$

$$\mathbf{A}_p^o = \begin{pmatrix} a_p^o(1,1) & .\;.\;.\;. & a^o(1,N_R) \\ a_p^o(2,1) & .\;.\;.\;. & a_p^o(2,N_R) \\ . & .\;.\;.\;. & . \\ . & .\;.\;.\;. & . \\ . & .\;.\;.\;. & . \\ a_p^o(N_O,1) & .\;.\;.\;. & a_p^o(N_O,N_R) \end{pmatrix}$$

$$\mathbf{W}_p^o = \begin{pmatrix} w_p^o(1,1) & .\;.\;.\;. & w_p^o(1,N_{H_p}) \\ w_p^o(2,1) & .\;.\;.\;. & w_p^o(2,N_{H_p}) \\ . & .\;.\;.\;. & . \\ . & .\;.\;.\;. & . \\ . & .\;.\;.\;. & . \\ w_p^o(N_O,1) & .\;.\;.\;. & w_p^o(N_O,N_{H_p}) \end{pmatrix}$$

$$\mathbf{A}_p^h = \begin{pmatrix} a_p^h(1,1) & .\;.\;.\;. & a_p^h(1,N_R) \\ a_p^h(2,1) & .\;.\;.\;. & a_p^h(2,N_R) \\ . & .\;.\;.\;. & . \\ . & .\;.\;.\;. & . \\ . & .\;.\;.\;. & . \\ a_p^h(N_{H_p},1) & .\;.\;.\;. & a_p^h(N_{H_p},N_R) \end{pmatrix}$$

$\mathbf{A}_p^o = (a_p^o(k,r))$, where $1 \leq k \leq N_O$ and $1 \leq r \leq N_R$, is the activation matrix of output layer and is handled by the processor, p.

(c) Broadcasts the local matrix, A_p^o, and receives all matrices, A_q^o, where $1 \leq q \leq N_O$ except p, from other processors.

Compute the matrix addition:

$$A^o = \sum_{k=1}^{N_O} A_k^o,$$

Add the bias value and then apply the activation function to each element of A^o.

$$a^o(k,r) \; = \; f\left(a^o(k,r) \; + \; b^o(k)\right), \quad 1 \leq k \leq N_O \text{ and } 1 \leq r \leq N_R.$$

$A^o = (a^o(k,r))$ is the activation matrix for output layer, and $a^o(k,r)$ is the activation value of the kth output neuron for the rth input pattern.

$\mathbf{B}^o = (b^o(k))$ is the bias vector for the output layer. Where $1 \leq k \leq N_O$.

(d) Compute the output layer error:

$e^o(k,r) = a^o(k,r) * (1 - a^o(k,r)) * (a^o(k,r) - t(k,r))$.

$e^o(k,r)$ is the error of the kth output neuron for the rth input pattern. Where $1 \leq k \leq N_O$ and $1 \leq r \leq N_R$.

(e) Compute the hidden layer error:

Let $\mathbf{E}^o = (e^o(k,r))$, where $1 \leq k \leq N_O$ and $1 \leq r \leq N_R$.

Compute the matrix multiplication, $\mathbf{W}^o\mathbf{E}_p^o = \left(\mathbf{W}_p^o\right)^T \times \mathbf{E}^o$.

Do the following computation:

$e_p^h(j,r) = A_p^h(j,r) * \left(1 - A_p^h(j,r)\right) * W^oE_p^o(j,r)$.

$e_p^h(j,r)$ is the error of the $\left(\sum_{r=1}^{p-1} N_{H_r} + j\right)^{th}$ hidden neuron for the rth input pattern.

Where $1 \leq j \leq N_{H_p}$ and $1 \leq r \leq N_R$.

(f) Compute the weight changes and then update weights at the output layer:

Compute, $\mathbf{W}_p^o = \mathbf{W}_p^o + \Delta\mathbf{W}_p^o$

where $\Delta\mathbf{W}_p^o$ is a matrix representing the weight changes for matrix $\mathbf{W}_p^o$ and is computed as follows:

Compute the matrix multiplication, $\mathbf{W}_{p_c}^o = \mathbf{E}^o \times \left(\mathbf{A}_p^h\right)^T$, and then compute,

$\Delta\mathbf{W}_{p_t}^o = \alpha\ \mathbf{W}_{p_c}^o + \Theta\mathbf{W}_{p_{t-1}}^o$

α is the learning rate and Θ is the momentum factor used to allow the previous weight change to influence the weight change in this time period, t.

(g) Compute the weight changes and then update weights at the hidden layer:

Compute, $\mathbf{W}_p^h = W_p^h + \Delta\mathbf{W}_p^h$

where $\Delta\mathbf{W}_p^h$ is a matrix representing the weight changes for matrix $\mathbf{W}_p^h$ and is computed as follows:

compute the matrix multiplication, $\mathbf{W}_{p_c}^h = \mathbf{E}_p^h \times \mathbf{X}^T$, and then do the computation,

$\Delta\mathbf{W}_{p_t}^h = \alpha\mathbf{W}_{p_c}^h + \Theta\mathbf{W}_{p_{t-1}}^h$

(h) Compute the bias changes and then update biases at the output layer:

$$b^o(k) = b^o(k) + \alpha \sum_{r=1}^{N_R} e^o(k,r), \quad \text{where } 1 \le k \le N_O.$$

(i) Compute the bias changes and then update biases at the hidden layer:

$$b_p^h(j) = b_p^h(j) + \alpha \sum_{r=1}^{N_R} e_p^h(j,r), \quad \text{where } 1 \le j \le N_{H_p}.$$

4. **End While**

5.3.3 Hybrid partitioning approach with incremental update back-propagation (HP with IBP)

The following describes the standard incremental update back-propagation algorithm for a cluster of processors. An HP scheme is used to partition the ANN. Each partitioned sub ANN is assigned to a single processor in the cluster.

1. Compute the number of hidden neurons per processor:

   $N_{H_p} = \left(\frac{N_H}{N_P}\right)$, and R be the remainder of $\left(\frac{N_H}{N_P}\right)$, where N_P be the number processors to be used to train the ANN and $1 \leq p \leq N_P$.

   $N_{H_p} = N_{H_p} + 1,$ where $1 \leq p \leq R$.

2. All allocated processors create their own weight matrices, $\mathbf{W}_p^h$ and $\mathbf{W}_p^o$, and start with randomly chosen values. Where $\mathbf{W}_p^h = \left(w_p^h(j,i)\right)$ is the weight matrix between hidden and input layers and is handled by processor p, and $w_p^h(j,i)$ is the weight between $\left(\sum_{r=1}^{p-1} N_{H_r} + j\right)^{th}$ hidden neuron and i^{th} input neuron, with $1 \leq j \leq N_{H_p}$ and $1 \leq i \leq N_I$.

   $\mathbf{W}_p^o = (w_p^o(k,j))$ is the weight matrix between output and hidden layers and is handled by the processor, p, and $w_p^o(k,j)$ is the weight between k^{th} output neuron and $\left(\sum_{r=1}^{p-1} N_{H_r} + j\right)^{th}$ hidden neuron, with $1 \leq k \leq N_O$ and $1 \leq j \leq N_{H_p}$.

3. **While** $MSE = \frac{1}{2N_R}\sum_{k=1}^{N_O}\sum_{r=1}^{N_R}(t(k,r) - a^o(k,r))^2$ is unsatisfactory and computational bounds are not exceeded, **Do Concurrently**

   (a) For each pattern $\mathbf{x}_r = (x_{r1},\ x_{r2},\ \cdots,\ x_{rN_I}),$ where $1 \leq r \leq N_R$:

      i. Compute the output of neurons in the hidden layer:

      $a_p^h(j) = \sum_{i=1}^{N_I} w_p^h(j,i)x_{ri} + b_p^h(j),$ where $1 \leq j \leq N_{H_p}$.

      $a_p^h(j) = f\left(a_p^h(j)\right)$

      $f(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function which is used to compute the activation of each neuron in the hidden and output layers. b is the bias.

ii. Compute the output of neurons in the output layer:

$$a_p^o(k) = \sum_{j=1}^{N_{H_p}} w_p^o(k,j) a_p^h(j), \quad \text{where} \quad 1 \le k \le N_O.$$

Broadcast $a_p^o(k)$ and receive $a_q^o(k)$, from processor, q. Where $1 \le q \le N_P$, and $q \ne p$.

$$a^o(k) = f\left(\sum_{p=1}^{N_P} a_p^o(k) + b^o(k)\right)$$

iii. Compute error terms for the neurons in the output layer:

$\delta^o(k) = (a^o(k) - y_{rk}) \ a^o(k) \ (1 - a^o(k))$, where $\mathbf{y}_r = (y_{r1}, \ y_{r2}, \ \cdots, \ y_{rN_O})$ is the actual output and $1 \le k \le N_O$.

iv. Compute error terms for the neurons in the hidden layer:

$$\delta_p^h(j) = a_p^h(j)\left(1 - a_p^h(j)\right) \sum_{k=1}^{N_O} \delta^o(k) w_p^o(k,j), \quad \text{where} \quad 1 \le j \le N_{H_p}.$$

v. Update both weight and bias changes in the output layer:

$$w_p^o(k,j) = w_p^o(k,j) + \eta \delta^o(k) a_p^h(j)$$

$$b^o(k) = b^o(k) + \eta \delta^o(k), \quad \text{where} \quad 1 \le k \le N_O \quad \text{and} \quad 1 \le j \le N_{H_p}.$$

vi. Update both weight and bias changes in the hidden layer:

$$w_p^h(j,i) = w_p^h(j,i) + \eta \delta_p^h(j) x_{ri}$$

$$b_p^h(j) = b_p^h(j) + \eta \delta_p^h(j), \quad \text{where} \quad 1 \le j \le N_{H_p} \quad \text{and} \quad 1 \le i \le N_I.$$

End-while

5.4 Implementations

In this case study, we implement the three variations of the parallel back-propagation ANN training algorithms discussed in the previous section, for a financial application, which is to predict the exchange fluctuation rate as determined by demand and supply conditions in the foreign exchange market. MPI is used to implement the parallel algorithms under DM and TCP-Linda is used to implement the parallel algorithms under VSM. The system will use past historical data, for the training phase.

The input data is taken from a data file which contains normalized exchange rate data. Since this system involves financial data, normalization has been used to preprocess the data. Sigmoid activation function is used in the ANN and this will scale the input data between 0.01 and 0.99. The following formula is used to normalize the input data:

$$\text{normalized data} \;=\; \left(\frac{\text{original data - lowest value}}{\text{highest value - lowest value}}\right)(0.98) + 0.01$$

For the pth input pattern, the input neurons are given as follows:

$$x_p,\ x_{p+1},\ x_{p+2},\ x_{p+3},\ x_{p+4},\cdots\cdots\cdots,\ x_{p+N_I-1},$$

and the output neuron would be x_{p+N_I}, where N_I be the number of neurons in the input layer. $x_1,\ x_2,\ x_3,\ x_4\cdots\cdots\cdots$ be the normalized past exchange rates stored in a text file.

In all, six algorithmic variations have been implemented on the high performance cluster, Monolith [WWW 13]. The Monolith cluster consists of over 396 nodes. Each node has two Intel Xeon processors at 2.2 GHz, 2 GBytes primary memory (ECC DDR), 512KB cache and a disk memory with 80 GBytes. In addition there is a common main disk storage of 7 TBytes. The operating system is Linux version 2.4.34-capl-smp. MPI (ScaMPI) on monolith runs on the SCI-network with a 2-rank ping-pong of around 250 MB/s and 4.5 micro seconds (large packet bandwidth and small packet latency respectively). TCP-Linda on monolith runs on TCP (which runs on a far from perfect 100 Mbps ethernet). Performance reaches of MPI on TCP is around 10 MB/s and 60-100 microseconds.

5.4.1 Distributed Memory Implementation

The fully connected multilayer perceptron ANN is partitioned into N_P partitions, with each partition mapped to N_P processors in the cluster. The implemented algorithm has three phases, which we call the forward phase, error propagation phase, and weight update phase.

Forward phase: In this phase, the activation value of the neurons local to the processors are computed. For a given input pattern, we can compute the activation value for all neurons in the hidden layer. But we need the activation values and weight connections, which are kept at every processor to calculate the activation value of the neurons in the output layer. First we compute the partial sum of the activation values and then exchange those activation values among processors to compute the activation value of the neurons in output layer. The *MPI_Sendrecv* subroutine, which is standardized in MPI [WWW 01] is used to exchange the partial sum of the activation values among processors.

Error propagation phase: In this phase, the error terms for the output layer neurons, δ^o and for the hidden layer neurons, δ^h), local to the processor are computed.

Weight update phase: In this phase, the weight connections between neurons local to the processors are updated. We need the value of ith input neuron and δ_h term in the jth hidden

neuron to update the weight connection between the jth hidden neuron and the ith input neuron. Also we need the activation value of jth hidden neuron and δ^o term in the kth output neuron to update the weight connection between the kth output neuron and the jth hidden neuron. Since both the activation value and the error term are kept in the processor, intercommunication between processors is not required.

5.4.2 Distributed Shared Memory Implementation

We follow almost the same approach as that in the distributed memory approach. But, in the forward phase of execution the partial sum of the activation values are exchanged between processors to calculate the activation value of the neurons in the output layer. To achieve the exchange of those partial sums of the activation values among processors, the *in* and the *out* operations defined in TCP-Linda [WWW 06] are used. In TCP-Linda, the operation, *eval* is used to create a process at run time. So, by dynamically changing the number of processes and obtaining the corresponding run time for the same set of input patterns, the TCP-Linda program is able to find the minimum running time and the corresponding optimal number of processors.

5.4.3 Results

We have considered three different ANNs, namely, ANN_1, ANN_2, and ANN_3. We use as the stopping criteria, the error value of 0.000045, or maximum number of iterations 120000.

Table 5.2 shows the timing values in seconds for the three parallel algorithms, HP-with-IBP, NHP-with-BBP, and HP-with-BBP, simulating the neural network architecture, ANN_1. Algorithms are implemented in MPI, and the number of processors assigned to run are from 1 to 7.

Table 5.3 and Table 5.4 show the timing values for the six scenarios arising from the three parallel algorithms mentioned above, simulating ANN_2 and ANN_3, respectively with algorithms implemented in MPI, and the number of processors from 1 to 15.

Table 5.5 and Table 5.6 show the timing values for the six scenarios arising from the three parallel algorithms, simulating ANN_1 and ANN_2, respectively with algorithms implemented in TCP-Linda, for TCP-Linda processes 1 to 7 and 1 to 9, respectively. Table 5.7 shows the number of MPI functions called at the training time of neural network.

Processors	HP with IBP	NHP with BBP	HP with BBP
1	1493	846	578
2	1552	817	351
3	2716	845	292
4	4516	883	256
5	6733	960	244
6	8460	1037	233
7	8915	1110	213

Table 5.2: Mean training time in seconds for ANN_1 using MPI

Processors	HP with IBP	NHP with BBP	HP with BBP
1	18371	15220	14600
2	6493	12962	7252
3	6735	11968	5318
4	7731	11859	3944
5	8086	9937	3226
6	10145	9980	2702
7	11548	10032	2625
8	13037	10153	2167
9	13896	10312	2038
10	15851	10437	1910
11	17236	10720	1849
12	19034	11077	1747
13	20560	11381	1603
14	22566	11544	1538
15	24012	12329	1469

Table 5.3: Mean training time in seconds for ANN_2 using MPI

Processors	HP with IBP	NHP with BBP	HP with BBP
1	80529	73825	61825
2	17446	54684	35795
3	14629	34393	22636
4	13275	27124	17694
5	12861	24548	13858
6	13427	22600	11943
7	14755	21942	10053
8	15666	21636	9487
9	16376	21981	8049
10	17987	21384	7330
11	18975	21499	6823
12	20089	22141	6401
13	21987	22646	6189
14	25263	22811	5722
15	26706	23294	5297

Table 5.4: Mean training time in seconds for ANN_3 using MPI

Processors	HP with IBP	NHP with BBP	HP with BBP
1	37803	10466	8443
2	40506	15104	14583
3	66762	21436	19553
4	108877	27356	26357
5	128821	36884	34029
6	148210	45770	41142
7	195128	54165	49098

Table 5.5: Mean training time in seconds for ANN_1 using TCP-Linda

Processors	HP with IBP	NHP with BBP	HP with BBP
1	243179	133315	99926
2	234916	124391	98944
3	239026	181067	96337
4	241074	213372	93260
5	249079	238871	100294
6	258061	240845	111750
7	260308	246381	125937
8	265721	250047	127757
9	289462	258647	134062

Table 5.6: Mean training time in seconds for ANN_2 using TCP-Linda

MPI Function Name	HP IBP	NHP BBP	HP BBP
MPI_Send	8	10.560004E+6	8
MPI_Recv	8	10.560004E+6	8
MPI_Send_recv	3211.200016E+6	0	2.40002E+6

Table 5.7: Number of MPI functions called at training time

5.4.4 Analysis of Results

In this section, we discuss the key performance metrics, the running time and the optimum number of processors for a given ANN configuration. For all three ANNs configurations, the parallel algorithms under the DM architectures show vastly better timing than VSM parallel algorithms.

Consider the algorithm HP with IBP, for ANN_1, the corresponding serial algorithms (Table 5.2 and Table 5.5; with processors=1) are optimum compared to the parallel algorithms written for both DM and VSM. For ANN_2, the DM algorithm shows a super linear speed up for two processors which there after deteriorates to a standard speed up. We have no exact explanation for this effect but we believe that this may be due to network structure and cache effects. For ANN2 the VSM algorithm shows the maximum speed up for two processors although nearly equal to the sequential implementation. For ANN_3, the optimum number of processors is 5 in the DM parallel algorithm.

Consider the algorithm NHP with BBP, for ANN_1, the corresponding serial algorithm (Table 5.2 and Table 5.5; with processors=1) is optimum compared to the parallel algorithm written for VSM, and the DM parallel algorithm with 2 processors shows better performance than the rest. For ANN_2, the DM parallel algorithm running with 5 processors shows better performance than the rest. Also VSM with 2 processors is optimal among its group. For ANN_3, the optimum number of processors is 10 in the DM parallel algorithm.

Now consider the variant (HP with BBP) designed by us, for ANN_1, where the serial algorithm is optimum compared to the parallel algorithm written for VSM. Also the DM parallel algorithm with 7 processors shows better performance than the rest. For ANN_2, the DM parallel algorithm running with 15 processors shows better performance than the rest. Also VSM with 5 processors is optimal among its group. For ANN_3, the optimum number of processors is 15 in the DM parallel algorithm. This parallel algorithm is more efficient than the other two parallel algorithms in either of the memory architectures.

Since the communication time per training epoch for a given input pattern is higher than the time for computations, there is little use of threads here as a programming tool.

One important observation is that the TCP-Linda program takes hugely more running time than the equivalent MPI program. The reason is that according to the code, the ANN assigned to each TCP-Linda process is read and is written back from/to the VSM, for all training tasks. These read and write operations are implemented in TCP-Linda by *in* and *out* operations. The *in* and *out* operations take considerable time to act on the user defined data types, such as “struct”.

Chapter 6

A Configurable DM/DSM Cluster

6.1 Distributed Shared Memory Sub Clusters Interconnected by MPI

We have carried out two case studies, one on the back-propagation neural network, and the other on the shallow water model both implemented using distributed memory and distributed shared memory programming models. Where as the shallow water model (Section 4.5.3) had performed well on distributed shared memory, the neural network had done so, on distributed memory (Section 5.4.3), reflecting the fact that there has to be a close match between the underlying computational problem and the interactions provided by the memory architectures, for effective performance improvement.

Henceforth we concentrate on tasks that have performed well under distributed shared memory model. The results for shallow water model have shown further that given sufficient computing resources, as the task domain is scaled up, the performance has gone down due to distributed shared memory replication overhead (Section 4.5.4). This situation is further complicated by the existence of a memory hierarchy (Section 3.1), due to which a further performance enhancement is only possible by an effective mixed mode programming model.

Having identified scalability and utilization of memory hierarchy as goals for effective parallel algorithm design on a cluster, we propose a hybrid and flexible programming approach utilizing both distributed memory and distributed shared memory models targeting tasks of interest to us as mentioned above.

The hybrid model to be proposed, consisting of configurable cells (or subclusters) will also enable arbitrary virtual topologies to be defined. In the era of supercomputing, efficient parallel algorithms have usually been defined with nodal topology in mind. With the possibility of our virtual topologies being defined, the reusability of established efficient algorithms become a reality.

We have proposed and have successfully implemented a new type of UPC/MPI hybrid cluster configuration to achieve the needs as mentioned above. The UPC/MPI hybrid cluster configuration is arranged by configuring the hardware cluster into a number of DSM sub clusters or cells interconnected by MPI. Each DSM cell is managed by UPC compiler. A hypothetical toroidal structure made of such UPC/MPI cells is shown in Figure 6.1.

We have defined two UPC/MPI hybrid cluster configurations, configuration-1 and configuration-2, as illustrated in Figure 6.2 and Figure 6.3, respectively. The shallow water equations (Section 4.3) with a four by three grid scheme is chosen to test our proposed UPC/MPI hybrid cluster configurations.

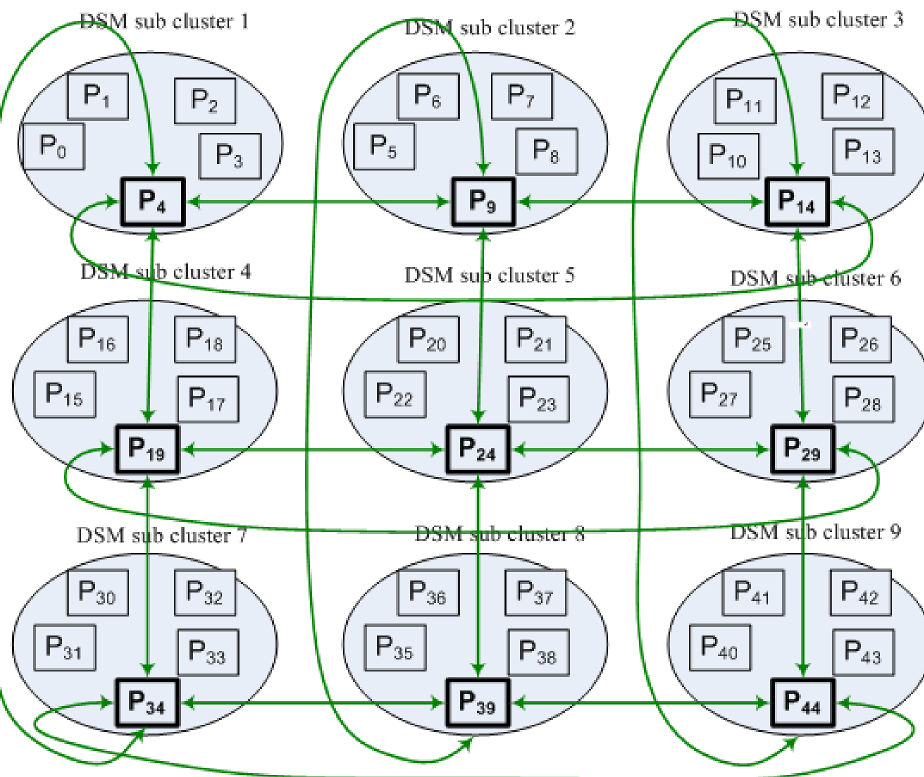

Figure 6.1: DSM sub clusters interconnected by MPI using two-dimensional toroidal topology.

6.2 Algorithm Design for Shallow Water Model

The domain decomposition method is used to parallelize the shallow water model. The physical or computational domain for this model is rectangular in shape. Details of the computational domain are given in Section 4.5.

A 12-node, four by three grid system, is used to parallelize the shallow water model with a domain size of 1226×900. Portions of the domains are assigned to each of the worker nodes, as

illustrated in Figure 4.2 where each sub-domain is labelled with a processor (worker) number. Data for each sub-domain is stored with each processor including water depth, coordinates, and initial disturbance. Since the model updates the solution explicitly using local data, each processor works independently of the others but requires data from neighboring workers to update solution along sub-domain boundaries. Communication occurs between two adjacent nodes during message passing.

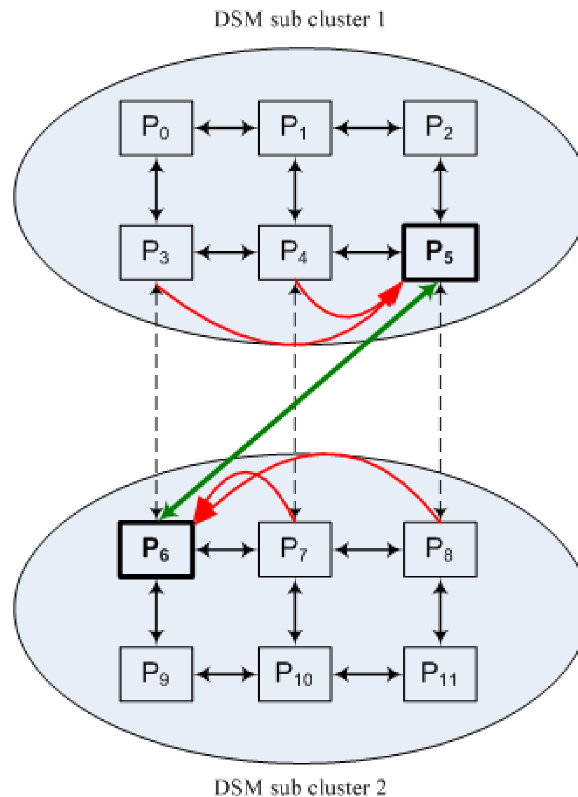

Figure 6.2: A 12-node cluster is configured as two DSM sub clusters interconnected by MPI (configuration-1).

The 12-node, four by three grid scheme, is selected to configure the hardware cluster into a number of DSM sub clusters interconnected by MPI. Configuration-1 arranges the 12-node cluster into equally sized two DSM sub clusters, as illustrated in Figure 6.2. The first DSM sub cluster includes computing nodes ranks from 0 to 5. Computing nodes with ranks from 6 to 11 forms second DSM sub cluster. Computing nodes with ranks 5 and 6 are assigned to interconnect the two DSM sub clusters for the purpose of exchanging data among them.

Configuration-2 arranges the 12-node cluster into four DSM sub clusters, $subcluster_n$, $1 \leqslant n \leqslant 4$, as illustrated in Figure 6.3. Computing nodes, P_k, $0 \leqslant k \leqslant 11$, assigned to the DSM sub cluster, $subcluster_n$, are given as follows:

$$k \in \{0,\ 1,\ 3,\ \mathbf{4}\} \quad \text{if} \ \ n = 1$$

$$k \in \{6,\ \mathbf{7},\ 9,\ 10\} \quad \text{if} \ \ n = 2$$

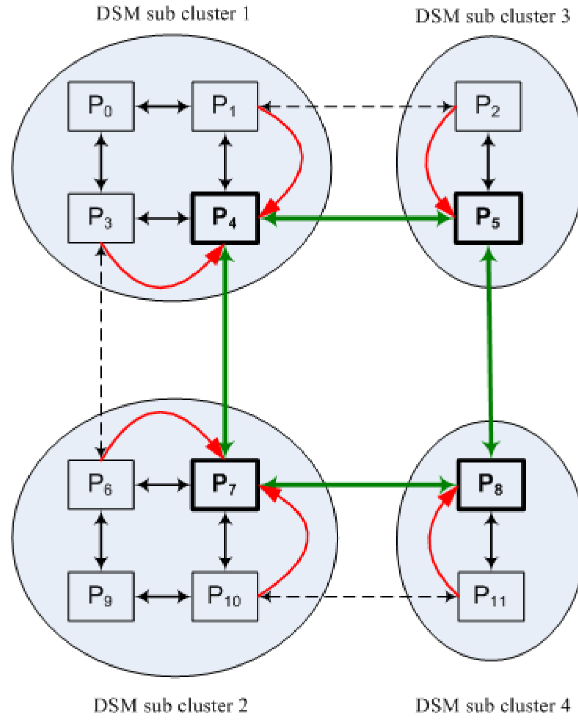

Figure 6.3: A 12-node cluster is configured as four DSM sub clusters interconnected by MPI (configuration-2).

$$k \in \{2,\ \mathbf{5}\} \qquad \text{if} \quad n = 3$$
$$k \in \{\mathbf{8},\ 11\} \qquad \text{if} \quad n = 4$$

Bold ranks of computing nodes in each sub cluster are chosen to handle data exchange between DSM sub clusters. UPC is used for intra subcluster communication and MPI is used for inter subcluster communication. The parallel algorithms for the two cluster configurations are outlined below.

6.2.1 Algorithm Design for Configuration-1

Algorithm 6.1: A hybrid UPC/MPI parallel algorithm for the shallow water model under cluster configuration-1.

1. Decompose rectangular domain into load-balanced rectangular sub-domains, the width and the length of a rectangular sub-domain are approximately, $\left(\frac{W}{N_w}\right)$ and $\left(\frac{L}{N_l}\right)$, respectively, where W and L are the width and length of domain, and $N_P = N_w * N_l$ is the number of processors to be used in the $(N_w \times N_l)$ grid scheme.

2. Specify parallel language related parameters, such as locations, neighbors, sub-domain sizes, and file names, of processors. Location of kth processor is (r_k, c_k), where $r_k = \left(\left(\frac{k}{gdm1}\right) + 1\right)$ and $c_k = (k - (r_k - 1) * gdm1 + 1)$ with $gdm1 = 3$ and $gdm2 = 4$ being dimension of the grid. For $k = 0, 1, 2, \cdots\cdots, (gdm1*gdm2-1)$, define $north_k$, $south_k$,

$east_k$, and $west_k$ to be the neighbors located in the side of north, south, east and west of the kth processor, then:

$$\begin{aligned} north_k &= k - gdm1 \quad \text{if} \;\; r_k > 1 \\ south_k &= k + gdm1 \quad \text{if} \;\; r_k < gdm2 \\ east_k &= k + 1 \quad \text{if} \;\; c_k < gdm1 \\ west_k &= k - 1 \quad \text{if} \;\; c_k > 1 \end{aligned}$$

3. Configure the 12-processor cluster into two equal sized sub clusters, subcluster-1 and subcluster-2. Processors with ranks from 0 to 5 are assigned to subcluster-1 and subcluster-2 is formed with the processors ranking from 6 to 11. Processors, $P_k,\ k \in \{5,\ 6\}$, are chosen to handle the message passing between the two sub groups, subgroup-1 and subgroup-2.

4. Input data and initial conditions:

   (a) Each processor, P_k, read the water depth, coordinates, and initial disturbance from the text file assigned in step 2.

   (b) Each processor, P_k, exchange sub-domain boundary data from $east_k$ to $west_k$, and from $west_k$ to $east_k$, in order.

   (c) Exchange sub-domain boundary data from north to south (name the following sub tasks as *N_to_S_data_exchange*):

      i. Processor, $P_k,\ k \in \{3,\ 4\}$, sends the boundary data to processor, P_5.

      ii. Processor, P_5, receives the data sent by the processors, $P_k,\ k \in \{3,\ 4\}$, and then sends this received data with its own boundary data to the processor, P_6.

      iii. Processor, P_6, receives the data sent by the processors, P_5, and then distributes the corresponding boundary data to processors, $P_k,\ k \in \{7,\ 8\}$, such that Data $(P_k) \longrightarrow P_{k+3},\ k \in \{3,\ 4,\ 5\}$.

      iv. Each processor, $P_k,\ k \notin \{3,\ 4,\ 5\}$, exchange boundary data from $north_k$ to $south_k$.

   (d) Exchange sub-domain boundary data from south to north (name the following sub tasks as *S_to_N_data_exchange*):

      i. Processor, $P_k,\ k \in \{7,\ 8\}$, sends the boundary data to processor, P_6.

      ii. Processor, P_6, receives the data sent by the processors, $P_k,\ k \in \{7,\ 8\}$, and then sends this received data with its own boundary data to the processor, P_5.

iii. Processor, P_5, receives the data sent by the processors, P_6, and then distributes the corresponding boundary data to processors, $P_k,\ k \in \{3,\ 4\}$, such that Data $(P_k) \longrightarrow P_{k-3},\ k \in \{6,\ 7,\ 8\}$

iv. Each processor, $P_k,\ k \notin \{6,\ 7,\ 8\}$, exchange boundary data from $south_k$ to $north_k$.

5. Set parameters and coefficients used at the open sea boundary.

6. Repeat the following steps for the pre-defined time-steps:

   (a) Each processor, P_k, exchange sub-domain boundary data from $east_k$ to $west_k$.

   (b) Exchange sub-domain boundary data from north to south by calling the sub algorithm, *N_to_S_data_exchange*.

   (c) Computation of the equation of continuity.

   (d) Setting of the open sea boundary condition.

   (e) Each processor, P_k, exchange sub-domain boundary data from $west_k$ to $east_k$.

   (f) Exchange sub-domain boundary data from south to north by calling the sub algorithm, *S_to_N_data_exchange*.

7. Gather computed results among all processors.

8. Compute the output on processor, P_k, where both $south_k$ and $west_k$ are equal to null.

6.2.2 Algorithm Design for Configuration-2

Algorithm 6.2: A hybrid UPC/MPI parallel algorithm for the shallow water model under cluster configuration-2.

First two steps 1 and 2 are identical to that given in Section 6.2.1.

3. Configure the 12-processor cluster into four sub clusters, subcluster-1, subcluster-2, subcluster-3, and subcluster-4, as follows:

   subcluster-n is formed by the processors, P_k,

   where $k \in \{0,\ 1,\ 3,\ \mathbf{4}\}$ if $n = 1$

   $k \in \{6,\ \mathbf{7},\ 9,\ 10\}$ if $n = 2$

   $k \in \{2,\ \mathbf{5}\}$ if $n = 3$

   $k \in \{\mathbf{8},\ 11\}$ if $n = 4$

   Bold processor numbers in each sub group are chosen to manage the message passing between sub groups.

4. Input data and initial conditions:

   (a) Each processor, P_k, read the water depth, coordinates, and initial disturbance from the text file assigned in step 2.

   (b) Exchange sub-domain boundary data from east to west (name the following sub tasks as *E_to_W_data_exchange*):

      i. Processor, P_2, sends the boundary data to processor, P_5.

      ii. Processor, P_5, receives the data sent by the processor, P_2, and then sends this received data with its own boundary data to the processor, P_4.

      iii. Processor, P_4, receives the data sent by the processors, P_5, and then distributes the corresponding boundary data to processor, P_1, such that
      Data $(P_k) \longrightarrow P_{k-1},\ k \in \{2,\ 5\}$.

      iv. Processor, P_{11}, sends the boundary data to processor, P_8.

      v. Processor, P_8, receives the data sent by the processor, P_{11}, and then sends this received data with its own boundary data to the processor, P_7.

      vi. Processor, P_7, receives the data sent by the processors, P_8, and then distributes the corresponding boundary data to processor, P_{10}, such that
      Data $(P_k) \longrightarrow P_{k-1},\ k \in \{8,\ 11\}$.

      vii. Each processor, $P_k,\ k \notin \{2,\ 5,\ 8,\ 11\}$, exchange boundary data from $east_k$ to $west_k$.

   (c) Exchange sub-domain boundary data from west to east (name the following sub tasks as *W_to_E_data_exchange*):

      i. Processor, P_1, sends the boundary data to processor, P_4.

      ii. Processor, P_4, receives the data sent by the processor, P_1, and then sends this received data with its own boundary data to the processor, P_5.

      iii. Processor, P_5, receives the data sent by the processors, P_4, and then distributes the corresponding boundary data to processor, P_2, such that
      Data $(P_k) \longrightarrow P_{k+1},\ k \in \{1,\ 4\}$.

      iv. Processor, P_{10}, sends the boundary data to processor, P_7.

      v. Processor, P_7, receives the data sent by the processor, P_{10}, and then sends this received data with its own boundary data to the processor, P_8.

      vi. Processor, P_8, receives the data sent by the processors, P_7, and then distributes the corresponding boundary data to processor, P_{11}, such that
      Data $(P_k) \longrightarrow P_{k+1},\ k \in \{7,\ 10\}$.

vii. Each processor, $P_k,\ k \notin \{1,\ 4,\ 7,\ 10\}$, exchange boundary data from $west_k$ to $east_k$.

(d) Exchange sub-domain boundary data from north to south (name the following sub tasks as *N_to_S_data_exchange*):

i. Processor, P_3, sends the boundary data to processor, P_4.

ii. Processor, P_4, receives the data sent by the processor, P_3, and then sends this received data with its own boundary data to the processor, P_7.

iii. Processor, P_7, receives the data sent by the processors, P_4, and then distributes the corresponding boundary data to processor, P_6, such that
Data $(P_k) \longrightarrow P_{k+3},\ k \in \{3,\ 4\}$.

iv. Processor, P_5, sends the boundary data to processor, P_8.

v. Processor, P_8, receives the data sent by the processor, P_5.

vi. Each processor, $P_k,\ k \notin \{3,\ 4,\ 5\}$, exchange boundary data from $north_k$ to $south_k$.

(e) Exchange sub-domain boundary data from south to north (name the following sub tasks as *S_to_N_data_exchange*):

i. Processor, P_6, sends the boundary data to processor, P_7.

ii. Processor, P_7, receives the data sent by the processor, P_6, and then sends this received data with its own boundary data to the processor, P_4.

iii. Processor, P_4, receives the data sent by the processor, P_7, and then distributes the corresponding boundary data to processor, P_3, such that
Data $(P_k) \longrightarrow P_{k-3},\ k \in \{6,\ 7\}$

iv. Processor, P_8, sends the boundary data to processor, P_5.

v. Processor, P_5, receives the data sent by the processor, P_8.

vi. Each processor, $P_k,\ k \notin \{6,\ 7,\ 8\}$, exchange boundary data from $south_k$ to $north_k$.

5. Set parameters and coefficients used at the open sea boundary.

6. Repeat the following steps for the pre-defined time-steps:

(a) Exchange sub-domain boundary data from east to west by calling the sub algorithm, *E_to_W_data_exchange*.

(b) Exchange sub-domain boundary data from north to south by calling the sub algorithm, *N_to_S_data_exchange*.

(c) Computation of the equation of continuity.

(d) Setting of the open sea boundary condition.

(e) Exchange sub-domain boundary data from west to east by calling the sub algorithm, *W_to_E_data_exchange*.

(f) Exchange sub-domain boundary data from south to north by calling the sub algorithm, *S_to_N_data_exchange*.

7. Gather computed results among all processors.

8. Compute the output on processor, P_k, where both $south_k$ and $west_k$ are equal to null.

6.3 Implementation

Data is partitioned into sub domains, and a four by three grid scheme is used for the parallel implementation of this model. With this grid scheme, there are two parallel variations: (1) the grid scheme is configured into two equally sized DSM sub clusters by configuration-1; and (2) the grid scheme is configured into four DSM sub clusters by configuration-2. Experiments use the Berkely UPC for data communication between computing nodes among DSM sub clusters and the Message Passing Interface (MPI) library for interconnecting DSM sub clusters.

Two algorithmic variations have been implemented on the high performance cluster, Upplanka. The Upplanka cluster consists of 14 computing nodes. Each node has Intel(R) Pentium(R) 4 CPU at 3.20GHz, 2 GBytes primary memory , 2 MBytes cache and a disk memory with 200 GBytes. The operating system is Linux version 2.6.9-42.0.10.EL. The MPI compiler is mpich2-1.1. UPC (the Berkeley Unified Parallel C compiler), v. 2.8.0, is installed with the UPC-to-C translator, v. 2.8.0, built at upplanka.ucsc.cmb.ac.lk (linux-x86_64).

Chapter 7

Performance of the Shallow Water Model on the Configurable Cluster

7.1 Review of MPI, UPC, and UPC/MPI for Configuration-1 and for Configuration-2

The high performance cluster, Upplanka with fourteen computing nodes, located at UCSC laboratory, University of Colombo School of Computing, Sri Lanka, was used to simulate the shallow water model with a (4×3) grid scheme. MPI compiler has been used to run the pure DM model whilst the Berkeley UPC compiler has been used to run the pure DSM model on Upplanka.

Two forms of UPC/MPI hybrid sub cluster configurations are made. The first configuration makes the Upplanka cluster with twelve computing nodes into two equal sized DSM sub clusters connected by MPI channels. The second configuration makes the Upplanka cluster with twelve computing nodes into four DSM sub clusters connected by MPI channels. UPC is used for intra DSM communication while MPI is used for inter DSM communication.

MPI	UPC	UPC + MPI (hybrid DSM/MPI)	
(DM)	(DSM)	configuration-1	configuration-2
9415.37	7797.70	7915.18	8068.58

Table 7.1: Average run time in seconds for the shallow water model with (4×3) grid scheme on Upplanka cluster.

Table 7.1 shows the average timing values obtained for the four shallow water parallel algorithms mentioned above for a (4×3) grid (Appendix V contains the running times). The parallel algorithm for the DSM exhibit better performance than the parallel algorithm for the DM. Though the DSM implicitly passes messages, the replication management subsystem has been optimized in UPC compared to MPI [WWW 14].

Parallel algorithm for the DSM exhibit a slightly better performance than either of the two parallel algorithms for the UPC/MPI hybrid cluster configurations while these two parallel algorithms exhibit overall superior performance than the parallel algorithm for DM.

Considering the two algorithms for the UPC/MPI hybrid cluster configurations, configuration-1 and configuration-2, the algorithm for configuration-1 exhibits a better performance than the algorithm for configuration-2, as the number of computing nodes per DSM sub cluster of configuration-2 becomes smaller (of size two), the system degenerates to a DM environment.

Unfortunately it has not been possible to replicate the same three scenarios, MPI, UPC, and UPC+MPI on a (6×8) grid as unlike in the Monolith [WWW 13], Upplanka resources are limited to only 14 nodes, whereas the requirement becomes 48 nodes. However, we predict but are unable to prove, that a cluster with a sufficent number of computing nodes implementing the proposed hybrid UPC/MPI programming model will have close if not better timing than the pure UPC programming model, and be more flexible to be programmed.

7.2 Evaluation

7.2.1 Extracting the capabilities of the memory hierarchy in a cluster

Thread capabilities have to be explicitly programmed into MPI which only exploits node level parallelism. In contrast, the UPC implementation inherently exploits thread level parallelism, and as such extracts the capabilities of memory architectures that exist on a cluster.

7.2.2 Programming flexibility of the language

Compared to a sequential algorithm, as MPI based parallel algorithm, needs to be fully restructured into a master and worker configuration. Parallelism has to be explicitly managed by the programmer with emphasis is placed on the low level communication details, which is a difficult task. However, the collective operations provided in MPI enables easy manipulation of data among multiple computing nodes. As MPI is a set of libraries, it inherits the features like fault tolerance and garbage collection from the base language (C, C++, or, FORTRAN).

When porting a sequential algorithm to an UPC parallel algorithm, the programming effort is comparatively low due to the underlying shared memory concepts. Programme development in UPC+MPI is also a matter of embedding MPI directives within the UPC environment, and

as such it is relatively easy.

7.2.3 Meta expressiveness

The proposed parallel programming model of UPC+MPI, raises the level of expressiveness of an algorithm such that independent of the underlying hardware architecture, virtual topologies of DSM islands can be configured, enabling re-use of efficient supercomputing codes in the past.

Chapter 8

Conclusion and Future Work

A parallel implementation of the shallow water model is chosen to study the impact of memory architectures, distributed memory and virtual shared memory, as one of our case studies. Data is partitioned into sub-domains, namely a (4×3) grid scheme and a (8×6) grid scheme which are used for the parallel implementation of this model. There are four versions of the parallel algorithm for each grid scheme: distributed memory without threads, distributed memory with threads, virtual shared memory without threads, and virtual shared memory with threads. These four parallel versions have been implemented on a high performance cluster, connected to the "Nordugrid" [WWW 13]. Experiments are realized using the Message Passing Interface (MPI) library, the C/Linda, and the Linux pthreads. Subject to the availability of memory, the virtual shared memory version without threads performs best, and as the task is scaled up, the threaded version becomes efficient in both DM and VSM implementations (Section 4.5.4).

In the course of our studies, we have also been able to present a fast parallel algorithm based on hybrid network partitioning and matrix multiplication for back-propagation learning neural networks on a computational cluster. The partitioned network is mapped onto nodes in the cluster. The timings show that for MPI/DM implementations the proposed algorithm is faster than any of the other known counterparts. The hybrid partition with matrix multiplication design also allows one to use a hierarchy of memory levels efficiently with message passing among partitioned networks and to shared memory within a node for matrix multiplication. As the ANNs get larger the parallel algorithms under both memory architectures show better running time than their serial counter parts. In Linda on VSM, the only advantage we have is its ability to identify the optimum number of processors dynamically (Section 5.5.4).

We then configured the hardware cluster into a configurable hybrid array of DSM subclusters connected by MPI channels. A shallow water model was run on two virtual configurations. The pure DSM algorithm exhibits a better performance than the proposed algorithms for the UPC/DSM hybrid cluster configurations while these proposed algorithms inturn exhibit better

performance than the pure DM algorithm (Section 7.1).

This thesis has presented a novel hybrid configurable DSM/DM programming model for a class of parallel problems that performs well on DSM, and also exhibits scalability and the ability to extract the memory hierarchy performance.

The proposed hybrid UPC/MPI (or DSM/DM) cluster programming paradigm can be positioned between a pure DM and a pure DSM model on a generic computing cluster. We have shown that our model extracts parallelism on a hierarchy of memory levels, easy to program and suitable for large scale clusters. Further it allows established efficient parallel algorithms written for specific topologies to be utilized on UPC/MPI on virtual topologies that can be defined on generic clusters.

Our study envisages the utilization of our proposed DSM/DM hybrid algorithms on a 48-node cluster to simulate the (8×6) grid scheme. Application of various types of topologies on this DSM/DM hybrid algorithms is also contemplated.

References

Amawy, A.E. & Kulasinghe, P. (1997). Algorithmic mapping of feedforward neural networks onto multiple bus systems, *IEEE Transaction on parallel and distributed systems*, vol. 8, no. 2, 1997, pp 130-136.

Anderson, J. & Rosenfeld, E. (1988). *Neurocomputing: foundations of research*, MIT Press, Cambridge, MA, USA, ISBN:0-262-01097-6.

Araiijo, M.A.A., Teixeira, E.P., Camargo, F.R. & Almeida, J.P.V. (2003). Parallel training for neural networks using PVM with shared memory, *J.P.V. Evolutionary Computation*, The 2003 Congress on vol. 2, IEEE, Dec. 2003, pp 1315-1322.

Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A, Plishker, W.L., Shalf, J., Williams, S.W & Yelick, K.A. (2006). The landscape of parallel computing research: a view from Berkeley, *Technical Report UCB/EECS-2006-183*, Electrical Engineering and Computer Science, University of California at Berkeley, Dec. 2006.

Atlas, L. E. & Suzuki, Y. (1989). Digital systems for artificial neural networks, *IEEE Circuits and Devices Magazine*, Nov. 1989, pp 20-24.

Aung, T.N., Nwe, A.A, Soe, K.M, Naing, T.T & Thein, N.L. (2005). Utilizing Multiple Networks for Interprocess Communication in Cluster Computing", 6th *Asia-Pacific Symposium on Information and Telecommunication Technologies (APSITT 2005)*, Nov. 2005, pp 345-349.

Babii, S. (2007). Performance evaluation for training a distributed back-propagation implementation, *Proceedings of the* 4th *International Symposium on Applied Computational Intelligence and Informatics (SACI'07)*, IEEE, May 2007, pp 273-278.

Bazanov, P., Kim, T.K., Kee, S.C. & Lee, S.K. (2002). Hybrid and parallel face classifier based on artificial neural networks and principal component analysis, *Proceedings of International Conference on Image Processing*, vol. 1, 2002, pp 916-919.

Blikberg, R. (2003). Nested Parallelism in OpenMP with Application to Adaptive Mesh Refinement, Ph.D. Thesis, University of Bergen, Norway.

Bova, S.W, Breshears, C.P., Cuicchi, C.E., Demirbilek, Z. & Gabb, H.A. (2000). Dual-level parallel analysis of Harbor Wave response using MPI and OpenMP, *International Journal of High Performance Computing Applications*, vol.14 , Issue 1, 2000, pp 49-64.

Brightwell, R.B. & Wen, Z. (2004). Advanced parallel programming models research and development opportunities, *SANDIA REPORT, SANDIA2004-3485*, 2004.

Buchau, A., Tsafak, S.M., Hafa, W., & Rucker, W.M. (2008). Parallelization of a Fast Multipole Boundary Element Method with Cluster OpenMP, *IEEE Transactions on Magnetics*, vol 44, no. 6, June 2008, pp 1338-1341.

Cai, X. & Langtangen, H.P (2007). Making Hybrid Tsunami Simulators in a Parallel Software Framework, *LNCS, Applied Parallel Computing, State of the Art in Scientific Computing*, vol. 4699/2008, 2007, pp 686-693.

Cantonnet, F., Yao, Y., Zahran, M. & Ghazawi, T.E. (2004). Productivity analysis of the UPC language, *proceedings of the* 18th *International Parallel and Distributed Processing Symposium*, IEEE Computer Society, 2004, pp 254a.

Chan, M.C., Wong, C.C. & Lam, C.C. (2000). Financial time series forecasting by neural network using conjugate gradient learning algorithm and multiple linear regression weight initialization. Computing in Economics and Finance 2000, no.61, July 2000.

Chatterjee, S., Jain, V.V., Lebeck, A.R., Mundhra, S. & Thottethodi, M. (1999). Nonlinear array layouts for hierarchical memory systems. *Proceedings of 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999, pp 444-453.

Chatterjee, S., Lebeck, A.R., Patnala, P.K. & Thottethodi, M. (1999). Recursive array layouts and fast parallel matrix multiplication, *in Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures*, Saint-Malo, France, June 1999, pp 222-231.

Chen, A.S., Leung, M.T. & Daouk, H. (2003). Application of neural networks to an emerging financial market: forecasting and trading the Taiwan stock index, vol. 30 , issue 6, *Emerging economics*, *Elsevier Science Ltd. Oxford, UK*, 2003, pp 901 - 923.

Cheng, L., Muralimanohar, N., Ramani, K., Balasubramonian, R. & Carter, J.B. (2006). Interconnect-Aware Coherence Protocols for Chip Multiprocessors, 33rd *International Symposium on Computer Architecture*, ISCA apos;06, 2006, pp 339-351.

Chu, L.C. & Wah, B.W. (1992). Optimal mapping of neural network learning on message-passing multicomputers, *Journal of Parallel and Distributed Computing*, vol.14, 1992, pp 319-339.

Crespo, M., Piccoli, F., Printista, M. & Gallard, R. (1999). A Parallel Approach for Backpropagation Learning of Neural Networks, *Journal on Computer Science and Technology - Special issue on Distributed and Parallel Processing*, vol. 1, no. 4, March 1999.

Dellar, P.J & Salmon, R. (2005). Shallow Water Equations with a Complete Coriolis Force and Topography, *Physics of Fluids 17*, September 2005, pp 1-19.

Frens, J.D. & Wise, D.S. (1997). Auto-blocking matrixmultiplication or tracking blas3 performance from source code, *in Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997, pp 206-216.

Ganeshamoorthy, K., Ranasinghe, D.N., Silva, K.P.M.K. & Wait, R. (2009). On the Performance of the Shallow Water Model on Distributed Memory Architectures, *The international journal on advances in ICT for emerging regions (ICTER)*, vol. 2, No1-2, December 2009.

Ganeshamoorthy, K. & Ranasinghe, D.N. (2008). On the Performance of Parallel Neural Network Implementations on Distributed Memory Architectures, *Proceedings of 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*, May 19-22, 2008, Lyon, France . IEEE 2008, pp 90-97.

Ganeshamoorthy, K., Ranasinghe, D.N., Silva, K.P.M.K. & Wait, R. (2007). Parallel implementation of shallow water model on distributed memory architectures, *Proceedings of the ISCA 20th International Conference on Parallel and Distributed Computing Systems (PDCS 2007)*, September 24-26, 2007, Las Vegas, Nevada, USA, ISCA 2007, pp 181-186.

Ganeshamoorthy, K., Ranasinghe, D.N., Silva, K.P.M.K. & Wait, R. (2007). Performance of Shallow Water Equations Model on the Computational Grid with Overlay Memory Architectures, *Proceedings of the Second International Conference on Industrial and Information Systems (ICIIS 2007)*, Faculty of Engineering, University of Peradeniya, Sri Lanka, 08-11 August, 2007, IEEE 2007, pp 415-420.

Gansterer, W.N. & Zottl, O. (2005). Message Passing vs. Virtual Shared Memory - A Performance Comparison, *The Springer International Series in Engineering and Computer Science, Distributed and Parallel Systems*, vol 777, December 2005, pp 39-46.

Ghosh, J. & Hwang, K. (1989). Mapping neural networks onto message passing multicomputers, *Journal of parallel and distributed computing*, vol. 6, no. 2, 1989, pp 291 - 330.

Goto, C. & Ogawa, Y. (1992). Numerical method of tsunami simulation with the leap-frog scheme, *UNESCO Intergovernmental Oceangraphic Commission, Manuals and Guides*, **35**, 1992, Dept. of Civil Engineering, Tohoku University, *Translated for the Tsunami Inundation Modelling Exchange Project, by N. Shuto.*

Harada, H., Tezuka, H., Takahashi, T., Sumimoto, S., Hori, A. & Ishikawa, Y. (1999). SCASH: Software DSM using High performance network using commodity hardware and software, 8th *workshop on scalable shared-memory multiprocessors*, ACM Press, May 1999, pp 26-27.

Haykin, S. (2001). *Neural Networks*, A Comprehensive Foundation, Second Edition, published by Addison WesLongman (Singapore) Pte.Ltd., 2001.

Heaton, J.T. (2005). `Introduction to Neural Networks with Java`, Published by Heaton Research, Inc., ISBN-13: 9780977320608.

Hill, M.D & Marty, M.R. (2008). Amdahls Law in the Multicore Era, *Published by the IEEE Computer Society*, vol 41, no. 7, July 2008, pp 33-38. [online]
http://www.cs.wisc.edu/multifacet/papers/ieeecomputer08_amdahl_multicore.pdf
[10/10/2009]

Hope, L. & Lam, E. (2008). A Review of Applications of Cluster Computing, [online] http://www.buyy
[15/04/2009]

Hu, J., Sasakawa, T., Hirasawa, K. & Zheng, H. (2007). A Hierarchical Learning System Incorporating with Supervised, Unsupervised and Reinforcement Learning, *Lecture Notes in Computer Science*, vol. 4491, 2007, pp 403-412.

Hung, S.L. & Adeli, H. (1994). A Parallel Genetic/Neural Network Learning Algorithm for MIMD Shared Memory Machines, *IEEE Transactions on Neural Networks*, vol. 5, Issue 6, Nov. 1994, pp 900-909.

Husbands, P. & Yelick, K. (2007). Multi-Threading and One-Sided Communication in Parallel LU Factorization, *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, Nevada, 2007, Article No. 31, ISBN:978-1-59593-764-3.

Jain, A.k., Mao, j. & Mohiuddin, K.M. (1996). Artificial Neural Networks: A Tutorial, *The flagship magazine of the IEEE Computer Society*, vol. 29, no. 3, ISSN: 0018-9162, March 1996, pp 31-44.

Jianga, C., Lia, K., Liub, N. & Zhang, Q. (2005). Implicit Parallel FEM Analysis of Shallow Water Equations, *Tsinghua Science & Technology*, vol. 10, Issue 3, June 2005, pp 364-371.

Kasim, H., March, V., Zhang, R. & See, S. (2008). Survey on parallel programming model, *Proceedings of the IFIP International Conference on Network and Parallel Computing*, LNCS, vol. 5245, October 2008, pp 266 - 275.

Keleher, P., Cox, A.L., Dwarkadas, S. & Zwaenepoel, W. (1994). TreadMarks: distributed shared memory on standard workstations and operating system, *Proceedings of the Technical Conference on USENIX Winter*, January 1994, pp 115-131.

Kumar, V., Grama, A., Gupta, A. & Karypis, G. (1994). *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms.* Addison-Wesley Pub Co., ISBN:0805331700.

Leverich, J., Arakida, H., Solomatnikov, A., Firoozshahian, A., Horowitz, M. & Kozyrakis, C. (2007). Comparing memory systems for chip multiprocessors, *ACM SIGARCH Computer Architecture News*, vol. 35, Issue 2, May 2007, pp 358-368.

Long, L.N. & Gupta, A. (2008). Scalable Massively Parallel Artificial Neural Networks, *Journal of Aerospace Computing, Information, and Communication (JACIC)*, vol. 5, no. 1, January 2008, pp 1-11.

Lotrič, U. & Dobnikar, A. (2005). Parallel implementations of feed-forward neural network using MPI and C# on .NET platform, *Proceedings of the International Conference in Adaptive and Natural Computing Algorithms*, Portugal, Publisher:Springer Vienna, December 2005, pp 534-537.

Ma, S., Ji, C. & Farmer, J. (1997). An efficient EM-based training algorithm for feed-forward neural networks, *Neural Networks*, vol. 10, no. 2, 1997, pp 243-256, ISSN:0893-6080.

Marchesi, M., Orlandi,G., Piazza, F. & Uncini, A. (1990). Liner array architecture implementing the back-propagation neural network, *Second Italian workshop on parallel architectures and neural networks, World scientific publishing*, 1990, pp 345-352. [online] http://www.uncini.com/research_activity/pdf/008_wirn90.pdf [11/02/2007]

Mattson, T.G. (1995). Programming Environments for Parallel and Distributed Computing: a Comparison of P4, PVM, Linda, and TCGMSG, *International Journal of High Performance Computing Applications*, vol. 9, no. 2, 1995, pp 138-161.

McCulloch, W.S & Pitts, W. (1998). A logical calculus of the ideas immanent in nervous activity, *Source: Neurocomputing: foundations of research book*, The MIT Press Cambridge, MA, USA, 1998, pp 15-27.

Mehrotra, K., Mohan, C. & Ranka, S. (1996). *Elements of Artificial Neural Networks*, ISBN-10:0-262-13328-8.

Nordström, T. & Svensson, B. (1992). Using and designing massively parallel computers for artificial neural networks, *Journal of parallel and distributed computing*, vol. 14, March 1992, pp 260-285.

Novokhodko,A. & Valentine, S. (2001). A parallel implementation of the batch backpropagation training of neural networks, *Proceedings of the International Joint Conference on Neural Networks (IJCNN'01)*, vol.3, IEEE, 2001 , pp 1783-1786.

Osowski, P.B.S. & Stodolski, M. (1996). Fast second order learning algorithms for feed-forward neural networks and its applications, *Neural Networks*, vol. 9, No. 9, 1996, pp 1583-1596.

Ouyang, Y., Yang, L., Sun, R. (2008), Research of Grid Platform Oriented to Intensive Data Processing, *Proceeding of the Seventh International Conference on Grid and Cooperative Computing*, October 2008, pp 226-231.

Patel, I. & Gilbert, J.R. (2008). An empirical study of the performance and productivity of two parallel programming models, *International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, IEEE, April 2008, pp 1 - 7.

Pethick, M., Liddle, M., Werstein, P. & Huang, Z. (2003). Parallelization of a backpropagation neural network on a cluster computer, *Proceedings of the* 15th *IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, Marina Del Rey, California, 2003, pp 574-582.

Robertson, N. & Rendell, A. (2003). OpenMP and NUMA Architectures I: Investigating Memory Placement on the SGI Origin 3000, January 2003, in Sloot, P.M.A et al. (eds) *Computational Science - ICCS 2003*, LNCS 2660, Springer-Verlag Berlin Heidelberg 2003, pp 648-656.

Rumelhart, D.E., Hinton, G.E. & Williams, R.J. (1986). Learning internal representations by error propagation, *Mit Press Computational Models of Cognition and Perception Series, Parallel distributed processing: explorations in the microstructure of cognition*, vol. 1, Cambridge, MA, USA, 1986, pp 318-362.

Sagan, H. (1994). Space-Filling Curves, Springer-Verlag, 1994. ISBN:0387942653.
Sanders, B.F & Pau, J.C. (2003). Parallel Implementation of an Explicit Finite-volume Shallow-water Model, 16th *ASCE Engineering Mechanics Conference*, University of Washinton, Seattle, July 2003. [online]
http://www.ce.washington.edu/em03/proceedings/papers/722.pdf [12/08/2006].

Seiffert, U. (2002). Artificial neural networks on massively parallel computer hardware, *Proceedings of the European symposium on artificial neural networks (ESANN'2002)*, Bruges (Belgium), April 2002, pp 319-330.

Shan, H., Singh, J.P., Oliker, L. & Biswas, R. (2001). Message passing vs. shared address space on a cluster of SMPs. *Proceedings of the* 15th *International Parallel and Distributed Processing Symposium, IEEE Computer Society*, vol. 1, April 2001, pp 10063b.

Singer, A. (1990). Implementation of artificial neural networks on the connection machine, *Parallel computing*, vol. 14, summer 1990, pp 305-315.

Sitangang K.I. & Lynett P. (2005). Parallel computation of a highly nonlinear Boussinesq equation model through domain decomposition, *International journal for numerical methods in fluids*, vol. 49, no.1, 2005, pp 57-74.

Skabar, A. & Cloete, I. (2002). Neural networks, financial trading and the efficient markets hypothesis, *Proceedings of the twenty-fifth Australasian conference on Computer science*, vol. 4, Melbourne, Victoria, Australia, ACM Press, 2002, pp 241-249.

Skalin, R. (1996). Scalability of Parallel Grid Point Limited Area Atmospheric Models I & II, *Manuscript*, Department of Mathematics Sciences, Norwegian University of Science and Technology, Trodheim, Norway, 1996.

Smith, L. & Bull, M. (2001). Development of mixed mode MPI/OpenMP applications, *Scientific Programming*, vol. 9, Issue 2,3, August 2001, , pp 83-98.

Solheim, J.G. (1995). The RENNS approach to neural computing. PhD Thesis, Norwegian Institute of Technology, Norway.

Sudhakar, V. & Murthy, C.S.R. (1998). Efficient mapping of back-propagation algorithm onto a network of workstations, *IEEE Trans. Man, Machine, and Cybernetics-Part B: Cybernetics*, vol. 28, no. 6, 1998, pp 841-848.

Sundararajan, N. & Saratchandran, P. (1998). *Parallel Architectures for Artificial Neural Networks*, Wiley-IEEE Computer Society Press, ISBN: 978-0-8186-8399-2.

Suresh, S., Omkar, S.N., & Mani, V. (2005). Parallel implementation of back-propagation algorithm in networks of workstations, *IEEE Transactions on parallel and distributed systems*, vol. 16, no. 1, January 2005, pp 24-34 .

Suri, N.N.R.R, Deodhare, D. & Nagabhushan, P. (2002). Parallel Levenberg-Marquardt-Based Neural Network Training on Linux Clusters, *ICVGIP 2002*, Ahmadabad, India, 2002. [online] http://www.ee.iitb.ac.in/ icvgip/PAPERS/248.pdf [16/04/2008]
.

Thomas, S., Cote, J., Staniforth, A., Lie, I. & Skalin, R. (1994). A Semi-Implicit Semi-Lagrange Shallow-Water Model for Massively Parallel Processors, *Proceedings of the* 6th *ECMWF Workshop on the Use of Parallel Processors in Meteorology*, November 1994, pp 407-423

Thulasiram, R.K., Rahman, R.M. & Thulasiraman, P. (2003). Neural network training algorithms on parallel architectures for finance applications, *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'03)*, IEEE, 6-9 Oct. 2003, pp 236-243.

Treleven, P. (1989). Neurocomputers, *Research Note 89/8*, Department of Computer Science, University College London, January 1989.

Wang, C.J., Wu, C.H. & Sivasundaram, S. (1989). Neural network simulation on shared-memory vector multiprocessors, *Conference on High Performance Networking and Computing, Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Reno, Nevada, United States, 1989, pp 197-204.

Wasserman, P.D. (1989). *Neural Computing: Theory and Practice*, Reprint Binding: Hard Cover Publisher: Van Nostrand Reinhold Co, New York, ISBN:0-442-20743-3.

Wei, G., Kirby, J.T., Grilli, S.T. & Subramanya, R. (1995). A fully nonlinear Boussinesq model for surface waves, Part I, Highly nonlinear unsteady waves, *Journal Fluid Mechanics 1995*; 294:71-92.

Werstein, P., Pethick, M. & Huang, Z. (2003). A performance comparison of DSM, PVM, and MPI, *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'2003)*, 27-29 Aug, IEEE, 2003, pp 476 - 482.

Williamson, D.L., Drake, J.B., Hack, J.J.R. & Swarztrauber, J.J. (1991). A standard Test Set for numerical Approximations to the Shallow Water Equations in Spherical Geometry, *Journal of Computational Physics*, 1991, pp 211-224,.

Wong, H.J. & Rendell, A.P. (2008). The design of MPI based distributed shared memory systems to support OpenMP on clusters, *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, Sep. 2008, pp 231-240.

Wu, M.S., Aluru, S. & Kendall, R.A. (2002). Mixed Mode Matrix Multiplication, *Proceedings of the IEEE International Conference on Cluster Computing*, 2002, pp 195-204.

Yasunaga, M. & Yoshida, E. (1998). Optimization of parallel BP implementation: training speed of 1056MCUPS on the massive, *Proceedings of the IEEE International Joint Conference on Neural Networks*, vol.1, 1998, pp 563-568.

Yoon, H., Nang, J.H., & Maeng, S.R. (1990). Parallel simulation of multilayered neural networks on distributed memory multiprocessors, *Microprocessing and Microprogramming,* vol. 29, 1990, pp 185-195.

Web References

[WWW 01] MPI Forum web page. http://www.mpi-forum.org/ [10/02/2006]

[WWW 02] PVM web page. http://www.csm.ornl.gov/pvm/ [15/02/2006]

[WWW 03] TCGMSG web page.
http://www.emsl.pnl.gov/docs/parsoft/tcgmsg/tcgmsg.html [15/02/2006]

[WWW 04] AVS/Express web page.
http://www.avs.com/software/soft_t/parallel_edition.html
http://www.avs.com/pdf/AVSExpressSlickClr.pd [05/03/2006]

[WWW 05] UPC web page. http://upc.lbl.gov/ [15/10/2008]

[WWW 06] TCP-Linda User Guide, Scientific Computing Associates IC., One Century Tower, 265 Church Street, New haven, CT 06510-7010 USA, September, 2005.
http://www.lindaspaces.com/products/ [01/02/2006]

[WWW 07] Cluster OpenMP*, *User's Guide, Version 9.1*, Copyright © 2005-2006 Intel Corporation, Document number: 309076-002 US.
http://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers
[12/01/2008]

[WWW 08] OpenMP web page. http://openmp.org/wp/ [17/06/2006]

[WWW 09] http://data1.gfdl.noaa.gov/∼arl/pubrel/m/atm_dycores/doc/ [01/10/2009]

[WWW 10] http://www.sea.ee/∼elken/DO5.pdf [20/06/2006]

[WWW 11] Göran Broström. Geophysical fluid dynamics with focus on shallow water dynamics, Large scale waves and coastal effects: Part 1, theory, Universitet Stockholms.
http://www.misu.su.se/∼ goran/shallow_water.html [01/10/2009]

[WWW 12] Pthreads webpage.
https://computing.llnl.gov/tutorials/pthreads/ [19/01/2007]

[WWW 13] *The Grid middleware project in the Nordic countries.*
http://nordugrid.org [25/03/2006]

[WWW 14] UPC Language Specifications V1.2, *A publication of the UPC Consortium*, May 31, 2005. upc.lbl.gov/docs/user/upc_spec_1.2.pdf [05/06/2008]

[WWW 15] The CORSO web page. http://earl.strain.at/space/Corso [05/01/2008]

[WWW 16] The portable batch system web page. www.openpbs.org/ [02/05/2002]

[WWW 17] Hope, L. & Lam, E. (2008). A Review of Applications of Cluster Computing, 2008. http://www.gridbus.org/~raj/csc433/ClusterApps.pdf [19/04/2009]

[WWW 18] MPICH2 webpage.
http://www.mcs.anl.gov/research/projects/mpich2/ [11/02/2008]

[WWW 19] LAM/MPI webpage. http://www.lam-mpi.org/ [11/02/2008]

[WWW 20] Open MPI webpage. http://www.open-mpi.org/ [11/02/2008]

[WWW 21] Moore's Law. http://www.intel.com/technology/mooreslaw/
http://www.webopedia.com/TERM/M/Moores_Law.html
http://en.wikipedia.org/wiki/Moore's_law [01/10/2009]

Appendix A

Theory of Shallow Water Models

A.1 Introduction

The equations governing the motion can be written

$$\frac{d\mathbf{u}}{dt} + f\mathbf{k} \times \mathbf{u} = -\frac{1}{\rho}\nabla p + v^2\mathbf{u} - g\mathbf{k} \tag{A.1.1}$$

where $\mathbf{u} = (u, v, w)$ is the velocity vector, $d/dt = \partial/\partial t + \mathbf{u}.\nabla$ is the total derivative, and $f = 2\Omega sin\phi$ is the Coriolis parameter. Neglecting the compressibility of water the continuity equation reads

$$\nabla.\mathbf{u} = 0 \tag{A.1.2}$$

The density depends most generally on the temperature and the salinity. Thus to complete the equations we should also include

$$\frac{dT}{dt} = v\nabla^2 T, \tag{A.1.3}$$

$$\frac{dS}{dt} = v\nabla^2 S. \tag{A.1.4}$$

A.2 Boundary condition

The equations are subject to certain boundary conditions that can be very complicated, especially at the sea surface. We will assume that the horizontal velocities are zero at solid walls, thus,

$$(u, v) = 0, \quad \text{at solid walls.} \tag{A.2.1}$$

At the free surface and the bottom kinematic conditions must be fulfilled:

$$w_{z=h} = \frac{\partial h}{\partial t} + \mathbf{u}_{h_{z=h}}.\nabla_h h \quad \text{at} \quad z = h, \tag{A.2.2}$$

$$w_{z=-H} = -\mathbf{u}_{h_{z=-H}}.\nabla_h H \quad \text{at} \quad z = -H, \tag{A.2.3}$$

where $\mathbf{u}_h$ is the horizontal velocity vector. If we do not consider any coupling to the atmosphere and neglect frictional terms and pressure continuity at the sea surface, the equations are well specified by boundary conditions. However, often the stress at the sea surface and the bottom are important for dynamics we choose to study. Thus, the vertical frictional terms cannot be neglected in the vicinity of the sea surface and the bottom. For instance, a wind blows over the sea surface it will give a stress at the sea surface creating a amass transport that must be considered. The condition for a stress at the sea surface is

$$v\frac{\partial \mathbf{u}_h}{\partial z} = \tau, \quad \text{at} \quad z = h(x, t), \tag{A.2.4}$$

and the conditions at the bottom reads

$$\mathbf{u}_h = 0, \quad \text{at} \quad z = H(x, t).$$

Applying this condition may require a special treatment of condition ($A.2.2$ & $A.2.3$), i.e., we may have to consider thin boundary layers.

The barotropic flow can be described by integrating Navier-Stokes, and continuity equations from the bottom, $-H(x, y)$, to the surface $h(x, y)$.

A.3 The integration: Mass continuity

Let us consider an integral of the continuity equation from the bottom to the sea surface as an illustrative example. The continuity equation, here also including possible changes in the density with time, reads,

$$\frac{\partial \rho}{\partial t} + \nabla.(\rho \mathbf{u}) = 0 \tag{A.3.1}$$

Rewriting

$$\frac{\partial \rho}{\partial t} + \nabla_h.(\rho \mathbf{u}_h) + \frac{\partial \rho w}{\partial z} = 0 \tag{A.3.2}$$

where ∇_h and $\mathbf{u}_h$ are the horizontal gradient and velocities, respectively. Integration from the bottom to the surface yields

$$\int_{-H}^{h} \frac{\partial \rho}{\partial t} dz + \int_{-H}^{h} \nabla_h.(\rho \mathbf{u}_h) dz + \rho \left(w_{z=h} - w_{z=-H} \right) = 0 \tag{A.3.3}$$

Now the chain rule for integration gives,

$$\int_{-H}^{h} \frac{\partial \rho}{\partial t} dz = \frac{\partial}{\partial t} \int_{-H}^{h} \rho dz - \rho_{z=h} \frac{\partial h}{\partial t} - \rho_{z=-H} \frac{\partial H}{\partial t} \tag{A.3.4}$$

$$\int_{-H}^{h} \nabla_h.(\rho \mathbf{u}_h) dz = \nabla_h \int_{-H}^{h} \rho \mathbf{u}_h dz - \rho \mathbf{u}_{h_{z=h}}.\nabla_h h - \rho \mathbf{u}_{h_{z=-H}}.\nabla_h H. \tag{A.3.5}$$

Equations (A.3.3), (A.3.4), and (A.3.5):

$$\begin{aligned} &\frac{\partial}{\partial t} \int_{-H}^{h} \rho dz - \rho_{z=h} \frac{\partial h}{\partial t} - \rho_{z=-H} \frac{\partial H}{\partial t} \\ &+\nabla_h \int_{-H}^{h} \rho u_h dz - \rho u_{h_{z=h}} \nabla_h h - \rho u_{h_{z=-H}} \nabla_h H \\ &+\rho \left(w_{z=h} - w_{z=-H} \right) = 0 \end{aligned} \tag{A.3.6}$$

Assuming that ρ does not vary with depth

$$\begin{aligned} &\frac{\partial}{\partial t} \int_{-H}^{h} \rho dz + \nabla_h \int_{-H}^{h} \rho u_h dz \\ &-\rho \left(\frac{\partial h}{\partial t} + u_{h_{z=h}}.\nabla_h h - w_{z=h} \right) \\ &-\rho \left(\frac{\partial H}{\partial t} + u_{h_{z=-H}}.\nabla_h H + w_{z=-H} \right) \\ &= 0 \end{aligned} \tag{A.3.7}$$

Recapitulating the kinematic boundary conditions at the sea surface and the bottom

$$w_{z=h} = \frac{\partial h}{\partial t} + \mathbf{u}_{h_{z=h}}.\nabla_h h,$$
$$w_{z=-H} = \mathbf{u}_{h_{z=-H}}.\nabla_h H. \tag{A.3.8}$$

And applying these boundary conditions to (*Equation A*.3.7) gives

$$\frac{\partial}{\partial t}\int_{-H}^{h} \rho dz + \nabla_h \int_{-H}^{h} \rho \mathbf{u}_h dz = 0. \tag{A.3.9}$$

This is a very general formula and can be used to include the influence of surface waves etc.. Assuming that the density and the velocity is constant with depth we find

$$\frac{\partial}{\partial t}\rho(h+H) + \nabla_h\left[\rho(h+H)\mathbf{u}_h\right] = 0. \tag{A.3.10}$$

It should be noted that this equation is easily derived using basic mass conservation. To continue, H is not usually not a function of time and for geophysical fluid dynamics we have $h \ll H$, thus

$$\frac{\partial}{\partial t}h + \nabla_h\left(H\mathbf{u}_h\right) = 0. \tag{A.3.11}$$

A.4 Equations in velocity form

To find the velocities describing the barotropic- or integrated- flow the equation must be integrated between the bottom and the sea surface, as was outlined above. The stress condition at the sea surface and the bottom should be included as well. The equation that governs the time evolution of the system- here assuming that $h \ll H$, neglecting the free surface gravitational waves and the horizontal and vertical diffusion -reads

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv = -g\frac{\partial h}{\partial x} + \frac{1}{H\rho_o}\left(\tau_x^S - \tau_x^B\right) \tag{A.4.1}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + fu = -g\frac{\partial h}{\partial y} + \frac{1}{H\rho_o}\left(\tau_y^S - \tau_y^B\right) \tag{A.4.2}$$

$$\frac{\partial h}{\partial t} + H\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + u\frac{\partial H}{\partial x} + v\frac{\partial H}{\partial y} = 0 \tag{A.4.3}$$

where $\tau_x^S \quad \tau_y^S$ are the surface stresses in the x and y direction, respectively. $\tau_x^B \quad \tau_y^B$ are the bottom stresses and are described by

$$\tau_x^B = \rho C_D u|u| \tag{A.4.4}$$

$$\tau_y^B = \rho C_D v|v| \tag{A.4.5}$$

A.5 Equations in transport form

It may be useful to consider the mass transport as variables instead of the velocity: Let us therefore introduce the mass transports $U = u(h+H), \quad V = v(h+H)$. Rewriting the equation with assuming that $h \ll H$ gives

$$\frac{\partial U}{\partial t} + H^{-1}\left(U\frac{\partial U}{\partial x} + V\frac{\partial U}{\partial y}\right) - H^{-2}\left(U^2\frac{\partial H}{\partial x} + UV\frac{\partial H}{\partial y}\right) - fV =$$

$$-gH\frac{\partial h}{\partial x} + \frac{1}{\rho_o}\left(\tau_x^S - \tau_x^B\right) \tag{A.5.1}$$

$$\frac{\partial V}{\partial t} + H^{-1}\left(U\frac{\partial V}{\partial x} + V\frac{\partial V}{\partial y}\right) - H^{-2}\left(UV\frac{\partial H}{\partial x} + V^2\frac{\partial H}{\partial y}\right) + fU =$$

$$-gH\frac{\partial h}{\partial y} + \frac{1}{\rho_o}\left(\tau_y^S - \tau_y^B\right) \tag{A.5.2}$$

$$\frac{\partial h}{\partial t} + \left(\frac{\partial U}{\partial x} + \frac{\partial V}{\partial y}\right) = 0 \tag{A.5.3}$$

where

$$\tau_x^B = \rho C_D U|U|/H^2 \tag{A.5.4}$$

$$\tau_y^B = \rho C_D V|V|/H^2 \tag{A.5.5}$$

A.6 Linear shallow water equations

Let us reglet the non-linear terms for the moment. The linear shallow water equations in transport form thus become

$$\frac{\partial U}{\partial t} - fV = -gH\frac{\partial h}{\partial x} + \frac{1}{\rho_o}\left(\tau_x^S - \tau_x^B\right) \tag{A.6.1}$$

$$\frac{\partial V}{\partial t} + fU = -gH\frac{\partial h}{\partial y} + \frac{1}{\rho_o}\left(\tau_y^S - \tau_y^B\right) \tag{A.6.2}$$

$$\frac{\partial h}{\partial t} + \left(\frac{\partial U}{\partial x} + \frac{\partial V}{\partial y}\right) = 0 \tag{A.6.3}$$

Appendix B

Matrix Multiplications

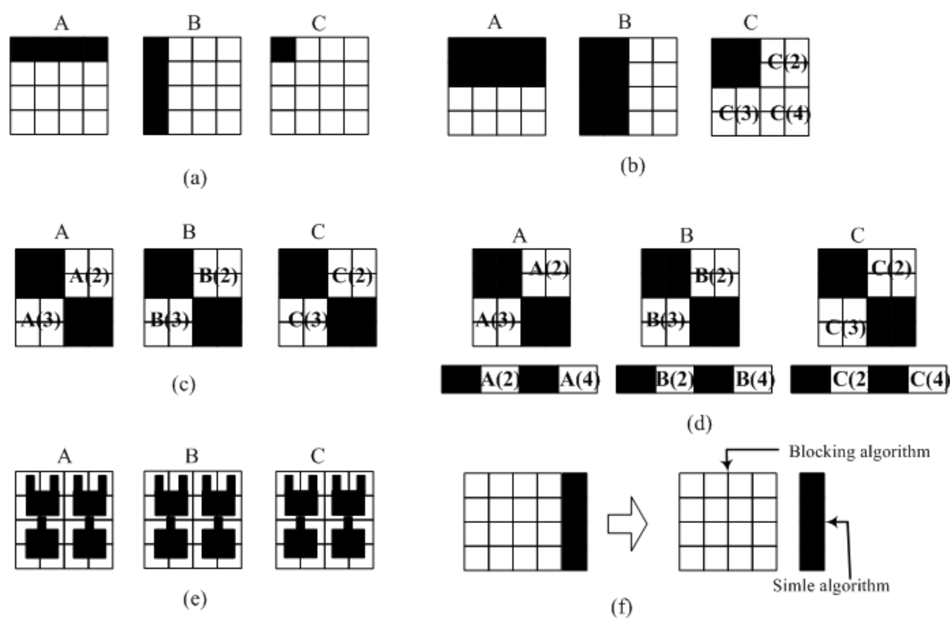

Figure B.1: Pictorial representation of cache level matrix multiplication algorithms (Wu et al., 2002).

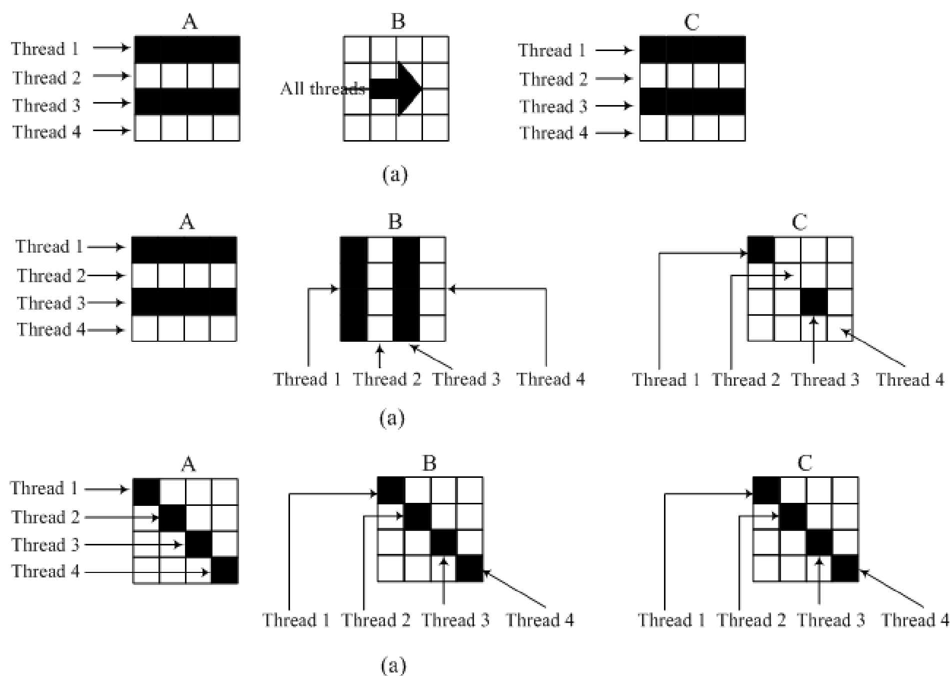

Figure B.2: Pictorial representation of shared memory matrix multiplication algorithms (Wu et al., 2002).

B.1 Cache layer matrix multiplication algorithms

Cache based algorithms which are vary from those that have high cache misses to those that effectively use cache, are given the following sub sections.

B.1.1 Simple three loops algorithm

Memory access patterns of the simple three loops algorithm is illustrated in Figure B.1 (a). This algorithm will incur the most cache misses among all other cache algorithms presented here. However, this algorithm includes fewest instruction count.

B.1.2 Blocking C algorithm

This algorithm compute the matrix C block by block, as shown in Figure B.1 (b). Since the algorithm computes according to the square patch of C, the corresponding portions of matrices A and B are not required to be square. When the matrix dimensions increase, the size of A and B patches also increase and thus incur more cache misses. This algorithm shows better performance when the sizes of matrices are small.

B.1.3 Blocking A algorithm

Figure B.1 (c), fix a block of A or B, outlines the memory access order of this algorithm. The algorithm keeps a square patch of matrix A in the cache as long as possible. For example, block A(1) is computed with block B(1) and result put into block C(1), then A(1) is multiplied by B(2) and stores the data in C(2). A(1) is then swapped out and replaced by A(2) to multiply B(3) and B(4), and so on.

B.1.4 Transform and blocking A algorithm

This algorithm first applies the transform operation to all three matrices and then starts the computation, as illustrated in Figure B.1 (d). This algorithm is same as the previous algorithm. The only difference is that the layout of all three matrices are transformed before computation starts. The layout of elements in each block is made consecutively by creating a block that is small enough to fit into cache and copying the appropriate portion to the newly allocated block.

B.1.5 Recursive algorithm

Chatterjee and coworkers (Chatterjee et al., 1999) and Frens and Wise (Frens & Wise, 1997) describe the recursive layout of matrix multiplication, as shown in Figure B.1 (e), that the data is transformed into layout according to different space-filling curve order (Sagan, 1994); then computations are done recursively according to that order. Because of the recursion, data has to be a power of 2. Different methods of Chatterjee et. al. (Chatterjee et al., 1999) and Frens and Wise (Frens & Wise, 1997) are used to handle the case when data size is not a power of 2. Wu and coworkers (Wu et al., 2002) implemented a simple version of the “U layout” which works on only square matrices. The blocking shell takes care of the case when data size are not a power of 2.

B.1.6 Strassen’s algorithm

In theory Strassen’s algorithm has better run time for matrix multiplication; it use additions and subtractions to reduce the times for multiplication. The algorithm processes data in small blocks recursively, which make it implicitly cache efficient. Meng-Shiou Wu (Wu et al.,

2002) et.al have found that recursion down to a single element reduced the performance and terminating the recursion at even a small block size increased the overall performance.

B.2 Shared memory layer matrix multiplication

Four shared memory matrix multiplication algorithms that can be easily implemented in the Pthreads programming model, are described bellow.

B.2.1 Overlapping matrix B

This algorithm divides matrix A into several rectangle blocks horizontally according to the number of threads, as illustrated in Figure B.2 (a). Each thread computes certain rectangle block of matrix A with whole matrix B and produces a complete contribution to a portion of matrix C. The algorithm has the possibility of causing read contention when different threads try to read matrix B.

B.2.2 Non-overlapping algorithm

This algorithm divides matrix A horizontally and matrix B vertically into blocks, as illustrated in Figure B.2 (b). Each thread first computes a block of matrix A multiplied by a block matrix B thus produces a full contribution of a square block of matrix C. In the next stage every thread still use the same block of matrix A, but shifts to another block of matrix B, thus producing another full contribution to a different square block of matrix C. In this way, if all threads are executed concurrently then different threads have less chance of accessing the elements of matrix B at the same time when the number of threads is less than the number of blocks of A and B pairs.

B.2.3 Blocking algorithm

This algorithm divides all three matrices into smaller square blocks, and each thread computes a square block of matrix A with a square block of B thus producing a partial square contribution to matrix C, as illustrated in Figure B.2 (c). The block computation order is the same as element computation order in simple three-loop algorithm.

B.2.4 Transform and blocking algorithm

This algorithm has the same computation order and work on the same shapes of matrices as the previous algorithm, except before the computation begins, all three matrices are transformed into blocks of consecutive data as in the above mentioned cache algorithm, and each thread works on three square patches with consecutive elements.

B.3 Distributed memory layer matrix multiplication

B.3.1 Two dimensional broadcasting algorithm

According to the number of physical nodes and physical grid, if the physical grid is $p \times q$, then matrices are partitioned into least common multiples of p and q parts. Each node takes turns broadcast the part of matrix A vertically or B horizontally or both, and computes according to the data it receives.

B.3.2 Cannon's algorithm

The algorithm is described in almost every parallel algorithm book such as Kumar et. al., (Kumar et al., 1994). Meng-Shiou Wu et al's (Wu et al., 2002) implementation, first form a grid using the available processors, then depending on the shape of the grid, distribute data accordingly. If the grid is square, distribute data as in Cannons original algorithm. If the grid is rectangular, find the least common multiple of two dimensions, use the least common multiple as the dimension of a virtual square grid and then distribute data according to this virtual grid.

Appendix C

Hardware Configurations

In recent years, the speed of CPUs grew steadily. Unfortunately, clock speed of CPUs stagnates now. Latest developments of CPUs go to multicore processors. Intel and AMD, which are the most important manufacturers for standard computer CPUs, introduced so-called quad-core processors [14]. Each of these processors possesses four cores. A dual-processor computer can execute eight independent operations in parallel. This trend makes parallelization of software not optional but indispensable.

A current modern computer consists of a multicore CPU with a shared memory as shown in Figure C.2. This means that all processors access the same memory. Data in one memory cell is accessible to all processors. In a computer cluster, parallelization is more complicated. CPUs and memory are distributed among the computing nodes as shown in Figure C.1. HPC cluster consists of symmetric multi processors (SMP's) with each SMP a chip multi processor (CMP).

C.1 Multiprocessors

Multiprocessor system is known as tightly-coupled system or parallel system. Generally, the processors are in close communication with each other. They share common data structures and a common system clock [11], [38], [48].

Advantages of Multiprocessor Systems:

- Reduced Cost: Multiple processors share the same resources. Separate power supply or mother board for each chip is not required. This reduces the cost.
- Increased Reliability: The reliability of system is also increased. The failure of one processor does not affect the other processors though it will slow down the machine. Several mechanisms are required to achieve increased reliability. If a processor fails, a job running on that processor also fails. The system must be able to reschedule the failed job or to alert the user that the job was not successfully completed.
- Increased Throughput: An increase in the number of processes completes the work in less time. It is important to note that doubling the number of processors does not halve the time to complete a job. It is due to the overhead in communication between processors and contention for shared resources etc.

C.2 Symmetric Multi Processors (SMP)

The most prevalent form of para architectures is the multiprocessor of small to moderate scale that provides a global physical address space and symmetric access to all of main memory from

any processor, often called an SMP.

SMP systems provide scalability. Additional CPUs can be added to absorb the increased transaction volume. If one CPU fails, the entire SMP system is down. Clusters of more than one SMP systems can be used to provide high availability. If one SMP system fails, the others continue to operate [5].

C.3 Chip Multi Processors (CMPs)

Chip multiprocessors (Leverich et al., 2007), (Cheng et al., 2006) are emerging as the architecture of choice for future high performance processors. The architecture of the CMP system with up to 16 processors is illustrated in Figure C.3. CMPs integrate several high-performance processing cores onto the same chip. A high performance interconnect and memory system are necessary to satisfy the data supply needs of all these cores especially given the ever increasing speed gap between processors and main memory systems. At the same time, power, temperature, complexity, and reliability are additional constraints that must be met by any design. The fact that now these components will be tightly integrated onto the same die presents opportunities and challenges that are very different than those that existed in previous multiprocessor systems.

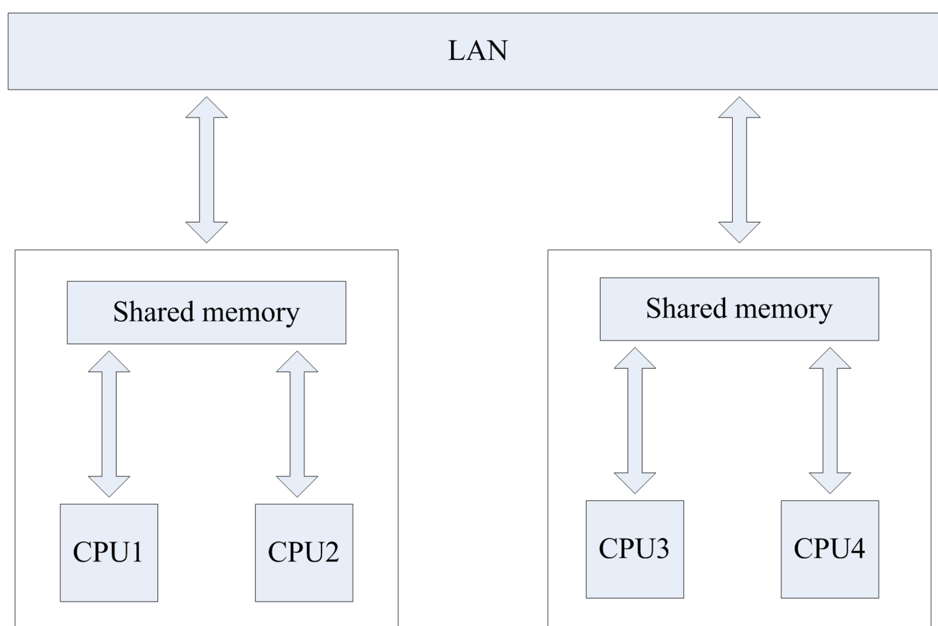

Figure C.1: Communication between threads on a distributed memory computer (Buchau et al., 2008).

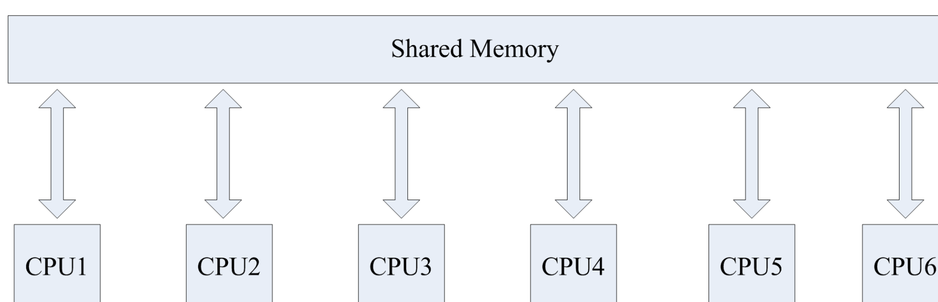

Figure C.2: Communication between threads on a shared memory computer (Buchau et al., 2008).

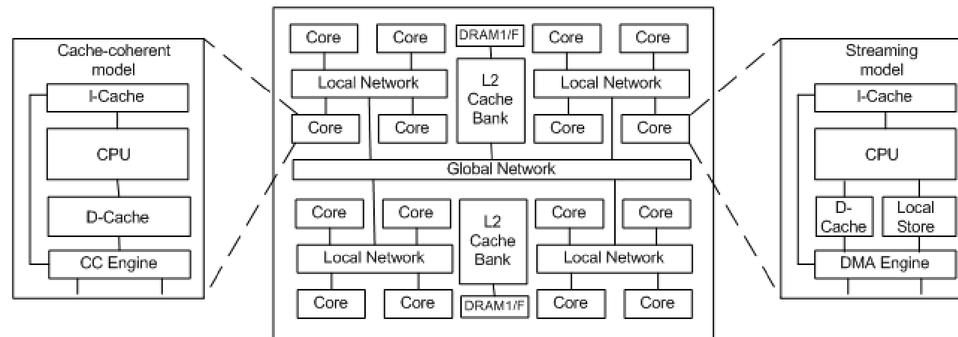

Figure C.3: The architecture of the CMP system with up to 16 processors. The core organizations for the cache-based and streaming models are shown on left and right side respectively (Leverich et al., 2007).

In CMP systems, there are two basic memory models for the on-chip memory (Leverich et al., 2007): *hardware managed, implicitly-addressed, coherent caches* and *software-managed, explicitly-addressed, local memories/streaming memory*. The advantage is that the hardware provides best-effort locality and communication management, even when the access and sharing patterns are difficult to statically analyze. With the cache-based model, all on-chip storage is used for private and shared caches that are kept coherent by hardware. With the streaming-memory model, part of the on-chip storage is organized as independently addressable structures. The advantage of streaming memory is that it provides software with full flexibility on locality and communication management in terms of addressing, granularity, and replacement policy.

For both the cache-based and streaming models, certain aspects of the on-chip memory system are set by VLSI constraints such as wire delay. Every node in the CMP system is directly associated with a limited amount of storage (first-level memory) that can be accessed within a small number of cycles. Nodes communicate by exchanging packets over an on-chip network that can range from a simple bus to a hierarchical structure. Additional memory structures (second-level memory) are also connected to the network fabric. The system scales with technology by increasing the number of nodes, the size of the network, and the capacity of second-level storage. Eventually, the scale of a CMP design may be limited by off-chip bandwidth or energy consumption.

Traditional desktop applications are difficult to analyze and favor cache-based systems. In contrast, many upcoming applications from the scientific computing domains, physical simulation, multimedia, and graphics are being targeted by both cache-based and streaming systems. The CMPs for the Xbox360 and PlayStation 3 differ dramatically in their on-chip memory model, as Xbox360s Xenon processor is a cache-based CMP and PlayStation 3s Cell processor is a streaming memory CMP (Leverich et al., 2007), (Cheng et al., 2006).

Appendix D

Feed-Forward ANN

ANN research was first initiated in 1940s with the publication of McCulloch & Pitts (1998) study on the perceptron model shown in Figure D.1. In 1960s, convincing demonstrations using the model were made (Philip, 1989). In the late 1960s, Minsky and Papert proved that severe restrictions in the learning of a single layer model did exist (Minsky & Papert, 1998). For the following twenty five years little research was undertaken. Then the MLP ANN, called back-propagation, was introduced by Rumelhart et al. (1986), and then initiated a remarkable increase in ANN research.

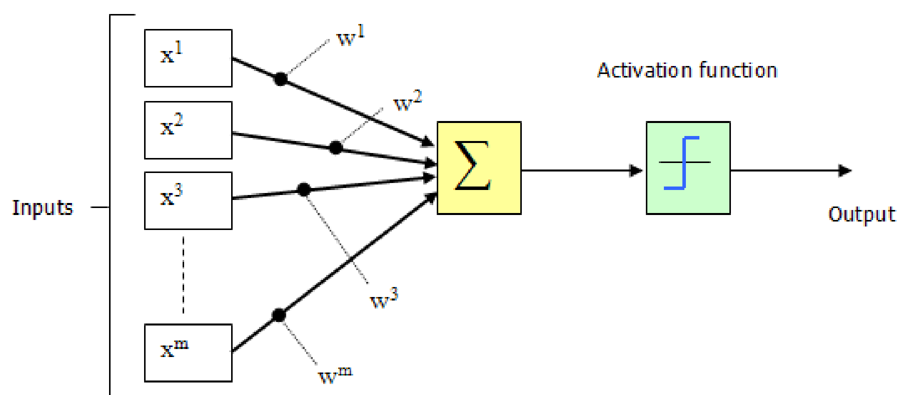

Figure D.1: The perceptron.

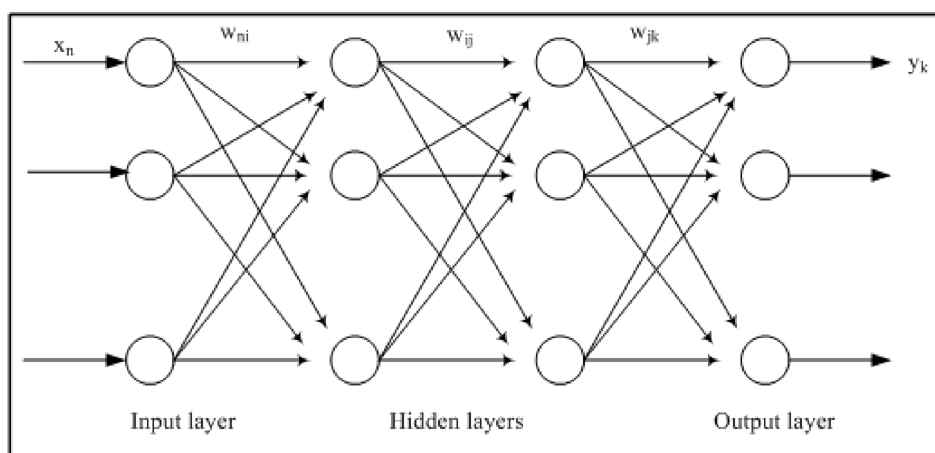

Figure D.2: A four-layer feed-forward ANN.

D.1 The Multi-layer ANN

A four-layer feed-forward ANN is shown in Figure D.2. This ANN is called fully connected, because there are all-to-all connections between two adjacent layers of neurons. The number of neurons in the input layer, hidden layers, and output layer are N_I, N_{H_1}, N_{H_2}, and N_O, respectively. The ANN can be extended to any number of layers by increasing hidden layers.

D.2 Back-propagation Algorithm

In our case study, we use a fully connected MLP ANN trained using the back-propagation learning algorithm. Generally a MLP ANN includes one input layer, one or more hidden layers, and one output layer. However, it has been observed that a single hidden layer MLP ANN with an enough number of hidden neurons is sufficiently able to solve any non-linearly separable problem (Haykin, 2001).

In a back-propagation training algorithm, a given set of input patterns are used for a two phase supervised algorithm based on the error correction learning. The first phase is referred to as the forward phase and the second is referred to as the backward phase (Pethick et al., 2003; Haykin, 2001). In the forward phase the synaptic weights remain unchanged throughout the network, and the function signals of the ANN are computed on a neuron-by-neuron basis. Thus the forward phase of computation begins at the first hidden layer by presenting it with the input pattern, and terminates at the output layer by computing the error signal for each neuron of this layer.

In the backward phase, starts at the output layer by passing the error signals leftward through the ANN, layer by layer, and recursively computing the local gradient (δ) for each neuron. This recursive process permits the synaptic weights of the ANN to undergo changes in accordance with the delta rule (Haykin, 2001; Mehrotra et al., 1996).

The presentation of the complete set of training patterns is known as an epoch. The ANN is trained for a number of epochs until the error measure, the sum of the squares of the difference between the expected and actual outputs, is lower than a specified value. When batch training the ANN instead of applying the update directly to the ANN, we sum the changes in a weight change matrix and apply the summed changes at the end of the epoch.

A usual MLP is shown in Figure D.2. The N_I input neurons are fully connected to the N_{H_1} hidden neurons, N_{H_1} hidden neurons are fully connected to N_{H_2} hidden neurons, and the N_{H_2} hidden neurons are fully connected to the N_O output neurons.

Let's define:

- input patterns: $\mathbf{X} = (x_1,\ x_2,\ x_3, \cdots\cdots,\ x_{N_I})$
- output patterns: $\mathbf{Y} = (y_1,\ y_2,\ y_3, \cdots\cdots,\ y_{N_O})$
- hidden weights: $\mathbf{W}^H = \left(w^h_{ji}\right)$ where $1 \le i \le N_I,\quad 1 \le j \le N_H$
- output weights: $\mathbf{W}^O = \left(w^o_{kj}\right)$ where $1 \le j \le N_H,\quad 1 \le k \le N_O$
- The θ is a bias incorporated into the training algorithm.

In the forward phase the hidden layer weight matrix $\mathbf{W}^H$ is multiplied by the input vector $\mathbf{X} = (x_1,\ x_2,\ x_3, \cdots\cdots,\ x_{N_I})^T$, to calculate the hidden layer output

$$y^h_j = f\left(\sum_{i=1}^{N_I} x_i * w^h_{ji}\ -\ \theta\right), \quad \text{where } 1 \le j \le N_H \tag{D.2.1}$$

The function $f(.)$ is a nonlinear activation function. Generally the sigmoid function

$$f(x) = \frac{1}{1 + e^{-x}} \tag{D.2.2}$$

is used. It compresses the output value to lie between 0 and 1. Moreover, the function is differentiable, which is a demand of training algorithm - see Equations D.2.1 and D.2.3.

The output from the hidden layer, y_j^h, is used to calculate the output of the ANN, y_k^o.

$$y_k^o = f\left(\sum_{j=1}^{N_H} y_j^h * w_{kj}^o - \theta\right), \text{where} 1 \leq k \leq N_O \tag{D.2.3}$$

The error, E_p, for a training pattern p is computed as follows

$$E_p = \frac{1}{2}\sum_{k=1}^{N_O}\left(d_{p,k} - y_{p,k}^o\right)^2 \tag{D.2.4}$$

The overall error for a training set of P patterns is

$$E = \sum_{p=1}^{P} E_p \tag{D.2.5}$$

In the expressions below the pattern index p has been omitted on all variables except for E_p to improve clarity. In the backward phase the target, d, and output, y^o, are compared and the difference is used to adopt the weights to reduce the error. The weights are adjusted according to the generalizes delta rule. This differs from the original delta rule, by the inclusion of non-linear activation function and hidden units (Jain et al., 1996). The rule is also called gradient descent, since it corresponds to performing the steepest descent in weight space where the height is equal to the error measure. It requires that the derivative of the error measure, with respect to each weight, is proportional (with negative constant) to the weight change computed by the delta rule (Jain et al., 1996).

The error in the output neurons (δ_k^o) and the error in the hidden neurons (δ_j^h) that will be used to update the weights are computed as follows:

$$\delta_k^o = y_k^o * (1 - y_k^o) * (y_k - y_k^o), \text{ where } 1 \leq k \leq N_O$$

$$\delta_j^h = y_j^h * (1 - y_j^h) * \sum_{k=1}^{N_o} \delta_k^o * W_{kj}^o \text{ , where } 1 \leq j \leq N_H$$

The main steps for back-propagation learning algorithm are given bellow.

---

1. Initialize the weight matrices, $\mathbf{W}^H$ and $\mathbf{W}^O$ to small random values. Where $\mathbf{W}^H = \left(w_{ji}^h\right)$, $1 \leq i \leq N_I$ and $1 \leq j \leq N_H$, is the weight matrix between the hidden layer and the input layer. $\mathbf{W}^O = \left(w_{kj}^o\right)$, $1 \leq k \leq N_O$, is the weight matrix between the output layer and the hidden layer.
2. Select an input pattern with corresponding output pattern from the training set and present the input pattern to the input layer of the ANN.
3. Calculate the actual outputs - forward phase.
4. According to the difference between actual and the desired outputs (error), adjust the weight matrices $\mathbf{W}^H$ and $\mathbf{W}^O$ to reduce the difference - backward phase.

5. Repeat from step 2 for all training input patterns.
6. Repeat from step 2 until the error is acceptably small.

D.3 Standard Incremental Update Back-propagation algorithm

The following outlines the standard sequential incremental update back-propagation algorithm (Sheng Ma & Farmer, 1997; Wang et al., 1989):

Start with randomly chosen weights

While $MSE = \frac{1}{2N_R} \sum_{p=1}^{N_R} \sum_{k=1}^{N_O} \left(y_{pk} - y^o_{pk}\right)^2$ is unsatisfactory and computational bounds are not exceeded, **do**

1. For each pattern $\mathbf{x}_p = (x_{p1},\ x_{p2},\ \cdots,\ x_{pN_I}),\quad \text{where}\ \ 1 \leq p \leq N_R :$

   (a) Compute the output of neurons in the hidden layer:
   $$net^h_{pj} = \sum_{i=1}^{N_I} w^h_{ji} x_{pi} + b^h_j, \quad \text{where}\ \ 1 \leq j \leq N_H.$$
   $$y^h_{pj} = f\left(net^h_{pj}\right)$$

   (b) Compute the output of neurons in the output layer:
   $$net^o_{pk} = \sum_{j=1}^{N_H} w^o_{kj} i_{pj} + b^o_k, \quad \text{where}\ \ 1 \leq k \leq N_O.$$
   $$y^o_{pk} = f\left(net^o_{pk}\right)$$

   (c) Compute error terms for the neurons in the output layer:
   $\delta^o_{pk} = \left(y_{pk} - y^o_{pk}\right)\ y^o_{pk}\ \left(1 - y^o_{pk}\right), \quad \text{where}\ \ 1 \leq k \leq N_O.$

   (d) Compute error terms for the neurons in the hidden layer:
   $$\delta^h_{pj} = y^h_{pj}\left(1 - y^h_{pj}\right) \sum_{k=1}^{N_O} \delta^o_{pk} w_{kj}, \quad \text{where}\ \ 1 \leq j \leq N_H.$$

   (e) Update both weight and bias changes in the output layer:
   $w^o_{kj} = w^o_{kj} + \eta \delta^o_{pk} y^h_{pj}$
   $b^o_k = b^o_k + \eta \delta^o_{pk}, \quad \text{where}\ \ 1 \leq k \leq N_O\ \ \text{and}\ \ 1 \leq j \leq N_H.$

   (f) Update both weight and bias changes in the hidden layer:
   $w^h_{ji} = w^h_{ji} + \eta \delta^h_{pj} x_{pi}$
   $b^h_j = b^h_j + \eta \delta^h_{pj}, \quad \text{where}\ \ 1 \leq j \leq N_H\ \ \text{and}\ \ 1 \leq i \leq N_I.$

End-while

D.4 Standard Parallel Batch Update Back-propagation Algorithm

To implement a batch update back-propagation scheme in parallel, the whole ANN is replicated among N_P processors and each processor carries out the learning process using $\left(\frac{N_R}{N_P}\right)$ patterns, where N_R is the size of the training set. Weight changes and bias changes are performed

in parallel and then the corresponding accumulated weight changes vectors, wc^h and wc^o, and the corresponding accumulated bias changes vectors, bc^h and bc^o, are exchanged between processors.

The following algorithm describes the steps of a batch update back-propagation parallel learning algorithm under the *'learn by epoch'* (Crespo et al., 1999).

Start with randomly chosen weights.

While $MSE = \frac{1}{2N_R}\sum_{p=1}^{N_R}\sum_{k=1}^{N_O}(y_{pk} - y^o_{pk})^2$ is unsatisfactory and computational bounds are not exceeded, **do**

1. For each pattern $\mathbf{x}_p = (x_{p1},\ x_{p2},\ \cdots,\ x_{pN_I}),$ where $1 \le p \le N_R$:
   (a) Compute the output of neurons in the hidden layer:
   $$net^h_{pj} = \sum_{i=1}^{N_I} w^h_{ji}x_{pi} + b^h_j, \quad y^h_{pj} = f\left(net^h_{pj}\right), \text{ where } 1 \le j \le N_H$$
   (b) Compute the output of neurons in the output layer:
   $$net^o_{pk} = \sum_{j=1}^{N_H} w^o_{kj}y^h_{pj} + b^o_k, \quad y^o_{pk} = f\left(net^o_{pk}\right), \text{ where } 1 \le k \le N_O$$
   (c) Compute error terms for the neurons in the output layer:
   $\delta^o_{pk} = \left(y_{pk} - y^o_{pk}\right)\ y^o_{pk}\ \left(1 - y^o_{pk}\right),$ where $1 \le k \le N_O.$
   (d) Compute error terms for the neurons in the hidden layer:
   $$\delta^h_{pj} = y^h_{pj}\left(1 - y^h_{pj}\right)\sum_{k=1}^{N_O}\delta^o_{pk}w_{kj}, \text{ where } 1 \le j \le N_H.$$
   (e) Compute both weight and bias changes in the output layer:
   $wc^o_{kj} = wc^o_{kj} + \eta\delta^o_{pk}y^h_{pj}$
   $bc^o_k = bc^o_k + \eta\delta^o_{pk}$
   (f) Compute both weight and bias changes in the hidden layer:
   $wc^h_{ji} = wc^h_{ji} + \eta\delta^h_{pj}x_{pi}$
   $bc^h_j = bc^h_j + \eta\delta^h_{pj}$

2. Broadcast local matrices for weight changes and bias changes, $\mathbf{wc}^h$, $\mathbf{wc}^o$, $\mathbf{bc}^h$ and $\mathbf{bc}^o$ and receive matrices for weight changes and bias changes, $\mathbf{wc}^h$, $\mathbf{wc}^o$, $\mathbf{bc}^h$ and $\mathbf{bc}^o$, from other processors.

3. Update weight changes in the output layer:
   $w^o_{kj} = w^o_{kj} + (wc^o_{kj}(local) + wc^o_{kj}(remote))$

4. Update weight changes in the hidden layer:
   $w^h_{ji} = w^h_{ji} + (wc^h_{ji}(local) + wc^h_{ji}(remote))$

5. Update bias changes in the output layer:
   $b^o_k = b^o_k + (bc^o_k(local) + bc^o_k(remote))$

6. Update bias changes in the hidden layer:
   $b^h_j = b^h_j + (bc^h_j(local) + bc^h_j(remote))$

End-while

The subindexes p, i, j and k identify the pth input pattern, i^{th} input neuron, j^{th} hidden neuron and k^{th} output neuron, respectively. w_{ij} is the weight corresponding to the connection between

neuron j and neuron i, wc_{ij} and bc_j are the accumulated change for the weight corresponding to the connection between neuron j and neuron i and the accumulated changes for the bias in the jth neuron respectively, **b** is the bias and N_I, N_H, and N_O identify the number of input, hidden, and output neurons, respectively. $f(x)$ is a nonlinear function that is often chosen to be of a sigmoidal form. $f(x) = \frac{1}{1+e^{-x}}$ is used in this algorithm.

Appendix E

Running Times for the Shallow Water Model

	MPI		TCP-Linda	
Nodes	wot	wt	wot	wt
12	9671.32	11032.31	2745.98	3319.65
	9705.12	11039.67	2769.08	3269.21
	9670.75	11065.97	2755.89	3275.39
	9701.62	11053.03	2721.77	3305.05
	9683.87	11076.73	2725.96	3319.58
Average	**9686.54**	**11053.54**	**2743.74**	**3297.78**
Median	**9683.87**	**11053.03**	**2745.98**	**3305.05**

Table E.1: Run time in seconds (wot - without thread; wt - with thread) were obtained from the Monolith high performance cluster

	MPI		TCP-Linda	
Nodes	wot	wt	wot	wt
48	11908.03	10261.39	3163.12	2861.09
	11939.37	10301.11	3189.47	2823.27
	11907.21	10289.21	3201.82	2844.71
	11955.39	10247.13	3156.71	2836.49
	11961.89	10269.81	3206.02	2861.16
Average	**11934.38**	**10273.73**	**3183.43**	**2845.34**
Median	**11939.37**	**10269.81**	**3189.47**	**2844.71**

Table E.2: Run time in seconds (wot - without thread; wt - with thread) were obtained from the Monolith high performance cluster

	MPI	UPC	Hybrid UPC+MPI	
			Configuration-1	Configuration-2
	9398.01	7780.05	7921.88	8047.77
	9391.82	7813.23	7915.46	8062.61
	9429.57	7787.63	7926.19	8081.27
	9440.78	7805.81	7914.45	8091.43
	9416.69	7801.71	7897.91	8059.82
Average	**9415.37**	**7797.69**	**7915.18**	**8068.58**
Median	**9416.69**	**7801.71**	**7915.46**	**8062.61**

Table E.3: Run time in seconds were obtained from the Upplanka high performance cluster