A Java Implemented Design-Pattern-Based System for Parallel Programming

By

Narjit Chadha

A Thesis

Submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba

October 2002

©2002 by Narjit Chadha

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79941-7

Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

A JAVA IMPLEMENTED DESIGN-PATTERN-BASED SYSTEM FOR PARALLEL PROGRAMMING

BY

NARJIT CHADHA

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

Master of Science

NARJIT CHADHA © 2002

Permission has been granted to the Library of The University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilm Inc. to publish an abstract of this thesis/practicum.

The author reserves other publication rights, and neither this thesis/practicum nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

Abstract

Parallel computing is slowly becoming more mainstream as the demand for computational power grows. Recently the focus of parallel computing has shifted away from expensive multiprocessor supercomputers to distributed clusters of commodity workstations. Widely accepted programming standards have been developed such as PVM and MPI. In addition, other tools have emerged that raise the level of abstraction of parallel programming and simplify repetitive, error prone tasks. This thesis explores existing parallel programming systems and presents a pattern based tool, MPI Buddy, that aims to decrease parallel program development time and reduce the number of errors due to parallelization.

MPI Buddy is designed as a design-pattern based, layered open system with a level of abstraction above MPI. It is constructed using Java and possesses a modular design allowing new design pattern modules to be added with ease. The intent is to allow MPI Buddy to have a user friendly interface, openness, moderate extensibility, and portability. In addition, the tool is intended to generate optimal communication code and be able to test code syntax from within. The design patterns incorporated were chosen from the most commonly used parallel communication and decomposition schemes. The uniqueness of this tool is its portability across different computer platforms, allowing the user to program parallel MPI applications on a PC, Apple, or any other platform which supports Java. Additionally, an installed version of MPI on the computing platform is necessary to compile the developed code from within MPI Buddy or run the developed applications.

The applications developed using MPI Buddy performed as well as the hand coded versions, but the less time was required in writing parallel programs with the tool. The benefits were more pronounced for smaller applications that use complex parallel communication. This tool produces error free MPI code and is also useful for educating novice programmers on parallel techniques and structures. It was inferred that data-parallel applications can be quickly prototyped in the field of signal and image processing using MPI Buddy.

Acknowledgments

I would like to thank my advisor, Dr. Aysegul Cuhadar, for accepting me as a Masters student at the University of Manitoba. Your commitment to see this project through to its completion despite the geographical distances that separated us was incredible. I cannot thank my co-advisor, Dr. Howard Card, enough for his efforts and support throughout my Masters project. He guided me throughout this project and supported me in keeping my goals in perspective. I want to thank Dr. Parimala Thulasiraman for her parallel computing advice during this project. Also, I want to express my appreciation to Dr. Bob McLeod for stepping in as a local advisor at the University of Manitoba when I was feeling disillusioned.

I want to thank Shawn Silverman for answering many questions that I had regarding the Java programming language and its hidden capabilities. Having you around made the task of programming the API go smoothly. Last, but not least, I want to thank my family and friends for their astounding support and advice over the last two years.

Contents

Abstract ii

Acknowledgments. iii

Contents iv

List of Figures ix

List of Tables xi

List of Equations xii

Chapter 1 Introduction 1

1.1 Parallel Computers 2

1.2 Parallel Programming 2

1.3 Design Patterns 3

1.4 Motivation and Objectives 3

1.5 Structure of the Thesis 4

Chapter 2 Parallel Computing Overview 5

2.1 Introduction 5

2.2 Requirements for Parallelism 5

2.2.1 Hardware Level 5

2.2.2 Operating System Level 6

2.2.3 Software Level 6

2.2.4 Techniques used to Exploit Parallelism 7

2.3 Parallel Computers 8

2.3.1 Classification of Parallel Computers 8

2.4 Types of Parallel Machine Architectures 9

2.4.1 Vector Processors 9

2.4.2 Dataflow Architectures 10

2.4.3 Systolic Architectures 11

2.4.4 Array Processors 11
2.4.5 Shared Memory MIMD 12
2.4.6 Distributed Memory MIMD (Message Passing Computers) 13
2.5 Challenges in Parallel Programming 14
2.5.1 Portability of Applications 14
2.5.2 Compatibility with Existing Computer Architectures 14
2.5.3 Expressiveness of Parallelism 15
2.5.4 Ease of Programming 15
2.6 Solutions to Parallel Programming Complexity 15
2.6.1 Raising the Level of Abstraction 15
2.6.2 Providing Tools to Simplify Repetitive Tasks 15
2.6.3 Design Patterns as a Unifying Idea 16
2.7 Design Pattern Advantages 17
2.7.1 Correctness 17
2.7.2 Maintainability and Reusability 17
2.7.3 Ease of Use 17
2.8 Limitations of Design Pattern Approaches 17
2.8.1 Efficiency 17
2.8.2 Flexibility 18
2.9 Summary 18

Chapter 3 Parallel Programming Systems 19
3.1 Overview 19
3.2 Attempts to Raise the Level of Abstraction 19
3.2.1 Message Passing Libraries (MPLs) and Remote Procedure Calls (RPCs) 19
3.2.2 Abstractions on top of MPLs and RPCs 20
3.2.3 Other High Level Programming Approaches 20
3.3 Classification of Tools for Parallel Programming by Functionality . . . 21
3.3.1 Basic Systems 22

3.3.2 Tool Kits . . . .
3.3.3 Integrated Development Environments (IDEs) 23
3.4 Two Distributed Programming Standards - PVM and MPI 23
3.4.1 PVM 23
3.4.2 MPI 25
3.5 Existing Design Pattern Based Systems 28
3.5.1 CODE 28
3.5.2 HeNCE 30
3.5.3 Tracs 32
3.5.4 Enterprise 33
3.5.5 DPnDP 35
3.6 Proposed Enhancements 37
3.7 Summary 38

Chapter 4 Design and Implementation of a Parallel Programming System (MPI Buddy) . 39
4.1 Introduction 39
4.2 Functionality Desired 39
4.3 The Java Programming Language 41
4.4 System Design Layout 42
4.4.1 Main Executable 43
4.4.2 Compilation 44
4.4.3 Help Modules 45
4.4.4 Printing 45
4.4.5 Design Patterns 45
4.5 Design Patterns Included 46
4.5.1 1D Scatter/Gather 46
4.5.2 Balanced 1D Send/Receive 47
4.5.3 2D Scatter/Gather 48
4.5.4 Block Cyclic Send/Receive 49
4.5.5 Cyclic Send/Receive 50

4.5.6 Dynamic 1D Master/Slave ... 51
4.5.7 1D Divide and Conquer ... 52
4.6 Programming Model ... 52
4.7 Summary ... 54

Chapter 5 Programming Experiments and Analysis ... 56
5.1 Introduction ... 56
5.2 Metrics used to Evaluate Performance ... 56
5.2.1 Objective Metrics ... 56
5.2.2 Subjective Metrics ... 57
5.3 Computing Platform Used in this Work ... 58
5.4 2D Discrete Wavelet Transform ... 59
5.4.1 Introduction ... 59
5.4.2 Analysis of the Problem ... 60
5.4.3 Parallel Decomposition Strategy ... 61
5.4.4 Approach to Solving Problem using MPI Buddy ... 63
5.4.5 Objective Analysis of the Tool ... 64
5.4.6 Subjective Analysis of the Tool ... 66
5.5 Fast Fourier Transform ... 67
5.5.1 Introduction ... 67
5.5.2 Analysis of the Problem ... 67
5.5.3 Parallel Decomposition Strategy ... 69
5.5.4 Approach to Solving Problem using MPI Buddy ... 71
5.5.5 Objective Analysis of the Tool ... 72
5.5.6 Subjective Analysis of the Tool ... 74
5.6 Overall Analysis of the Tool ... 74

Chapter 6 Conclusions and Future Work ... 76
6.1 Review of this Work ... 76
6.2 Future Work ... 78

6.2.2 Better GUI 78
6.2.3 Automatically Color Code MPI and C Keywords 78
6.2.4 Integrate a Performance Visualization Tool 78
6.2.5 Add Support for Other MPI Communication 79
6.3 Conclusion 79

References 80

Appendix A Software Listing for 2D Discrete Wavelet Transform 85
A1: Software Listing for Sequential 2D DWT Program 86
A2: Software Listing for Parallel Hand Coded 2D DWT Program 99
A3: Software Listing for MPI Buddy Coded 2D DWT Program 110

Appendix B Software Listing for 1D Fast Fourier Transform 114
B1: Software Listing for Sequential 1D FFT Program 115
B2: Software Listing for Parallel Hand Coded 1D FFT Program 119
B3: Software Listing for MPI Buddy Coded 1D FFT Program 123

List of Figures

Figure 2.1 Flynn's Taxonomy ... 9
Figure 2.2. Register-memory vector computer ... 10
Figure 2.3 Array processor layout ... 12
Figure 2.4 UMA (a) and NUMA (b) shared memory MIMD machine architectures ... 13
Figure 2.5 Structure of the Intel Paragon ... 13
Figure 2.6 Beowulf layout ... 14
Figure 2.7 Relationship between an architectural skeleton, a virtual machine, and the final program code. ... 16

Figure 3.1 Tradeoff between abstraction and flexibility ... 21
Figure 3.2 Classification of parallel programming systems by functionality ... 23
Figure 3.3 Message passing between workstations using PVM ... 24
Figure 3.4 PVM process spawning ... 25
Figure 3.5 MPI execution example (2 process system ... 27
Figure 3.6 A screen shot of CODE (version 2.2) [Berg02] ... 29
Figure 3.7 Screen Shot of HeNCE ... 31
Figure 3.8 Enterprise Screen Shot ... 34
Figure 3.9 Enterprise Assets (Design Patterns) ... 34
Figure 3.10 Structure of a DPnDP application ... 36

Figure 4.1 A layered open system ... 40
Figure 4.2 Layout of the MPI Buddy System ... 43
Figure 4.3 Screen shot of MPI Buddy (main window) ... 44
Figure 4.4 Compile GUI ... 44
Figure 4.5 1D Scatter/Gather ... 47
Figure 4.6 Balanced 1D Send/Receive ... 48
Figure 4.7 2D Scatter/Gather ... 48
Figure 4.8 Block Cyclic Send/Receive ... 50

Figure 4.9 Cyclic Send/Receive 51
Figure 4.10 Dynamic 1D Master/Slave 51
Figure 4.11 1D Divide and Conquer 52
Figure 4.12 MPI Buddy program layout 53
Figure 4.13 Programming approach 54

Figure 5.1 Platform Used 59
Figure 5.2 One Stage of 2D Discrete Wavelet Transform 60
Figure 5.3 Communication approach (3 processors) 61
Figure 5.4 Selecting 2D Scatter/Gather design pattern parameters 63
Figure 5.5 Adding user code to the 2D DWT program 64
Figure 5.6 DWT execution time versus machine size (D=6) 65
Figure 5.7 DWT speedup versus machine size (D=6) 66
Figure 5.8 Iterative fast Fourier transform (FFT) 69
Figure 5.9 Parallel FFT algorithm (4 processors, 8 data elements) 70
Figure 5.10 Selecting 1D Scatter/Gather design pattern parameters 72
Figure 5.11 Execution time versus machine size for parallel FFT 73
Figure 5.12 Speedup versus machine size for parallel FFT 73

List of Tables

Table 5.1: Timings (in seconds) for 2D discrete wavelet transform program (512x512 image, filter size 8) 66

Table 5.2 : Timings (in seconds) for FFT applications (N=20, data size 2^{20}) 73

List of Equations

(5.1) Speedup 57

(5.2) Efficiency 57

(5.3) Cost 57

(5.4) K-level wavelet discrete wavelet transform communication time 62

(5.5) Discrete wavelet transform computation time 62

(5.6) Discrete wavelet transform speedup 62

(5.7) Discrete Fourier transform equation 67

(5.8) Subdividing discrete Fourier transform 68

(5.9) Odd/even point divisions of discrete Fourier transform 68

(5.10) First N/2 point computation of discrete Fourier transform sequence 68

(5.11) Second N/2 point computation of discrete Fourier transform sequence 68

(5.12) Communication time for parallel fast Fourier transform 71

(5.13) Root processor computation time for fast Fourier transform 71

(5.14) Overall parallel execution time for fast Fourier transform implementation 71

Chapter 1

Introduction

Scientists and engineers require elevated computational power to run demanding applications involving weather prediction, simulation and modeling, DNA mapping, nuclear physics, astronomy, code breaking, image processing, and computer animation to name a few. One solution to overcome this limitation is to improve the operating speed of processors and other components so that they can offer the computational power demanded by certain applications. This solution is currently viable, but future improvements are likely to be constrained by the speed of light, thermodynamic laws, and the high costs of processor fabrication [Buyy99]. Another feasible alternative is to use multiple processors together, coordinating their computation. These are known as parallel computers and have evolved since the 1950s. The movies Titanic and Shrek both used parallel computers in rendering complex moving images [Comp02, SGI01].

The idea behind parallel computing is that if one processor can provide k units of computational power, then n processors should be able to provide n.k units of computational power. If these processors can work on a problem simultaneously, the parallel case should only require $1/n$ th the time of the single processor situation [WiAl99]. Of course, all problems cannot regularly be divided in such an optimal manner in practice, but significant execution time improvements can still be achieved.

A major barrier to the widespread adoption of parallel computing is that writing efficient and portable parallel programs is difficult because parallel programs must express both the sequential computations and the interactions among the sequential computations which define the parallelism. There is a need for tools that allow the programmer to bridge the complexity gap between sequential

and parallel programming without extensive retraining. In this thesis, a parallel programming environment is presented that can assist developers in writing parallel software. The system customizes and duplicates common parallel programming patterns which can be inserted into a parallel program.

1.1 Parallel Computers

Traditionally, most people have associated parallel computers with expensive multiprocessor machines such as the Thinking Machines CM-5 or the Cray MTA. These machines are powerful and strive to fulfill resource requirements lacking in a common personal desktop such as CPU and memory. Multiprocessor supercomputers have not proliferated extensively due to their prohibitive costs, large sizes, high power consumption, and the difficulty in interfacing common peripherals with them [Webb94]. In addition, these machines quickly lose the status of a "supercomputer" as the performance of available processors typically increases 50% annually [Zava99].

Recently, the focus of parallel computing has moved away from individual multiprocessor machines to distributed clusters of machines. It was found that parallel machines can be built economically by using commodity workstations interconnected by a fast interconnection network such as ethernet or gigabit ethernet. These virtual "supercomputers" have been found to produce execution speedups approaching those of fast multiprocessor machines, but have further advantages in that the cost of these workstations computers is low, the latest processors can be incorporated into the systems as they become available, and peripherals interfacing is easy. Also, the Unix and Linux operating systems allow for easy high level communication tool development.

1.2 Parallel Programming

The growth of cluster based parallel computing environments has spawned the development of various parallel programming tools. These tools employ macros, functions, abstract data types, and objects to allow the user to deal with the complexity of parallel programming. Two standards, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI), have been developed and accepted for programming the socket level communication requirements between distributed

processors. These tools consist of message passing libraries and remote procedure calls that raise the level of abstraction for the programmer. However, understanding how to build complex parallel programs can still be quite challenging for the novice parallel programmer. Indeed, software developers often fear that the time saved by executing parallel versus serial applications may not justify the time involved in developing, debugging, and testing these parallel programs. Today, building software tools to aid in parallel application development is an important research topic in the field of parallel computing.

1.3 Design Patterns

A pattern is a recurring solution to a standard problem [Schm95]. Programming design patterns are modeled on successful past programming experiences. These patterns, once modeled, can then be reused in other programs. Typically, hand coding a program from scratch results in better execution time performance, but may consume immense amounts of time that cannot be tolerated.

Design patterns for parallel programming provide a mechanism to address commonly occurring data decomposition and communication structures. Such structures include master/slave, workpool, and divide and conquer. These few structures exist in most parallel programs and the complexity of these structures can be masked through the use of parallel design patterns. The term design pattern in this thesis refers to a parallel design pattern.

1.4 Motivation and Objectives

Parallel design pattern based systems must have the mechanisms to cover most of the commonly found parallel communication structures, but must also be flexible enough to give the user flexibility to work on less common problems. In addition, most parallel programming systems are limited to certain architectures and operating systems.

The objective of this thesis is to demonstrate the creation of an open platform independent design-pattern based system for distributed parallel programming. The system design criteria

includes the generation of code that can be used over a wide range of cluster architectures, along with a good degree of performance portability. A secondary objective will be to assess the ease of use of the tool and the efficiency of the code developed using this system against hand developed code.

1.5 Structure of the Thesis

This chapter provides an introduction to parallel computing and parallel programming along with the motivation and objectives of this thesis. Chapter 2 undertakes to describe parallel computing in more detail and describes the methods of conquering the programming complexity which includes the unifying idea of utilizing design patterns. Chapter 3 describes classifications of parallel programming systems and then provides examples of existing systems along with observed limitations and proposed enhancements. In chapter 4, the layout of an open platform independent parallel programming system, MPI Buddy, is described including the design patterns incorporated into the system. Chapter 5 illustrates the use of the MPI Buddy system in programming various parallel applications and makes attempts to qualitatively and quantitatively assess the value of the tool developed. Finally, chapter 6 draws conclusions from this research and points out future directions for continuing research.

Chapter 2

Parallel Computing Overview

2.1 Introduction

Before one can properly grasp the problems with parallel programming, an overview of parallel computing appears helpful This chapter provides an overview of parallel computing. Requirements for parallelism are discussed before an overview of parallel computers and existing parallel architectures is given. The challenges inherent with parallel programming and the solutions to these challenges are described, including the use of design patterns. Finally, the advantages and disadvantages of design pattern use are inferred.

2.2 Requirements for Parallelism

Parallelism is not inherent in any computer system by default. There should be support available at the hardware, operating system, and software (application) level. If one of these levels does not provide support for parallelism, then parallel program development may not be possible.

2.2.1 Hardware Level

The system hardware should support parallelism at the instruction level for efficient fine grain parallelism. This requires that the system memory, the system buses, and CPU must all be capable of supporting activities in parallel. Multiprocessor workstations are examples of systems in which the hardware supports instruction-level parallel activities. Workstation clusters do not support parallelism at the instruction level, but use program parallelism intended for coarse grain problem decompositions.

2.2.2 Operating System Level

The operating system manages the allocation of resources during the execution of user programs [Thul01]. The operating system is also involved with processor scheduling, memory maps, and interprocessor communication.

In order to run processes in parallel, there needs to be a mechanism to handle process startup, termination, and allocation. Another desirable feature is process migration. Communication and synchronization among processors is important for the sharing of information between the processors [Siu96].

Some workstations clusters use different operating systems over different types of processors. Heterogeneity is a concept that allows as many workstations to cooperate as possible, without regard to their underlying architectural differences. This improves the utility of the cluster. However, heterogeneity requires data type and protocol translation, which devours computer resources as data type and protocol translation is required across the processors of the cluster.

Finally, operating systems provide essential security measures for the system. These include file ownership and permission properties (i.e. Unix). Administration is another property which many operating systems provide.

2.2.3 Software Level

The complexity of handling parallel program development falls to parallel program development tools including parallel programming systems, parallel debuggers, and compilers.

Developers are required to understand the complex patterns of interactions between all sequential processes and each process in isolation. This has resulted in research into now parallel programming models and systems to make the job of parallel programming easier. Examples of parallel programming systems include CODE [BHDM95], Hence [BDGM94], and Enterprise [SSLP93].

Parallel program debuggers can let the user trace run-time activities and locate programming mistakes. The debuggers available mostly provide event interaction related information at a lower levels, so users may have difficulty comprehending the results. Other debuggers are used to evaluate the execution performance of parallel programs. These include tools such as ParaGraph [HeFi97], ATEMPT [Kran96, VGKS95], ULTRA [CoGG00], and PS [AMMV98].

Compilers are necessary to allow the programmer to utilize low level features in the operating system and hardware which can exploit parallelism. Compilers translate source code into object code. Additionally, compilers assign variables to registers and memory and reserve functional units for operators. Following compilation, an assembler translates the compiled object code into machine code so that it can be recognized by the machine hardware. There are many parallelizing compilers available today which can automatically detect parallelism in sequential source code and others which have been specifically developed for parallel code (i.e. MPI compilers)

Other tools available such as Globus allow parallel applications to be run across workstation clusters on different local area networks (LANs). Globus is a toolkit that provides the basic infrastructure for communication, authentication, network information, and data access [Glob00]. It has support for parallel programming standards such as MPI, and also takes care of the resource management across different clusters containing different machine architectures.

2.2.4 Techniques used to Exploit Parallelism

Parallelism can be exploited at algorithm design time, programming time, compile time and runtime. If the basic infrastructure for parallelism is available, there must be a way for the user to program parallel applications which will have desired behaviors. One technique is to directly program socket streams or other interprocessor communication. This technique results in the highest speedups in the parallelized versions, but comes with a high time cost of programming the application. Other techniques include the use of parallelizing compilers on sequential code, and programming using higher level tools.

2.3 Parallel Computers

Parallel computers have been considered as early as 1955. The first “parallel” computer built is a disputed item among scholars. Likely candidates include the IBM STRETCH and Livermore Automatic Research Computer (LARC), both of which were conceived in 1956 and were produced by 1959 [Wils94]. In 1962, Burroughs introduced the D825, a symmetrical multiple-instruction multiple-data multiprocessor (MIMD) with 1-4 CPUs and 1-16 memory modules. The vast majority of earlier parallel computers were single machines with a shared memory and multiple processors. Starting in the mid 1970s, work started being done on developing distributed memory computers in which message passing was required to gain access to all memory elements. Since then, there have been two recognized tracks of parallel computer development : the shared memory track and the message passing track..

2.3.1 Classification of Parallel Computers

Flynn has organized computers into a taxonomy based upon their functionality [Dunc90] as shown in Figure 2.1. The divisions he made are :

- **Single Instruction over Single Data Stream (SISD)** : These are representative of sequential computers.

- **Multiple Instruction Single Data (MISD)** : The same data stream flows though a linear array of processors, which execute different instructions. These are also known as systolic arrays.

- **Single Instruction over Multiple Data Streams (SIMD)** : These machines apply a single instruction or set of instructions to multiple data streams. Instructions from a program are broadcast to many processors. Each processor executes the same instruction in synchronism, but uses different data.

- **Multiple Instruction Multiple Data (MIMD)** : Each processor has its own instruction(s) to execute on its own set of data. Most parallel computers are of this type.

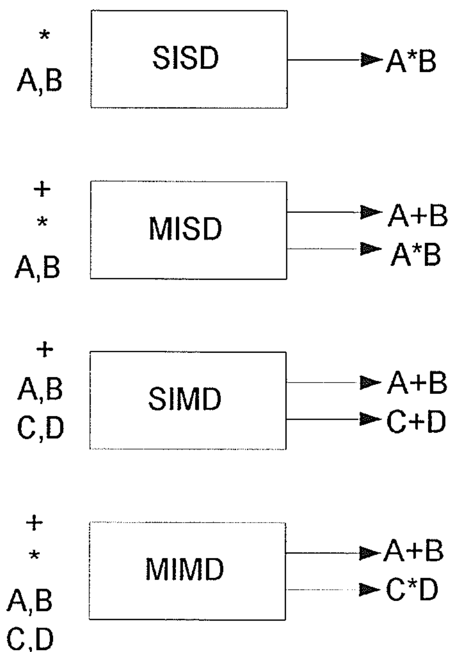


Figure 2.1 Flynn's Taxonomy

2.4 Types of Parallel Machine Architectures

2.4.1 Vector Processors

Vector processors are representative of most of the earlier supercomputers. These machines execute single instructions on sequences of data (i.e. vectors or pipelines) instead of on single items; they are examples of SIMD machines. Using vector instructions results in more efficient memory access than single instructions as a large amount of work can be done on the input vector before a new memory access is required . Another advantage of these architectures is that they can be optimized to solve problems while removing data hazards. The first vector computer was the CDC Star-100, introduced in 1972. This machine could execute instructions by taking two input vectors from memory, compute the result vector, and write it directly to the memory [HiTa72].

In 1976, Seymour Cray founded Cray Research and introduced the Cray-1 [Patt02]. The

Cray-1 was the first vector computer to have fast scalar and vector performance. The Cray-1 abandoned the memory to memory approach of the Star 100 and instead introduced a register memory architecture. The Cray-1 performed almost everything fast for its generation and became the first commercially successful vector supercomputer. Fig 2.2 shows the architectural layout of the a register-memory vector computer. Vector computers continue to hold a niche in the supercomputing industry and include such recent models as the Cray SV1, Cray SV2, Alex Informatics AVX3, Connection Machines CM-5, Intel iWarp, and many others.

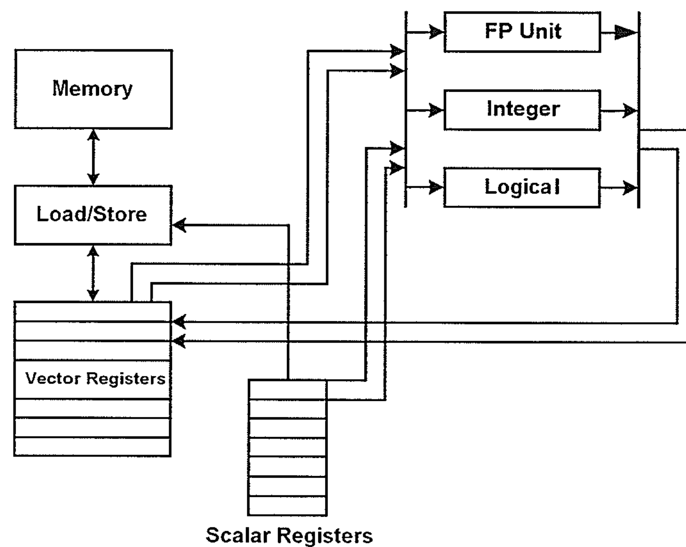


Figure 2.2. Register-memory vector computer

2.4.2 Dataflow Architectures

Duane Adams of Stanford University defined the term "dataflow" while describing graphical models of computation for his PhD thesis in 1968. In 1974, Jack Dennis and David Misunas at MIT published the first description of a dataflow computer. In 1977, Al Davis and Burroughs together built the DDM1, the first operational dataflow computer.

Dataflow computer architectures are intended to allow for data driven computation. This form of computation differs considerably from the von Neumann machine model. The von Neumann model involves program driven control of machine instructions, whereas in the dataflow model, the instructions are driven by data availability. These architectures work on the assumption that

programs can be represented as directed graphs of data dependencies [ArCu86]. The availability of data activates matching instructions and computation proceeds. There are two categories of dataflow architectures: static and dynamic. Static dataflow architectures use primitive functions to represent nodes. Dynamic dataflow architectures use subgraphs to represent nodes.

2.4.3 Systolic Architectures

H. T. Kung and Charles Leiserson published the first paper describing systolic computation in 1978. The term "systolic" is used because of the analogy of these systems with the circulatory system of the human body. In the circulatory system, the heart send and receives a large amount of blood as a result of the frequent and rhythmic pumping of small amount of blood though arteries and veins [Kris89]. In systolic computer systems, the heart would correspond to the global memory as the source and destination of data. The arterial-venous network would similarly correspond to processors and communication links. Systolic architectures are extensions of the pipelining concept, except multidimensional, multidirectional flow is permitted including feedback. Data can be used, reused and both new data and partial results may move in the system. There are two categories of systolic architectures: systolic trees, and systolic mesh automata (systolic arrays). The Intel iWarp is and example of the latter [GrOh98].

2.4.4 Array Processors

These architectures are another example of the SIMD machine model developed by Flynn. In 1968, IBM delivered the first array processor (the 2938). Array processors are interconnected in a rectangular mesh or a grid arrangement. Each node has 4 directly connected neighbors, except at those nodes at the boundaries. These architectures are useful for applications in matrix processing and image processing where each node can be identified with the matrix element or a picture element (pixel) [Kris89]. The array processor has a control unit which controls the instructions within processing element in the array. The array processor also has a data level concurrent hardware module, 2D array geometry, and synchronized control. An example of an array processor layout is shown in Figure 2.3 below.

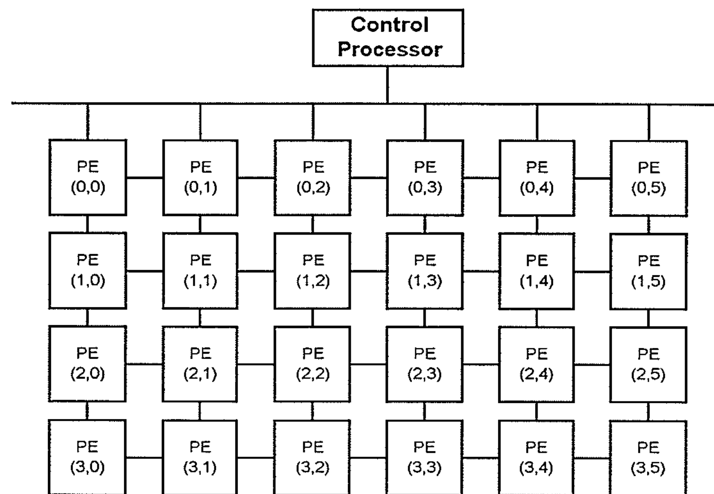


Figure 2.3 Array processor layout

2.4.5 Shared Memory MIMD

This is a fairly mature parallel computer architecture, with the first machines appearing in the early 1960s.The main feature of this class of machines is that communication and cooperation between processes may occur using normal memory access instructions. These machines are constructed with a singly addressed memory shared amongst all the processors in the machine. The processor elements may be connected to each other and the memory elements in a variety of configurations including a bus, crossbar, multistage network configuration. There are symmetric multiprocessor configurations (SMP) configurations available that allow for a uniform memory access (UMA) time by all the processors. Usually ,these systems involve bus or crossbar connections and do not scale well. Other shared memory MIMD machines exhibit non-uniform memory access (NUMA) time which means that some processors can access some memory elements faster than others. These machines are more scalable than their UMA counterparts. Examples are of each type of shared memory MIMD machine are given Figure 2.4.

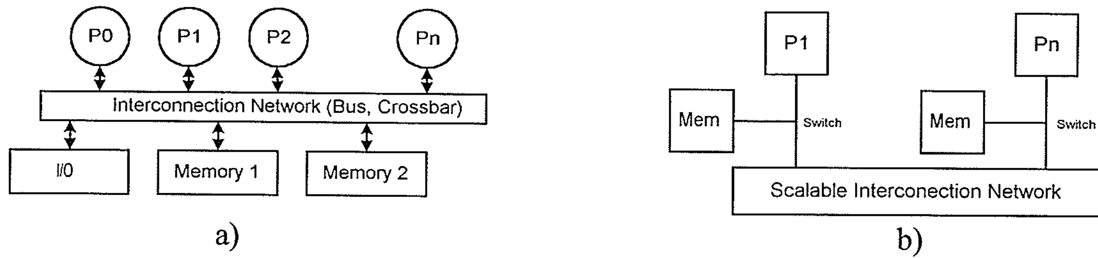


Figure 2.4 UMA (a) and NUMA (b) shared memory MIMD machine architectures.

2.4.6 Distributed Memory MIMD (Message Passing Computers)

These machines make up the message passing track of parallel computers and include single computers with more than one processor and distributed memories (multiprocessors) and multiple computers connected by a high bandwidth network (multicomputers). Examples of the former include the IBM SP-2 and the Intel Paragon. These machines have special direct memory access (DMA) mechanisms which facilitate data exchange between nodes. The structure of the Intel Paragon multiprocessor is given in Figure 2.5.

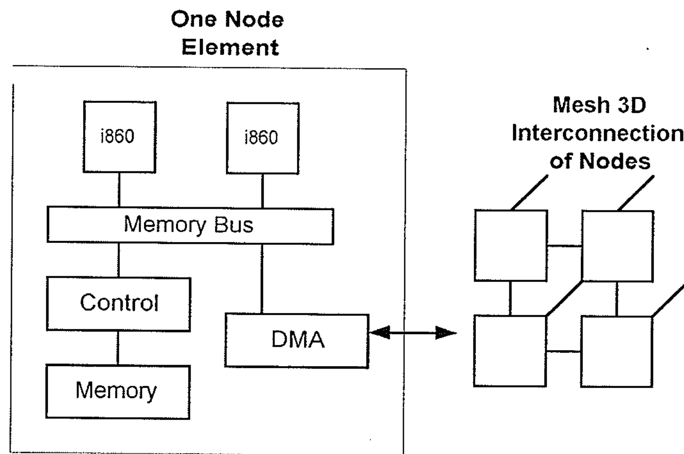


Figure 2.5 Structure of the Intel Paragon

Multicomputers (a.k.a. cluster of workstations or network of workstations) are implemented using workstations (nodes) with point to point connections. Each computer has a private local memory and communication occurs by message passing primitives through the network. In the evolution of multicomputers, the Beowulf has been created. Beowulfs are high performance platforms built entirely out of commodity off-the-shelf components. An example of a Beowulf layout is shown in

Figure 2.6 below.

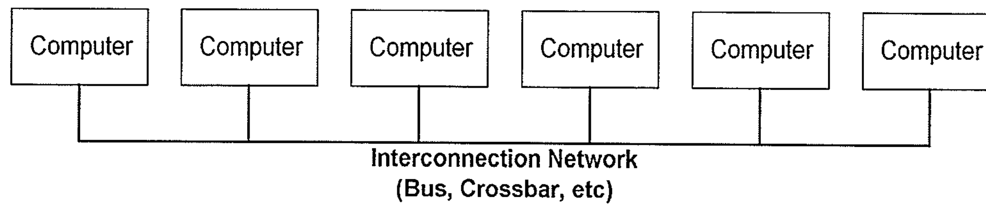


Figure 2.6 Beowulf layout

Beowulf setups are the dominant focus of parallel computing today due to their scalability and cost effectiveness.

2.5 Challenges in Parallel Programming

Parallel programming introduces many unique challenges to the developer. Human thinking is sequential so the programming of parallel applications takes some thought outside of conventional thinking. The challenges evident in parallel program development are described in this section.

2.5.1 Portability of Applications

This is the most challenging attribute to achieve since there are many different types of parallel computer architectures, each supporting different programming styles. As well, parallel code may not perform the same way on different architectures.

2.5.2 Compatibility with Existing Computer Architectures

It is important to have programming standards that can be used on existing computers. It is important to work in parallel programming environments with architecture independent languages, compilers, and software tools. This gives the developer flexibility in where he or she wishes to program and not compromise the finished parallel application.

2.5.3 Expressiveness of Parallelism

It is important for the developer to understand what is being programmed. Programming tools should exhibit the parallel features of each node and the interactions between nodes. This may be accomplished through the introduction of visual graphs or other easy to understand approaches.

2.5.4 Ease of Programming

Many parallel programming software methods present great challenges to the developer. If familiar sequential concepts are employed in a parallel programming tool, the tool is more capable of gaining wide acceptance [Simo97]. Few individuals will put more time into program development than the final application is worth.

2.6 Solutions to Parallel Programming Complexity

2.6.1 Raising the Level of Abstraction

Programmers often feel that working with low level primitives can be quite difficult, even though they are the most flexible among all parallel programming primitives. Raising the level of abstraction hides the details of parallelism from developers, while making certain parallel programming tasks easier. The goal is to allow the programmer to solve the problem in a high level model without worrying about the difficult and unnecessary low level details.

2.6.2 Providing Tools to Simplify Repetitive Tasks

A solution to programming common, complex, and error prone tasks is to provide software tools that automate the implementation of these tasks. An example of a commercial sequential software tool is the Visual Studio by Microsoft for easily programming complex graphical user interface (GUI) applications for the Windows environment. Application code is generated automatically with the user only specifying certain parameters and then filling in the specifics for the program. Other advanced tools are available such as interface builders, advanced compilers, debugger, visualization tools, profilers, and simulators to assist the developer in various phases of the software development cycle. Similar tools are available for parallel programming. Some of these

parallel program development tools will be discussed in chapter 3.

2.6.3 Design Patterns as a Unifying Idea

The idea behind a "pattern" is to describe a recurring structure, and then use this model again in other similar situations. "Design patterns" are used in everyday life, from fax cover sheets to word processor style sheets. In each of these cases, there is a template specified containing the same fields, and the user only needs to fill the missing information into the fields provided by these templates [GHJV94]. Expert designers do not feel the need to "reinvent the wheel", but rather prefer to reuse solutions that have worked well for them in the past.

For parallel programming, there are computation and communication structures that do not appear in sequential programming. In generating a parallel program through the use of design patterns, developers instantiate a design pattern to obtain communication skeletons into which they can insert their own application specific code. An architectural skeleton is a basic communication pattern devoid of any user code. Upon the insertion of code by the user, a virtual machine is obtained. A virtual machine is an application-specific specialization of a skeleton [GoSP99]. By filling a virtual machine with complete application code, the final program code is achieved. Figure 2.7 Illustrates this approach.

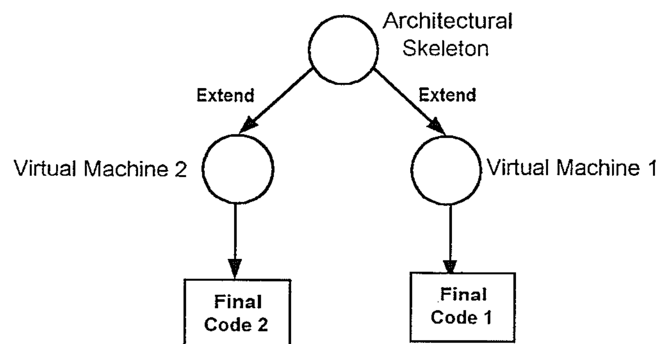


Figure 2.7 Relationship between an architectural skeleton, a virtual machine, and the final program code.

2.7 Design Pattern Advantages

Design patterns have been found to have the properties of correctness, maintainability and reusability, and efficiency which have made them favorable to use by programmers.

2.7.1 Correctness

Communication and synchronization can be very complex and error prone. Furthermore, the code developed can be difficult to debug. Using design patterns, the programmer can use previously developed structures which have been tested repeatedly for correctness. This saves time involved for development, debugging, and testing. The developer can then concentrate on the actual specific algorithm for the problem he or she is developing and not worry about specific communication implementation details.

2.7.2 Maintainability and Reusability

Design patterns are able to reproduce frequently used communication and synchronization structures of parallel programs. Also, design patterns separate computation, communication, and processor binding specifications of parallel programs, so that each one can be modified independently (called separation of the specifics). This promotes usability and makes programs easier to maintain. In addition, the programmer is better able to understand the nature of each of the parts of the parallel program better.

2.7.3 Ease of Use

Design patterns allow developers to approach complex problems at a higher level of abstraction. The parallel part of a program is what flusters sequential programmers. By allowing the design patterns to take care of the parallel structures found in the program, the programmer can concentrate on the sequential components of the program.

2.8 Limitations of Design Pattern Approaches

2.8.1 Efficiency

Programs developed in a high level design pattern model are generally less efficient than

those developed using low level primitives. Efficiency refers both to the execution time and amount of unnecessary code generated for each of the development styles. Using design patterns, there may be excess code generated to ensure correctness over a broad range of platforms, and a slower execution time when compared to the low level primitive approach.

2.8.2 Flexibility

Raising the level of abstraction can lower the flexibility. Most design-pattern-based systems provide a limited number of patterns. A design pattern system is of no added use to the developer if the communication or data decomposition structures are not available in the system. Most generated structures cannot be easily modified. Thus, developers may feel trapped in the high level model.

2.9 Summary

This chapter has provided an overview of parallel computing today. Support for parallelism must exist at the hardware level, operating system level, and software level to even contemplate parallel application development. Providing this support does exist, parallel programming itself presents challenges with respect to the portability of applications, compatibility with existing computer architectures, the expressiveness of parallelism ,and the ease of programming. Two solutions to parallel programming are rasing the level of abstraction while programming and providing tools that simplify repetitive tasks. Design patterns are presented as a unifying idea as they possess the benefits of correctness, maintainability and reusability, and they are easy to use. Design patterns might have drawbacks of compromising efficiency and flexibility. The next chapter will discuss parallel programming systems that exist and their relative merits and shortcomings.

Chapter 3

Parallel Programming Systems

3.1 Overview

There have been numerous parallel programming languages and systems developed over the past forty years to allow the programmer to work with greater ease and efficiency. By 1989, over 100 languages were already documented for parallel and distributed computing [BaST89]. This number has grown significantly and widespread programming standards have developed such as PVM and MPI. This chapter documents attempts that have been made to make the task of parallel programming simpler along with providing examples of programming systems and their benefits and shortcomings.

3.2 Attempts to Raise the Level of Abstraction

As mentioned in Chapter 2, rasing the level of abstraction is one of the primary methods to deal with parallel programming complexity. The use of low level primitives for message passing often frustrates users due to the high complexity of the socket interface. Basic systems, tool kits, and integrated development environments have all been used to raise the level of abstraction for the programmer and allow for a more automated programming approach.

3.2.1 Message Passing Libraries (MPLs) and Remote Procedure Calls (RPCs)

MPLs raise the level of abstraction of socket level communication by using processes and communication channels. They allow processes to communicate with each other through message passing (sending and receiving messages). Examples of MPLs include the PVM and MPI libraries which have revolutionized multicomputer parallel programming. RPCs also involve message

passing and allow a procedure to be called on a remote machine. MPLs and RPCs have become accepted as standard models for parallel program development, but this level of abstraction may still be too low for the development of larger parallel applications.

3.2.2 Abstractions on top of MPLs and RPCs

These are abstractions which hide the details of MPLs or RPCs while using the beneficial attributes of these models underneath. Some examples of these systems include OOMPI [Osl02] and mpC [Mpc02]. OOMPI is an object oriented interface to the MPI-1 C++ standard. OOMPI keeps all of the MPI-1 functionality, but also offers new object oriented abstractions which promise to expedite the MPI programming process by allowing programmers to take full advantage of C++ features. The other tool, mpC, was developed and implemented on the top of MPI as a programming environment facilitating and supporting efficiently portable modular parallel programming. mpC uses the ANSI C standard as the programming language. This environment does not compete with MPI, but tries to strengthen its advantages (portable modular programming) and to weaken its disadvantages (a low level of parallel primitives and difficulties with efficient portability). It has the properties of efficient portability, meaning that an application running efficiently on a particular multiprocessor will run efficiently after porting to other multiprocessors). Users can consider mpC as a tool facilitating the development of complex and/or efficiently portable MPI applications.

3.2.3 Other High Level Programming Approaches

There are many high level programming paradigms that do not fall into the two previous categories. C/C++-Linda expresses parallelism through a distributed tuple space [CGMS94], a repository for different kinds of shared data such as database records or requests for computation. Linda is available across many different architectural platforms and the management of the tuple space is provided transparently across heterogeneous nodes [Losh94]. Another paradigm, ABC++, involves a library that supports distributed active objects on top of C++. Parallelism is described through C++ objects that have their own threads. There are other approaches such as Balance, which is a library of executable commands that allows the user to distribute the parallel workload evenly to the computers connected in one or more networks [BEST99]. The system can be run as a user level system or executed by the root to act as a system scheduling tool for microprocessors and

interconnected computers.

Enterprise was a breakthrough in high level programming tools. In brief, Enterprise is a graphical programming environment complete with a code generation mechanism, graphic visualization tools, a compiler, and a debugger. Enterprise allows programmers to express applications though a set of design patterns. Enterprise uses neither PVM, nor MPI underneath, but rather low level C augmented by new semantics for procedure calls that allows them to be executed in parallel [WIMN95]. This project will be discussed in more detail in section 3.5.4.

Raising the level of abstraction makes parallel program development easier, but at the risk of compromising flexibility. Figure 3.1 shows the relationship between flexibility and the level of abstraction for various existing parallel programming tools.

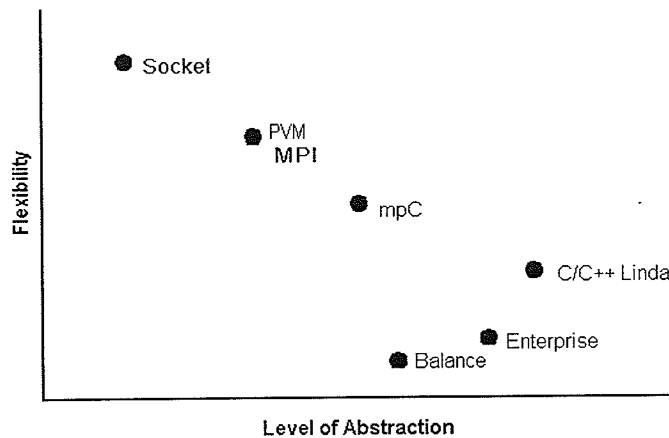


Figure 3.1 Tradeoff between abstraction and flexibility (adapted from [Siu96])

3.3 Classification of Tools for Parallel Programming by Functionality

The previous section provided a classification scheme for tools based upon their level abstraction. Another scheme to classify parallel programming tools is based on their functionality. Parallel programming tools are utilized to enhance the comprehensibility of complex problems and to improve the correctness of the coding approach. They provide functionality such as programming environments, parallel debuggers, performance monitors, and project management tools. Other tools such as PVM Simulator (PS), allow users to predict the performance of a parallel application on a

different architecture without actually running the simulation on that architecture [AMMV98]. This negates the need of investing money in a computer system that may later prove to be insufficient. Generally, the more integrated tools a programming system supports, the easier it is to develop parallel programs. To ensure a higher level of efficiency in programming, the level of abstraction provided by a tool must complement the base programming model. As an example, consider a GUI which uses graphs to represent communication between nodes of a distributed network. This approach works quite well as the graph model coincides well with communication patterns. As a counterexample, A GUI that expresses the structure of communication in a confusing manner would not be useful. Parallel programming systems can be divided into basic systems, tool kits, and integrated development environments based upon their overall functionality. Examples of systems falling into each category are shown in Figure 3.2. This classification scheme is independent of the previous classification scheme involving level of abstraction.

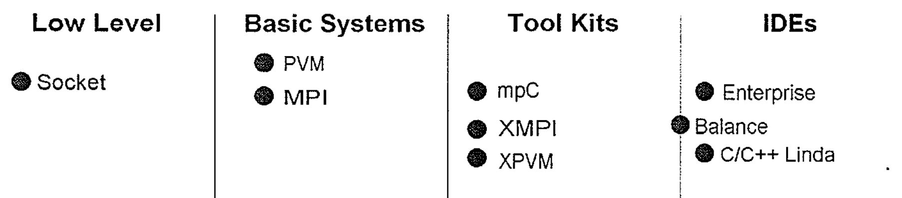


Figure 3.2 Classification of parallel programming systems by functionality

3.3.1 Basic Systems

Basic systems only provide the core functionality for developing parallel programs in a particular paradigm. These systems are often used by researchers who wish to try out new libraries and paradigms, but have no need to develop the product into a full system (yet). These systems provide enhancements over the basic programming paradigm and do not severely limit flexibility. Some examples include PVM and MPI which have matured into very useful products on their own. Other systems in this classification include ABC++ and Orca [Siu96].

3.3.2 Tool Kits

These systems are loosely coupled tools developed for a particular parallel programming paradigm. There are loosely coupled tools available for debugging, performance monitoring, and allocation of a parallel executable among processors. Commonly, a tool kit is developed once a

program paradigm has matured and is widely used. The concern with tool kits is the ability of the tools to be applicable in the various phases of the application development cycle for the desired programming paradigm. XPVM and PADE are examples of tool kits for developing PVM programs. XPVM is a graphical console and monitor for PVM [Kohl02]. XMPI and mpC are examples of tool kits for developing MPI programs. XMPI is an X/Motif based graphical user interface for running, debugging and visualizing MPI programs [LAM01].

3.3.3 Integrated Development Environments (IDEs)

An IDE is a complete development environment which integrates all the tools for developing, debugging, executing, and maintaining a parallel program. These environments are uncommon as they take a very long time to develop and require a high level of expertise on the part of the developer. These environments commonly provide higher level abstractions which make the job of programming easier for the user. Some examples of IDEs include Enterprise and Tracs. These two systems both have support for designing, developing, and maintaining parallel programs.

3.4 Two Distributed Programming Standards - PVM and MPI

PVM and MPI are two basic message passing libraries which have evolved into widely accepted programming standards for distributed heterogenous workstation clusters. They are both discussed in some detail in this section.

3.4.1 PVM

PVM (Parallel Virtual Machine) was the result of the efforts of a single research group working at Oak Ridge National Laboratories and Emroy University, thereby allowing it to have a large degree of flexibility in its design and also enabling it to respond incrementally to the experiences of a large user community [GrLu97]. The design and implementation teams were the same so the design and implementation of this tool were completed quickly.

PVM consists of a collection of library routines that the user can employ within C or FORTRAN programs. Using PVM, the user writes a completely separate and different program for

each type of computer on the network. This programming style is referred to as the Multiple Program Multiple Data (MPMD) model. The routing of messages between computers is done with the PVM daemon, which is installed by PVM on the computers which form the virtual machine (Figure 3.3). A daemon is a special operating system process that stays resident and performs system level operations for a user when required or carries out background system tasks. A process (master) may spawn other processes (slaves) dynamically during run time (see Figure 3.4).

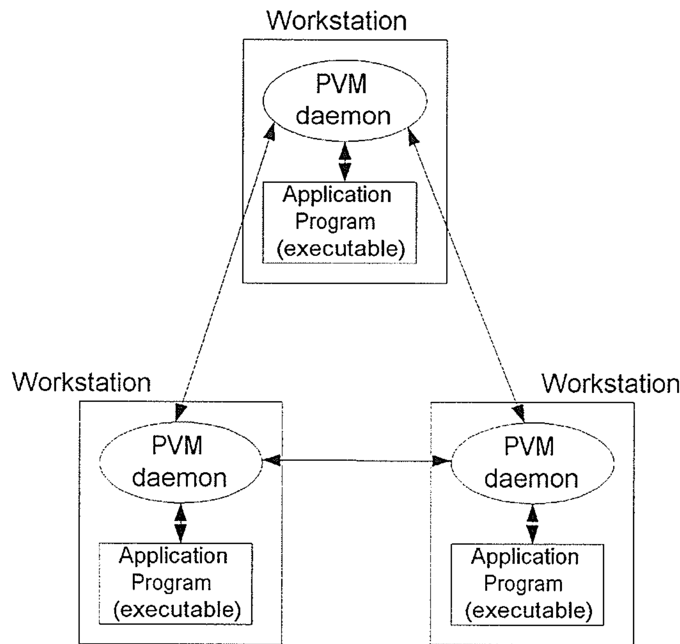


Figure 3.3 Message passing between workstations using PVM

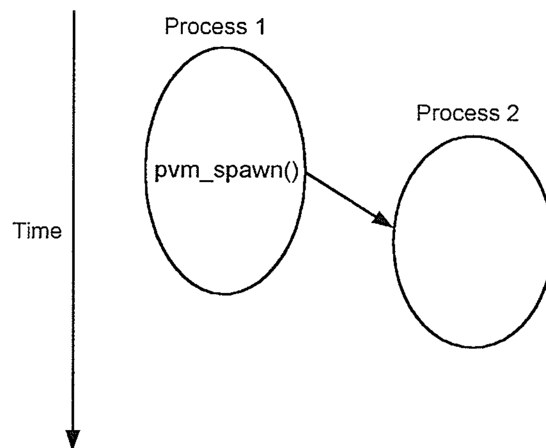


Figure 3.4 PVM process spawning

The execution model which fits PVM the closest is the MIMD model. The user must define a parallel virtual machine before running PVM, which contains a list of machines which will work together. Some of the features available in PVM include process control, fault tolerance, dynamic process groups, communication, and multiprocessor integration [Losh94]. PVM performs well over networked collections of heterogenous hosts [GeKP96].

3.4.2 MPI

MPI (Message Passing Interface) was designed by the MPI Forum, a collection of implementors, library writers, and users. Each group working on the MPI project design did so without any specific final implementation in mind, but with the hope that the implementation would be carried out by participating software vendors [GrLu97]. Because MPI was broadly planned and developed as a standard, it has become the most widely used parallel programming tool available today. MPI implementations are available for C, C++, and Fortran. MPI has advantages over PVM in that it possesses a richer set of communication functions and higher communication performance can be expected over a homogenous cluster of machines [GeKP96]. MPI also has the ability to specify a logical communication topology.

Using MPI, the programmer writes a single program which executes on all processes. Usually one process is mapped to each processor. Depending on control statements (i.e. if process_rank==1), only certain processes will execute certain statements. This programming methodology is known as the Single Program Multiple Data (SPMD) model. In earlier versions of MPI, a process could not spawn another process. However MPI 2 allows for dynamic process creation in a manner similar to PVM.

All global variable declarations will be duplicated in each process using MPI. Memory space for dynamic variables (pointer structures) only need to allocated by processes requiring the variable. MPI uses communicators to send and receive messages. These can be intracommunicators for communicating withing a group, or intercommunicators for communication between groups. A group simply defines a collection of processes. MPI has support for blocking, non-blocking, and collective communication of data.

When an MPI program is started, the number of processes, say p, is supplied to the program from the invoking environment. The number of processes in use can be determined from within the MPI program by using the MPI_Comm_size routine. Most MPI implementations developed will provide some useful error information when an error is encountered during execution, unlike PVM which simply aborts the program execution. The information provided is dependent on the MPI implementation and is not defined in the MPI standard.

Figure 3.5 shows an execution example of a typical MPI program. First the global variables are declared. Each of these variables will be present in all processes. Next, the MPI initialization statements follow which set up the processes for communication. Following this part, process 0 sends 10 terms of integer type to process 1. Finally, the MPI processes are shut down with the MPI_Finalize() statement. MPI code will not be valid following this statement.

```
/*- - - - - -global variable declarations - - - - - - - - - */
int my_rank;    /*for rank of current process */
int p; /*for number of processes */
int *first;
int tag =1;
MPI_Status starus;  /*return status for MPI_Recv*/
/*----------------------------------------------- - - - - - - - - - - - -*/

MPI_Init(&argc,&argv); /*start up Initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank); /*find out process rank->my_rank*/
MPI_Comm_size((MPI_COMM_WORLD,&p); /*find out number of processes ->p*/
first = malloc(10*sizeof(int));
if (my_rank==0) {
        for (i=0;i<10;i++)
        first=i;
        MPI_Send(first,10,MPI_INT, my_rank+1,tag,MPI_COMM_WORLD);
}
else if (my_rank!=0) {
        MPI_Recv(first,10,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
}
MPI_Finalize();
```

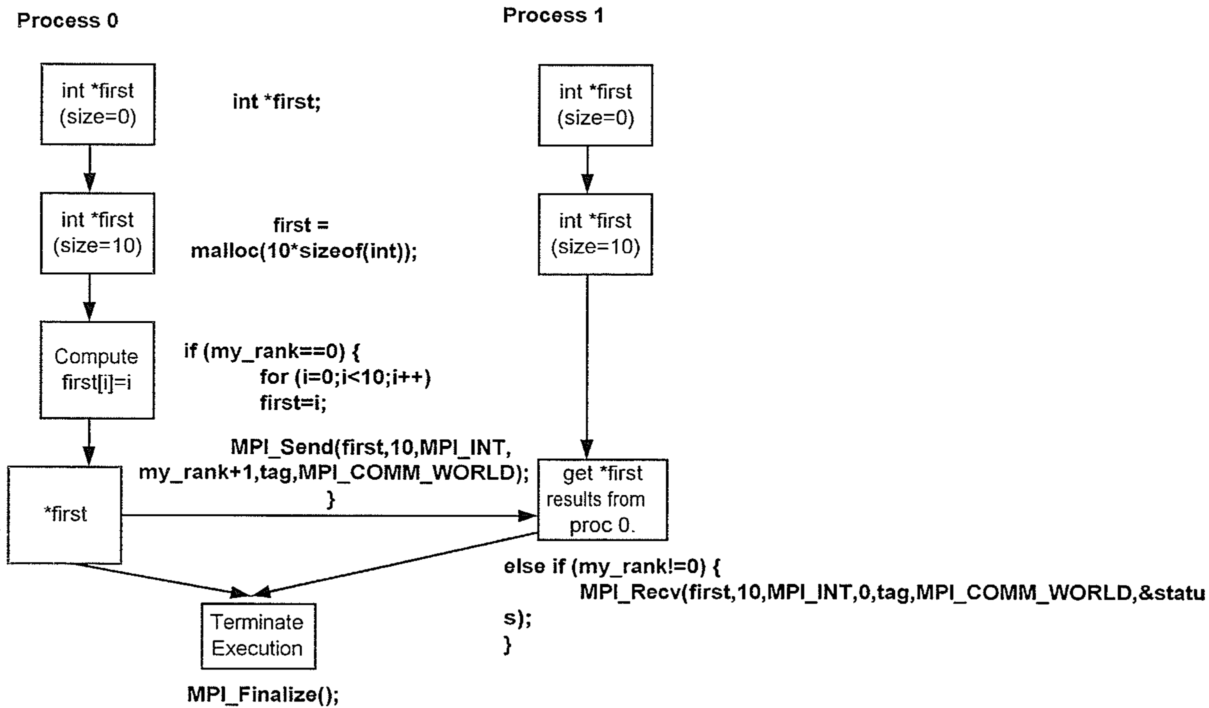


Figure 3.5 MPI execution example (2 process system)

There are numerous MPI implementation available including MPICH, developed by Argonne National Laboratory at the University of Chicago[GrLu96], and LAM which was developed by the Ohio Supercomputer Center at Ohio State University [Ohio96]. In addition, implementations of MPI such as MPICH-G2 have been developed for use with the Globus grid resource management system

[Glob00].

3.5 Existing Design Pattern Based Systems

Largely as a result of the increased popularity of multiprocessor workstations and multicomputer workstation clusters, research has accelerated to develop viable design pattern based programming systems which attempt to provide tools that enable the user to program more efficiently and complete complex parallel tasks with ease. Many of the approaches discussed in this section employ separation of the specifics, advanced GUIs, and templates for parallel programming.

3.5.1 CODE

Computationally Oriented Display Environment (CODE) was developed at the University of Texas at Austin and allows the user to compose sequential programs into a parallel one [Berg02]. Using CODE, the parallel program is expressed using a directed graph, where data flows on arcs connecting the nodes represent the sequential programs. The sequential programs may be written in any language, and CODE will produce parallel programs for a variety of architectures, as its model is architecture-independent. The CODE system can produce parallel programs for shared memory and distributed memory architectures, including clusters of workstations.

The developer builds the parallel program in two steps. In the first step, the developer specifies the contents of each node (i.e. sequential subroutines, input/output ports, internal variables, and rules governing how the node is run). The second step involves connecting the different nodes together using the GUI to show the interaction among them. CODE translates the graph into a complete parallel program. A screen shot of code is shown in Figure 3.6 below.

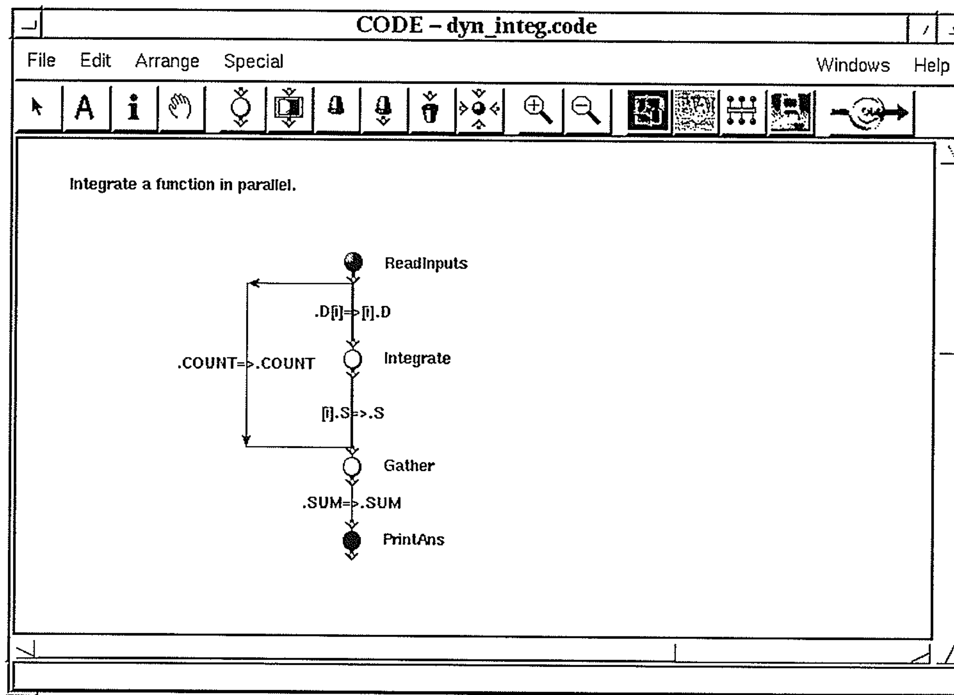


Figure 3.6 A screen shot of CODE (version 2.2) [Berg02]

CODE uses the dataflow model to represent communication and synchronization in parallel programs. The data flow model assumes that computation proceeds, depending on the availability of data. In CODE, the design patterns are the elements in the dataflow graph such as a sequential computation node or a common shared variable. High level design patterns such as divide and conquer are not available in CODE which would describe the structure and behavior of a collection of elements. CODE does enable the recursive embedding of graphs so that a constructed dataflow graph can be used as a single black box node.

CODE advocates the use of separation of the specifics, which means that parallel aspects of the application are kept separate from its sequential functionality. The first version of CODE appeared in the mid 1980s when the visual aspect of programming was the most important part. The new versions of CODE are designed to run on the Unix operating system and are compatible with PVM and MPI based networks.

No MPL programming skills are required to build parallel programs using CODE. Users

work at the procedural level, stipulating how a computation is done [Beck96]. With CODE, a transition is made from how something is done to what the developer is trying to do. CODE allows the user to write a book by writing an outline and then having the tool fill in the rest. The user need to only build multiple sequential programs, connect them using arcs, and CODE takes care of the parallel book keeping.

Some limitations are that the CODE environment can only run only over a Unix/Linux operating system and the full GUI of CODE is available only for Sun workstations. Also, from a programming perspective, it is believed that the use of dataflow elements and complex firing rules still involve too low of a level when building large and complex parallel programs [BHDM95].

3.5.2 HeNCE

Heterogenous Network Computing Environment (HeNCE) is an X-window based software environment designed to assist scientists in developing parallel programs that run across a network of workstations [Siu96]. HeNCE was developed at the University of Tennessee and is similar to CODE in its intent to provide a GUI specifying a directed graph, which shows the interaction among nodes. HeNCE also uses separation of the specifics in which the developer first specifies the sequential computation in each node and the communication between nodes using a process graph. Design patterns in HeNCE are represented by graphical icons and includes higher level parallel programming abstractions such as replication, loop, pipeline, etc. Other structures such as master/slave can easily be constructed with the provided design patterns and basic nodes.

The HeNCE model uses control flow graphs, as opposed to the dataflow oriented graphs of CODE. HeNCE generates PVM code based on the graphs constructed by the user. PVM, as discussed, is an accepted parallel programming standard so the code developed is portable. HeNCE also relies on the PVM system for process initialization and communication. The programmer never has to write explicit PVM code. During or after execution, HeNCE displays an event-ordered animation of the application execution, allowing visualization of the relative computational speeds, processor utilization, and load imbalances [Netl94]. Again, analogous to CODE, the developer can

easily decompose existing C (or FORTRAN) source code into pieces which can be executed in parallel over an existing network of workstations or supercomputers. In this way, existing programs may be reused and unused performance can be tapped out of existing machines.

HeNCE is limited to run under a Unix operating system. User feedback on HeNCE indicates that it is not flexible enough to express more complex parallel algorithms [BHDM95]. HeNCE was conceived as a research project, rather than a development tool and never gained a high level of popularity among users. Its development has been discontinued ,but is still used due to its legacy value as an early automated parallel programming tool [Netl94]. A screen shot of the HeNCE environment is shown below in Figure 3.7.

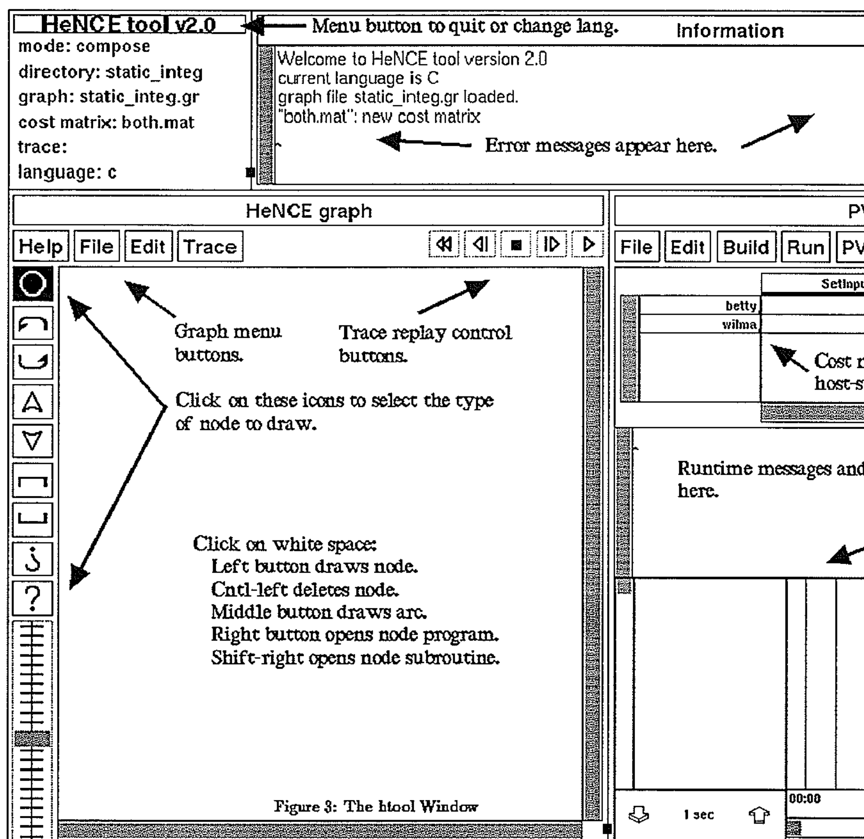


Figure 3.7 Screen Shot of HeNCE (from [Netl94])

3.5.3 Tracs

Tracs was a result of work carried out at the University di Pisa [BCDP95] with the design goal of creating an environment that can facilitate the development of distributed applications involving groups of networked, heterogenous machines. Tracs enforces the use of an appropriate methodology for distributed application design. The parallel application design is split into two phases, the first devoted to finding the basic design patterns components, the second to building an actual application out of the components. Tracs provides separation of the specifics, in a similar fashion as HeNCE.

The modular approach of Tracs permits code reuse and allows the developer to structure their applications in an organized manner with well defined interfaces between the components. Tracs provides many advanced utilities that fit into the overall framework and whose operations are independent of one another. It provides the ability to automatically create components and to simplify the definition of components in the application itself.

Tracs specifies three components which are the task model, the message model, and the architecture model. Nodes communicate synchronously or asynchronously by messages though unidirectional channels. A channel must be associated with a message model which handles the packing, unpacking, and translation of the data. A developer starts by specifying the sequential computation in the task model [Siu96]. Following this, the task model is combined with attributes such as ports, services, logical names, and messaging models. When all the tasks are defined, the developer connects all the task models and binds them to processor. The final code is generated based on the models.

The most significant contribution of Tracs is its use of high level design patterns. The powerful graphical interface can facilitate the addressing of complex design patterns such as task farm, ring, array, grid, and tree. Its architecture model has raised the abstraction of design patterns from a single process to a collection of processes. Support is provided for C, C++, and Fortran.

Tracs forces all design patterns to be graphical, which can cause difficulties with the

representation of some patterns that cannot be represented conveniently (i.e. divide and conquer). The graphical interface of Tracs is rich in its strategy, but can limit the expressiveness of the system. Tracs also can only run in the Unix environment.

3.5.4 Enterprise

Enterprise was developed at the University of Alberta in Edmonton [WIMN95, SSLP93]. It is an integrated environment complete with a compiler, a debugger, graphics visualization tools, and a performance debugger that allows developers to produce distributed applications with ease. It also uses separation of the specifics. There is a rich graphical interface which the user may utilize to build parallel applications, with the system automatically inserting the code necessary to handle the communication and synchronization [Ente02]. The code generated is C code that is supplemented by new semantics for procedure calls that allow them to be run in parallel.

The developer uses a programming model that resembles a business organization to represent parallel structures such as pipeline, master/slave, and divide and conquer and does not have to deal with low programming details such as marshalling data, sending/receiving messages, and synchronization. The developer specifies the desired design pattern technique at a high level by manipulating icons using the GUI (see Figure 3.8). All of the communication protocols that are required are inserted automatically into the user's code. The user is given control of the amount of parallelism required through Enterprise's high level mechanism.

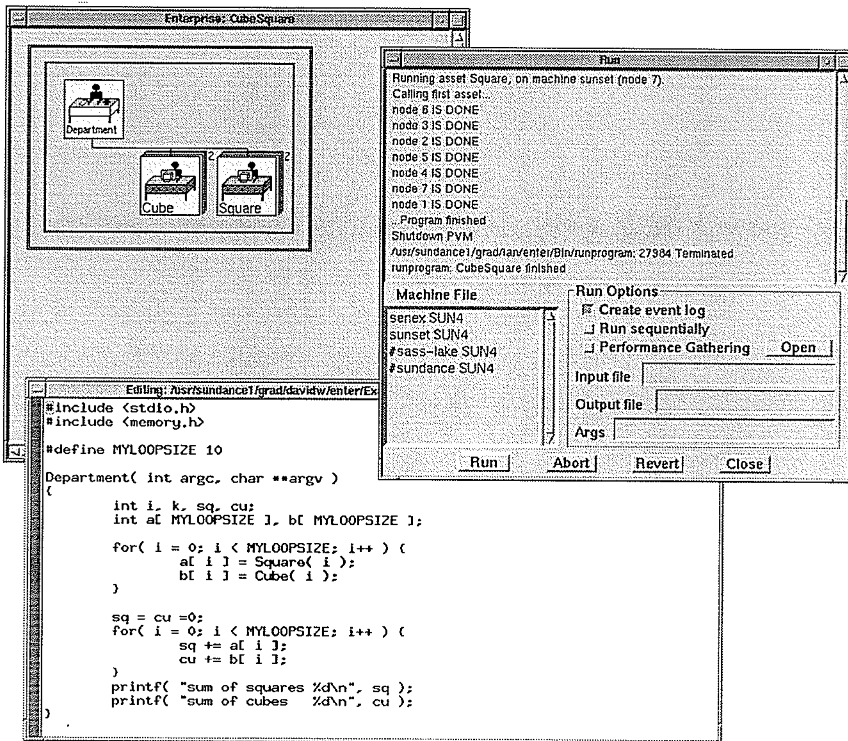


Figure 3.8 Enterprise Screen Shot (from [Ente02])

Programmers draw a diagram of parallelism inherent in their applications using the business model or enterprise analogy. Tasks are subdivided into smaller tasks and assets are allocated to perform the tasks. Parallelism is determined by the number and types of assets used. Graphical icons representing assets such as an individual, (assembly)line, division, and others are provided (Figure 3.9). A department, for example, can divide the tasks among components that can then perform the tasks concurrently.

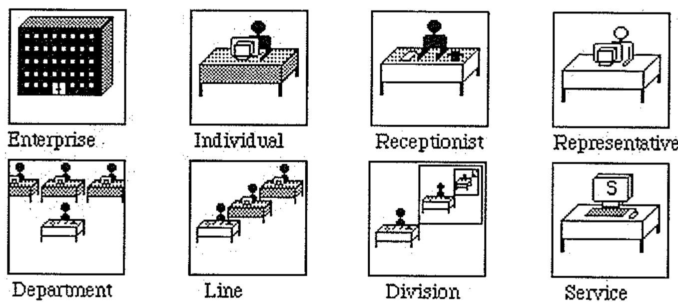


Figure 3.9 Enterprise Assets (Design Patterns) [IMMN95]

The fact that C code is generated greatly enhances the ability of the Enterprise to produce portable code. In addition, the high level of abstraction gives novice users the ability to program complex parallel programs.

The Enterprise programming environment itself can only run on the Unix operating system which is limitation. As well, many programmer have found that Enterprise is too inflexible in the code it produces. In a monitored programming experiment, graduate students produced less code using Enterprise than using PVM, but required more time create optimal code. The lessons learned from Enterprise are that design patterns can be used to quickly and correctly develop parallel programs, but these programs do not produce the performance of hand-crafted parallel programs using MPLs, and that users do not like to lose control and flexibility of low-level primitives within a higher level model.

3.5.5 DPnDP

Design Patterns and Distributed Processes (DPnDP) [Siu96, SiSi97] was developed at the University of Waterloo by Stephen Siu and Ajit Singh as a parameterized design-pattern based system. The programming system was designed with two enhancements over other existing systems, openness and extensibility. Openness is the ability to bypass the high level programming model and directly access low level primitives for the purpose of optimizing performance and enhancing flexibility. A system which permits easy access to low level primitives has a high degree of openness, while a closed system forces the user to stay within the automated coding approach. Extensibility refers to the ability to add new design patterns to the system, thereby increasing the system's utility. In a non-extensible system, if a required pattern is not provided by the system, the system has no advantage over direct low level coding.

DPnDP is an open system in two ways. First, developers can create any arbitrary process structure using a combination of single process design patterns (singletons) and multi-process design patterns. Users are not restricted to only the high-level design patterns. Second, developers can access low-level message passing primitives if they want to tune the performance or to use specialized message passing features such as ``groups" in PVM. Therefore, developers can develop applications, partially using design pattern and partially using low-level message passing primitives.

When users decides to use low-level message passing primitives over the high level automated code generation mechanism, they are responsible for ensuring correctness.

All design patterns in DPnDP share a uniform interface for definition and development. Other components in DPnDP access them only by using this interface. Therefore, a design pattern does not have to know the implementations of other design patterns to work with them. This context insensitivity allows system developers to add new design patterns incrementally into DPnDP without having to know the implementation of other patterns or the system. Furthermore, existing design patterns can be used as building blocks to create new design patterns.

The DPnDP parallel programming model assumes a MIMD machine architecture and an operating system that supports process creation and message passing among the processes. A parallel program is represented by a directed graph when using the GUI. Each node in the graph is a singleton or a multiprocess design pattern. Node in the graph communicate and synchronize through message passing, represented by directed arrows as shown in Figure 3.10.

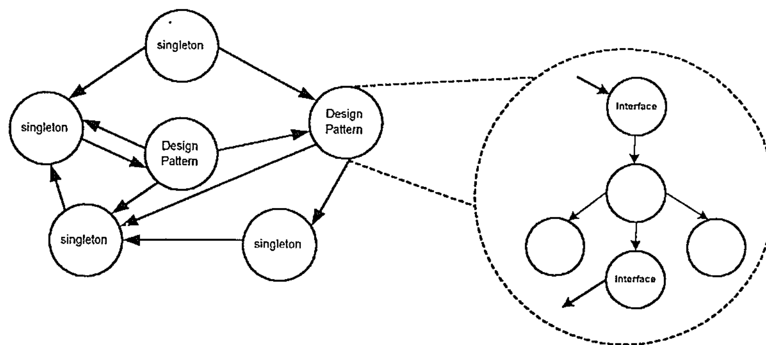


Figure 3.10 Structure of a DPnDP application [SiSi97]

Each process (represented by a node) in a DPnDP application operates in a loop in that it waits for incoming messages on any of its ports from other processes. When a message arrives at a port of a process, the process notifies the appropriate user provided message handler to process the message. Design patterns are provided that implement various types of process structures and interactions found in parallel systems, but the application specific procedures are unspecified allowing the user to fill in his/her code. When using a design pattern, the user only deals with communication that is application specific. All other communication needed for process management is taken care by the

automatically generated code.

DPnDP has been implemented and run using a network of workstations that run under the Solaris operating system. Preliminary results from simulations indicate that the performance of code produced by DPnDP is within 10% of hand coded PVM for similar problems. Openness and Extensibility are improvements over other pattern based systems. However, the programming system can only be used in Unix/Linux environments.

3.6 Proposed Enhancements

The above systems provide a insight into existing design-pattern based parallel programming systems. These systems have simplified parallel programming significantly for intermediate users, but at the same time have imposed bounds on the user. Every system discussed has limitations, ranging from the programming model being too low in the case of CODE to a lack of flexibility in HeNCE. There is a well defined tradeoff between the ease of use of a system and the system's flexibility. Tracs and Enterprise suffer from a lack of expressivity of higher level design patterns.

Enhancements are required to the programming model of most of the above systems in order that the programmer should be able to use mechanisms provided by the system to cover the common problems, but also include mechanisms to cover the remaining types of uncommon problems [BHDM95]. In addition, the programmer should be able to use the programming environment over a diverse range of platforms such as a Unix workstation, a Windows PC, an Apple Macintosh, etc. The programming environment should function irrespective of whether the platform can actually run the resultant parallel program. The developed program can always be ported to the intended platform(s).

While systems such as DPnDP have been proposed and developed to tackle the issues of openness and extensibility [Siu96, SiSi97], there is little to show for programming portability. Overcoming the portability issue for the programming environment is not an easy task as every OS and hardware platform has its own rules for handling events and graphics. While MPI and C are

standards used in parallel programming today, there is almost no system which can aid the user in programming MPI code using C on almost any widely used computing system. This thesis project sets out to demonstrate that by building a parallel programming system through a non-platform specific language such as Java, a portable system is possible.

3.7 Summary

This chapter has provided a description of existing parallel programming systems. Raising the level of abstraction can be done by providing the user with message passing libraries(MPLs) and remote procedure calls (RPCs), abstractions on top of MPLs and RPCs, and through other more unique approaches. Parallel programming systems are broadly classified into basic systems, tool kits, and integrated development environments based on their functionality. PVM and MPI have evolved as two accepted MPL standards for distributed programming and design pattern based systems have emerged such as CODE, HeNCE, Tracs, Enterprise, and DPnDP to assist the programmer. Most of these systems are closed, not extensible, and generally only function as programming tools on particular computer architectures or operating systems. Improvements in these areas appear desirable. Chapter 4 discusses the design and implementation of MPI Buddy, an open and portable design-pattern based system.

Chapter 4

Design and implementation of a Parallel Programming System (MPI Buddy)

4.1 Introduction

This chapter describes the design and implementation of the MPI Buddy system for parallel program development using MPI. The uniqueness of the MPI Buddy system is the ability for the developer to program the application from almost any platform. The functionality desired from the programming system is discussed before the actual layout of the implementation is presented. Next, the design patterns included in the system are discussed. The chapter concludes with the programming model that is intended to be used with this system.

4.2 Functionality Desired

1. User Friendly Interface

A user friendly interface is important for allowing the user to conveniently work with the higher level automated code generation mechanism and simultaneously providing access to low level primitives.

2. Openness

Openness is a system attribute that is key to allowing the user to have the flexibility to customize the automatically generated code from the system. Openness gives the user the ability to mix the high level model with low level primitives when necessary. Openness can be achieved in the manner shown in Figure 4.1.

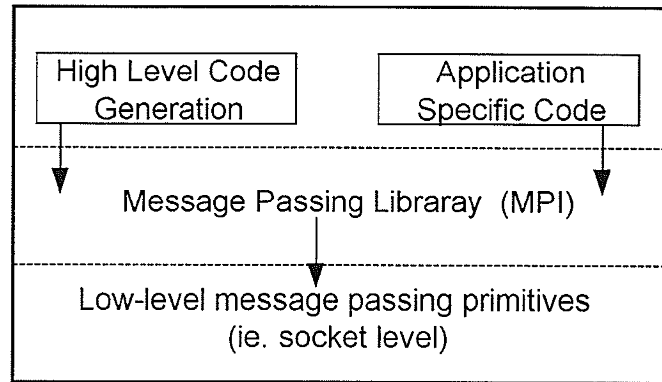


Figure 4.1 A layered open system (Adapted from [Siu96])

One disadvantage of an open system is that correctness is compromised as the system has no control over the code directly entered by the user. Another disadvantage is reusability of the code becomes compromised by the user inserting application specific code.

3. Design Pattern Based

As explained in chapter 2, design patterns offer many advantages including correctness, maintainability and reusability, and ease of use. Therefore, it is imperative that the system use parallel design patterns and that they be parameterized so a single pattern can be adapted to what the developer specifies by simply filling in the parameters using a GUI.

4. Extensibility

The system should allow new design patterns to be added conveniently by simply adding another module onto the system and completing the links in the system code. Ideally, it would be beneficial if new design patterns could be added to the system easily through a common interface, but this design concept could not be realized in this work due to time constraints.

5. Generation of Optimal Code

The code generated by the system should not only be correct, but it should be optimal in terms of communication requirements. The developer should be able to use the automatically generated MPI code wherever possible and expect to get good parallel results. The assumption is that the user's

parallel design is efficient in the first place and the target environment is a cluster of single processor workstations.

6. Ability to Test Code Syntax Correctness

The system should be able to compile the MPI code to determine whether the MPI program is functioning correctly. This is especially crucial to an open system which allows the user to directly modify the generated code. Forcing the user to exit the system to compile the code would delay the development process.

7. Portability of System

The developer should be able to work different architectural platforms using widely accepted operating systems. This is possible if the system is developed using the Java programming language and Java's Swing based components are exploited. The Swing GUI components appear visually the same regardless of the platform being used. The Java programming language is described in more detail in section 4.3.

4.3 The Java Programming Language

The Java programming language has revolutionized the world of programming, allowing developers to easily create multimedia-intensive, platform-independent, object-oriented code for conventional, Internet, Intranet, and Extranet-based applets and applications [DeDe99].

The Java programming language has also been considered for parallel program development directly. A Java class library jmpi already exists [Dinc98]. Jmpi is a 100% Java-based implementation of the Message Passing Interface (MPI). jmpi supports all the MPI-1 functionality as well as the thread safety and dynamic process management of MPI-2. jmpi is built upon JPVM, which implements message passing by communication over TCP sockets. Java has the advantages of being easy to learn, keeping projects manageable, and simplifying the development and testing of parallel programs. Java is platform independent and extremely portable. The Java programming language also has the advantage in that the language was designed for networks (and computer

clusters) and has built in communication routines.

However, the overhead involved with Java far exceeds that of the C language and can result in very slow executing programs, especially message passing ones. Java programs run on average 10 times slower than those written in C. This makes no difference in building a programming system, as the rich graphical support for coding the API in Java gives a graphical richness to the application, while at the same time the actual application code developed from the Java application will use the low level, high speed C language with MPI support. This style of interface programming incorporates the best virtues of both the Java and C languages: ease of graphical development and efficient low-level code.

4.4 System Design Layout

This section details the layout of the "MPI Buddy" system that was developed using Java with the intent of providing a level of abstraction above MPI. The Swing components of Java 2 were exploited to give the programming system a consistent appearance across computing platforms. The system was designed as shown in Figure 4.2.

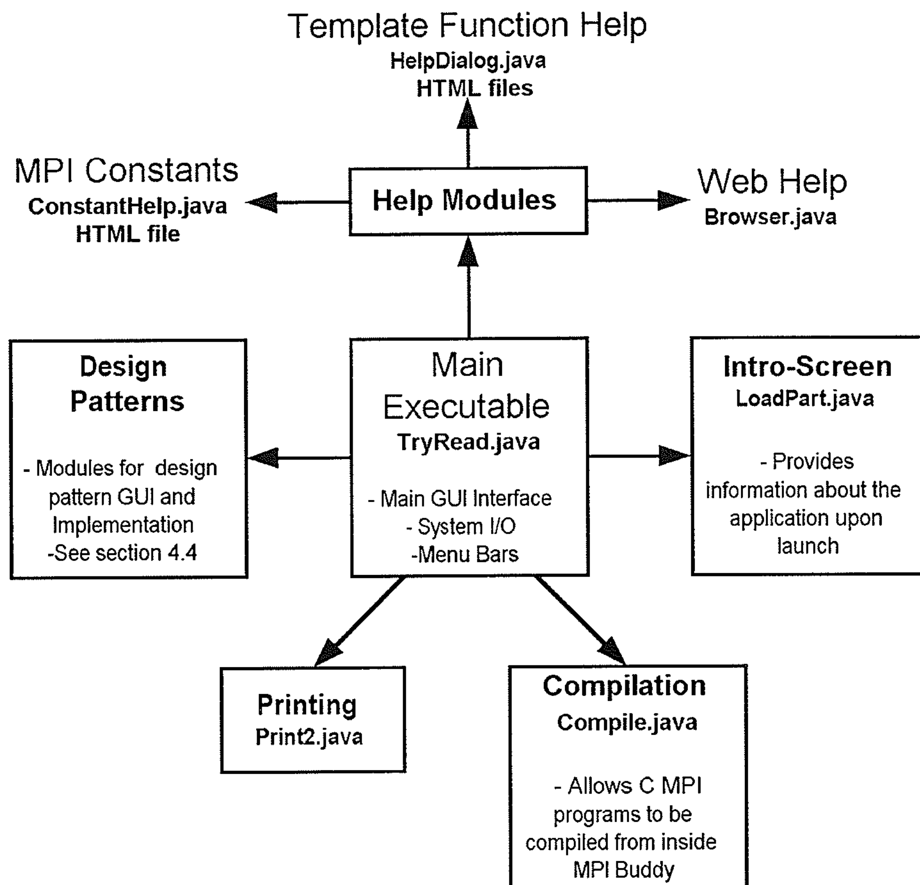


Figure 4.2 Layout of the MPI Buddy System

The main module is responsible for the launch of the application and links when the user selects various menu selections to other classes in the application which perform specific functions. This approach was determined to be logical and consistent with the Object Oriented Modular style of programming. The main features (Figure 4.2) of the MPI Buddy application are described below.

4.4.1 Main Executable

This class produces the main GUI window that displays the user's code (Figure 4.3). The screen and file I/O for the text MPI C code is handled by the main executable. The code in this module contains links to the other program classes, which can be activated by selecting options from the pull down menu. In addition, there is another text area produced by this class which returns the output results of compilation attempts on C code by the user. Much of the code of this class is only executed once action events are instantiated by the user.

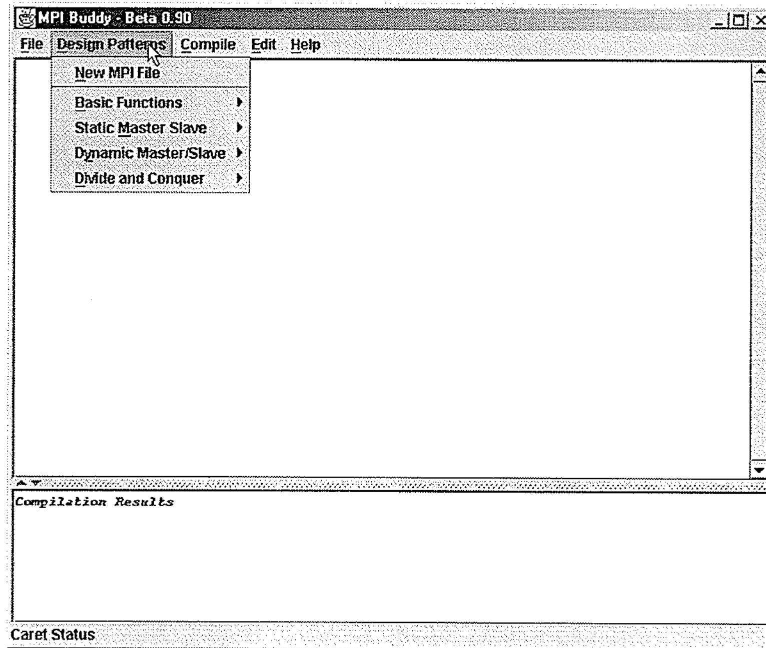


Figure 4.3 Screen shot of MPI Buddy (main window)

4.4.2 Compilation

The Compile class allows the user to compile a C MPI program (or basic C program), providing that the underlying operating system supports the compile command used. The user is given full control of the compilation command to be used and may modify the command line statement directly in the text box provided by the GUI (Figure 4.4). This gives added flexibility to the application and ensures that the compilation command will not be limited only to one compiler or platform.

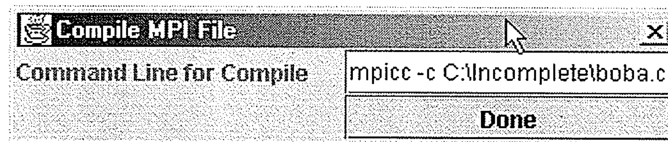


Figure 4.4 Compile GUI

4.4.3 Help Modules

It was determined early in the development process that the application would require a learning curve for the user. Fortunately, the Java language allows for the easy displaying of information in html files and also provides easy access to the world wide web. These capabilities were exploited in order to provide the user with support. The help menu accessible in the main window provides the following support:

- **Template Help**: Html files were created during the development process that document the intricacies of each design pattern type and how to use them. Using a Java JEditorPane, the contents of the html file are displayed.

- **MPI Constant Help** : A JEditorPane is used to display the information about the various MPI types (stored in a html file).

- **Web Help** : A simple web browser is provided that sends the user to a main web site for MPI development upon launch (http://www-unix.mcs.anl.gov/mpi/www/www3/). The user may click on the hyperlinks or type in a URL to access whatever else is required from the world wide web. This feature requires that the computer the user is working on is connected to the internet.

4.4.4 Printing

It was deemed important that the user should be able to print out his/her document, consistent with other APIs. The Java Printable interface was implemented and code was written to ensure that the proper number of pages to print was automatically calculated.

4.4.5 Design Patterns

The design patterns were developed independently of one another, but all inherit from a common base class (DesignMaster.class). This base class contains the common variables used to create the communication code as specified by the user. These include the communicator name, the process variable declaration, current rank variable for each process, and the default MPI_Status designation. The design patterns are described in much more detail in section 4.5 .

4.5 Design Patterns Included

The design patterns of the MPI Buddy system were chosen from recurring parallel programming decomposition paradigms such as divide and conquer, 1D Master/Slave, 2D Master/Slave, etc. The design patterns incorporated into the system were found to be among the most utilized. One pattern, the dynamic 1D Master/Slave was provided mainly to give the programmer an idea of how to create such an approach. Dynamic workpool approaches are appropriate for cluster environments when the individual workloads are sufficiently coarse grained or the individual workstation speeds differ substantially. There are almost an infinite number of other communication and decomposition schemes available than those provided in the system, but time did not facilitate their implementation.

The GUI interfaces for the design patterns were developed with the goal in mind of providing choices to the user that will enable custom communication code to be created along with the generation of all the required variables. There are tool tips provided to the user when entering the required parameters that will enable him/her to easily understand what is required. Additionally , help on each design pattern is available in the help menu (help->template functions). The code produced by the various design pattern modules was developed so that design patterns can be reused within a single program with similar or different input values for the required parameters. The patterns included in the system are described in this section.

4.5.1 1D Scatter/Gather

The first function of the 1D Scatter/Gather module is to distribute (scatter) an array of data evenly over the p processes. This setup assumes that the initial array has a number of elements that is evenly divisible by the number of processes. If the data is not evenly divisible, an error will be produced in the resultant program. The approach is shown below in Figure 4.5 .

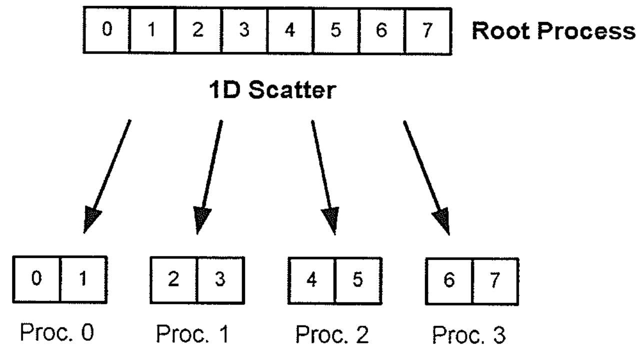


Figure 4.5 1D Scatter/Gather

The second function of this module is to gather up the data that was previously scattered. The user is provided with a choice of the type of gather operation required. More flexibility is available depending on the type of gather operation specified by the user. The user has the choice of No Gather (leave the data among the processes), MPI_Gather, MPI_Allgather, and MPI_Reduce. The parameters that the user must fill with the GUI include the send and receive buffers, the communicator, root process rank, and MPI datatype (int,float,double,etc).

4.5.2 Balanced 1D Send/Receive

This module is analogous to the 1D Scatter/Gather module, except now there is no requirement for the load (data) size to be evenly divisible by the number of processes. If the data is evenly divisible by number of processes, using the 1D Scatter/Gather module will result in more efficient code being produced. The code produced by this module works as follows: The data is sent out in a manner such that the root process (0) always gets *naverage* consecutive terms from the original send buffer, where *naverage* is the data size divided by the number of processes. The next process will get *naverage terms* + a leftover term if there are remaining terms (These terms are always consecutive in the new receive buffer). This continues for the remaining process until all the remaining terms (i.e. terms above naverage) are used up. The total data size variable is *totalCount* upon code generation. The data sent may also be received again into the root process. If the user selects "Yes" in the Receive drop down menu.

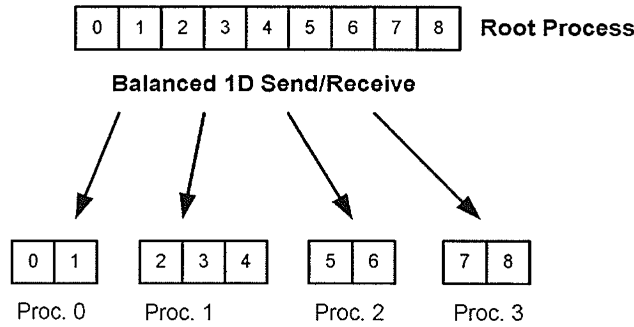


Figure 4.6 Balanced 1D Send/Receive

4.5.3 2D Scatter/Gather

This module implements a 2 dimensional scatter/gather approach. The module has the function similar to the 1D Scatter/Gather module, except now the data is scattered using equal sized blocks. The code created by this module works by first setting up the required variables. They are the *totalsize* for the total data size, *hsize* for the block size, *loc_hsize* for the amount from each block to go to every process, and finally *vsize* for the number of blocks. Following this, the data is scattered blockwise so that each process gets *loc_hsize* terms after each scatter. This approach is illustrated below in Figure 4.7.

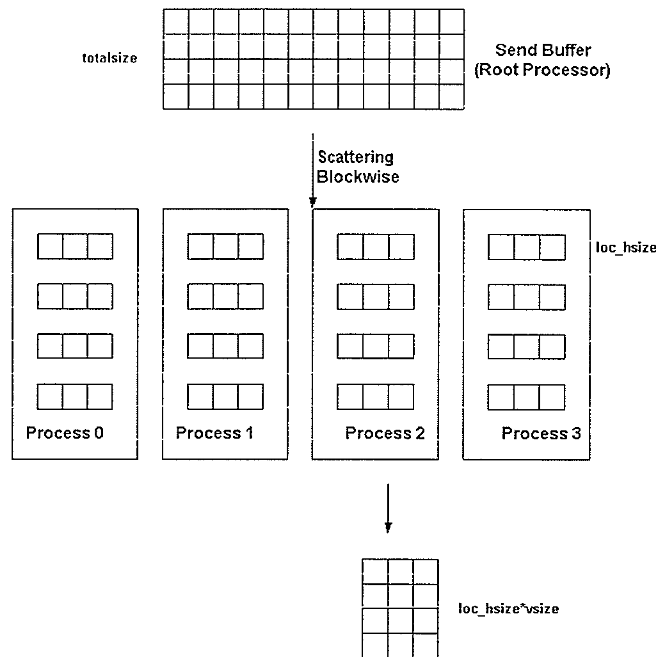


Figure 4.7 2D Scatter/Gather

Using this approach there are two constraints; First, the data size must have a number of blocks to cover the entire data size (i.e. the Send Buffer). Second, each block size must be evenly divisible by the number of processes. If either of these conditions is not met, an error will result. If the first condition is met, but not the second, the developer can use the Block/Cyclic Send/Receive Approach instead.

There is also the choice to receive the data back into the root process following the initial 2D scatter. If this option is chosen, the data will be received into the final receive buffer in a blockwise gather fashion (so the elements will be in order).

4.5.4 Block Cyclic Send/Receive

This module implements a block cyclic send/receive data decomposition scheme. The approach distributes data cyclically from blocks of data. The code created by this module works by first setting up the required variables: They are *dataSize* for the total data Send Buffer size, *hsize* for the block size, *vsize* for number of blocks, *naverage* for the integer average in each process, and *nremain* for the total number of leftover terms above naverage. The communication proceeds in rounds with the root process always being 0. The total number of points after each round (for each process) are in *mypoint*s. The variable *mytotalpoints* contains all the points after the completion of the block cyclic send/receive. Following this, the data is scattered blockwise so that each process gets *loc_hsize* terms after each scatter. This approach is illustrated below in Figure 4.8.

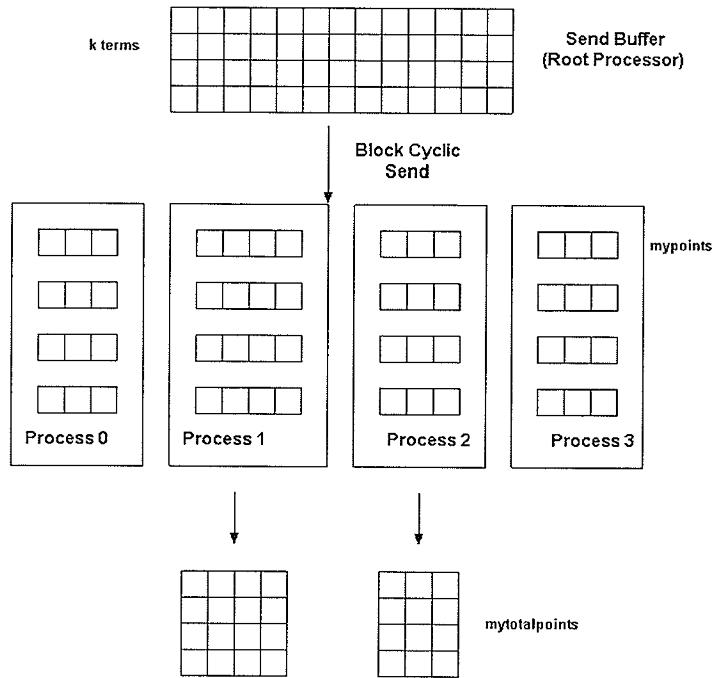


Figure 4.8 Block Cyclic Send/Receive

4.5.5 Cyclic Send/Receive

The function of this module is to distribute an array of data in a cyclic fashion. In other words, each element of the array will be distributed to a process determined by the equation: $i\%p$, where i is the position of the element in the array and p is the total number of processes. Figure 4.9 illustrates this communication setup. There are no constraints on using this module.

To save communication time, the total data to be sent to the processes is calculated by the root process (process 0) and simultaneously the slave processes calculate how many terms they will require. The total data size for the Send Buffer is given by the variable *dataSize*. For all processes, *mypoints* gives the total number of points in each process, and the receive buffer specified by the user gives the data in each process.

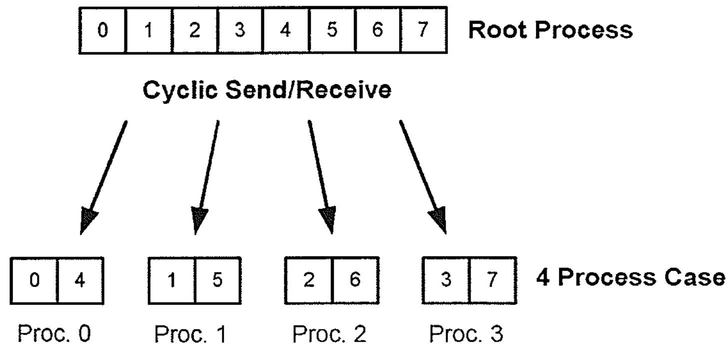


Figure 4.9 Cyclic Send/Receive

4.5.6 Dynamic 1D Master/Slave

The function of the Dynamic 1D Master/Slave (workpool) module is to allocate processor resources based upon the availability of data. A process working in parallel to solve the job is allocated a certain amount of work. Upon finishing the work, the idle process checks if more work is available. If more work is available, the process is assigned new work. When the total work is completed, the dynamic 1D Master/Slave function exits. Figure 4.10 shows the approach employed This approach is useful for heterogeneous clusters where individual job completion times are unpredictable.

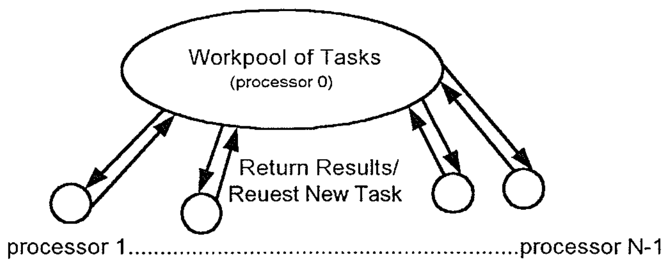


Figure 4.10 Dynamic 1D Master/Slave

The root process (process 0) keeps track of what task (block of data) is in which process and takes care of the task communication to the other slave processes. The slave processes receive the tasks and manipulate the data received, before sending the results back to the root process. When a result is returned to the root process, it is inserted into a final receive buffer and the root process sends out the next task until all tasks are completed. This module produces a function which the user can name using the module GUI. This function is designed to be manipulated to perform the work desired in

each task.

4.5.7 1D Divide and Conquer

The function of the 1D Divide and Conquer module is to distribute a 1D array of data using the divide and conquer approach. This approach uses a balanced tree to distribute the data to the processes. The 1D Divide and Conquer approach is illustrated in Figure 4.11 for a 8 process system.

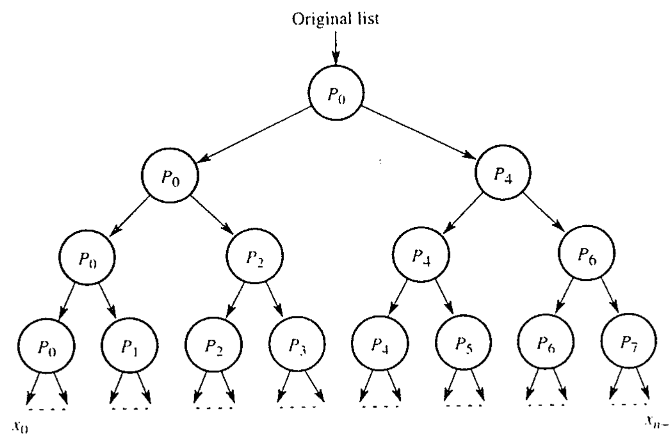


Figure 4.11 1D Divide and Conquer (from [WiAl99], p.114)

The type of distribution for the Send Buffer data may be chosen by the user using the pull down menu. The user also has the option of collecting the results up the tree again to the root process (process 0) if requested in another pull down menu. This module will produce one procedure that depends on the type of distribution selected by the user (i.e. regular, keep even/send odd, keep odd/send even). Another procedure is produced if the user selects "yes" from the Receive pull down menu which allows the user to specify computation that can be performed on the way up the tree to a single array.

4.6 Programming Model

The programming model for MPI buddy uses a Java Swing based GUI which the user utilizes to write his or her code with. The user must go through several small steps in order to produce efficient application code with minimal programming time.

1. Define a New MPI File

The developer selects Design Patterns->New MPI File from the main menu. This opens a window which requests parameters such as the default message tag, the default communicator, the name to denote the variable containing the number of processes, and other essential parameters in an MPI program. The user fills in the requested information and selects the DONE button. The main text screen will be cleared of any existing program code and replaced with new code containing the MPI header/ender.

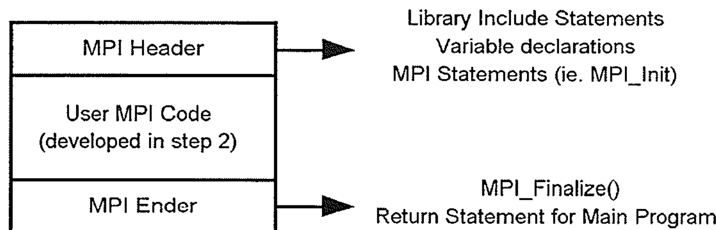


Figure 4.12 MPI Buddy program layout

The MPI header/ender contains all the necessary information for starting up and shutting down an MPI program. In addition, there are include statements in the C code for the commonly used C libraries (stdlib.h,stdio.h) along with the mpi library (mpi.h). The user may stop at this point and use only the header/ender to develop the MPI program further if he or she chooses. This code by itself is correct and will compile and execute with no work performed. However, the many design patterns provided by the tool make programming many parallel problems easier by following the remaining steps.

2. Utilize the Required Design Pattern(s)

This involves the user first clicking the left mouse button in the code at the place in the program where the data decomposition/ communication structure is to be inserted. This will require that the user has a good knowledge of C programming to understand what is intended to be constructed. The design pattern to be inserted can be found from the menu : *Design Patterns->(type of pattern)->(pattern)*. Parameters will be requested of the user and upon filling these out and clicking DONE, the code skeleton will be inserted automatically. The user may modify and add to the code as

deemed necessary. The model for this approach is shown in Figure 4.13.

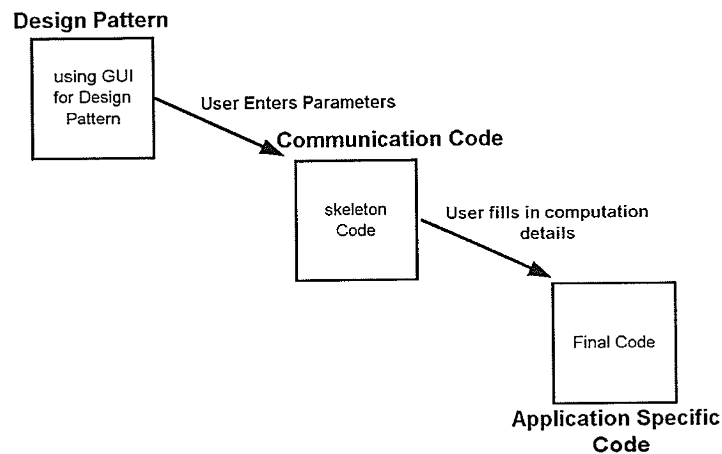


Figure 4.13 Programming approach

3. <u>Save, Check, and Compile Final Code</u>

The user must save the MPI C file that was developed if the aspiration is use it on a Linux/Unix cluster. The file will be saved in the standard ASCII text format for portability. Following this, if the underlying architecture supports MPI program execution, the program may be compiled and the results will appear in the lower text area of the GUI. Alternatively, at this stage one may opt to work with the standard MPI compile statements directly in a Unix/Linux environment.

4.7 Summary

This chapter has discussed the design intentions and layout of the MPI Buddy system. It was considered important that the tool be portable, design pattern based, contain a user friendly user interface, produce optimal code, and have the ability to test the code for correct syntax. The ability to extend the number of design patterns was deemed to be a very good trait as well and was partially accomplished through a modular system design. The system was developed with the use of the Java programming language since this language has built in portability properties. The design patterns included in the system were discussed in detail and even though the design patterns included are among the most commonly used, they are only a small subset of the total number of parallel patterns in existence.

The programming model of MPI Buddy starts by having the user declare a new MPI file. Following this, the user selects the design patterns necessary for the application being developed. Filling in the generated code skeleton with application specific computation code completes the coding process. The programming examples and analysis of the utility of the MPI Buddy tool is discussed in the next chapter.

Chapter 5

Programming Experiments and Analysis

5.1 Introduction

This chapter describes how various parallel programming problems can be tackled using MPI Buddy. Examples are provided involving problems such as the 2D discrete wavelet transform and the iterative fast Fourier transform. Each of these problems is approached using the design patterns provided in the MPI Buddy programming system. Observations and difficulties experienced using MPI Buddy are documented. Finally an analysis of the programming system model is made.

5.2 Metrics used to Evaluate Performance

Parallel and distributed programmers use various metrics to gauge the performance of a parallel implementation. Metrics such as run time, speedup, efficiency, and cost reveal information regarding the benefits of parallelism over a sequential version of the algorithm. Other softer metrics are required to assess the benefits of a parallel programming system. These include the time spent developing the code and the ease of programmability. The ability to port the applications and the fact MPI Buddy can run on different computing platforms are benefits inherit in the design of the system.

5.2.1 Objective Metrics

<u>Run Time</u>

This is the most primitive hard metric to gauge the performance of a parallel application. It is basically a raw comparison of the best sequential algorithm run time t_s to the parallel run time t_p.

Speedup

Speedup is a metric that captures the relative benefit of solving the problem in parallel over using a single processor system. Speedup is defined as :

$$S(p) = \frac{Execution\ time\ on\ 1\ processor}{Execution\ time\ with\ p\ processors} = \frac{t_s}{t_p} \qquad (5.1)$$

The best sequential algorithm is employed for t_s in the above equation.

Efficiency

Efficiency, E is a measure of the fraction of time for which the processor is usefully employed with work (computation) and not idling.

$$E = \frac{Execution\ time\ with\ 1\ processor}{Execution\ time\ with\ a\ multiprocessor \times number\ of\ processors} = \frac{S(p)}{p} \qquad (5.2)$$

An ideal efficiency approaches 1, representing the rare case when all the parallel processors are employed on the computation at all times. In this case the speedup S(p) would be p.

Cost

Cost is an index which indicates whether the parallel execution time is proportional to the sequential execution time. Cost is defined as :

$$Cost = Parallel\ Execution\ time \times total\ number\ of\ processors\ used = \frac{t_s . p}{S(p)} = \frac{t_s}{E} \qquad (5.3)$$

5.2.2 Subjective Metrics

Time spent developing code and ease of programmability are crucial to recognizing how the MPI Buddy system benefits the developer. Understanding that MPI Buddy is geared at distributed clusters of workstations, since it uses MPI underneath, it is necessary that the code generated be correct and perform well against hand coded MPI applications.

In evaluating a programming system, the time spent developing code and ease of programmability are the most important measures to assess, yet they are the most difficult to quantify. One must be able to realize whether the tool is worth using. Every programmer will react differently to a programming system, depending on his/her programming knowledge and personal preferences. Still, it is believed that generalizations can be drawn from using representative problems and conferring with a small group of individuals that are familiar with MPI.

Sections 5.4-5.5 illustrate the use of the MPI Buddy system for parallel programming. The objective metrics of run time and speedup are used to gauge the performance of the hand coded implementation. A comparison is made between the MPI Buddy and the hand coded versions to understand if MPI Buddy does generate efficient code. A brief explanation of the time spent developing the code using the hand coded and MPI Buddy scenarios and a description of the ease of programming the application is also provided for both situations.

5.3 Computing Platform Used in this Work

The computing platform used for running the developed parallel applications was a dedicated Beowulf Cluster with eight nodes, located in the Department of Computer Science at the University of Manitoba. Each compute node is a Dual Pentium III workstation operating at 550 MHz with 512 MB RAM. The setup of the platform is shown in Figure 5.1.

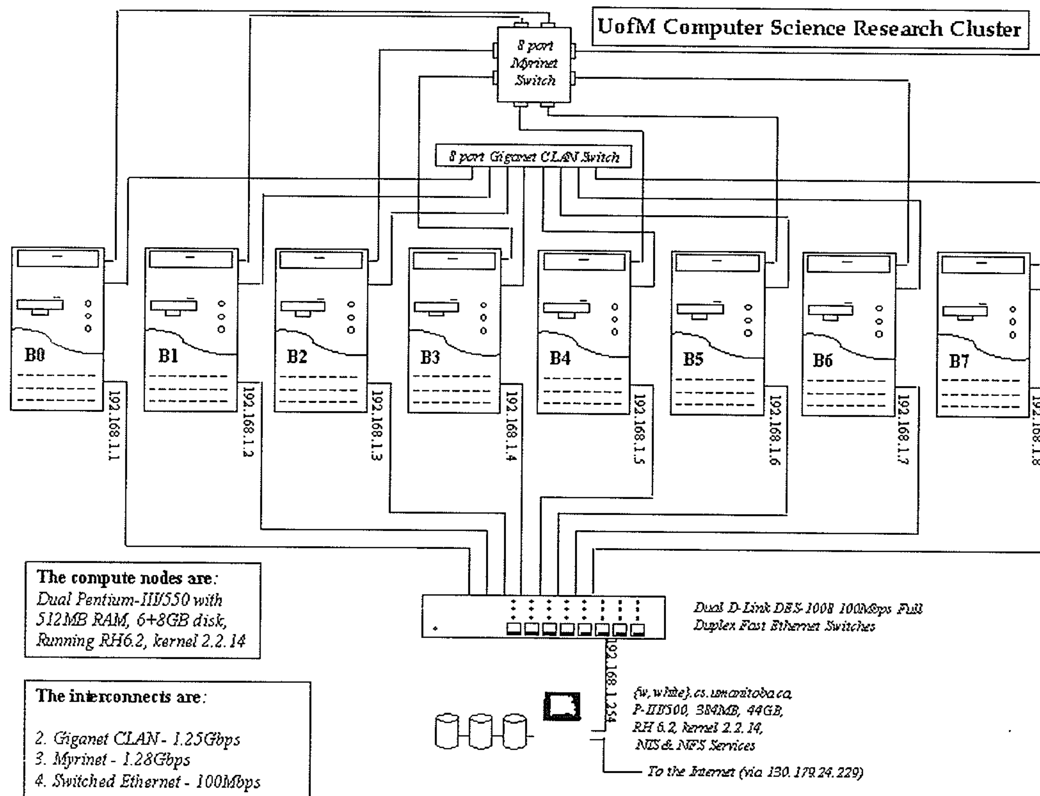


Figure 5.1 Platform Used

The experiments were run using a 100 Mbps Fast Ethernet network. The Fast Ethernet connection has a high communication latency. Therefore communication differences between the hand coded and MPI Buddy coded version of the application will be obvious. The Myrinet and Gigabit connections for the cluster were not operational at the time of this work.

5.4 2D Discrete Wavelet Transform

5.4.1 Introduction

Wavelet transforms have been successfully used for the multiresolution representation of one-dimensional and multi-dimensional signals. The most common application of this representation is data compression. The new image compression standards such as JPEG 2000 [MGBB00] use the wavelet transform prior to the encoding step. Data compression is not the only application of the

wavelet transforms. They have been used in many different areas, such as data analysis, numerical analysis, signal processing and image processing. In such applications, the problem size is typically large and the computation of the wavelet transform can be prohibitively costly, even though its complexity is linear in the number of input samples, with a constant factor that depends on the length of the filter. However, the algorithm is well suited for parallel processing because of the repetitive nature of the computations and the regularity of the
input data [ChCC02].

5.4.2 Analysis of the Problem

The orthogonal wavelet decomposition of a 2D signal x[n,m] can be computed by convolving with a filter in one dimension (rows), retaining every other column, followed by convolution of the resulting signal by another filter in the other dimension (columns) and retaining every other row [KHTG98]. With a pyramid algorithm, further stages of the 2D-wavelet transform can be computed by recursively applying the procedure to the approximation coefficients of the previous stage (LL subband). The 2D DWT algorithm will produce four subbands at every stage, each one is 1/4 the size of the input data size to that stage. These four subbands are denoted HH, HL, LH, and LL with the first letter denoting the filter used horizontally and the second letter the filter used vertically (H=High pass, L=Low Pass). One stage of the 2D DWT is shown below in Figure 5.2.

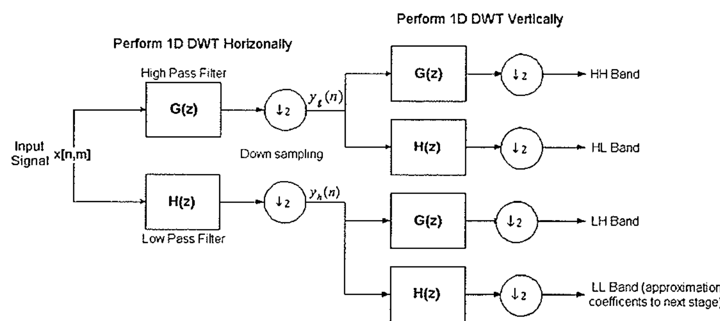


Figure 5.2 One Stage of 2D Discrete Wavelet Transform

The image is assumed to be periodic in nature along both rows and columns when performing the 2D DWT. This infers that should the convolution need terms beyond the edge of the image, those

terms are supplied from the other side of the image. For the 2D DWT discussion, the images are assumed to be of size NxN and the number of processors is P.

5.4.3 Parallel Decomposition Strategy

The design strategy for the parallelization of the wavelet transform algorithm should be capable of meeting the requirements for execution speedup by minimizing the inter-processor communications. While developing the parallel algorithm, we must consider the fact that the computational complexity decreases exponentially at each stage. The crucial point in the implementation of the algorithm is to choose the optimal distribution of the initial two-dimensional data and the intermediate subband coefficients on the processors.

The approach is to distribute the data columnwise such that each processor contains N/P columns of data and then perform the convolution in the x-direction to obtain the approximation and detail coefficients for the local data on each processor. Only the border data is to be communicated between neighboring processors as the information about a particular position is mapped to a particular processor. Now all the necessary data to perform the convolutions in the y direction exists locally. A detailed analysis of this strategy is given in [PaJa96]. This strategy requires a communication step with an amount of data proportional to the length of the filter. Exactly D-2 terms must be communicated from each row, where D is the size of the filter [NiHe00]. The wavelet transformation occurs locally at each processor. Consequently each processor will have a local LL, LH, HL, and HH subband. The communication approach is given in Figure 5.3 for the algorithm using 3 processors (P=3).

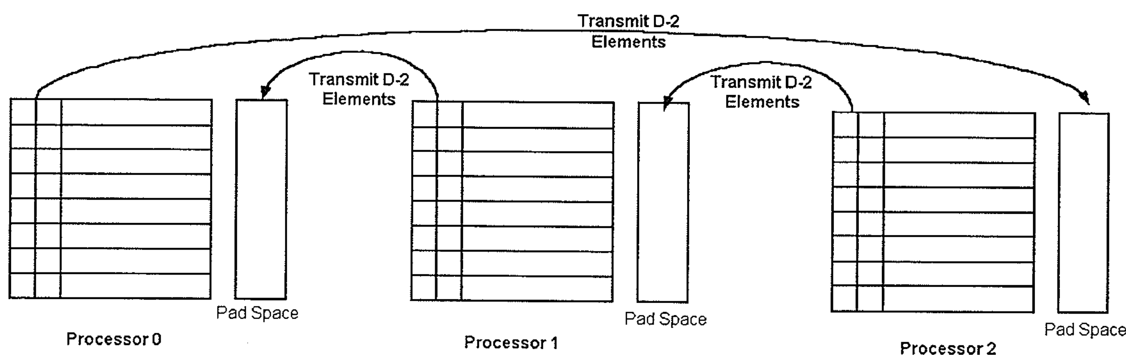


Figure 5.3 Communication approach (3 processors)

The total communication time required to perform a K-level wavelet transform will be

$$\begin{aligned} T_{comm} &= \sum_{k=1}^{K} 2t_s + \frac{(2D-4)}{2^{k-1}} N t_w \\ &= 2Kt_s + (4D-8)(1-\frac{1}{2^K}) N t_w \end{aligned} \tag{5.4}$$

in which t_s is the startup time for the message, t_w is the time required to transfer a double precision number through the communication link, and N is the size of the message. In each stage of the algorithm, the approximation coefficients of the previous stage are used for computation of new subband coefficients. The amount of computation decreases by a factor of 4 at each stage. Assuming that addition and multiplication operations require a time of t_c then the time for computation can be estimated as

$$T_{comp} \approx \frac{8}{3}(D+1)\frac{N^2}{P} t_c \tag{5.5}$$

since $\frac{N^2}{P}$ elements will be mapped onto each processor, each requiring D+1 multiplications and D+1 additions at each stage of the wavelet decomposition. The sequential implementation has the complexity $\Theta(N^2)$ and the above parallel implementation will have a processor-time product $\Theta(N^2)$. Hence, the parallel implementation will be cost-optimal. The speedup for this implementation can be estimated as

$$S(NxN, P) \approx \frac{1}{\frac{1}{P} + \frac{8(4D-8)t_w}{3(D+1)Nt_c}} \tag{5.6}$$

When N is increased, a nearly linear speedup should theoretically be observed with this implementation. If the system allows for overlapping computation and communication steps, better speedup results can be obtained.

5.4.4 Approach to Solving Problem using MPI Buddy

This problem was solved using the same algorithm as the hand coded parallel version. The steps in coding using MPI Buddy are as follows:

1. Create the MPI Header/Ender by selecting Design Patterns->New MPI File from the menu bar. The default settings are selected and a functional MPI C code file is created.

2. Save the file as a C file so that updates can be made. Select : File->Save As. From the menu bar and enter the file name desired. The file may now be save whenever deemed necessary by selecting : File ->Save .

3. Use the algorithm to select the desired communication structure. From Figure 5.3, it is obvious that the data needs to be divided columnwise, with N/P columns existing on each processor. Therefore 2D Scatter/Gather is selected from Design Patterns->Static Master Slave-> 2D Scatter/Gather. The appropriate parameters are then filled (Figure 5.4).

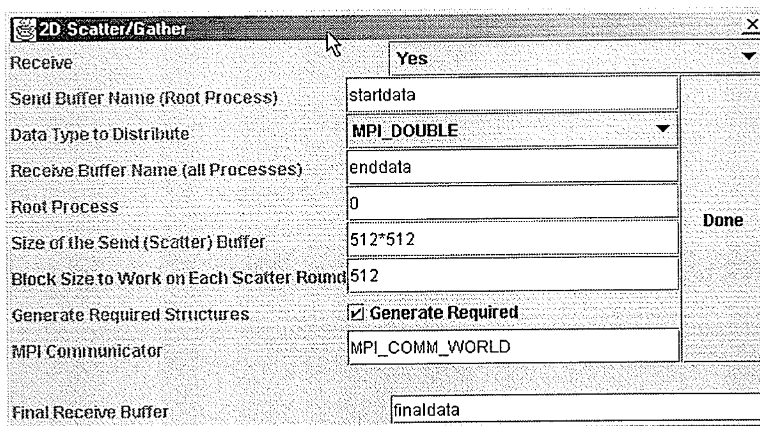


Figure 5.4 Selecting 2D Scatter/Gather design pattern parameters

4. Add in necessary application specific code for the program and define user parameters and variables to be used as shown in Figure 5.5. This also involves the use of other .h files that may need to be used. The "mpi.h", "stdlib.h", and "stdio.h" header files were included in the step 1 automatically.

```
C:\incomplete\waveletbuddyfile.c
File  Design Patterns  Compile  Edit  Help
        locdata=malloc(totalsize/p*sizeof(double));
        enddata=malloc(totalsize*sizeof(double));
        hsize=512;
        loc_hsize=hsize/p;
        vsize=512*512/hsize;

        if (my_rank==0) {
                /*User Should define mainImage->data here*/

                mainImage=(image*)malloc(sizeof(image));
                inputfile=malloc(81);
                outputfile=malloc(81);
                inputfile="barbara.pgm";
                outputfile="outfile.pgm";
                steps=3; /*number of steps in the transform*/
                getheader(mainImage,inputfile);
                startdata=mainImage->data;
        }

        thefilter=makedaub(FALSE,daub8coeffs,8,0);
        validwavelet=initwavelet(thefilter);
Compilation Results
Caret position : 1952   Current Line Number: 65   Total Lines of Code: 187
```

Figure 5.5 Adding user code to the 2D DWT program

The use of other design patterns is valid, but not necessary in this case. For the MPI_Bcast 's required, the user may insert these by hand or use the Design Patterns->Basic Functions->MPI_Bcast selection from the menu bar.

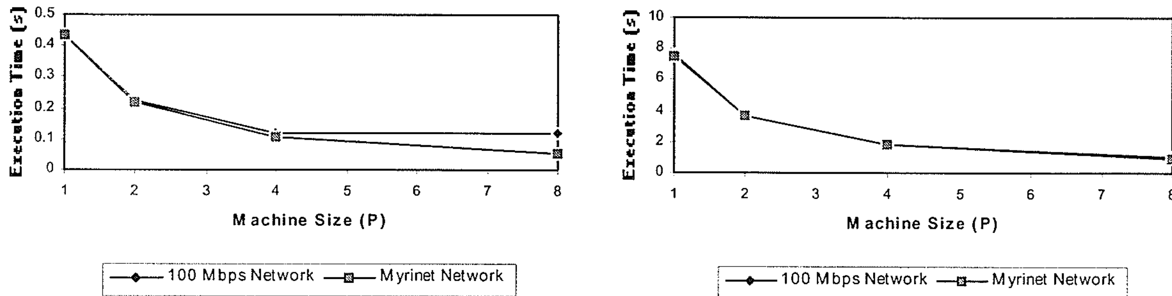
5. Final code is obtained. The code may be compiled from within MPI Buddy if the underlying operating system is installed with a working version of MPI.

The code for sequential, hand coded parallel, and MPI Buddy coded versions of the application are given in Appendix A.

5.4.5 Objective Analysis of the Tool

The code from the hand coded version and the MPI Buddy coded version appear to be similar in structure with the hand coded version. It is expected that the two version will perform equally well. Past experiments have shown that a linear speedup is observed for both the MPI Buddy

coded version and the hand coded version [ChCC02]. Figure 5.6 gives plots of execution time versus machine size using a filter size of 6 for 3 stages of the 2D DWT algorithm.

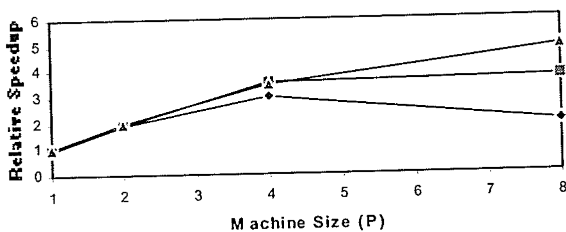


a) 512x512 Image **b)** 2048x2048 Image

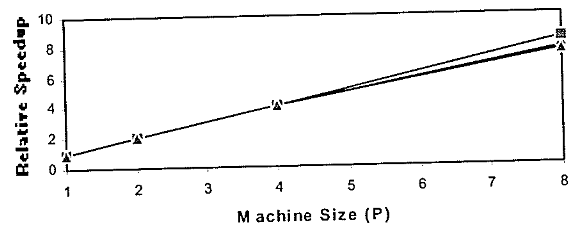
Figure 5.6 DWT Execution time versus machine size (D=6) [ChCC02]

From these plots, it can be seen that as the image size becomes larger, the execution time of the implementation reduces almost linearly with an increasing number of processors. There is a steeper drop off in execution time for the Myrinet interconnection with an increase in machine size than for the 100 Mbps fast ethernet connection. This is the result of the properties of the Myrinet connection that allow it to function with greater data transmission speed and lower latency. For the largest image size used (2048x2048), the 100 Mbps fast ethernet connection performed almost as well as the Myrinet interconnection as computation dominated greatly over communication.

Plots of speedup versus machine size are given in Figure 5.7 using a filter size of 6 for both the Fast Ethernet and Myrinet interconnects for 3 stages of the algorithm. The results from running the simulations using the Fast Ethernet connection reveals that the speedups are compromised by excessive communication time, except for the larger image sizes. Using the Myrinet network, almost linear speedups were obtained for all the image sizes.



a) 100 Mbps Fast Ethernet



b)Myrinet Interconnection

Figure 5.7 DWT Speedup versus machine size (D=6) [ChCC02]

The above discussion pertains to the hand coded version run over the same cluster with functional Myrinet and fast ethernet connections. During this work, the Myrinet connection was not functional. Also, much of the parallel communication occurs before and after the 2D DWT itself in order to have all the results on a single processor. The analysis employed to measure the ability of MPI Buddy to produce efficient communication code is to include all communication including the initial scatter and final gather for both the hand coded and MPI Buddy coded versions. These result appear in Table 1. For a machine size of 8, there is a slowdown using the 100 Mbps network as communication dominates the execution time. The results show the MPI Buddy Coded version performs as well as the hand coded version.

	1 Processor	**2 Processors**	**4 Processors**	**8 Processors**
Hand Coded	0.580198	0.447595	0.448913	0.769215
MPI Buddy	0.576976	0.44694	0.441617	0.76683

Table 5.1: Timings (in seconds) for 2D discrete wavelet transform program (512x512 image, filter size 8)

5.4.6 Subjective Analysis of the Tool

The 2D discrete wavelet transform is a problem which can be decomposed easily in parallel. The tool provides the 2D Scatter/Gather functionality which is the key aspect in decomposing the problem. Data type support is provided for all the basic MPI types and user defined types so the

wavelet program could be constructed efficiently. In addition, the basic generation of the header saves programming time and allows the developer to concentrate on the problem at hand.

The time spent programming the main code of the 2D DWT was approximately 1 hour using the MPI Buddy system and considerably longer coding by hand. This metric is difficult to measure as likely the time to program an application will depend on the user's familiarity with the application, MPI Buddy, and MPI. There are advantages though for the parallel decomposition of the problem using MPI Buddy such as it is less error prone than hand coding.

5.5 Fast Fourier Transform

5.5.1 Introduction

The fast Fourier transform (FFT) is basically a fast discrete Fourier transform (DFT). These functions operate on a discrete set of data transforming the time domain to the frequency domain, where further processing often follows. If there is a periodic sequence x(n) with period N, then the Fourier series representation output will consist of N harmonically related exponential functions. Fourier transforms can represent naturally occurring signals well, but perform more poorly than wavelets in discerning singularities and edges. The FFT is designed to work on data sizes with the number of points being a power of 2. [Walk96 , PrMa96]. Fourier transforms find applications in signal processing, and financial forecasting among others.

5.5.2 Analysis of the Problem

The FFT algorithm is derived by starting with the discrete Fourier transform:

$$X_k = \frac{1}{N}\sum_{j=0}^{N-1} x_j w^{jk} \text{ where } w = e^{-2\pi i/N} \text{ and } N = \text{input data size} \qquad (5.7)$$

The capital letter X denotes frequency domain signal and the lower case x, the time domain signal. The summation may be divided into two parts, even and odd:

$$X_k = \frac{1}{N}\left[\sum_{j=0}^{(N/2-1)} x_{2j} w^{2jk} + \sum_{j-0}^{(N/2)-1} x_{2j+1} w^{(2j+1)k}\right]$$

$$X_k = \frac{1}{2}\left[\frac{1}{(N/2)}\sum_{j=0}^{(N/2)-1} x_{2j} w^{2jk} + w^k \frac{1}{(N/2)}\sum_{j=0}^{(N/2)-1} x_{2j+1} w^{2jk}\right] \quad (5.8)$$

$$X_k = \frac{1}{2}\left[\frac{1}{(N/2)}\sum_{j=0}^{(N/2)-1} x_{2j} e^{-2\pi i\left(\frac{jk}{N/2}\right)} + w^k \frac{1}{(N/2)}\sum_{j=0}^{(N/2)-1} x_{2j+1} e^{-2\pi i\left(\frac{jk}{N/2}\right)}\right]$$

Each summation point is a N/2 DFT operating on N/2 even points and N/2 odd points respectively. Therefore, the complete sequence can be divided into 2 parts:

$$X_k = \frac{1}{2}\left[X_{even} + w^k X_{odd}\right] \quad (5.9)$$

The complete sequence k=0,1, ...N can be calculated by dividing it into 2 parts as shown by equations 10 and 11.

$$X_k = \frac{1}{2}\left[X_{even} + w^k X_{odd}\right] \quad (5.10)$$

$$X_{k+N/2} = \frac{1}{2}\left[X_{even} + w^{k+N/2} X_{odd}\right] = \frac{1}{2}\left[X_{even} - w^k X_{odd}\right] \quad (5.11)$$

This approach works as $w^{k+N/2} = -w^k$ where $0 \le k < N/2$. X_k and $X_{k+N/2}$ can be computed using two N/2 point transforms. When performing the iterative FFT, each of the N/2 point transforms can be decomposed into two N/4 point transforms and the decomposition should continue until only single points are to be transformed. A one point transform is simply the value of the point. The twiddle factors w are found by recognizing that as the number of terms reduces by a factor of 2, the powers of w increase by a factor of 2 (i.e. $w = e^{-2\pi i/N}$).

The algorithm is shown in Figure 5.8a for a 8 point FFT. The terms are to be arranged in reverse bit order when first performing the 1D DFT. This requirement ensures that the end result will be in the correct order. At every level there is a butterfly computation (shown in Figure 5.8b).

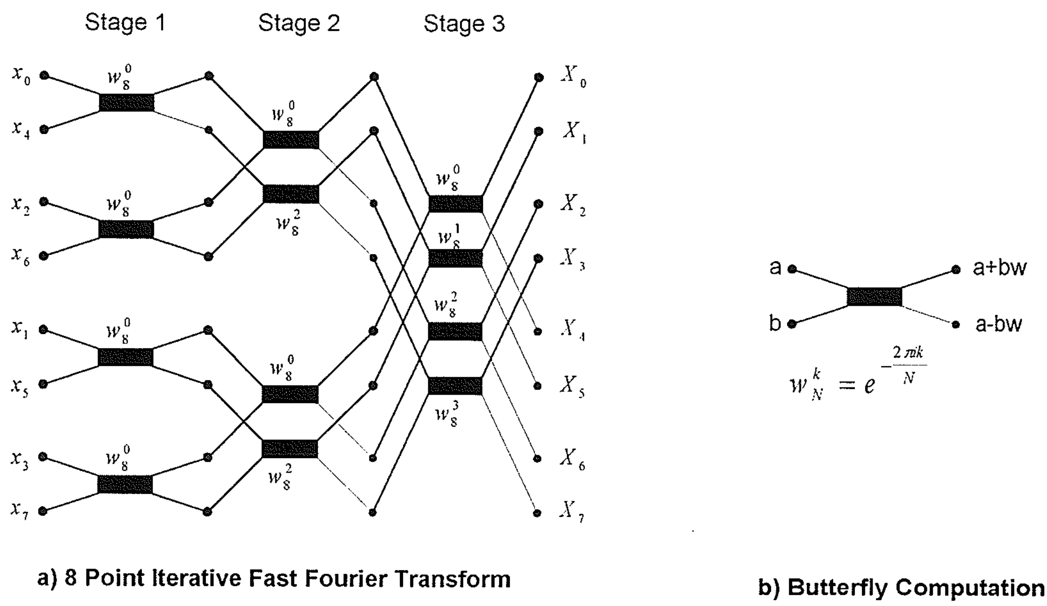


Figure 5.8 Iterative fast Fourier transform (FFT)

Each butterfly computation involves one complex multiplication and two complex additions. For a data size of $N = 2^x$, there are N/2 butterflies per stage of the computation process and $\log_2 N$ stages. For the sequential case, the complexity will be of $O(N \log_2 N)$ as there are a total of $(N/2)\log_2 N$ complex multiplications and $N \log_2 N$ complex additions.

5.5.3 Parallel Decomposition Strategy

This algorithm can be readily decomposed easily in parallel. The data is scattered so that processor 0 will receive the first N/P elements, processor 1 will get the next N/P elements, etc, where N denotes the number of terms in the initial array. Following this part, every processor will effectively perform a FFT simultaneously on the data it has received. Upon the completion of the local FFTs, the results are gathered into processor 0. Now the remainder of the FFT proceeds until completion. The parallel algorithm is illustrated in Figure 5.9.

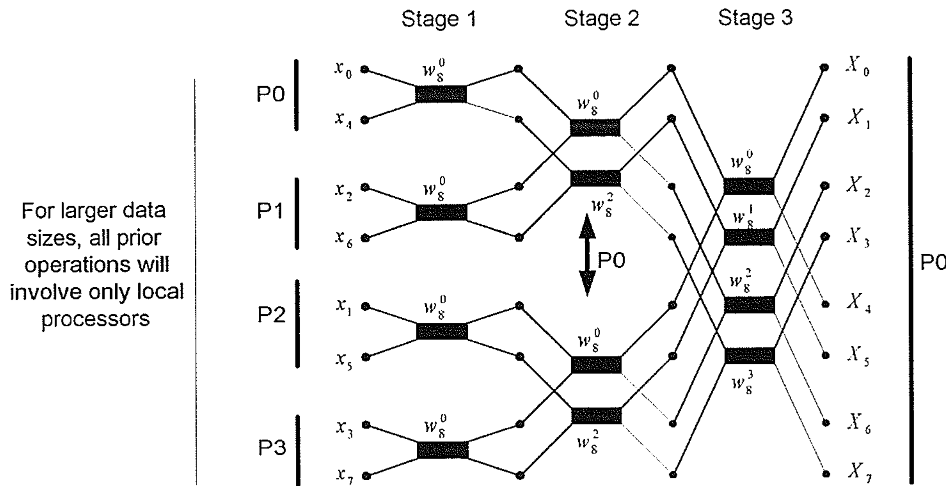


Figure 5.9 Parallel FFT algorithm (4 processors, 8 data elements)

The algorithm described performs well because it greatly reduces the amount of non-local computation. There are a total of log(N) stages for this algorithm, and only the last log(P) stages will proceed using the root processor alone. For instance if there are 4 processors and 2^{16} data element, then 2^{14} FFT iterations will proceed locally and only 2 iterations will proceed using the root processor alone. The fact that the processor numbers are limited allows for the bulk of the program to proceed in parallel. Indeed, for large data sizes, this parallel algorithm performs significantly better than its serial counterpart.

The analysis of the parallel decomposition strategy can be understood by first assuming that there are P processors available that are able to work in parallel. The time required to execute the first FFT iterations in parallel is $\frac{3N}{2P}\log(N/P)$, given that there are log(N/P) stages involving all processors and the there are two complex additions and one complex multiplication per butterfly stage. Only one parallel communication is required that sends the data from all the slave processes to the master. Assuming only one processor can send data through the communication link at a time, the communication time can be approximated by:

$$T_{comm} = (P-1).t_s + N.\frac{(P-1)}{P}t_c \qquad (5.12)$$

where t_s is the startup time to send a complex number and t_c is the time required to transfer a complex number through the communication link. For the last log(P) stages, only the root processor computes the butterfly computations. The computation for this part is simply :

$$T_{root} = N\log_2 P + \frac{N}{2}\log_2 P \qquad (5.13)$$

The overall time requirement for the parallel version is therefore :

$$T_{parallel} = (P-1).t_s + N\frac{(P-1)}{P}t_c + \frac{3N}{2P}\log_2(N/P) + \frac{3N}{2}\log_2 P \qquad (5.14)$$

This is a significant improvement over the sequential version of the FFT, particularly for larger data sizes.

5.5.4 Approach to Solving Problem using MPI Buddy

This problem was solved using the same algorithm as the hand coded parallel version. The steps in coding using MPI Buddy are as follows:

1. Create the MPI Header/Ender by selecting Design Patterns->New MPI File from the menu bar. The default settings are selected and a functional MPI C code file is created.

2. Save the file as a C file so that updates can be made. Select : File->Save As. From the menu bar and enter the file name desired. The file may now be save whenever deemed necessary by selecting : File ->Save .

3. Use the algorithm to select the desired communication structure. From Figure 5.9, it is evident that the data needs to be 1D scattered, with N/P consecutive elements going to each processor. One can assume this data will already be in a bit reversed order so long as this is hand programmed later into the program. 1D Scatter/Gather is selected from Design Patterns->Static Master Slave-> 1D

Scatter/Gather. The appropriate parameters are then filled in (Figure 5.10). The data structure Complex is defined in complex.h so the data type is "Other/User Defined".

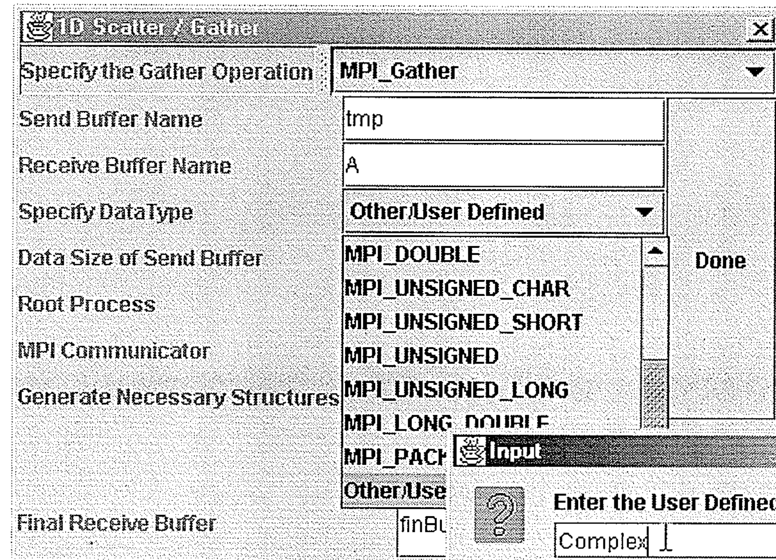


Figure 5.10 Selecting 1D Scatter/Gather design pattern parameters

4. Add in necessary application specific code for the program and define user parameters and variables to be used. This also involves the use of other .h files that may need to be used. Most of this program relies on the user knowledge of the algorithm, thus a great deal of hand coding is necessary.

5. Final Code is obtained. The code may be compiled from within MPI Buddy if the underlying operating system is installed with a working version of MPI.

The code for the sequential, parallel ,and MPI Buddy versions of the FFT are provided in Appendix B.

5.5.5 Objective Analysis of the Tool

Again, the MPI Buddy coded version of the application appears to run as fast as the hand coded version, likely due to the fact the two programs were constructed using the same algorithm.

The timing results for running the hand coded FFT implementation on different machine sizes using a data size of 2^{20} are provided in Figure 5.11. The corresponding speedup graph is illustrated in Figure 5.12.

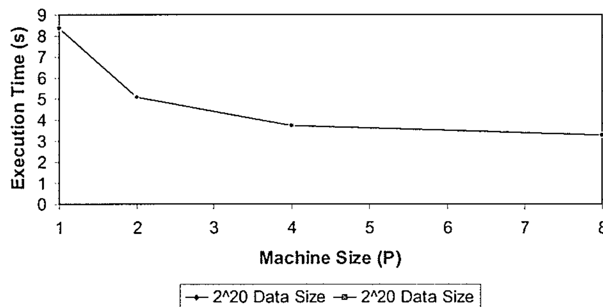


Figure 5.11 Execution Time versus machine size for parallel FFT

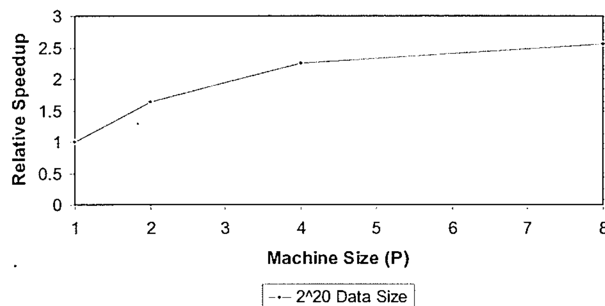


Figure 5.12 Speedup versus machine size for parallel FFT

Using the timings for the entire program and not just the FFT core, a comparison of the performance between the hand coded and MPI Buddy coded versions can be made. Using the full programs was necessitated by the fact the code from the hand tailored version differed from the MPI Buddy coded version in where communication structures and other program parts were placed. The average timing results of running the entire 1D FFT program for a data size of 2^{20} are shown in Table 2.

	1 Processor	**2 Processors**	**4 Processors**	**8 Processors**
Hand Coded	10.06795	7.514335	6.442555	6.163328
MPI Buddy	9.989772	7.540516	6.382316	6.123404

Table 5.2 : Timings (in seconds) for FFT applications (N=20, data size 2^20)

It is evident, that the two programs perform equally well. The 100 Mbps interconnection used results in progressive speedups for increases in machine size, but there are diminishing returns as communication begins to dominate for larger machine sizes when using a fixed problem size.

5.5.6 Subjective Analysis of the Tool

The FFT can be highly parallelized, but the actual process of doing so is fairly complex. Again, MPI Buddy provides the necessary design pattern that makes the task easier. Support for user defined data types gives one the flexibility to use specific C structure types and MPI Buddy takes care of scattering the structure automatically. The code for the 1D scatter/gather is easily generated. Again ,the benefits discussed for the 2D DWT implementation also apply.

The main observed benefit in programming this application using MPI Buddy was the automatic generation of the header/ender and the automatic generation of the MPI communication code. However, the time benefit was minimal, if any, over the use of more common C programming environments for the experienced MPI user.

5.6 Overall Analysis of the Tool

The analysis of the tool which can be inferred from the experiments conducted is that the tool performs well in coding small parallel applications. This is expected to be especially true in generating automatic MPI code for the beginner user. The benefits become more pronounced when coding difficult communication structures such as the 2D scatter/gather, as opposed to simpler structures (i.e. 1D scatter/gather). For all communication structures, efficient communication code is generated. MPI Buddy is advantageous in that it gives the user the ability to program MPI applications from almost any platform, a change from the programming systems discussed in chapter 3. This tool can useful for the rapid prototyping of data-parallel signal/image processing applications.

Drawbacks experienced were that the high level programming model may confuse the user when parameters are entered into the design pattern GUI. As well, the user is expected to enter application specific code himself/herself into the skeleton code which is generated. This is fine for those who have worked with MPI, but it may prove difficult for the beginner user. Also, there was not much support for inserting MPI design patterns into procedures other than the main() procedure. This was a limitation inherit in the design of the MPI Buddy system which automatically generates needed variables, but only in the main() program procedure. The lack of a color coded API for data types, automatic indentations, and other features found in such APIs as Visual C++ made sequential additions uncomfortable. The lack of an advanced GUI that allows the user to program more naturally appears to be the limiting factor with respect to ease of use for the user. It is well documented that providing the user with graph models for programming facilitates ease of use [Siu96], but time constraints did not facilitate this implementation.

Chapter 6

Conclusions and Future Work

6.1 Review of this Work

Parallelism is not inherent on any computer system by default. Support should be available at the hardware, operating system, and developmental levels. Provided that support does exist, parallelism can be exploited while programming the application, at compile time, and during execution. Various parallel computer architectures have been developed, most of these taking the form of expensive supercomputers which utilize multiple processors simultaneously. The demand for increased computational power at reasonable cost has led to the concept of distributed clusters. These "machines" often take the form of a network of conventional workstations interconnected through high speed networks. The use of these computing environments is widespread in certain scientific and engineering domains, but have not proliferated further because of the lack of appropriate programming tools to aid the novice parallel programmer.

Approaches to overcoming the complexity of parallel programming are to raise the level of abstraction and provide tools that simplify repetitive tasks. Rasing the level of abstraction can be done by providing the user with message passing libraries, abstractions on top of message passing libraries, and other unique methods. Two message passing libraries, MPI and PVM, have emerged as standards for parallel programming. In addition, complex programming systems have been developed to generate parallel code easily. Some of these systems use PVM or MPI underneath, while others directly socket program processor communication using C. A shortcoming observed with all the systems studies is that they are platform dependent. The vast majority of these systems were also closed in their design, indicating that inserting user code was difficult to impossible. MPI Buddy, a open platform non-specific MPI programming tool was proposed as a solution.

MPI Buddy was designed as a design pattern based, layered open system with a level of abstraction above MPI. It was constructed using Java and possesses a modular design allowing new design pattern modules to be added with ease. The intent was allow MPI Buddy to possess a user friendly interface, openness, moderate extensibility, and portability. In addition, the tool was intended to generate optimal communication code and be able to test code syntax from within. The design patterns incorporated were chosen from the most commonly used parallel communication and decomposition schemes. They include the 1D Scatter/Gather, Balanced 1D Send/Receive, 2D Scatter/Gather, Block Cyclic Send/Receive, Cyclic Send/Receive, and 1D Divide and Conquer approaches. Another design pattern, Dynamic 1D Master/Slave, was included for use on heterogeneous clusters when the individual job completion times are unpredictable.

The programming model for MPI Buddy is user friendly. The user starts out by indicating from the menu bar that a new MPI file is to be created. Following this, the desired design pattern is selected from the design pattern menu and automatically inserted into the C code. The user may modify the code as desired. Finally ,the developer saves the code. Providing the operating system environment has MPI installed, compilation can also be completed, with the results piped to the MPI Buddy window.

Code was produced using MPI Buddy for applications involving the 2D discrete wavelet transform and the 1D fast Fourier transform. The MPI Buddy coded applications and hand coded applications were compared for performance differences on a dedicated 100 Mbps fast ethernet connected cluster. There were no significant differences in the run times of the applications which lends evidence to the notion that MPI Buddy performs as well as hand coded versions of the same application. With respect to subjective measures, it was determined that the tool is a time saver for small applications, particularly when the user is a beginner MPI developer possessing C programming knowledge. Consequently, the tool can be useful for educating novice programmers on parallel programming techniques.

6.2 Future Work

The experience of using the tool has allowed for the identification of shortcomings which can be improved on. These have been identifies as :

6.2.2 Better GUI

Though the user interface is sufficient to program the majority of the parallel problems faced in computer engineering, the menu driven GUI is not as natural as a user controlled graph model that allows the user to naturally represent communication by extending lines between blocks (representing processes).

6.2.3 Automatically Color Code MPI and C Keywords

With MPI Buddy, keywords and data types are simply incorporated as regular code. Those who have worked with Windows 98 know that using Notepad for programming can be very difficult, while the use of Visual C++ makes programming easier. If the code produced can automatically highlight C types and MPI types in different colors, programming will be easier. Similarly, allowing the code to automatically indent itself if one is working in a loop will make programming easier.

6.2.4 Integrate a Performance Visualization Tool

The developer should be able to evaluate the performance of the code produced using MPI Buddy easily from within the tool. This is a standard element of more integrated programming environments. If this integration is made, then the user can modify the code as necessary to easily achieve the desired application functionality and omit parallel bottlenecks.

6.2.5 Include Additional Parallel Design Patterns

Support for the most common parallel design patterns was included in the MPI Buddy system, but by no means is this a complete pattern catalogue. There are many other complex patterns which could be included in the system such as a 2D dynamic master/slave approach.

6.2.6 Add Support for Other MPI Communication

The communication code produced by MPI Buddy is close to optimal, providing the target is a cluster of one processor machines. However, there are some clusters that use multiprocessor computers. In these configurations, often one processor communicates while the others compute. For these cases, providing coding support for non-blocking communication would result in betters speedups

6.3 Conclusion

This thesis has reviewed the popular tools used in reducing the difficulties associated with parallel programming. A Java Implemented open design-pattern based system, MPI Buddy was developed and tested. This tool was determined to be useful for prototyping data parallel applications in the field of signal and image processing. The main benefit of the tool was portability across different computer platforms. The tool performed as expected, with the benefits more pronounced for small parallel applications involving complicated communication. Intended improvements of the system have been listed as future works.

References

[AMMV98] Rocco Aversa, Antonio Mazzeo, Nicola Mazzocca, Umberto Villano, "Heterogenous System Performance Prediction and Analysis Using PS", IEEE Concurrency, pp.20-29, July-September 1998.

[ArCu86] Arvind and D. E. Culler, "Dataflow Architectures," Annual Reviews in Computer Science, Annual Reviews Inc., Palo Alto, CA ,pp. 225-253, 1986.

[BaST89] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum, "Programming Languages for Distributed Computer Systems", ACM Computing Surveys, September 1989.

[BCDP95] A. Bartoli, P. Corsini, G. Dini, C.A. Prete, "Graphical Design of Distributed Applications through Reusable Components", IEEE Parallel and Distributed Techonology, vol. 3, no. 1, pp. 37-51, Spring 1995.

[BDGM94] A. Beguelin, J. Dougarra, G.A. Geist, R. Mancheck, and V. Sunderarn, *HeNCE: A User's Guide*. Carnegie Mellon University and Oak Ridge National Laboratory, June 1994.

[Beck96] Alan Beck, "Visual Programming May Come of Age with CODE", *HPCwire*, viewed on http://www.cs.utexas.edu/users/code/CODE-HPCwire-article.html , August 23, 1996.

[Berg02] Emery Berger, "The CODE Visual Programming System", University of Texas Austin, http://www.cs.utexas.edu/users/code/ , current as of April 23, 2002.

[BEST99] The BEST Group, "The Balance System v1.0.2", http://balance.cyber.ust.hk/intro.html , August 1999.

[BHDM95] J.C. Browne, S. Hyder, J. Dongarra, K. Moore, and P. Newton, "Visual Programming and Debugging for Parallel Computing", *IEEE Parallel and Distributed Technology*, vol 3, no. 1, pp. 75-83, 1995.

[Buyy99] Rajkumar Buyyam, High Performance Cluster Computing: Architectures and Systems , vol. 1, Prentice Hall PTR, NJ, USA, 1999, 881pp.

[ChCC02] Narjit Chadha, Aysegul Cuhadar, and Howard Card, "A Parallel Implementation of the 2D Discrete Wavelet Transform", *Proc. 20th IASTED International Multi-Conference on Applied Informatics*, Innsbruck, Austria, February 18-21, 2002.

[CGMS94] N. Carriero, D. Gelernter, T. Mattson, and A. Sherman, "The Linda Alternative to Message-Passing Systems", *Parallel Computing*, vol. 20, no.4, pp. 633-655, 1994.

[CoGG00] W.E. Cohen, W.D. Garrett, and R.K. Gaede, "Parallel Program Traces for Accurate

Prediction of Proposed Cluster Performance", Proceedings of the Second Workshop on Cluster-Based Computing, http://www.crhc.uiuc.edu/~steve/wcbc00/ , May 2000.

[Comp02] Compaq Computer Corporation, "Titanic Sets Sail with Alpha", http://www.compaq.com/hpc/film/titanic.html , current as of February 27, 2002.

[DeDe99] H.M. Deitel and P.J. Deitel, Java, How to Program.(3rd ed.). Upper Saddle River, NJ: Prentice Hall, 1999,1355 pp.

[Dinc98] Kivanc Dincer, "jmpi and a Performance Instrumentation Analysis and Visualization Tool for jmpi", Europar-98, Southampton, UK, 1998.

[Dunc90] Ralph Duncan, "A Survey of Parallel Computer Architectures", *IEEE Computer*, pp. 5-16, February 1990.

[Ente02] The Enterprise Group, "The Enterprise Programming Environment", Software Systems Research Group, Department of Computer Science, University of Alberta, http://www.cs.ualberta.ca/~systems/enterprise-overview.html , current as of May 2, 2002.

[GeKP96] G.A. Geist, J.A. Kohl, and P.M. Papadopoulos, "PVM and MPI: a Comparison of Features", *Calculateurs Paralleles* , Vol. 8 No. 2, 1996.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Glob00] The Globus Project, "The Globus Quick Start Guide v1.1.3", http://www.globus.org/toolkit/documentation/QuickStart.pdf , September 2000, 38 pp.

[GoSP01] Dhrubajyoti Goswami, Ajit Singh, and Bruno Richard Preiss, "Building Parallel Applications using Design Patterns", A chapter in the upcoming book: "Advances in Software Engineering: Topics in Comprehension, Evolution and Evaluation", Springer-Verlag, New York, 2001, 24 pages.

[GoSP99] Dhrubajyoti Goswami, Ajit Singh and Bruno R. Preiss, "Architectural Skeletons: The Re-Usable Building-Blocks for Parallel Applications". *In proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pp. 1250-1256, Las Vegas, June 1999.

[GrLu96] William Gropp and Ewing Losk, *Users Guide for mpich, a Portable Implementation of MPI - Version 1.2.1.* Argonne National Laboratory, University of Chicago, 1996, 68 pp.

[GrLu97] William Gropp and Ewing Losk, "Why are PVM and MPI so Different", *Proceedings of the 4th European PVM/MPI Users' Group Meeting,* November 1997.

[GrOh98] Thomas Gross and David R. O'Hallaron, *iWarp -Anatomy of a Parallel Computer.* MIT Press, 1998, 530pp.

[HeFi97] Michael Heath and Jennifer Finger, "ParaGraph: A Tool for Visualizing Performance of Parallel Programs", National Center for Supercomputing Applications, http://www.ncsa.uiuc.edu/Apps/MCS/ParaGraph/manual/manual.html , May 1997.

[HiTa72] R.G. Hintz and D.P. Tate. "Control Data STAR-100 Processor Design", *Proc. Compcon 72'*, IEEE Computer Society Conference, New York, pp. 1-4, 1972.

[IMMN95] Paul Iglinkski, Steve MacDonald, Chris Morrow, Diego Noviillo, Ian Parsons, Jonathan Shaeffer, Duane Szafron, and David Woloschuk, *Enterprise User's Manual - Version 2.4.* Department of Computer Science, University of Alberta. 1995, 65pp.

[KHTG98] A. Khokhar, G. Heber, P. Thulasiraman, and G.R. Gao, "Load Adaptive Algorithms and Implmentations for the 2D Discrete Wavelet Transform on Fine-Grain Multithreaded Architectures", *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, Session 14, IEEE, 1998.

[KoGe95] J.A. Kohl and G.A. Geist, *XPVM 1.0 Users Guide*. Computer Science and Mathematics Division. Oak Ridge National Laboratory, April 1995.

[Kohl02] Jim Kohl, "XPVM: A Graphical Console and Monitor for PVM", http://www.netlib.org/utk/icl/xpvm/xpvm.html , current as of April 2002.

[Kran96] Dieter Kranzlmuller, "Debugging Massively Parallel Programs with ATEMPT", www.gup.uni-linz.ac.at:8001/papers/abstracts/Kran96c.html , July 1996.

[Kris89] E.V. Krishnamurthy, *Parallel Processing - Principles and Practice.* Addison-Wesley, Singapore, 1989, 332 pp.

[LAM01] LAM / MPI Parallel Computing, "XMPI -- A Run/Debug GUI for MPI", http://www.lam-mpi.org/software/xmpi/ , December 2001.

[Losh94] David Loshin, High Performance Computing Demystified, AP Professional, Cambridge MA, 1994, 261 pp.

[MGBB00] M. Marcellin, M. Gormish, A. Bilgin, and M. Boliek, "An Overview of JPEG-2000", *Proc. Data Compression Conference*, J.A. Storer and M. Cohn, eds., Snowbird, Utah, pp.523-541, Mar.28-Mar.30, 2000.

[Mpc02] mpC Team, "The mpC Parallel Programming Environment", Institute for System Programming, Russian Academy of Sciences, http://www.ispras.ru/~mpc/ , current as of March 28, 2002.

[NeBr92] P. Newton and J.C. Browne, "The CODE 2.0 Graphical Parallel Programming Language", *Proc. ACM Int. Conf. on Supercomputing*, July, 1992.

[Netl94] Netlibrary, "HeNCE (Heterogeneous Network Computing Environment)", http://www.netlib.org/hence/ , June 1994.

[NiHe00] O.M. Nielson amd M. Hegland, "Parallel Performance of Fast Wavelet Transforms", International Journal of High Performance Computing, vol. 11, no. 1, p.55-74, 2000.

[Ohio96] Ohio Supercomputer Center, *MPI Primer/ Developing with LAM*. The Ohio State University, 1996, 86 pp.

[PaJa96] J. N. Patel and L. H. Jameison, Scalability of 2-D Wavelet Transform Algorithms: Analytical and Experimental Results on Coarse Grained Parallel Computers, *In Proceedings of the 1996 IEEE Workshop on VLSI Signal Processing*, San Franciso, USA, pp. 376-385, 1996.

[Patt02] Jason Patterson, "The History Of Computers During My Lifetime - The 1970s", http://www.pattosoft.com.au/jason/Articles/HistoryOfComputers/1970s.html , current as of March 16, 2002.

[PrMa96] John Proakis and Dimitris G. Manolakis, Digital Signal Processing, third ed., Upper Saddle River, NJ: Prentice Hall, 1996, 968pp.

[Schm95] Douglas Schmidt, "Using Design Patterns to Develop Reusable Object-Oriented Communication Software", *CACAM*, 38, 10, October 1995.

[SGI01] SGI Inc, "PDI/DreamWorks Uses SGI Firepower to Visualize the Adventures of Shrek", http://www.sgi.com/features/2001/aug/shrek/index.html , 2001.

[Simo97] Mauricio De Simone, "Active Expressions : A Language-Based Model for Expressing Concurrent Patterns", *Masters Thesis*. Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, 1997, 100 pp.

[SiSi97] Stephen Siu and Ajit Singh, "Design Patterns for Parallel Computing Using a Network of Processors", *proceeding of the 6th International Symposium on High Performance Distributed Computing (HPDC'97)*, pp. 293-304, 1997.

[Siu96] Stephen Siu, "Openness in Design-Pattern-Based Parallel Programming Systems", *Masters Thesis*, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, 1996, 102 pp.

[SSLP93] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons, "The Enterpriser Model for developing Distributed Applications", *IEEE Parallel and Distributed Technology*, vol 1, no. 3, pp. 85-96, 1993.

[Thul01] Parimala Thulasiraman, *Advances in Parallel Computing*. CS785 Class Notes; Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, 2001.

[VGKS95] Jens Volkert, Siegfried Grabner, Dieter Kranzlmuller, and Richard Schall, "ATEMPT - A Tool for Event ManiPulaTion", www.gup.uni-linz.ac.at:8001/research/debugging/atempt/ , September 1995.

[Walk96] James S. Walker, *Fast Fourier Transforms*, 2nd ed. Boca Raton, FL: CRC Press, 1996. 464 pp.

[Webb94] Jon A. Webb, "High Performance Computing in Image Processing and Computer Vision", *ICRP*, Jerusalem, Oct 9-13, 1994.

[WiAl99] Barry Wilkinson and Michael Allen, Parallel Programming, Prentice Hall, Upper Saddle River, New Jersey, 1999, 431 pp.

[Wils99] G. Wilson, "The History of the Development of Parallel Computing", http://ei.cs.vt.edu/~history/Parallel.html , 1994.

[WIMN95] D. Woloschuk, P. Iglinski, S. MacDonald, D. Novillo, I. Parsons, J. Schaeffer and D. Szafron, "Performance Debugging in the Enterprise Parallel Programming System", *CASCON'95 Conference CDRom Proceedings*, Toronto, November 1995.

[Zava99] Andrea Zavanella, "Skeletons and BSP: Performance Portability for Parallel Programming", PH.D. Thesis, Dipartimento di Informatica Dottorato di Ricerca in Informatica, Universita Degli Studi di Pisa, December 1999, 198 pp.

Appendix A:

Software Listing for 2D Discrete Wavelet Transform
A1: Sequential Case
A2: Parallel Hand Coded Case
A3: MPI Buddy Coded Case

A1: Software Listing for Sequential 2D DWT Program

```
/*startproj.c/
/*A wavelet transform program - developed by Narjit Chadha */
/* Summer 2001 */

#include <stdarg.h>
#include "image.h"
#include "filter.h"
#include "mpi.h" /*use for timings*/

double *wavetrans(wavelet usewavelet,image *theimage,int steps);

int main (int argc, char *argv[])
{
        /*place coefficient variables that may have to be used */
        double daub4coeffs[] = { 0.4829629131445341, 0.8365163037378077,
                        0.2241438680420134, -0.1294095225512603 };

        double daub6coeffs[] = { 0.3326705529500825, 0.8068915093110924,
                        0.4598775021184914, -0.1350110200102546,
                        -0.0854412738820267, 0.0352262918857095 };

        double daub8coeffs[] = { 0.2303778133088964, 0.7148465705529154,
                            0.6308807679398587, -0.0279837694168599,
                            -0.1870348117190931, 0.0308413818355607,
                            0.0328830116668852, -0.0105974017850690 };

/*start the program*/
        char *inputfile,*outputfile;
        int steps,p;
        filterset *thefilter;
        wavelet validwavelet;
        double *transformed; /*contains the transformed data*/
        image *mainimage=(image*) malloc(sizeof(image)); /*the image to be used in this project */
        double starttime,endtime; /*use for timings*/
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&p);

        inputfile=malloc(81); /*assume the input file name will not exceed 80 chars*/
        outputfile=malloc(81);
        printf("\nEnter the name of the pgm image file to be used: ");
        gets(inputfile);
        printf("\nEnter the name of the output raw file :");
        gets(outputfile);
        printf("\nEnter the number of steps in the wavelet transform");
        scanf("%d",&steps);
        getheader(mainimage,inputfile);
        /*for daub4 filterset*/
/*      thefilter = makedaub(FALSE,daub4coeffs,4,0); */
/*      thefilter = makedaub(FALSE,HarrCoeffs,2,0); */
/*      thefilter = makedaub(FALSE,daub6coeffs,6,0); */
        thefilter = makedaub(FALSE,daub8coeffs,8,0);
```

```
        /*now enter the wavelet tranform -error below here*/
        validwavelet=initwavelet(thefilter);
        MPI_Barrier(MPI_COMM_WORLD);
        starttime=MPI_Wtime();

        /*compression begins*/
        transformed=wavetrans(validwavelet,mainimage,steps);
        /*find end time*/
        MPI_Barrier(MPI_COMM_WORLD);
        endtime=MPI_Wtime();

        /*now write to the output pgm file so that 2d discrete wavelet tranformed */
        /*image can be displayed*/
        writeoutput(transformed,outputfile,mainimage->hsize,mainimage->vsize);
        printf("\nThe elapsed time is %e \n", endtime-starttime);
        MPI_Finalize();
        return 0;
}

double *wavetrans(wavelet usewavelet,image *theimage,int steps)
{
        int i;
        int lowsizeH=theimage->hsize;
        int lowsizeV=theimage->vsize;
        int highsizeH,highsizeV;
        int symmetric=usewavelet.symmetric;
        int npad=usewavelet.npad;
        int hsize=lowsizeH;
        int vsize=lowsizeV;
        double *tempin,*tempout;
        double *tempdata;
        tempdata=(double*)malloc(hsize*vsize*sizeof(double));
        tempin=(double*)malloc((2*npad+max2(hsize,vsize))*sizeof(double));
        tempout=(double*)malloc((2*npad+max2(hsize,vsize))*sizeof(double));
        /*initialize the tranform first*/

        copy(theimage->data,tempdata,hsize*vsize); /*copy image to a temp location*/
        while (steps--)
        {
                if ((lowsizeH<=2 || lowsizeV<=2) &&symmetric==1) {
                        warning("reduce the number of transform setps of increase the signal size");
                        warning("or switch to a periodic extension wavlet set");
                        error("low pass subband is too small");
                }
                /*now do a convolution with the low pass part of each row */
                for (i=0;i<lowsizeV;i++) {
                        copy(tempdata+(i*hsize),tempin+npad,lowsizeH);
                        /*convolve with low and high pass filters*/
                        transform(usewavelet,tempin,tempout,lowsizeH,symmetric);
                        /*now copy back to the image*/
                        copy(tempout+npad,tempdata+(i*hsize),lowsizeH);
                }
                /*now covolve on the low pass portion of each column*/
                for (i=0;i<lowsizeH;i++) {
```

```
                /*copy each column i into the data array*/
                copy2(tempdata+i,hsize,tempin+npad,lowsizeV);
                /*now convove with low and high pass filters*/
                transform(usewavelet,tempin,tempout,lowsizeV,symmetric);
                copy3(tempout+npad,tempdata+i,hsize,lowsizeV);
            }
            /*stay in while loop - do row and column convolutions until steps*/
            /*have all been completed*/
            highsizeH=lowsizeH/2;
            lowsizeH=(lowsizeH+1)/2;
            highsizeV=(lowsizeV)/2;
            lowsizeV=(lowsizeV+1)/2;
        } /*end of while*/
        /*
        free(tempout);
        free(tempin);*/
        /*error above here ??? why*/
        /*tempdata contains the data required -definitely correct*/
        return tempdata; /*contains the modified image*/
}
/*make a four variable copy function*/
```

```
/globals.h/
/*------------------------------------------------------------------------*/
/* Baseline Wavelet Transform Coder Construction Kit
        Geoff Davis
  gdavis@cs.dartmouth.edu
  http://www.cs.dartmouth.edu/~gdavis
  Copyright 1996 Geoff Davis 9/11/96
  Permission is granted to use this
  software for research purposes as
  long as this notice stays attached to this software.*/
/*globals.h -file contains all the globals necessary for my DWT program*/
/*modified by Narjit Chadha - Summer 2001 */
/*------------------------------------------------------------------------*/
#include <math.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
/*----------------------------------------------------------*/
/*define type PGM*/
#define PGM
/*------------------------------------------------------------------------*/
/* standard #defines*/
/*------------------------------------------------------------------------*/
#define TRUE  1
#define FALSE 0

#define BACKSPACE 8
#define BS      8
#define ESC     27
/*------------------------------------------------------------------------*/
```

```
/*useful constants*/
/*------------------------------------------------------------------------*/

#define Pi 3.14159265358979
#define TwoPi 2.0*Pi
#define Sqrt2 sqrt(2.0)
#define Log2  log(2.0)

/*------------------------------------------------------------------------*/
/*helpful inline functions -comment out if defined on compiler*/
/*------------------------------------------------------------------------*/
#define min2(x,y) (((x)<(y))?(x):(y))
#define max2(x,y) (((x)>(y))?(x):(y))
/*------------------------------------------------------------------------*/
void error (char *format, ...);
void warning (char *format, ...);
```

```
#include "globals.h"
/*globals.c*/
/*----------------------------------------*/
#ifdef DEBUG
static FILE *debug_file;
static int debug_file_open = FALSE;
#endif
/*----------------------------------------*/

void error (char *format, ...)
{
  va_list list;

  va_start (list, format);

  printf ("Error: ");
  vprintf (format, list);
  va_end (list);
  printf ("\n");

#ifdef DEBUG
  if (debug_file_open) {
    fprintf (debug_file, "Error: ");
    vfprintf (debug_file, format, list);
    fprintf (debug_file, "\n");
    fflush (debug_file);
  }
#endif

  assert(0);
}


void warning (char *format, ...)
{
```

```
  va_list list;

  va_start (list, format);

#ifdef DEBUG
  if (debug_file_open) {
    fprintf (debug_file, "Warning: ");
    vfprintf (debug_file, format, list);
    fprintf (debug_file, "\n");
    fflush (debug_file);
  }
#endif

  printf ("Warning: ");
  vprintf (format, list);
  va_end (list);
  printf ("\n");
}
```

```
/*-----------------------------------------------------------*/
/*image.h */
/*the header file for all of the image routines */
/* Narjit Chadha Summer 2001 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "globals.h"

typedef struct
{
            int hsize,vsize,maxg;
            long size;
   double* data; /*data for the image*/
} image;

void loaddata(image *animage,FILE *infile);
void getheader(image *theimage,char *afile);
void sizeimage(image *theimage,int xsize,int ysize);
void skipcomments(FILE *infile, unsigned char* ch);
unsigned int getpgmval(FILE *infile);
void copy (double *data1, double *data2,int size);
void copy2(double *data1,int stride1,double *data2,int length);
void copy3(double *data1,double *data2,int stride2,int length);
void writeoutput(double *tranformed,char *outputname,int xsize,int ysize);
void mklocimage(image* locimage,double *locdata,int loc_hsize,int loc_vsize);
```

```
/*image .c */
/*this file will take care of reading in a pgm image and outputting the final
```

```
image to a file */
/* Developed by Narjit Chadha , Summer 2001*/


#include "image.h"

void getheader(image *theimage,char *afile)
{
	FILE *infile;
	unsigned char ch =' '; /*use the unsigned type for images*/
	char filetype;
	int xsize,ysize,maxg;
	infile=fopen(afile,"rb");
	if (infile==NULL) {
		error("Unable to open the file %s\n",afile);
	}
	theimage->hsize=0;
	theimage->vsize=0;
	theimage->data=NULL;
	/*assume file is a pgm file-->i.e. the default supported type*/
	while((ch!='P')&&(ch!='#')) {ch=fgetc(infile);}
	skipcomments(infile,&ch);
	filetype=getc(infile); /*should be 5 or 6*/
	/*now get the relevant information about the file including hsize,vsize */
	xsize=(int)getpgmval(infile);
	ysize=(int)getpgmval(infile);
	maxg=(int)getpgmval(infile);
	/*just make sure that the program can execute*/
	if ((theimage->hsize<=0)&&(theimage->vsize<=0)){
		sizeimage(theimage,xsize,ysize);
		if (theimage->data==NULL)
			error("Trouble allocating memory for image with dimensiions %d by %d\n",xsize,ysize);
	}
	else {
		if ((xsize!=theimage->hsize)||(ysize!=theimage->vsize)) {
			error("File dimensions and image settings are in Conflict!\n");
		}
	}
	if (filetype=='5') {
		theimage->maxg=maxg;
		printf("File %s , of type PGM is %d by %d with max gray level %d\n",
afile,theimage->hsize,theimage->vsize,theimage->maxg);
		loaddata(theimage,infile); /*write a routine to load the image data)*/
	}
	fclose(infile);
}

void loaddata(image *animage,FILE *infile)
{
	long i;
	unsigned char *tmp;
	long fp;
	tmp=(unsigned char*)malloc(animage->size*sizeof(unsigned char));
	fp = -1*animage->size;
```

```
	fseek(infile,fp,SEEK_END);
	if (fread(tmp,animage->size,sizeof(unsigned char),infile)!=1)
		error("problem with input file");
	for (i=0;i<animage->size;i++) {
		animage->data[i]=(double)tmp[i];
	}
	free(tmp);
}

void sizeimage(image *theimage,int xsize,int ysize)
{
	int i,j;
	long imagesize;
	double *tmpvalue;
	imagesize=xsize*ysize;

	tmpvalue=(double*)malloc(imagesize*sizeof(double));
	for (i=0;i<imagesize;i++) {
		tmpvalue[i]=0;
	}
	for (j=0;j<min2(theimage->vsize,ysize); j++) {
		for (i=0; i<min2(theimage->hsize,xsize);i++) {
			printf("\nnothing should print");
			tmpvalue[j*xsize+i]=theimage->data[j*theimage->hsize+i];
		}
	}
	theimage->hsize=xsize;
	theimage->vsize=ysize;
	if (theimage->data!=NULL)
		free(theimage->data);
	theimage->data=tmpvalue; /*allocated space for the image*/
	theimage->size=imagesize;
}

void skipcomments(FILE *infile, unsigned char* ch)
{
	while((*ch=='#')) {
		while (*ch!='\n') {*ch=getc(infile);}
		while (*ch<' ') {*ch=getc(infile);}
	} /*i.e. bypass all the commments in the pgm file*/
}


unsigned int getpgmval(FILE *infile)
{
	unsigned int tmp;
	unsigned char ch;
	do {ch=getc(infile);} while ((ch<=' ')&&(ch!='#'));
	skipcomments(infile,&ch);
	ungetc(ch,infile);
	if (fscanf(infile,"%u",&tmp)!=1) {
		printf("%s\n", "Error parsing the file.");
		exit(1);
	}
```

```
	return(tmp);
}

void copy (double *data1, double *data2,int size)
{
	int temp=size;
	while (temp--) {
		*data2++=*data1++;
	}
}

void copy2(double *data1,int stride1,double *data2,int length)
{
	int temp=length;
	while (temp--) {
		*data2++=*data1;
		data1+=stride1;}
}

void copy3(double *data1,double *data2,int stride2,int length)
{
	int temp=length;
	while (temp--) {
		*data2=*data1++;
		data2+=stride2;}
}

void writeoutput(double *transformed,char *outputname,int xsize,int ysize)
{
	unsigned char *buffer;
	int i;
	double max,min,scale,hold;
	FILE *outfile;
	buffer=(unsigned char*)malloc(xsize*ysize*sizeof(unsigned char));

	outfile=fopen(outputname, "wb+");
	if (outfile==NULL) {
		error("unable to open the file %s\n", outputname);
	}
	fprintf(outfile, "P5\n#%s\n%d %d\n255\n",outputname,xsize,ysize);
	/* have a function called for image scaling*/
	max=0.0;
	min=0.0;
	for (i=0;i<xsize*ysize;i++) {
		if (transformed[i]>max)
			max=transformed[i];
		if (transformed[i]<min)
			min=transformed[i];
	}
	/*now scale all the values in the array to write to the file - values from 0-255*/
	scale=max-min;
	/*(255/scale)+min*(255/scale);*/
	for (i=0;i<xsize*ysize;i++) {
		hold=(255/scale)*transformed[i]+min*(255/scale);
```

```
                    buffer[i]=(unsigned char)(hold);
        }
        fwrite(buffer,xsize*ysize,1,outfile);
        fclose(outfile);
/*      free(buff); */
}

void mklocimage(image* locimage,double *locdata,int loc_hsize,int loc_vsize)
{
        locimage->data=locdata;
        locimage->hsize=loc_hsize;
        locimage->vsize=loc_vsize;
        locimage->size=(loc_hsize*loc_vsize);
}
```

```
/*filter.h*/
#include "globals.h"

/* a header file for the filters and various filter functions */

typedef struct {
        int size,firstindex,center;
        double *coeff;
} filter;

typedef struct {
        int symmetric;
        filter *analysislow, *analysishigh, *synthesislow, *synthesishigh;
} filterset;

typedef struct {
        filter *analysislow, *analysishigh;
        filter *synthesislow, *synthesishigh;
        int symmetric;
        int npad;
} wavelet;


filterset *makedaub(int symmetric,double *anlow, int anlowsize, int anlowfirst);
filter *makefilter2(filter *usefilter);
filter *makefilter1(int filtersize,int firstindex,double *coeff);
wavelet initwavelet(filterset *afilterset);
void transform(wavelet awavelet,double *input,double *output,int size,int symt);
void symmetric_ext (double *output, int size, int left_ext, int right_ext, int npad, int symmetry);
void periodic_ext (double *output, int size, int npad);
```

```
/*the set of files for the wavelet filter functions */
/*filter.c*/
/*designed by Narjit Chadha - Summer 2001*/

#include "filter.h"
#include "globals.h"
```

```
filterset *makedaub(int symmetric,double *anlow, int anlowsize, int anlowfirst)
{
	int i,sign;
	filterset *tempfs;
	tempfs=(filterset*)malloc(sizeof(filterset));
	tempfs->analysislow=makefilter1(anlowsize,anlowfirst,anlow);
	/*assume the wavelets are orthogonal*/
	tempfs->synthesislow=makefilter2(tempfs->analysislow);
	tempfs->analysishigh=makefilter1(tempfs->analysislow->size,2-tempfs->analysislow->size-tempfs->analys
islow->firstindex,NULL);
	tempfs->symmetric=symmetric; /*must copy the symmetries*/
	if (tempfs->analysislow->firstindex%2){
		sign=1;
	}
	else sign=-1;
	for(i=0;i<tempfs->analysislow->size;i++) {

tempfs->analysishigh->coeff[1-i-tempfs->analysislow->firstindex-tempfs->analysishigh->firstindex]=
			sign*tempfs->analysislow->coeff[i];
		assert(1-i-tempfs->analysislow->firstindex-tempfs->analysishigh->firstindex>=0);

assert((1-i-tempfs->analysislow->firstindex-tempfs->analysishigh->firstindex)<(tempfs->analysishigh->size));
		sign*=-1;
	}
	/*copy the high pass analysis filter to the synthesis filter */
	tempfs->synthesishigh=makefilter2(tempfs->analysishigh);
	return tempfs;
}


filter *makefilter2(filter *usefilter)
{
	int i;
	filter *tmpfilter;
	tmpfilter=(filter*)malloc(sizeof(filter));
	tmpfilter->coeff=NULL;
	tmpfilter->size=usefilter->size;
	tmpfilter->firstindex=usefilter->firstindex;
	tmpfilter->center=-(usefilter->firstindex);
	tmpfilter->coeff=(double*)malloc(usefilter->size*sizeof(double));
	if(usefilter->coeff!=NULL){
		for (i=0;i<tmpfilter->size;i++)
			tmpfilter->coeff[i]=usefilter->coeff[i];
	}
	else {
		for (i=0;i<tmpfilter->size;i++)
			tmpfilter->coeff[i]=0;
	}
	return tmpfilter;
}

filter *makefilter1(int filtersize,int firstindex,double *coeff)
{
	int i;
```

```
	filter *afilter;
	afilter=(filter*)malloc(sizeof(filter));
	afilter->size=filtersize;
	afilter->firstindex=firstindex;
	afilter->center=-firstindex;
	afilter->coeff=(double*)malloc(afilter->size*sizeof(double));
	if(coeff!=NULL){
		for (i=0;i<afilter->size;i++)
			afilter->coeff[i]=coeff[i];
	}
	else {
		for (i=0;i<afilter->size;i++)
			afilter->coeff[i]=0;
	}
	return afilter;
}

wavelet initwavelet(filterset *afilterset)
{
	wavelet tempw;
	tempw.analysislow=afilterset->analysislow;
	tempw.analysishigh=afilterset->analysishigh;
	tempw.synthesislow=afilterset->synthesislow;
	tempw.synthesishigh=afilterset->synthesishigh;
	tempw.symmetric=afilterset->symmetric;
	tempw.npad=max2(tempw.analysislow->size,tempw.analysishigh->size);
	return tempw;
}


void transform(wavelet awavelet,double *input,double *output,int size,int symt)
{
	int i,j;
	int lowsize=(size+1)/2;
	int leftext,rightext;
	if (awavelet.analysislow->size%2) {
		/*i.e.an odd filter length*/
		leftext=1;
		rightext=1;
	}else {
		leftext=2;
		rightext=2;
	}

	if (symt)
	{
		symmetric_ext(input,size,leftext,rightext,awavelet.npad,1);
	}
	else {
		periodic_ext(input,size,awavelet.npad);
	/*i.e. add the necessary extensions to make the wavelet transform work*/
	/*the detail will become xxxxxxxx->HHHHGGGG */
	}
	/*first use low pass filter*/
```

```
	for (i=0;i<lowsize;i++) {
		output[awavelet.npad+i]=0.0;
		for (j=0;j<awavelet.analysislow->size;j++) {
			output[awavelet.npad+i]+=input[awavelet.npad+2*i+awavelet.analysislow->firstindex+j]
*(awavelet.analysislow->coeff[j]);
		}
	}

	/*now use high pass filter*/
	for (i=lowsize;i<size;i++) {
		output[awavelet.npad+i]=0.0;
		for (j=0;j<awavelet.analysishigh->size;j++) {

output[awavelet.npad+i]+=input[awavelet.npad+2*(i-lowsize)+awavelet.analysishigh->firstindex+j]*(awavelet.anal
ysishigh->coeff[j]);
		}
	}
}

/*--------------------------------------------------------------------------*/
/* Do symmetric extension of data using prescribed symmetries*/
 /* Original values are in output[npad] through output[npad+size-1]*/
/*  New values will be placed in output[0] through output[npad] and in*/
/*     output[npad+size] through output[2*npad+size-1] (note: end values may*/
/*     not be filled in) */
/*  left_ext = 1 -> extension at left bdry is   ... 3 2 1 | 0 1 2 3 ...*/
/*  left_ext = 2 -> extension at left bdry is ... 3 2 1 0 | 0 1 2 3 ...*/
/*  right_ext = 1 or 2 has similar effects at the right boundary*/

 /*  symmetry = 1  -> extend symmetrically*/
/*  symmetry = -1 -> extend antisymmetrically*/

void symmetric_ext (double *output, int size, int left_ext, int right_ext, int npad, int symmetry)
{
 int i,originalfirst,originallast,originalsize,period;
 int first = npad, last = npad + size-1;
 int nextend;


 if (symmetry == -1) {
  if (left_ext == 1)
   output[--first] = 0;
  if (right_ext == 1)
   output[++last] = 0;
 }
 originalfirst = first;
 originallast = last;
 originalsize = originallast-originalfirst+1;

 period = 2 * (last - first + 1) - (left_ext == 1) - (right_ext == 1);

 if (left_ext == 2)
  output[--first] = symmetry*output[originalfirst];
 if (right_ext == 2)
```

```
   output[++last] = symmetry*output[originallast];

  /* extend left end*/
  nextend = min2 (originalsize-2, first);
  for (i = 0; i < nextend; i++) {
   output[--first] = symmetry*output[originalfirst+1+i];
  }

  /* should have full period now -- extend periodically*/
  while (first > 0) {
   first--;
   output[first] = output[first+period];
  }

  /*extend right end*/
  nextend = min2 (originalsize-2, 2*npad+size-1 - last);
  for (i = 0; i < nextend; i++) {
   output[++last] = symmetry*output[originallast-1-i];
  }

  /*should have full period now -- extend periodically*/
  while (last < 2*npad+size-1) {
   last++;
   output[last] = output[last-period];
  }
}

/*------------------------------------------------------------------------*/
/* Do periodic extension of data using prescribed symmetries */
/*  Original values are in output[npad] through output[npad+size-1] */
/*  New values will be placed in output[0] through output[npad] and in */
/*    output[npad+size] through output[2*npad+size-1] (note: end values may */
/*    not be filled in) */

void periodic_ext (double *output, int size, int npad)
{
  int first = npad, last = npad + size-1;

  /* extend left periodically*/
  while (first > 0) {
   first--;
   output[first] = output[first+size];
  }

  /* extend right periodically*/
  while (last < 2*npad+size-1) {
   last++;
   output[last] = output[last-size];
  }
}

/*------------------------------------------------------------------------*/
```

A2: Software Listing for Parallel Hand Coded 2D DWT Program

*Code is the same as A1, except for startproj.c, and filther.c, and filter.h (shown below modified).

```
/*A Parallel wavelet transform program - developed by Narjit Chadha */
/*This is an MPI implementation of 2D DWT that can operate on images and filters
of various sizes. Right now the only restriction is that the filters must be
asymmetric. i.e. size 2,4,6,8,10,etc. This will mean periodic extensions will
need to be added when performing the wavelet transform*/
/*This program functions by reading in an image, producing a wavelet filter,
distributing N/p columns of the image to each processor, conducting a local
DWT on each part of the image, and finally gathering the local results and outputting
the final result to a file*/

/* Developed - Summer 2001 */

#include <stdarg.h>
#include "image.h"
#include "filter.h"
#include "mpi.h" /*for the mpi communication/data structures*/

double *wavetrans(wavelet usewavelet,image *theimage,int steps,int my_rank,int p,MPI_Status status);

void main (int argc,char* argv[])
{
        /*place coefficient variables that may have to be used - need only reside on root processor*/
        double Harrcoeffs[] = {0.707106781, 0.707106781};

        double daub4coeffs[] = { 0.4829629131445341, 0.8365163037378077,
                        0.2241438680420134, -0.1294095225512603 };

        double daub6coeffs[] = { 0.3326705529500825, 0.8068915093110924,
                        0.4598775021184914, -0.1350110200102546,
                        -0.0854412738820267, 0.0352262918857095 };

        double daub8coeffs[] = { 0.2303778133088964, 0.7148465705529154,
                        0.6308807679398587, -0.0279837694168599,
                        -0.1870348117190931, 0.0308413818355607,
                        0.0328830116668852, -0.0105974017850690 };


/*start the program*/
        char *inputfile,*outputfile;
        double *locdata,*finalarray;
        int steps; /*keep loc_v size the same*/
        int loc_hsize,loc_vsize;
        long position;
        filterset *thefilter;
        wavelet validwavelet;
        double *transformed; /*contains the transformed data*/
        int my_rank,i;
```

```
int p; /*number of processes*/
int root=0; /*rank of the root processor*/
int tag=0; /*tag for the message*/
image *mainimage,*loc_image;
double starttime,endtime;
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&p); /*find out number of processors*/
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);  /*find out the rank of each processor*/
if (my_rank==0) {
mainimage=(image*) malloc(sizeof(image)); ; /*the image to be used in this project */
inputfile=malloc(81); /*assume the input file name will not exceed 80 chars*/
outputfile=malloc(81);
/*      printf("\nEnter the name of the pgm image file to be used: ");
gets(inputfile); */
inputfile="barbara.pgm";

/*printf("\nEnter the name of the output raw file :");
gets(outputfile);*/
outputfile="barb1.pgm";

/*printf("\nEnter the number of steps in the wavelet transform");
scanf("%d",&steps);*/
steps=3;

getheader(mainimage,inputfile);
loc_hsize=(mainimage->hsize)/p;
loc_vsize=mainimage->vsize; /*this stays the same*/
finalarray=(double*)malloc(mainimage->hsize*mainimage->vsize*sizeof(double));
} /*end of if my_rank==0*/
/*for daub4 filterset*/
/*      thefilter = makedaub(FALSE,Harrcoeffs,2,0); */
/*      thefilter = makedaub(FALSE,daub4coeffs,4,0); */
/*      thefilter = makedaub(FALSE,daub6coeffs,6,0); */
thefilter = makedaub(FALSE,daub8coeffs,8,0);

/*now enter the wavelet tranform -error below here*/
validwavelet=initwavelet(thefilter);
/*can do on all processors -faster!*/
/*compression begins -do on multiple processors!*/

/*scatter the image so that each processor has N/p columns of data -this is how
 the communication efficient DWT is supposed to function*/
MPI_Bcast(&steps,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&loc_hsize,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&loc_vsize,1,MPI_INT,0,MPI_COMM_WORLD);
/*divide up the image*/

loc_image=(image*)malloc(sizeof(image));  /*local array of elements*/
locdata=(double*)malloc(loc_hsize*loc_vsize*sizeof(double));

MPI_Barrier(MPI_COMM_WORLD);
starttime=MPI_Wtime();
```

```
	for (i=0;i<loc_vsize;i++) {
		position=i*mainimage->hsize;

MPI_Scatter(mainimage->data+position,loc_hsize,MPI_DOUBLE,locdata+i*loc_hsize,loc_hsize,MPI_DOUBLE,0,
MPI_COMM_WORLD);
	}
	/*make a simple function to covert the new data into new images -eliminate fxn call to save time*/
  /* mklocimage(loc_image,locdata,loc_hsize,loc_vsize); */
	loc_image->data=locdata;
	loc_image->hsize=loc_hsize;
	loc_image->vsize=loc_vsize;
	loc_image->size=(loc_hsize*loc_vsize);
	/*revamp transformed to allow for parallel computation*/

	transformed=wavetrans(validwavelet,loc_image,steps,my_rank,p,status);

	for (i=0;i<loc_vsize ;i++){
		position=i*loc_image->hsize;

MPI_Gather(transformed+position,loc_hsize,MPI_DOUBLE,finalarray+(i*mainimage->hsize),loc_hsize,MPI_DOU
BLE,0,MPI_COMM_WORLD);
	}
	MPI_Barrier(MPI_COMM_WORLD);
	endtime=MPI_Wtime();

	/*now write to the output pgm file so that 2d discrete wavelet tranformed */
	/*image can be displayed*/
	if (my_rank==0) {
	writeoutput(finalarray,outputfile,mainimage->hsize,mainimage->vsize);
	printf("\nelapsed time is %e \n",endtime-starttime);
	}
	MPI_Finalize();
	return;
}

double *wavetrans(wavelet usewavelet,image *theimage,int steps,int my_rank,int p,MPI_Status status)
{
	/*thie routine has been modified to allow for parallel comutations*/
	int i;
	int lowsizeH=theimage->hsize;
	int lowsizeV=theimage->vsize;
	int highsizeH,highsizeV;
	int symmetric=usewavelet.symmetric;
	int npad=usewavelet.npad;
	int nonlocal=npad-2; /*number of coefficients that must be transmitted nonlocally*/
	int hsize=lowsizeH;
	int vsize=lowsizeV;
	double *tempin,*tempout,*hold;
	double *tempdata;
	long position=hsize;
	double *holdpad;
	tempdata=(double*)malloc(hsize*vsize*sizeof(double));
	tempin=(double*)malloc((2*npad+max2(hsize,vsize))*sizeof(double)); /*i.e. padding on both sides*/
	tempout=(double*)malloc((2*npad+max2(hsize,vsize))*sizeof(double));
```

```
	holdpad=(double*)malloc(nonlocal*lowsizeV*sizeof(double)); /*hold values for sends/receives*/
	hold=(double*)malloc(nonlocal*lowsizeV*sizeof(double));
	/*initialize the tranform first*/

	copy(theimage->data,tempdata,hsize*vsize); /*copy image to a temp location*/

	while (steps--)
	{
		if ((lowsizeH<=2 || lowsizeV<=2) &&symmetric==1) {
			warning("reduce the number of transform setps of increase the signal size");
			warning("or switch to a periodic extension wavlet set");
			error("low pass subband is too small");
		}
		/*now do a convolution with the low pass part of each row */
		/*make an array of data that needs to be communicated*/
		for (i=0;i<lowsizeV;i++) {
			copy(tempdata+i*hsize,holdpad+i*nonlocal,nonlocal);
		}
		/*only communicate terms once per round*/
		if ((my_rank%2)==0) {
				/*i.e. do the even ranks first*/

MPI_Send(holdpad,nonlocal*lowsizeV,MPI_DOUBLE,(my_rank+p-1)%p,0,MPI_COMM_WORLD);

MPI_Recv(hold,nonlocal*lowsizeV,MPI_DOUBLE,(my_rank+p+1)%p,0,MPI_COMM_WORLD,&status);
			}
			else if ((my_rank%2)==1) {
				/*now do the odd procesor ranks*/

MPI_Recv(hold,nonlocal*lowsizeV,MPI_DOUBLE,(my_rank+p+1)%p,0,MPI_COMM_WORLD,&status);

MPI_Send(holdpad,nonlocal*lowsizeV,MPI_DOUBLE,(my_rank+p-1)%p,0,MPI_COMM_WORLD);
			}

		for (i=0;i<lowsizeV;i++) {
			copy(tempdata+(i*hsize),tempin+npad,lowsizeH);
			copy(hold+i*nonlocal,tempin+(npad+lowsizeH),nonlocal); /*ie the new padded values */
			/*convolve with low and high pass filters*/
			/*first complete sends and receives in this routine for horizontal*/

			transform2(usewavelet,tempin,tempout,lowsizeH,symmetric);
			/*now copy back to the image*/
			copy(tempout+npad,tempdata+(i*hsize),lowsizeH);
		}
		/*now covolve on the low pass portion of each column*/
		/*each processor contains all necessary information for padding here*/

		for (i=0;i<lowsizeH;i++) {
			/*copy each column i into the data array*/
			copy2(tempdata+i,hsize,tempin+npad,lowsizeV);
			/*now convove with low and high pass filters*/
			transform(usewavelet,tempin,tempout,lowsizeV,symmetric);
			copy3(tempout+npad,tempdata+i,hsize,lowsizeV);
		}
```

```
		/*stay in while loop - do row and column convolutions until steps*/
		/*have all been completed*/
		highsizeH=lowsizeH/2;
		lowsizeH=(lowsizeH+1)/2;
		highsizeV=(lowsizeV)/2;
		lowsizeV=(lowsizeV+1)/2;
	} /*end of while*/
	/*
	free(tempout);
	free(tempin);*/
	/*error above here ??? why*/
	/*tempdata contains the data required -definitely correct*/
	return tempdata; /*contains the modified image*/
}
/*make a four variable copy function*/
```

```
/*filter.h*/
#include "globals.h"

/* a header file for the filters and various filter functions */

typedef struct {
	int size,firstindex,center;
	double *coeff;
} filter;

typedef struct {
	int symmetric;
	filter *analysislow, *analysishigh, *synthesislow, *synthesishigh;
} filterset;

typedef struct {
	filter *analysislow, *analysishigh;
	filter *synthesislow, *synthesishigh;
	int symmetric;
	int npad;
} wavelet;


filterset *makedaub(int symmetric,double *anlow, int anlowsize, int anlowfirst);
filter *makefilter2(filter *usefilter);
filter *makefilter1(int filtersize,int firstindex,double *coeff);
wavelet initwavelet(filterset *afilterset);
void transform(wavelet awavelet,double *input,double *output,int size,int symt);
void symmetric_ext (double *output, int size, int left_ext, int right_ext, int npad, int symmetry);
void periodic_ext (double *output, int size, int npad);
void transform2(wavelet awavelet,double *input,double *output,int size,int symt);
```

```
/*the set of files for the wavelet filter functions */
/*designed by Narjit Chadha - Summer 2001*/
/*filter.c*/
```

```
#include "filter.h"
#include "globals.h"

filterset *makedaub(int symmetric,double *anlow, int anlowsize, int anlowfirst)
{
        int i,sign;
        filterset *tempfs;
        tempfs=(filterset*)malloc(sizeof(filterset));
        tempfs->analysislow=makefilter1(anlowsize,anlowfirst,anlow);
        /*assume the wavelets are orthogonal*/
        tempfs->synthesislow=makefilter2(tempfs->analysislow);
        tempfs->analysishigh=makefilter1(tempfs->analysislow->size,2-tempfs->analysislow->size-tempfs->analys
islow->firstindex,NULL);
        tempfs->symmetric=symmetric; /*must copy the symmetries*/
        if (tempfs->analysislow->firstindex%2){
                sign=1;
        }
        else sign=-1;
        for(i=0;i<tempfs->analysislow->size;i++) {

tempfs->analysishigh->coeff[1-i-tempfs->analysislow->firstindex-tempfs->analysishigh->firstindex]=
                        sign*tempfs->analysislow->coeff[i];
                assert(1-i-tempfs->analysislow->firstindex-tempfs->analysishigh->firstindex>=0);

assert((1-i-tempfs->analysislow->firstindex-tempfs->analysishigh->firstindex)<(tempfs->analysishigh->size));
                sign*=-1;
        }
        /*copy the high pass analysis filter to the synthesis filter */
        tempfs->synthesishigh=makefilter2(tempfs->analysishigh);
        return tempfs;
}


filter *makefilter2(filter *usefilter)
{
        int i;
        filter *tmpfilter;
        tmpfilter=(filter*)malloc(sizeof(filter));
        tmpfilter->coeff=NULL;
        tmpfilter->size=usefilter->size;
        tmpfilter->firstindex=usefilter->firstindex;
        tmpfilter->center=-(usefilter->firstindex);
        tmpfilter->coeff=(double*)malloc(usefilter->size*sizeof(double));
        if(usefilter->coeff!=NULL){
                for (i=0;i<tmpfilter->size;i++)
                        tmpfilter->coeff[i]=usefilter->coeff[i];
        }
        else {
                for (i=0;i<tmpfilter->size;i++)
                        tmpfilter->coeff[i]=0;
        }
        return tmpfilter;
}
```

```
filter *makefilter1(int filtersize,int firstindex,double *coeff)
{
	int i;
	filter *afilter;
	afilter=(filter*)malloc(sizeof(filter));
	afilter->size=filtersize;
	afilter->firstindex=firstindex;
	afilter->center=-firstindex;
	afilter->coeff=(double*)malloc(afilter->size*sizeof(double));
	if(coeff!=NULL){
		for (i=0;i<afilter->size;i++)
			afilter->coeff[i]=coeff[i];
	}
	else {
		for (i=0;i<afilter->size;i++)
			afilter->coeff[i]=0;
	}
	return afilter;
}

wavelet initwavelet(filterset *afilterset)
{
	wavelet tempw;
	tempw.analysislow=afilterset->analysislow;
	tempw.analysishigh=afilterset->analysishigh;
	tempw.synthesislow=afilterset->synthesislow;
	tempw.synthesishigh=afilterset->synthesishigh;
	tempw.symmetric=afilterset->symmetric;
	tempw.npad=max2(tempw.analysislow->size,tempw.analysishigh->size);
	return tempw;
}


void transform(wavelet awavelet,double *input,double *output,int size,int symt)
{
	int i,j;
	int lowsize=(size+1)/2;
	int leftext,rightext;
	if (awavelet.analysislow->size%2) {
		/*i.e.an odd filter length*/
		leftext=1;
		rightext=1;
	}else {
		leftext=2;
		rightext=2;
	}

	if (symt)
	{
		symmetric_ext(input,size,leftext,rightext,awavelet.npad,1);
	}
	else {
		periodic_ext(input,size,awavelet.npad);
	/*i.e. add the necessary extensions to make the wavelet transform work*/
```

```
		/*the detail will become xxxxxxxx->HHHHGGGG */
		}
		/*first use low pass filter*/
		for (i=0;i<lowsize;i++) {
				output[awavelet.npad+i]=0.0;
				for (j=0;j<awavelet.analysislow->size;j++) {

output[awavelet.npad+i]+=input[awavelet.npad+2*i+awavelet.analysislow->firstindex+j]*(awavelet.analysislow->c
oeff[j]);
				}
		}

		/*now use high pass filter*/
		for (i=lowsize;i<size;i++) {
				output[awavelet.npad+i]=0.0;
				for (j=0;j<awavelet.analysishigh->size;j++) {

output[awavelet.npad+i]+=input[awavelet.npad+2*(i-lowsize)+awavelet.analysishigh->firstindex+j]*(awavelet.anal
ysishigh->coeff[j]);
				}
		}
}

/*---------------------------------------------------------------------------*/
/* Do symmetric extension of data using prescribed symmetries*/
 /* Original values are in output[npad] through output[npad+size-1]*/
/*   New values will be placed in output[0] through output[npad] and in*/
/*      output[npad+size] through output[2*npad+size-1] (note: end values may*/
/*      not be filled in) */
/*   left_ext = 1 -> extension at left bdry is   ... 3 2 1 | 0 1 2 3 ...*/
/*   left_ext = 2 -> extension at left bdry is ... 3 2 1 0 | 0 1 2 3 ...*/
/*   right_ext = 1 or 2 has similar effects at the right boundary*/

 /*  symmetry = 1  -> extend symmetrically*/
/*   symmetry = -1 -> extend antisymmetrically*/

void symmetric_ext (double *output, int size, int left_ext, int right_ext, int npad, int symmetry)
{
  int i,originalfirst,originallast,originalsize,period;
  int first = npad, last = npad + size-1;
  int nextend;


  if (symmetry == -1) {
    if (left_ext == 1)
      output[--first] = 0;
    if (right_ext == 1)
      output[++last] = 0;
  }
  originalfirst = first;
  originallast = last;
  originalsize = originallast-originalfirst+1;

  period = 2 * (last - first + 1) - (left_ext == 1) - (right_ext == 1);
```

```
  if (left_ext == 2)
    output[--first] = symmetry*output[originalfirst];
  if (right_ext == 2)
    output[++last] = symmetry*output[originallast];

  /* extend left end*/
  nextend = min2 (originalsize-2, first);
  for (i = 0; i < nextend; i++) {
    output[--first] = symmetry*output[originalfirst+1+i];
  }

  /* should have full period now -- extend periodically*/
  while (first > 0) {
    first--;
    output[first] = output[first+period];
  }

  /*extend right end*/
  nextend = min2 (originalsize-2, 2*npad+size-1 - last);
  for (i = 0; i < nextend; i++) {
    output[++last] = symmetry*output[originallast-1-i];
  }

  /*should have full period now -- extend periodically*/
  while (last < 2*npad+size-1) {
    last++;
    output[last] = output[last-period];
  }
}

/*----------------------------------------------------------------------------*/
/* Do periodic extension of data using prescribed symmetries */
/*   Original values are in output[npad] through output[npad+size-1] */
/*   New values will be placed in output[0] through output[npad] and in */
/*      output[npad+size] through output[2*npad+size-1] (note: end values may */
/*      not be filled in) */

void periodic_ext (double *output, int size, int npad)
{
  int first = npad, last = npad + size-1;

  /* extend left periodically*/
  while (first > 0) {
    first--;
    output[first] = output[first+size];
  }

  /* extend right periodically*/
  while (last < 2*npad+size-1) {
    last++;
    output[last] = output[last-size];
  }
}
```

```
/*--------------------------------------------------------------------------*/

void transform2(wavelet awavelet,double *input,double *output,int size,int symt)
{
	/*a transform routine for the horizonal part of the wavelet transform. This tranform
	part functions by having each processor send and receive D elements from it's neighbour.
	This approach seems to work */

	int i,j;
	int first,last;
	int lowsize=(size+1)/2;
	int leftext,rightext;
	if (awavelet.analysislow->size%2) {
		/*i.e.an odd filter length*/
		leftext=1;
		rightext=1;
	}else {
		leftext=2;
		rightext=2;
	}


	if (symt)
	{
		symmetric_ext(input,size,leftext,rightext,awavelet.npad,1);
	}

	else {
		first = awavelet.npad;
		last = awavelet.npad + size-1;

	/* extend left periodically*/
	while (first > 0) {
  first--;
  input[first] = input[first+size];
		}
	/*i.e. add the necessary extensions to make the wavelet transform work*/
	/*the detail will become xxxxxxxx->HHHHGGGG */
	}

	/*first use low pass filter*/
	for (i=0;i<lowsize;i++) {
		output[awavelet.npad+i]=0.0;
		for (j=0;j<awavelet.analysislow->size;j++) {
			output[awavelet.npad+i]+=input[awavelet.npad+2*i+awavelet.analysislow->firstindex+j]
*(awavelet.analysislow->coeff[j]);
		}
	}

	/*now use high pass filter*/
	for (i=lowsize;i<size;i++) {
		output[awavelet.npad+i]=0.0;
		for (j=0;j<awavelet.analysishigh->size;j++) {
			output[awavelet.npad+i]+=input[awavelet.npad+2*(i-lowsize)+awavelet.analysishigh->fi
```

```
rstindex+j]*(awavelet.analysishigh->coeff[j]);
                    }
          }
}
```

A3: Software Listing for MPI Buddy 2D DWT Program

*Except for the main program file, startproj.c, all other files are the same as the regular parallel case.

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
/*user may insert other include statements below this line */
#include "image.h"
#include "filter.h"

double *dotransform(wavelet usewavelet,double *theimage,int steps,int my_rank,int p,MPI_Status status,int hsize,
int vsize);
/*user must define using his/her own non-standard data structure types */

void main( int argc, char* argv[])
{
        /*--Automatic Code Generation of MPI Header / Ender --*/
        int i;
        int vsize;
        int hsize;
        int loc_hsize;
        int position;
        int totalsize;
        double *startdata;
        double *locdata;
        double *enddata;
        int my_rank;        /*rank # of current processes*/
        int p;      /*variable for number of processes*/
        int tag= 0;         /*default tag for send/recv*/
        MPI_Status status;          /*return Status for MPI_Recv*/
        /*user may put other user defined variable declarations below this line */
        wavelet validwavelet;
        image* mainImage;
        filterset *thefilter;
        char *inputfile;
        char *outputfile;
        int steps;
        double starttime,endtime;

        double daub8coeffs[] = { 0.2303778133088964,  0.7148465705529154,
                             0.6308807679398587, -0.0279837694168599,
                            -0.1870348117190931,  0.0308413818355607,
                             0.0328830116668852, -0.0105974017850690 };

        /*-----Start Up MPI-----*/
        MPI_Init(&argc,&argv);

        /*Find out Process Rank*/
        MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
        /*Find out the number of processes*/
```

```
MPI_Comm_size(MPI_COMM_WORLD,&p);

/*User may insert Application Specific Code Below*/

totalsize=512*512;    /*the total data size to distribute*/
startdata=malloc(totalsize*sizeof(double));
locdata=malloc(totalsize/p*sizeof(double));
enddata=malloc(totalsize*sizeof(double));
hsize=512;
loc_hsize=hsize/p;
vsize=512*512/hsize;

if (my_rank==0) {
        /*User Should define mainImage->data here*/

        mainImage=(image*)malloc(sizeof(image));
        inputfile=malloc(81);
        outputfile=malloc(81);
        inputfile="barbara.pgm";
        outputfile="outfile.pgm";
        steps=3; /*number of steps in the transform*/
        getheader(mainImage,inputfile);
        startdata=mainImage->data;
}

thefilter=makedaub(FALSE,daub8coeffs,8,0);
validwavelet=initwavelet(thefilter);

MPI_Bcast(&steps,1,MPI_INT,0,MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);
starttime=MPI_Wtime();

for (i=0;i<vsize; i++) {
        position=i*hsize;

MPI_Scatter(startdata+position,loc_hsize,MPI_DOUBLE,locdata+(i*loc_hsize),loc_hsize,MPI_DOUBLE,0,MPI_C
OMM_WORLD);
};

/*Every process has local data in locdata */
/*user may insert own information here*/
locdata=dotransform(validwavelet,locdata,steps,my_rank,p,status,loc_hsize,vsize);

for (i=0;i<vsize;i++) {
        position=i*loc_hsize;

MPI_Gather(locdata+position,loc_hsize,MPI_DOUBLE,enddata+i*hsize,loc_hsize,MPI_DOUBLE,0,MPI_COMM_
WORLD);
}

MPI_Barrier(MPI_COMM_WORLD);
endtime=MPI_Wtime();
```

```
        /* process 0 has result in enddata */
        if (my_rank==0) {
                writeoutput(enddata,outputfile,mainImage->hsize,mainImage->vsize);
                printf("\nelapsed time is %e \n",endtime-starttime);
        }


        /*End of Application Specific Code*/
        MPI_Finalize();
        return;
}

double *dotransform(wavelet usewavelet,double *theimage,int steps,int my_rank,int p,MPI_Status status,int hsize,
int vsize)
{
        /*thie routine has been modified to allow for parallel comutations*/
        int i;
        int lowsizeH=hsize;
        int lowsizeV=vsize;
        int highsizeH,highsizeV;
        int symmetric=usewavelet.symmetric;
        int npad=usewavelet.npad;
        int nonlocal=npad-2; /*number of coefficients that must be transmitted nonlocally*/
        double *tempin,*tempout,*hold;
        double *tempdata;
        long position=hsize;
        double *holdpad;
        hsize=lowsizeH;
        vsize=lowsizeV;
        tempdata=(double*)malloc(hsize*vsize*sizeof(double));
        tempin=(double*)malloc((2*npad+max2(hsize,vsize))*sizeof(double)); /*i.e. padding on both sides*/
        tempout=(double*)malloc((2*npad+max2(hsize,vsize))*sizeof(double));
        holdpad=(double*)malloc(nonlocal*lowsizeV*sizeof(double));  /*hold values for sends/receives*/
        hold=(double*)malloc(nonlocal*lowsizeV*sizeof(double));
        /*initialize the tranform first*/

        copy(theimage,tempdata,hsize*vsize); /*copy image to a temp location*/

        while (steps--)
        {
                if ((lowsizeH<=2 || lowsizeV<=2) &&symmetric==1) {
                        warning("reduce the number of transform setps of increase the signal size");
                        warning("or switch to a periodic extension wavlet set");
                        error("low pass subband is too small");
                }
                /*now do a convolution with the low pass part of each row */
                /*make an array of data that needs to be communicated*/
                for (i=0;i<lowsizeV;i++) {
                        copy(tempdata+i*hsize,holdpad+i*nonlocal,nonlocal);
                }
                /*only communicate terms once per round*/
                if ((my_rank%2)==0) {
                                /*i.e. do the even ranks first*/
```

```
MPI_Send(holdpad,nonlocal*lowsizeV,MPI_DOUBLE,(my_rank+p-1)%p,0,MPI_COMM_WORLD);

MPI_Recv(hold,nonlocal*lowsizeV,MPI_DOUBLE,(my_rank+p+1)%p,0,MPI_COMM_WORLD,&status);
                        }
                        else if ((my_rank%2)==1) {
                                /*now do the odd procesor ranks*/

MPI_Recv(hold,nonlocal*lowsizeV,MPI_DOUBLE,(my_rank+p+1)%p,0,MPI_COMM_WORLD,&status);

MPI_Send(holdpad,nonlocal*lowsizeV,MPI_DOUBLE,(my_rank+p-1)%p,0,MPI_COMM_WORLD);
                        }

                for (i=0;i<lowsizeV;i++) {
                        copy(tempdata+(i*hsize),tempin+npad,lowsizeH);
                        copy(hold+i*nonlocal,tempin+(npad+lowsizeH),nonlocal); /*ie the new padded values */
                        /*convolve with low and high pass filters*/
                        /*first complete sends and receives in this routine for horizontal*/

                        transform2(usewavelet,tempin,tempout,lowsizeH,symmetric);
                        /*now copy back to the image*/
                        copy(tempout+npad,tempdata+(i*hsize),lowsizeH);
                }
                /*now covolve on the low pass portion of each column*/
                /*each processor contains all necessary information for padding here*/

                for (i=0;i<lowsizeH;i++) {
                        /*copy each column i into the data array*/
                        copy2(tempdata+i,hsize,tempin+npad,lowsizeV);
                        /*now convove with low and high pass filters*/
                        transform(usewavelet,tempin,tempout,lowsizeV,symmetric);
                        copy3(tempout+npad,tempdata+i,hsize,lowsizeV);
                }
                /*stay in while loop - do row and column convolutions until steps*/
                /*have all been completed*/
                highsizeH=lowsizeH/2;
                lowsizeH=(lowsizeH+1)/2;
                highsizeV=(lowsizeV)/2;
                lowsizeV=(lowsizeV+1)/2;
        } /*end of while*/

        return tempdata; /*contains the modified image*/
}
```

Appendix B

Software Listing for 1D Fast Fourier Transform
B1: Sequential Case
B2: Parallel Hand Coded Case
B3: MPI Buddy Coded Case

B1: Software Listing for Sequential 1D FFT Program

```
##include "complex.h"
#include "mpi.h"
/* this will be the benchmark timing program to compare against*/
/*make this into a suitable MPI program for comparison*/
/*modified by Narjit Chadha, March 13, 2001*/

void main(int argc, char* argv[])

{
    unsigned int    N;
    unsigned long   length,half_length;
    unsigned long   i;
                        double start,finish;
    double  interval = 2.0 * M_PI;
    double  factor;
    double  max = 0.0;
    Complex*        data;

    /**************************************************/
            MPI_Init(&argc,&argv);
  data = (Complex*) malloc(length*sizeof(Complex));
            /* now start the program */
            printf("\nEnter the power of 2 for the data : ");
            scanf("%d",&N);
            length=1<<N;
            MPI_Barrier(MPI_COMM_WORLD);
            start=MPI_Wtime();

            half_length=length>>1;
            factor= 1.0/(double) length;
            data= (Complex*) malloc(length*sizeof(Complex));
            for (i=0; i<length; i++) {
                        data[i].re=(double)i;
                        data[i].im=(double)i;
            }
            for (i = 0; i < length; i++) {
                        data[i].re = (double)i;
                        data[i].im = (double)i;
            }


            fft_1(data, N, -1);

            MPI_Barrier(MPI_COMM_WORLD);
            finish=MPI_Wtime();

            printf("\nthe elapsed time is %e : ", finish-start);

            free(data);

            return;
}
```

```
/* complex.h - header file for complex.c */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define M_PI 3.14159265358979323846

struct acomplex {
            double re;
```

```
        double im;
};

typedef struct acomplex Complex;


void copy(Complex* z, Complex a);
void polar(Complex* c, double r, double t);
void cmult(Complex* c, Complex a, Complex b);
void cexp(Complex* c, Complex a);
void scale(Complex* c, double x, Complex a);
int bit_reversal(unsigned int N, unsigned long l);
void fft_1(Complex* data, unsigned int N, int isign);
```

```
#include    "complex.h"


/*****************************************************************************/
/*complex.c contains mathimatical routines for complex analysis*/
/*****************************************************************************/

void
copy(Complex* z, Complex a)
{
        z->re = a.re;
        z->im = a.im;
}

/*****************************************************************************/

void
polar(Complex* c, double r, double t)
{
        c->re = r * cos(t);
        c->im = r * sin(t);
}

/*****************************************************************************/

void
cmult(Complex* c, Complex a, Complex b)
{
        c->re = a.re*b.re - a.im*b.im;
        c->im = a.re*b.im + a.im*b.re;
}

/*****************************************************************************/

void
cexp(Complex* c, Complex a)
{
        polar(c, exp(a.re), a.im);
}

/*****************************************************************************/

void
scale(Complex* c, double x, Complex a)
{
        c->re = x * a.re;
        c->im = x * a.im;
}

/*****************************************************************************/
```

```
int bit_reversal(unsigned int N, unsigned long l)
{
		int			i, result = 0;
		unsigned long		pow;

		for (i = 0; i < abs(N); ++i) {
			pow = 1<<i;
			if (l & pow)
				result += 1<<(N-i-1);
		}
		return result;
}

void
fft_1(Complex* data, unsigned int N, int isign)
{
	unsigned long	length = 1 << N;
	unsigned long	half_length = length >> 1;

	unsigned long	i, m, pow, t, t_pow;

	Complex*	w;
	Complex*	tmp;
	Complex		z1, z2, phase;

	z1.re = 0.0;
	z1.im = 0.0;
	z2.re = 0.0;
	z2.im = 0.0;

	phase.re = 0.0;
	phase.im = M_PI * ((double) isign) / (double) half_length;

	/************************************************/


	w = (Complex*) malloc(half_length*sizeof(Complex));

	w[0].re = 1.0;
	w[0].im = 0.0;
	cexp(w+1, phase);

	for (i = 2; i < half_length; ++i) {
		scale(&z1, ((double) i), phase);
		cexp(w+i, z1);
		/* cmult(w+i, w[i-1], w[1]); */
	}

	/************************************************/

	tmp = (Complex*) malloc(length*sizeof(Complex));
	for (i = 0; i < length; ++i)
		copy(tmp+bit_reversal(N, i), data[i]);

	/************************************************/

	for (m = 0; m < N; ++m) {
		pow = 1 << m;

		for (t = 0; t < length; ++t) {
			if (!(t & pow)) {
				t_pow = (t*(half_length/pow)) % half_length;
				copy(&z1, tmp[t]);
				cmult(&z2, w[t_pow], tmp[t+pow]);
				tmp[t].re = z1.re + z2.re;
				tmp[t].im = z1.im + z2.im;
				tmp[t+pow].re = z1.re - z2.re;
```

```
                    tmp[t+pow].im = z1.im - z2.im;
                }
            }
        }

        /************************************************/

        for (i = 0; i < length; ++i)
            copy(data+i, tmp[i]);

        /************************************************/

        free(tmp);
        free(w);
}

/*****************************************************************************/
```

B2: Software Listing for Parallel Hand Coded 1D FFT Program

*complex.h and complex.c remain the same as for the sequential case given in B1.

```
#include <stdio.h>
#include <stdlib.h>
#include "complex.h"
#include "mpi.h"
#define M_PI 3.14159265358979323846

/*this is the mpi code for a parallel itertive fast fourier tranform*/
/*made by Narjit Chadha, March 1, 2001*/

void main (int argc,char* argv[])
{
        int my_rank;  /*rank of the process*/
        int p;      /*the number of processes*/
        int root=0;  /*rank of the root processor=0*/
        double start, finish;
        unsigned int N;
        unsigned long length,half_length,loc_size,count,loc_N;
        unsigned long startmark,endmark;
        unsigned long j,i,s,m,k;
        double factor;
        double max=0.0;
        double interval=2.0*M_PI;
        int direction;

        Complex* data;
        Complex w;
        Complex wm;
        Complex* A;
        Complex* tmp; /* to hold data only for processor 1*/
        Complex u,t,phase;
        double one=1;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&p);
        MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

        /* now start the program using MPI*/
        MPI_Barrier(MPI_COMM_WORLD);
        start=MPI_Wtime();

        if (my_rank==0) {
                /*
                printf("\nEnter the power of 2 to determine the data : ");
                scanf("%d",&N);
                */
                N=20;
                length=1<<N;
                factor=1.0/(double)length;
        } /* if my_rank==0*/
```

```
        if (my_rank==0){
        factor= 1.0/(double) length;
        data= (Complex*) malloc(length*sizeof(Complex));
        for (i=0; i<length; i++) {
        data[i].re=(double)i;
        data[i].im=(double)i;
        } /*processor 0 has the data*/
                /*now do bit reversal to reorder terms for algorithm*/
        tmp = (Complex*) malloc(length*sizeof(Complex));
        for (i = 0; i < length; ++i) {
        copy(tmp+bit_reversal(N, i), data[i]);
        }
    }


    /*now broadcast these parts -packing is a waste of computaional power*/
    MPI_Bcast(&N,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
    MPI_Bcast(&factor,1,MPI_DOUBLE,0,MPI_COMM_WORLD);

    length=1<<N;
    half_length=length>>1;
    loc_size=length/p; /*number of elements in each local array*/
    /*now distribute the tmp to processors in blockwise fashion*/
    A = (Complex*) malloc(loc_size*sizeof(Complex)); /*keep as a global*/

    MPI_Scatter(tmp,loc_size*sizeof(Complex),MPI_CHAR,A,loc_size*sizeof(Complex),MPI_CHAR,0,MPI_
COMM_WORLD);
    /*A of size loc_size*/
    /*every processor has an array of size loc_size - must perform local operations
    and communications operations*/

    u.re=0.0;
    u.im=0.0;
    t.re=0.0;
    t.im=0.0;
    phase.re=0.0; /*this initializaion is still necessary*/
    count=loc_size;
    loc_N=0;
    while (count>1)
    {
        count=count/2;
        loc_N++;
    }
    /*loc_N=N/p; number of iterations to perform locally- assume N is divisible by p*/
    /*do local computations first-assume balanced array*/
    for (s=1; s<(loc_N+1); s++) {
            m=1<<s;
            w.re=1.0;  /*w=1 to start off*/
            w.im=0.0;
            phase.re=0.0;
            phase.im=-interval/m;
            cexp(&wm,phase);  /*w(1).re=exp(phase.re)*cos(t),w(1).img=exp(phase.re)*sin(t)*/
```

```
				for (j=0; j<m/2;j++)
			{
				for (k=j; k<loc_size;k=k+m) {

					copy(&u,A[k]);  /*z1=u*/
					cmult(&t, w, A[k+m/2]);  /*z2=t*/
					A[k].re = u.re + t.re;
					A[k].im = u.im + t.im;
					A[k+m/2].re = u.re - t.re;
					A[k+m/2].im = u.im - t.im;
				}
				cmult(&w,w,wm);
			}
		}

		MPI_Gather(A,loc_size*sizeof(Complex),MPI_CHAR,tmp,loc_size*sizeof(Complex),MPI_CHAR,0,MPI_
	COMM_WORLD);
		/*Gather everything into processor 0 tmp before proceeding further*/
		if (my_rank==0) {

			for (s=loc_N+1; s<N+1; s++) {
					m=1<<s;
					w.re=1.0;  /*w=1 to start off*/
					w.im=0.0;
					phase.re=0.0;
					phase.im=-interval/m;
					cexp(&wm,phase);
/*w(1).re=exp(phase.re)*cos(t),w(1).img=exp(phase.re)*sin(t)*/
					for (j=0; j<m/2;j++) {
					for (k=j; k<length;k=k+m) {

						copy(&u,tmp[k]);  /*z1=u*/
						cmult(&t, w, tmp[k+m/2]);  /*z2=t*/
						tmp[k].re = u.re + t.re;
						tmp[k].im = u.im + t.im;
						tmp[k+m/2].re = u.re - t.re;
						tmp[k+m/2].im = u.im - t.im;
					}
					cmult(&w,w,wm);
				}
			}
			for (i = 0; i < length; ++i)
				copy(data+i, tmp[i]);
			free(tmp);
		} /*my_rank==0*/

		MPI_Barrier(MPI_COMM_WORLD);
		finish=MPI_Wtime();

		if (my_rank==0) {
			printf("\nthe elapsed time is %e : ", finish-start);
			/*now print out the data */
		/*	for (i=(length-10); i<(length); i++)
			{
```

```
                        printf("\ndata at %d is %lf +j*%lf", i,data[i].re,data[i].im);
                } */
        } /*if my_rank==0*/

        free(A);
        free(data);
        MPI_Finalize();
        return;
}
```

B3: Software Listing for MPI Buddy Coded 1D FFT Program

*complex.h and complex.c remain the same as for the sequential case given in B1.

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
/*user may insert other include statements below this line */
#include "complex.h"
#define M_PI 3.14159265358979323846
/*user must define using his/her own non-standard data structure types */

void main( int argc, char* argv[])
{
        /*--Automatic Code Generation of MPI Header / Ender --*/
        int totalCount;  /*the total size of data to distribute*/
        int sendCount;
        Complex *tmp;
        Complex *A;
        Complex *finBuffer;
        int my_rank;        /*rank # of current processes*/
        int p;      /*variable for number of processes*/
        int tag= 0;           /*default tag for send/recv*/
        MPI_Status status;            /*return Status for MPI_Recv*/
        /*user may put other user defined variable declarations below this line */
        unsigned int N;
        unsigned long length, half_length, loc_size,count,loc_N;
        double start,finish;
        unsigned long j,i,s,m,k;
        double factor;
        double max=0.0;
        double interval=2.0*M_PI;
        int direction;

        Complex *data;
        Complex w;
        Complex wm;
        Complex u,t,phase;
        double one=1;

        /*-----Start Up MPI-----*/
        MPI_Init(&argc,&argv);

        /*Find out Process Rank*/
        MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
        /*Find out the number of processes*/
        MPI_Comm_size(MPI_COMM_WORLD,&p);

        /*User may insert Application Specific Code Below*/
        MPI_Barrier(MPI_COMM_WORLD);
        start=MPI_Wtime();
        if (my_rank==0) {
```

```
		/*printf("\nEnter the power of 2 to determine the data : ");
		scanf("%d", &N);*/
		N=20;
		length=1<<N;
		factor=1.0/(double)length;
	}


	MPI_Bcast(&N,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
	MPI_Bcast(&factor,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
	length=1<<N;
	half_length=length>>1;
	loc_size=length/p;

	totalCount=(length)*sizeof(Complex);   /*the total data size to send*/
	sendCount=totalCount/p;
	tmp=malloc(totalCount);
	A=malloc(sendCount);
	finBuffer=malloc(totalCount);

	if (my_rank==0) {
		/*User Should define tmp here*/
		data=(Complex*)malloc(length*sizeof(Complex));
		for (i=0;i<length;i++) {
			data[i].re=(double)i;
			data[i].im=(double)i;
			copy(tmp+bit_reversal(N,i),data[i]);
		}
	}



	MPI_Scatter(tmp,sendCount, MPI_CHAR, A, sendCount, MPI_CHAR, 0, MPI_COMM_WORLD);

	/*scattered data in all processes in A */
	/*------Enter Program Specific Code Here------*/
	u.re=0.0;
	u.im=0.0;
	t.re=0.0;
	t.im=0.0;
	phase.re=0.0;
	count=loc_size;
	loc_N=0;
	while (count>1)
	{
		count=count/2;
		loc_N++;
	}

	for (s=1; s<(loc_N+1); s++) {
			m=1<<s;
			w.re=1.0;   /*w=1 to start off*/
			w.im=0.0;
			phase.re=0.0;
```

```
                phase.im=-interval/m;
                cexp(&wm,phase);

                for (j=0; j<m/2;j++)
        {
                for (k=j; k<loc_size;k=k+m) {

                        copy(&u,A[k]);  /*z1=u*/
                        cmult(&t, w, A[k+m/2]);  /*z2=t*/
                        A[k].re = u.re + t.re;
                        A[k].im = u.im + t.im;
                        A[k+m/2].re = u.re - t.re;
                        A[k+m/2].im = u.im - t.im;
                }
                cmult(&w,w,wm);
        }
}


MPI_Gather(A, sendCount, MPI_CHAR, finBuffer, sendCount, MPI_CHAR, 0, MPI_COMM_WORLD);
 /*result in process 0 array finBuffer */
if (my_rank==0) {
        tmp=finBuffer;
        for (s=loc_N+1; s<N+1; s++) {
                        m=1<<s;
                        w.re=1.0;
                        w.im=0.0;
                        phase.re=0.0;
                        phase.im=-interval/m;
                        cexp(&wm,phase);
                        for (j=0; j<m/2;j++) {
                        for (k=j; k<length;k=k+m) {

                                copy(&u,tmp[k]);  /*z1=u*/
                                cmult(&t, w, tmp[k+m/2]);  /*z2=t*/
                                tmp[k].re = u.re + t.re;
                                tmp[k].im = u.im + t.im;
                                tmp[k+m/2].re = u.re - t.re;
                                tmp[k+m/2].im = u.im - t.im;
                        }
                        cmult(&w,w,wm);
                }
        }
        for (i = 0; i < length; ++i)
                copy(data+i, tmp[i]);
        free(tmp);
} /*my_rank==0*/

MPI_Barrier(MPI_COMM_WORLD);
finish=MPI_Wtime();

if (my_rank==0) {
                printf("\nthe elapsed time is %e : ", finish-start);
/*      for (i=(length-10); i<(length); i++)
```

```
		{
			printf("\ndata at %d is %lf +j*%lf", i,data[i].re,data[i].im);
		}*/
	} /*if my_rank==0*/



	/*End of Application Specific Code*/
	MPI_Finalize();
	return;
}
```