IMPLEMENTATION PATTERNS FOR PARALLEL PROGRAM AND A CASE STUDY

By

EUNKEE KIM

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

Dedicated to Grace Kim, Jineun Song and parents

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my adviser, Dr. Beverly A. Sanders, for providing me with an opportunity to work in this exciting area and for providing feedback, guidance, support, and encouragement during the course of this research and my graduate academic career.

I wish to thank Dr. Joseph N. Wilson and Dr. Stephen M. Thebaut for serving on my supervisory committee.

Finally I thank Dr. Berna L. Massingill for allowing me to use machines in Trinity University and for technical advice.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS iv

LIST OF TABLES ix

LIST OF FIGURES x

ABSTRACT xi

CHAPTER

1 INTRODUCTION 1

1.1 Parallel Computing 1
1.2 Parallel Design Patterns 1
1.3 Implementation Patterns for Design Patterns 2
1.4 Implementation of Kernel IS of NAS Parallel Benchmark Set Using Parallel Design Patterns. 2
1.5 Organization of the Thesis 3

2 OVERVIEW OF PARALLEL PATTERN LANGUAGE 4

2.1 Finding Concurrency Design Space 4
2.1.1 Getting Started 4
2.1.2 Decomposition Strategy 4
2.1.3 Task Decomposition 5
2.1.4 Data Decomposition 5
2.1.5 Dependency Analysis 6
2.1.6 Group Tasks 6
2.1.7 Order Tasks 7
2.1.8 Data Sharing 7
2.1.9 Design Evaluation 7
2.2 Algorithm Structure Design Space 8
2.2.1 Choose Structure 8
2.2.2 Asynchronous Composition 9
2.2.3 Divide and Conquer 9

2.2.4 Embarrassingly Parallel 10
2.2.5 Geometric Decomposition 10
2.2.6 Pipeline Processing 11
2.2.7 Protected Dependencies 11
2.2.8 Recursive Data 11
2.2.9 Separable Dependency 12
2.3 Supporting Structures Design Space 12
2.3.1 Program-Structuring Group 12
2.3.1.1 Single program and multiple data 12
2.3.1.2 Fork join 13
2.3.1.3 Master worker 13
2.3.1.4 Spawn 13
2.3.2 Shared Data Structures Group 13
2.3.2.1 Shared queue 13
2.3.2.2 Shared counter 14
2.3.2.3 Distributed array 14
2.4 Chapter Summery 14

3 PATTERNS FOR IMPLEMENTATION 15

3.1 Using Message Passing Interface (MPI) 15
3.1.1 Intent 15
3.1.2 Applicability 15
3.1.3 Implementation 15
3.1.3.1 Simple message passing 16
3.1.3.2 Group and communicator creation 17
3.1.3.3 Data distribution and reduction 22
3.2 Simplest Form of Embarrassingly Parallel 23
3.2.1 Intent 23
3.2.2 Applicability 23
3.2.3 Implementation 24
3.2.4 Implementation Example 25
3.2.5 Example Usage 27
3.3 Implementation of Embarrassingly Parallel 28
3.3.1 Intent 28
3.3.2 Applicability 28
3.3.3 Implementation 28
3.3.4 Implementation Example 29
3.3.5 Usage Example 38
3.4 Implementation of Pipeline Processing 39
3.4.1 Intent 39
3.4.2 Applicability 39
3.4.3 Implementation 39
3.4.4 Implementation Example 41
3.4.5 Usage Example 47
3.5 Implementation of Asynchronous-Composition 48
3.5.1 Intent 48

3.5.2 Applicability 49
3.5.3 Implementation 49
3.5.4 Implementation Example 50
3.6 Implementation of Divide and Conquer 54
3.6.1 Intent 54
3.6.2 Motivation 54
3.6.3 Applicability 55
3.6.4 Implementation 55
3.6.5 Implementation Example 57
3.6.6 Usage Example 61

4 KERNEL IS OF NAS BENCHMARK 63

4.1 Brief Statement of Problem 63
4.2 Key Generation and Memory Mapping 63
4.3 Procedure and Timing 64

5 PARALLEL PATTERNS USED TO IMPLEMENT KERNEL IS 66

5.1 Finding Concurrency 66
5.1.1 Getting Started 66
5.1.2 Decomposition Strategy 67
5.1.3 Task Decomposition 68
5.1.4 Dependency Analysis 68
5.1.5 Data Sharing Pattern 68
5.1.6 Design Evaluation 69
5.2 Algorithm Structure Design Space 70
5.2.1 Choose Structure 70
5.2.2 Separable Dependencies 71
5.2.3 Embarrassingly Parallel 71
5.3 Using Implementation Example 71
5.4 Algorithm for Parallel Implementation of Kernel IS 72

6 PERFORMANCE RESULTS AND DISCUSSIONS 74

6.1 Performance Expectation 74
6.2 Performance Results 74
6.3 Discussions 76

7 RELATED WORK AND CONCLUSIONS AND FUTURE WORK 78

7.1 Related work 78
7.1.1 Aids for Parallel Programming 78
7.1.2 Parallel Sorting 80
7.2 Conclusions 81
7.3 Future Work 82

APPENDIX

A KERNEL IS OF THE NAS PARALLEL BENCHMAKRK 83

B PSEUDORANDOM NUMBER GENERATOR 90

C SOURCE CODE OF THE KERNEL IS IMPLEMENTATION 94

D SOURCE CODE OF PIPELINE EXAMPLE 107

E SOURCE CODE OF DIVIDE AND CONQUER 115

LIST OF REFERENCES 124

BIOGRAPHICAL SKETCH 127

LIST OF TABLES

Table page

4-1 Parameter Values to be used for Benchmark 65

6-1 Performance Results for Class A and B 75

A-1 Values to be used for Partial Verification 88

A-2 Parameter Values to be used for Benchmark 89

LIST OF FIGURES

Figure page

3-1 The Algorithm Structure Decision Tree 9

3-2 Usage of Message Passing 40

3-3 Irregular Message Handling 50

3-4 Message Passing for Invocation of the Solve and Merge Functions 56

4-1 Counting Sort 67

6-1 Execution Time Comparison for Class A 76

6-2 Execution Time Comparison for Class B 76

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

IMPLEMENTATION PATTERNS FOR PARALLEL PROGRAM AND A CASE STUDY

By

Eunkee Kim

December 2002

Chairman: Beverly A. Sanders
Major Department: Computer and Information Science and Engineering

Design patterns for parallel programming guides the programmer through the entire process of developing a parallel program. In this thesis, implementation patterns for parallel program are presented. These patterns help programmers to implement a parallel program after designing it using the parallel design patterns.

Parallel Integer sorting, one of the kernels of Parallel Benchmark set of the Numerical Aerodynamic Simulation Program (NAS), has been designed and implemented using the parallel design pattern as a case study. How the parallel design pattern was used in implementing Kernel IS and its performance are discussed

CHAPTER 1
INTRODUCTION

1.1 Parallel Computing

Parallel computing is what a computer does when it carries out more than one computation at a time using many processors. An example of parallel computing (processing) in our daily life is an automobile assembly line: at every station somebody is doing part of the work to complete the product. The purpose of parallel computing is to overcome the limit of the performance of a single processor. We can increase the performance of a sequential program by parallelizing exploitable concurrency in a sequential program and using many processors at once.

Parallel programming has been considered more difficult than sequential programming. To make it easier for programmers to write a parallel program, a parallel design pattern has been developed by B.L. Massingill, T.G. Mattson, and B.A Sanders.[1-4]

1.2 Parallel Design Patterns

The Parallel pattern language is a collection of design patterns for parallel programming.[5-6] Design patterns are a high level description of a solution to a problem.[7] Parallel pattern language is written down in a systematic way so that a final design for a parallel program can result by going through a sequence of appropriate patterns from a pattern catalog. The structure of the patterns is designed to parallelize even complex problems. The top-level patterns help to find the concurrency in a problem and decompose it into a collection of tasks. The second-level patterns help to find an

appropriate algorithm structure to exploit concurrency that has been identified. The parallel design patterns are described in more detail in Chapter 2.

1.3 Implementation Patterns for Design Patterns

Several implementation patterns for design patterns in the algorithm structure design space were developed as a part of the parallel pattern language and presented in this thesis. These implementation patterns are solutions of problems mapping high-level parallel algorithms into programs using a Message Passing Interface (MPI) and programming language C.[8] The patterns of the algorithm structure design space capture recurring solutions to the problem of turning problems into parallel algorithms. These implementation patterns can be reused and provide guidance for programmers who might need to create their own implementations after using the parallel design pattern. The implementation patterns of design patterns in the algorithm structure design space are in Chapter 3.

1.4 Implementation of Kernel IS of NAS Parallel Benchmark Set Using Parallel Design Patterns

The Numerical Aerodynamic Simulation (NAS) Program, which is based at NASA Ames Research Center, developed parallel benchmark set for the performance evaluation of highly parallel supercomputers. The NAS Parallel Benchmark set is a "Paper and Pencil" benchmark.[9] All details of this benchmark set are specified only algorithmically. The Kernel IS of NAS Parallel Benchmark set is a parallel sorting over large numbers of integers. A solution to the Kernel IS is designed and implemented using parallel pattern language as a case study. The used patterns are following.

Getting Start, Decomposition Strategy, Task Decomposition, Dependency Analysis, Data Sharing, and Design Evaluation patterns of Find Concurrency Design Space are

used. Choose Structure pattern, Separable Dependency, and Embarrassingly Parallel patterns of the algorithm structure design space are also used. The details of how these patterns are used and the final design of the parallel program for Kernel IS are in Chapter 5.

1.5 Organization of the Thesis

The research that has been conducted as part of this thesis and its organization are as follows:

- Analysis of the parallel design patterns (Chapter 2)
- Implementation patterns for patterns in Algorithm structure design space (Chapter 3)
- A description of Kernel IS of the NAS Parallel Benchmark set (Chapter 4)
- Design and implementation of Kernel IS through the parallel design patterns (Chapter 5)
- Performance results and discussions (Chapter 6).
- Conclusions and future work (Chapter 7).

CHAPTER 2
OVERVIEW OF PARALLEL PATTERN LANGUAGE

Parallel pattern language is a set of design patterns that guide the programmer through the entire process of developing a parallel program.[10] The patterns of parallel pattern language, as developed by Masingill et al., are organized into four design spaces: Finding Concurrency Design Space, Algorithm Structure Design Space, Supporting Structure Design Space, and Implementation Design Space.

2.1 Finding Concurrency Design Space

The finding concurrency design space includes high-level patterns that help to find the concurrency in a problem and decompose it into a collection of tasks.

2.1.1 Getting Started

The getting started pattern helps to start designing a parallel program. Before using these patterns, a user of this pattern needs to be sure that the problem is large enough or needs to speed up and understand the problem. The user of this pattern needs to do the following tasks:

- Decide which parts of the problem require most intensive computation.
- Understand the tasks that need to be carried out and data structure that are to be manipulated.

2.1.2 Decomposition Strategy

This pattern helps to decompose the problem into relatively independent entities that can execute concurrently. To expose the concurrency of the problem, the problem can be decomposed along two different dimensions:

- Task Decomposition: Break the stream of instructions into multiple chunks called tasks that can execute simultaneously.
- Data Decomposition: Decompose the problem's data into chunks that can be operated relatively independently.

2.1.3 Task Decomposition

The Task Decomposition pattern addresses the issues raised during a primarily task-based decomposition. To do task decomposition, the user should try to look at the problem as a collection of distinct tasks. And these tasks can be found in the following places:

- Function calls may correspond to tasks.
- Each iteration of a loop, if the iterations are independent, can be the tasks.
- Updates on individual data chunks decomposed from a large data structure.

The number of tasks generated should be flexible and large enough. The tasks should have enough computation.

2.1.4 Data Decomposition

This pattern looks at the issues involved in decomposing data into units that can be updated concurrently. The first point to be considered is whether the data structure can be broken down into chunks that can be operated on concurrently. An array-based computation and recursive data structures are examples of this approach. The points to be considered in decomposing data are as follows:

- Flexibility in the size and number of data chunks to support the widest range of parallel systems
- The size of data chunks large enough to offset the overhead of managing dependency
- Simplicity in data decomposition

2.1.5 Dependency Analysis

This pattern is applicable when the problem is decomposed into tasks that can be executed concurrently. The goal of a dependency analysis is to understand the dependency among the tasks in detail. Data-sharing dependencies and ordering constraints are the two kinds of dependencies that should be analyzed. The dependencies must require little time to manage relative to computation time and easy to detect and fix errors. One effective way of analyzing dependency is following approach.

- Identify how the tasks should be grouped together.
- Identify any ordering constraints between groups of tasks.
- Analyze how tasks share data, both within and among groups.

These steps lead to the Group Tasks, the Order Tasks, and the Data Sharing patterns.

2.1.6 Group Tasks

This pattern constitutes the first step in analyzing dependencies among the tasks of problem decomposition. The goal of this pattern is to group tasks based on the following constraints. Three major categories of constraints among tasks are as follows:

- A temporal dependency: a constraint placed on the order in which a collection of tasks executes.
- An ordering constraint that a collection of tasks must run at the same time.
- Tasks in a group are truly independent.

The three approaches to group tasks are as follows:

- Look at how the problem is decomposed. The tasks that correspond to a high-level operation naturally group together. If the tasks share constraints, keep them as a distinct group.
- If any other task groups share the same constraints, merge the groups together.
- Look at constraints between groups of tasks.

2.1.7 Order Tasks

This pattern constitutes the second step in analyzing dependencies among the tasks of problem decomposition. The goal of this pattern is to identify ordering constraints among task groups. Two goals are to be met in defining this ordering:

- It must be restrictive enough to satisfy all the constraints to be sure the resulting design is correct.
- It should not be more restrictive than it need be.

To identify ordering constraints, consider the following ways tasks can depend on each other:

- First look at the data required by a group of tasks before they can execute. Once this data have been identified, find the task group that created it, and you will have an order constraint.
- Consider whether external services can impose ordering constraints.
- It is equally important to note when an order constraint does not exit.

2.1.8 Data Sharing

This pattern constitutes the third step in analyzing dependencies among the tasks of problem decomposition. The goal of this pattern is to identify what data are shared among groups of tasks and how to manage access to shared data in a way that is both correct and efficient. The following approach can be used to determine what data is shared and how to manage the data:

- Identify data that are shared between tasks.
- Understand how the data will be used.

2.1.9 Design Evaluation

The goals of this pattern are to evaluate the design so far and to prepare for the next phase of the design space. This pattern therefore describes how to evaluate the design

from three perspectives: suitability for the target platform, design quality, and preparation for the next phase of the design.

For the suitability for the target platform, the following issues included:

- How many processing elements are available?
- How are data structures shared among processing elements?
- What does the target architecture imply about the number of units of execution and how structures are shared among them?
- On the target platform, will the time spent doing useful work in a task be significantly greater than the time taken to deal with dependencies?

For the design quality, simplicity, flexibility, and efficiency should be considered.

To prepare the next phase, the key issues are as follows:

- How regular are the tasks and their data dependencies?
- Are interactions between tasks (or task groups) synchronous or asynchronous?
- Are the tasks grouped in the best way?

2.2 Algorithm Structure Design Space

The algorithm structure design space contains patterns that help to find an appropriate algorithm structure to exploit the concurrency that has been identified.

2.2.1 Choose Structure

This pattern guides the algorithm designer to the most appropriate Algorithm-Structure patterns for the problem. Consideration of Target Platform, Major Organizing Principle and Algorithm-Structure Decision Tree are the main topics of this pattern.

The two primary issues of Consideration of Target Platform are how many units of execution (threads or processes) the target system will effectively support and the way information is shared between units of execution. The three major organizing principles are “organization by ordering,” “organization by tasks,” and “organization by data.” The

Algorithm-Structure Decision Tree is in Figure 3.1. We can select an algorithm-structure using this decision tree.

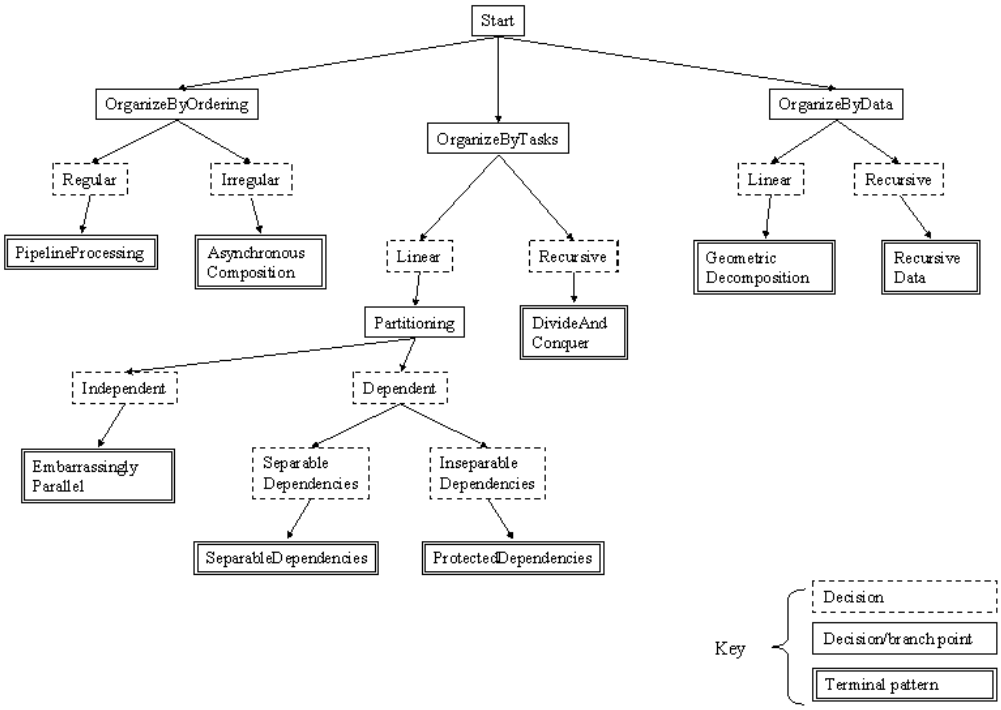

Figure 3-1 The Algorithm Structure Decision Tree

2.2.2 Asynchronous Composition

This pattern describes what may be the most loosely structured type of concurrent program in which semi-independent tasks interact through asynchronous events. Two examples are the Discrete-Event Simulation and Event-Driven program. The key issues in this pattern are how to define the tasks/entities, how to represent their interaction, and how to schedule the tasks.

2.2.3 Divide and Conquer

This pattern can be used for the parallel application program based on the well-known divide-and-conquer strategy. This pattern is particularly effective when the amount of

work required to solve the base case is large compared to the amount of work required for the recursive splits and merges. The key elements of this pattern are as follows:

- Definitions of the functions such as “solve,” “split,” “merge,” “baseCase,” “baseSolve”
- A way of scheduling the tasks that exploits the available concurrency

This pattern also includes Correctness issues and Efficiency issues.

2.2.4 Embarrassingly Parallel

This pattern can be used to describe concurrent execution by a collection of independent tasks and to show how to organize a collection of independent tasks so they execute efficiently.

The key element of this pattern is a mechanism to define a set of tasks and schedule their execution and also a mechanism to detect completion of the tasks and terminate the computation. This pattern also includes the correctness and efficiency issues.

2.2.5 Geometric Decomposition

This pattern can be used when the concurrency is based on parallel updates of chunks of a decomposed data structure, and the update of each chunk requires data from other chunks. Implementations of this pattern include the following key elements:

- A way of partitioning the global data structure into substructures or “chucks”
- A way of ensuring that each task has access to all the data needed to perform the update operation for its chunk, including data in chunks corresponding to other tasks
- A definition of the update operation, whether by points or by chunks
- A way of assigning the chunks among units of execution (distribution), that is a way of scheduling the corresponding tasks

2.2.6 Pipeline Processing

This pattern is for algorithms in which data flow through a sequence of tasks or stages. This pattern is applicable when the problem consists of performing a sequence of calculations, each of which can be broken down into distinct stages on a sequence of inputs. So for each input, the calculations must be done in order, but it is possible to overlap computation of different stages for different inputs. The key elements of this pattern are as follows:

- A way of defining the elements of the pipeline where each element corresponds to one of the functions that makes up the computation
- A way of representing the dataflow among pipeline elements, i.e., how the functions are composed
- A way of scheduling the tasks

2.2.7 Protected Dependencies

This pattern can be used for task-based decompositions in which the dependencies between tasks cannot be separated from the execution of the tasks, and the dependencies must be dealt with in the body of the task. The issues of this pattern are as follows:

- A mechanism to define a set of tasks and schedule their execution onto a set of units of executions
- Safe access to shared data
- Shared memory available when it is needed

2.2.8 Recursive Data

This pattern can be used for parallel applications in which an apparently sequential operation on a recursive data is reworked to make it possible to operate on all elements of the data structure concurrently. The issues of this pattern are as follows:

- A definition of the recursive data structure plus what data is needed for each element of the structure

- A definition of the concurrent operations to be performed
- A definition of how these concurrent operations are composed to solve the entire problem
- A way of managing shared data
- A way of scheduling the tasks onto units of executions
- A way of testing its termination condition, if the top-level structure involves a loop.

2.2.9 Separable Dependency

This pattern is used for task-based decompositions in which the dependencies between tasks can be eliminated as follows: Necessary global data are replicated and (partial) results are stored in local data structures. Global results are then obtained by reducing results from the individual tasks. The key elements of this pattern are as follows:

- Defining the tasks and scheduling their execution
- Defining and updating a local data structure
- Combining (reducing) local objects into a single object

2.3 Supporting Structures Design Space

The patterns at this level represent an intermediate stage between the problem-oriented patterns of the algorithm structure design space and the machine-oriented patterns of the Implementation-Mechanism Design Space.

Patterns in this space fall into two main groups: Program-Structuring Group and Shared Data Structures Group.

2.3.1 Program-Structuring Group

Patterns in this group deal with constructs of structuring the source code.

2.3.1.1 Single program and multiple data

The computation consists of *N* units of execution in parallel. All *N* UEs (Generic term for concurrently executable entity, usually either process or thread) execute the same

program code, but each operates on its own set of data. A key feature of the program code is a parameter that differentiates among the copies.

2.3.1.2 Fork join

A main process or thread forks off some number of other processes or threads that then continue in parallel to accomplish some portion of the overall work before rejoining the main process or thread.

2.3.1.3 Master worker

A master process or thread sets up a pool of worker processes or threads and a task queue. The workers execute concurrently with each worker repeatedly removing a task from the task queue and processing it until all tasks have been processed or some other termination condition has been reached. In some implementations, no explicit master is present.

2.3.1.4 Spawn

A new process or thread is created, which then executes independently of its creator. This pattern bears somewhat the same relation to the others as GOTO bears to the constructs of structured programming.

2.3.2 Shared Data Structures Group

Patterns in this group describe commonly used data structures.

2.3.2.1 Shared queue

This pattern represents a "thread-safe" implementation of the familiar queue abstract data type (ADT), that is, an implementation of the queue ADT that maintains the correct semantics even when used by concurrently executing units of execution.

2.3.2.2 Shared counter

This pattern, as with the Shared Queue pattern, represents a “thread-safe” implementation of a familiar abstract data type, in this case a counter with an integer value and increment and decrement operations.

2.3.2.3 Distributed array

This pattern represents a class of data structures often found in parallel scientific computing, namely, arrays of one or more dimensions that have been decomposed into subarrays and distributed among processes or threads.

2.4 Chapter Summery

This chapter describes an overview of the parallel pattern languages for an application program which guides programmers from the design process of the program to the implementation point. The Next chapter illustrates how these design patterns have been used to design and implement the Kernel IS of the NAS Parallel Benchmark.

CHAPTER 3
PATTERNS FOR IMPLEMENTATION

In this chapter, we will introduce patterns for the implementation of patterns in the algorithm structure design space. Each pattern contains Implementation Example for each pattern in the algorithm structure design space. The Implementation Examples are implemented in an MPI and C environment. The MPI is standards for "Message Passing Interface" in Distributed Memory Environments. The Implementation Examples may need modification occasionally, but will be reusable and helpful for an implementation of the parallel program designed using parallel pattern language.

3.1 Using Message Passing Interface (MPI)

3.1.1 Intent

This pattern is an introduction of Message Passing Interface (MPI).

3.1.2 Applicability

This pattern is applicable when the user of parallel pattern language has finished the design of his parallel program and considers implementing it using MPI.

3.1.3 Implementation

The MPI is a library of functions (in C) or subroutines (in Fortran) that the user can insert into a source code to perform data communication between processes. The primary goals of MPI are to provide source code portability and allow efficient implementations across a range of architectures.

3.1.3.1 Simple message passing

Message passing programs consist of multiple instances of a serial program that communicates by library calls. The elementary communication operation in MPI is "point-to-point" communication, that is, direct communication between two processes, one of which *sends* and the other *receives.* This message passing is the basic mechanism of exchanging data among processes of a parallel program using MPI. This message passing is also a good solution about how to order the executions of tasks belongs to a group and how to order the executions of groups of tasks.

To communicate among processes, communicators are needed. Default communicator of MPI is MPI_COMM_WORLD which includes all processes. The common implementation of MPI executes same programs concurrently at each processing element (CPU or workstation). The following program is a very simple example of using MPI_Send (send) and MPI_Recv (receive) functions.

```
/* simple send and receive */

/* Each process has the same copy of the following program and executes it */
#include <stdio.h>
#include <mpi.h>

#define BUF_SIZE 100
int tag =1;
void main (int argc, char **argv) {

  int myrank;               /* rank (identifier) of each process*/
  MPI_Status status;       /* status object */
  double a[BUF_SIZE];  /*Send(Receive) buffer */

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank of process*/

  if( myrank == 0 ) {
     /* code for process 0 */
     /* send Message */
     MPI_Send(
```

```
            a,                  /*initial address of send buffer */
            BUF_SIZE,           /*number of elements in send buffer */
            MPI_DOUBLE,         /*datatype of each send buffer element */
            1,                  /*rank of destination */
            tag,                /* message tag */
            MPI_COMM_WORLD /*communicator */
            );
 }
 else if( myrank == 1 ){
  /* code for process 1 */
  /*receive message */
  MPI_Recv( a,              /*initial address of receive buffer */
            BUF_SIZE,       /*number of elements in send buffer */
            MPI_DOUBLE, /*datatype of each receive buffer element */
            0,              /*rank of source */
            tag,            /*message tag */
            MPI_COMM_WORLD, /*communicator */
            &status         /*status object */
            );
 }
 /* more else if statement can be added for more processes */
 /* switch statements can be used instead of if else statement s*/

 MPI_Finalize();        /* Terminate MPI */
}
```

3.1.3.2 Group and communicator creation

A communicator in MPI specifies the communication context for communication operations and the set of processes that share this communication context. To make communication among processes in a program using MPI, we must have communicators. There are two kinds of communicators in MPI: intra-communicator and inter-communicator. Intra-communicators are for message passing among processes belong to the same group. Inter-communicators are for message passing between two groups of processes. As mentioned previously, the execution order of tasks belonging to one group can be implemented using intra-communicator and message-passing. The execution order of groups of tasks can be implemented using inter-communicator and message-passing.

To create a communicator, a group of processes is needed. MPI does not provide mechanism to build a group from scratch, but only from other, previously defined groups.

The base group, upon which all other groups can be created, is the group associated with the initial communicator MPI_COMM_WORLD. A group of processes can be created by the five steps as follows:

1. Decide the processing units that will be included in the groups
2. Decide the base groups to use in creating new group.
3. Create group.
4. Create communicator.
5. Do the work.
6. Free communicator and group.

The first step is decide how many and which processing units will be included in this group. The points to be considered are the number of available processing units, the number of tasks in one group, the number of groups that can be executed concurrently, etc., according to the design of the program.

If only one group of tasks can executed because of dependency among groups than the group can use all available processing units and may include them in that group. If there are several groups that can be executed concurrently, then the available processing units may be divided according to the number of groups that can be executed concurrently and the number of tasks in each group.

Since MPI does not provide a mechanism to build a group from scratch, but only from other, previously defined groups, the group constructors are used to subset and superset existing groups. The base group, upon which all other groups can be defined, is the group

associated with the initial communicator MPI_COMM_WORLD, and processes in this group are all the processes available when MPI is initialized.

There are seven useful group constructors in MPI. Using these constructors, groups can be constructed as designed. The first three constructors are similar to the union and intersection of set operation in mathematics. The seven group constructors are as follows: The MPI_GROUP_UNION creates a group which contains all the elements of the two groups used. The MPI_GROUP_INTERSECTION creates a group which contains all the elements that belong in both groups at the same time. The MPI_GROUP_DIFFERENCE creates a group which has all the elements that do not belong in both groups at the same time. The MPI_GROUP_INCL routine creates a new group that consists of the specified processes in the array from the old group. The MPI_GROUP_EXCL routine creates a new group that consists of the processes not specified in the array from the old group. The MPI_GROUP_RANGE_INCL routine includes all processes between one specified process to another process, and the specified processes themselves. The MPI_GROUP_RANGE_EXCL routine excludes all processes not between the specified processes, and also excludes the specified processes themselves.

For the creation of communicators, there are two types of communicators in MPI: intra-communicator and inter-communicator. The important intra-communicator constructors in MPI are as follows: The MPI_COMM_DUP function will duplicate the existing communicator. The MPI_COMM_CREATE function creates a new communicator for a group.

For the communication between two groups, the inter-communicator for the identified two groups can be created by using the communicator of each group and peer-

communicator. The routine for this purpose is MPI_INTERCOMM_CREATE. The peer-communicator must have at least one selected member process from each group. Using duplicated MPI_COMM_WORLD as a dedicated peer communicator is recommended.

The Implementation Example Code is as follows:

```
/*****************************************/
/* An Example of creating intra-communicator   */
/*****************************************/
#include <mpi.h>
main(int argc, char **argv)
{
 int myRank, count, count2;
 int *sendBuffer, *receiveBuffer, *sendBuffer2, *receiveBuffer2;
 MPI_Group MPI_GROUP_WORLD, grprem;
 MPI_Comm commSlave;
 static int ranks[] = { 0 };
 /* ... */
 MPI_Init(&argc, &argv);
 MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
 MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

 /* Build group for slave processes */
 MPI_Group_excl(
                MPI_GROUP_WORLD, /* group */
                 1,                          /* number of elements in array ranks */
                 ranks,                      /* array of integer ranks in group not to appear
                                                in new group */
                 &grprem);                   /* new group derived from above */

 /* Build communicator for slave processes */
 MPI_Comm_create(
                MPI_COMM_WORLD, /* communicator */
                grprem,                     /* Group, which is a subset of the group of
                                               above communicator */
                &commSlave);                /* new communicator */

 if(myRank !=0)
   {
    /* compute on processes other than root process */
    /* ... */

    MPI_Reduce(sendBuffer, receiveBuffer, count, MPI_INT, MPI_SUM, 1,
                commSlave);
```

```
    /* ... */
   }

  /* Rank zero process falls through immediately to this reduce,
     others do later… */
  MPI_Reduce(sendBuffer2, receiveBuffer2, count2, MPI_INT, MPI_SUM, 0,
             MPI_COMM_WORLD);
  MPI_Comm_free(&commSlave);
  MPI_Group_free(&MPI_GROUP_WORLD);
  MPI_Group_free(&grprem);
  MPI_Finalize();
}


/*****************************************/
/* An example of creating inter-communicator    */
/*****************************************/
#include <mpi.h>

main(int argc, char **argv)
{
  MPI_Comm myComm;       /* intra-communicator of local sub-group */
  MPI_Comm myInterComm; /* inter-communicator between two group */
  int membershipKey;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* User generate membership Key for each group */
  membershipKey = rank % 3;

  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(
          MPI_COMM_WORLD, /* communicator */
          membershipKey,          /* Each subgroup contains all processes
                                     of the same membershipKey */
          rank,                   /* Within each subgroup, the processes are
                                     ranked in the order defined by
                                     this rank value*/
          &myComm                 /* new communicator */
           );

  /* Build inter-communicators. Tags are hard-coded */
  if (membershipKey == 0)
    {
```

```
    MPI_Intercomm_create(
                        myComm, /* local intra-communicator */
                        0,          /* rank of local group leader */
                        MPI_COMM_WORLD, /* "peer" intra-communicator */
                        1,          /* rank of remote group leader
                                       in the "peer" communicator */
                        1,          /* tag */
                        &myInterComm /* new inter-communicator */
                        );
  }
 else
  {
   MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,0,1,
                        &myInterComm);
  }

 /* do work in parallel */

 if(membershipKey == 0)
  MPI_Comm_free(&myInterComm);
 MPI_Finalize();
}
```

3.1.3.3 Data distribution and reduction

There are several primitive functions for distribution of data in MPI. The three basic functions are as follows: The MPI_BROADCAST function replicates data in one root process to all other processes so that every process has the same data. This function can be used for a program which needs replication of data in each process to improve performance.

The MPI_SCATTER function of MPI can scatter data from one processing unit to every other processing unit with the same amount (or variant amounts) of data, and each processing unit will have a different portion of the original data. The Gather function of MPI is the inverse of Scatter.

For the reduction of data distributed among processes, the MPI provides two primitive MPI_REDUCE and MPI_ALL_REDUCE functions. These reduce functions are useful

when it is needed to combine locally computed subsolutions into one global solution. The MPI_REDUCE function combines the elements provided in the input buffer of each process in the group using the specified operation as a parameter, and returns the combined value in the output buffer of the process with rank of root. The MPI_ALL_REDUCE function combines the data in the same way as the MPI_REDUCE, but every process has the combined value in the output buffer. This function is beneficial when the combined solution can be used as an useful information for the next phase of computation.

The Implementation Example Code is not provided in this section, but the Implementation Example Code of 3.2.4 can be used as an Implementation Example Code. The source code of the Kernel IS implementation is also a good Implementation Example Code.

3.2 Simplest Form of Embarrassingly Parallel

3.2.1 Intent

This is a solution to the problem of how to implement the simplest form of Embarrassingly Parallel pattern in the MPI and C environment.

3.2.2 Applicability

This pattern can be used after the program is designed using patterns of the finding concurrency design space and patterns of algorithm structure design space. And, the resulted design is a simplest form of the Embarrassingly Parallel pattern. The simplest form of Embarrassingly Parallel pattern satisfies the conditions as follows: All the tasks are independent. All the tasks must be completed. Each task executes same algorithm on a distinct section of data.

3.2.3 Implementation

The common MPI implementations have a static process allocation at initialization time of the program. It also executes same program at each processing unit (or workstation). Since all the tasks executes same algorithm on a different data, the tasks can be implemented as one high level function which will be executed in each process.

The simplest form of Embarrassingly Parallel pattern can be implemented by the following five steps.

1. Find out the number of available processing units.
2. Distribute or replicate data to each process.
3. Execute the tasks.
4. Synchronize all processes.
5. Combine (reduce) local results into a single result.

In MPI, the number of available processing units can be found by calling the MPI_COMM_SIZE function and passing the MPI_WORLD_COM communicator as a parameter.

In most cases, after finding out the number of available processing units, data can be divided by that number and distributed into each processing unit. There are several primitive functions for distribution of data in MPI. The three basic functions are *Broadcast*, *Scatter*, and *Gather*. These functions are introduced in 3.1.3.3.

After computing local results at each process, synchronize all processes to check that all local results are computed. This can be implemented by calling the MPI_BARRIER function after each local result is solved at each processing unit because the MPI_BARRIER blocks the caller until all group members have called it.

The produced local results can be combined to get the final solution of the problem. The *Reduce* functions of MPI are useful in combining subsolutions. The Reduce function of MPI combines the elements, which are provided in the input buffer of each process in the group, using the specified operation as a parameter, and returns the combined value in the output buffer of the process with rank of root. The operations are distributed to each process in many MPI implementations so they can improve overall performance of the program if it takes considerable time to combine subsolutions in one processing unit.

3.2.4 Implementation Example

```
#include <mpi.h>
#define SIZE_OF_DATA  2000   /* size of data to be distributed into each process */
  int root = 0;              /* The rank(ID) of root process */
  int myRank;                /* process ID */

 /*= Modification1=========================================*/
 /*= The data type of each variable should be modified      =*/
  int* data;                /*= The data to be distributed into each process  =*/
  int* localData;           /*= distributed data that each process will have =*/
  int* subsolution;         /*= subsolutaion data                            =*/
  int* solution;            /*= solution data                                =*/
 /*=========================================================*/

  int sizeOfLocalData;     /* number of elements in local data array */
  int numOfProc;           /* number of avaiable process          */
  int numOfElement;        /* number of element in subsolution */


  /*************************************************/
  /** Highest level function which solves subproblem              */
  /*************************************************/
  void solve()
  { /*= Implementation1=============================*/
    /*=The code for solving subproblem should be implemented  =*/
    /*=========================================*/
  }


  /*************************************************/
  /*  Main Function which starts program                         */
  /*************************************************/
```

```
main(argc, argv)
  int argc;
  char **argv;
{
  MPI_Status status;

  /* MPI initialization */
  MPI_Init(&argc, &argv);

  /* Finding out the rank (ID) of each process */
  MPI_Comm_rank(MPI_COMM_WORLD,&myRank);

  /*Finding out the number of available processes   */
  MPI_Comm_size(MPI_COMM_WORLD,&numOfProc);

  /* Dividing the data by the number of processes   */
  sizeOfLocalData = SIZE_OF_DATA/numOfProc;

  /* Dynamic memory allocation for local data */
  localData=(int *)malloc(sizeOfLocalData*sizeof(int));

  /*****************************************************************/
  /*     Distribute data into local data of each process           */
  /*****************************************************************/
 MPI_Scatter(  data,              /* Address of send buffer (data) to distribute   */
               sizeOfLocalData,   /* Number of element sent to each process      */
               MPI_INT,           /*MODIFICATION2==================*/
                                  /*= data type of the send buffer              =*/
                                  /*================================*/
               localData,         /* Address of receive buffer (local data)      */
               sizeOfLocalData,   /* Number of element in receive buffer (local  */
                                  /* data)                                       */
               MPI_INT,           /*= MODIFICATION2================*/
                                  /*= Data type of receive buffer               =*/
                                  /*================================*/
               root,              /* Rank of sending process                     */
               MPI_COMM_WORLD  /* Communicator                                   */
             );

  /*solve sub-problem in each process */
  solve();

  /*Synchronize all processes to check all subproblems are solved */
  MPI_Barrier(MPI_COMM_WORLD);

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

```
    /* Combine sub-solutions to get the solution                    */
    /*************************************************************/
    MPI_Reduce(  subsolution,        /* address of send buffer (subsolution) */
                 solution,           /* address of receive buffer (solution)  */
                 numOfElement,       /* number of elements in send buffer
                                        (subsolution)                 */
                 MPI_INT,          /*= MODIFICATION3============*/
                                   /*=  data type of the send buffer      =*/
                                   /*================================*/
                 MPI_MULT,         /*= MODIFICATION4 ============*/
                                   /*=  reduce operation                  =*/
                                   /*================================*/
                 root,               /* rank of root process             */
                 MPI_COMM_WORLD  /* communicator                         */
               );

    MPI_Finalize();
}
```

3.2.5 Example Usage

The steps to reuse the Implementation Example are as follows:

- Implement the blocks labeled as IMPLEMENTATION1. This block should contain code for the “solve” function.
- Modify the data type of the variables which are in the block labeled as MODIFICATION1.
- Modify the data type parameters which are in the block labeled as MODIFICATION2. Each data type parameter must be one of MPI data type which matches with the type of the data to receive and send.
- Modify the data type parameters which are in the block labeled as MODIFICATION3. Each data type parameter must be one of MPI data type which matches with the type of the data to receive and send
- Modify the reduce operator parameter which are in the block labeled as MODIFICATION4. This parameter should be one of MPI operators.

Consider a simple addition of each element in an integer array with size 1024. What should be modified is the solve function and fifth parameter of MPI_Reduce function as follows:

void solve()

```
{
        for(i=0;i<sizeOfLocalData;i++)
        {
        subSolution=subSolution+localData[i];
        }
}

MPI_Reduce(    subsolution,
                    solution,
                    numOfElement,
                    MPI_INT,            /*Modified */
                    MPI_SUM,            /* Modified */
                    root,
                    MPI_COMM_WORLD
                );
```

3.3 Implementation of Embarrassingly Parallel

3.3.1 Intent

This pattern shows how to implement Embarrassingly Parallel pattern in MPI and C environment

3.3.2 Applicability

This pattern can be used after the program is designed using patterns of finding concurrency design space and patterns of the algorithm structure design space. A simple form of the Embarrassingly-Parallel-Algorithm-Structure pattern is that the order of priority of all the tasks is known and all tasks must be completed and all the tasks are independent. The Implementation Example is for the case that the resulted design is a simple form of the Embarrassingly Parallel Algorithm Structure pattern.

3.3.3 Implementation

The common implementation of MPI executes same programs concurrently at each processing element and makes message passing on a needed basis. Since the design of the parallel program has independent tasks, those tasks can be executed without restriction on the order of tasks execution.

If each task has known amount of computations, the implementation of a parallel program is straightforward in MPI and C environment. Each task can be implemented as a function and can be executed at each process using process ID (rank) and *if else* statement or *switch* statement. Since the amount of computation of each task, the load balance should be achieved by distributing the tasks among processes when the program is implemented.

If the amount of computation is not known, there should be some dynamic tasks scheduling mechanism for the load balance. The MPI does not provide primitive process scheduling mechanism. One way of simulating dynamic tasks scheduling mechanism is using a task-name queue and primitive message passing of MPI. We implemented this mechanism as follows: Each task is implemented as a function. The root process (with rank 0) has a task-name queue which contains the names of tasks (functions). Each process sends message to the root process to get task name, whenever it is idle or finished its task. The root process with rank 0 waits for messages from other processes. If the root process receives messages and the task queue is not empty, it sends back the name of task in the task queue to the sender process of that message. When the process, which sent message for a task name, receives the task name, it executes the task (function). The Implementation Example is provided in 3.3.4.

The MPI routines, which is useful for the combining the locally computed subsolutions into a global solution, are described in 3.1.3.3

3.3.4 Implementation Example

```
#include<mpi.h>
#include <stdlib.h>

#define EXIT -1 /* exit message */
#define TRUE 1
```

```
#define FALSE 0
#define ROOT 0
#define NUM_OF_ALGO 9

int oneElm =1; /* one element */
int tag =0 ;
int algoNameSend= -1;
int idleProc = -1;
int algoNameRecv= -1;

/* Size of Data to send and Receive */
int sizeOfData;

/* number of tasks for each algorithm */
int numOfTask[NUM_OF_ALGO] ;

int isAlgorithmName = TRUE;
int moreExitMessage = TRUE;
MPI_Status status;


/*********************************************************/
/* Simple Queue implementation.                          */
/*********************************************************/

int SIZE_OF_Q = 20;
int* queue;
int qTail = 0;
int qHead = 0;

/*******************************/
/* insert                      */
/*******************************/
void insert(int task){
  if(qHead-qTail==1)
    { /* when queue is full, increase array */
      int* temp =  (int *) malloc(2*SIZE_OF_Q*sizeof(int));
      int i;
      for(i=0;i<SIZE_OF_Q;i++)
           {
             temp[i]=remove();
           }
      queue=temp;
      SIZE_OF_Q = SIZE_OF_Q*2;
    }
  else if(qTail<SIZE_OF_Q-1 && qHead-qTail >2){
```

```
   /* queue is not full and there is more than one space */
   qTail++;
   queue[qTail]= task;

  }
  else{
   /* there is just one more space in queue */
   qTail=0;
   queue[qTail]= task;
  }
}
/*************************************/
/* check whether or not queue is empty          */
/*************************************/
int isEmpty()
{
  if(qTail==qHead){/* queue is empty */
    return 1;
  }
  else{
    return 0;
  }
}
/*************************/
/* remove                          */
/*************************/
int remove()
{
  if(qHead<SIZE_OF_Q-1){
    qHead++;
    return queue[qHead];
  }
  else {    qHead =0;
  return queue[SIZE_OF_Q - 1];
  }
  return 0;
}

/*********************************************************************/
/* Each function should correspond to an algorithm for tasks.                 */
/* If several tasks have same algorithm, then one function for them           */
/* More algorihtm for tasks can be added as a function                        */
/*********************************************************************/

void algorithm1(){
  /*= DATA TRANSFER 1  ================================*/
```

```
 /*= the data which will be used by this algorithm in local process      =*/
/*= More of this block can be added on needed basis                      =*/

 int* data; /* Modify the data type*/
 /* receive the size of data to receive */
MPI_Recv(&sizeOfData,oneElm,MPI_INT,ROOT,
              tag,MPI_COMM_WORLD,&status);

 /* dynamic memory allocation */
 data = (int*/* Modify the data type*/ )
        malloc(sizeOfData*sizeof(int/* Modify the data type*/));

 /* Receive the input data  */
 MPI_Recv(&data,sizeOfData,
           MPI_INT,     /*= <- Modify the data type  =*/
           ROOT,tag,MPI_COMM_WORLD,&status);
/*=                                                                      =*/
/*=======================================================*/

 /*=IMPLEMENTATION1  ================================*/
 /*= code for this algorithm should be implemented                      =*/
 /*=======================================================*/

 /* Memory deallocation ex) free(data);*/
}
void algorithm2(){

 /*= DATA TRANSFER 1  ================================*/
 /*= the data which will be used by this algorithm in local process      =*/
/*= More of this block can be added on needed basis                      =*/
 int* data; /* Modify the data type*/
 /* receive the size of data to receive */
MPI_Recv(&sizeOfData,oneElm,MPI_INT,ROOT,
              tag,MPI_COMM_WORLD,&status);

 /* dynamic memory allocation */
 data = (int*/* Modify the data type*/ )
        malloc(sizeOfData*sizeof(int/* Modify the data type*/));

 /* Receive the input data  */
 MPI_Recv(&data,sizeOfData,
           MPI_INT,     /*= <- Modify the data type  =*/
           ROOT,tag,MPI_COMM_WORLD,&status);
/*=                                                                      =*/
/*=======================================================*/
```

```
 /*=IMPLEMENTATION1  ================================*/
 /*= code for this algorithm should be implemented                =*/
 /*=================================================*/

 /* Memory deallocation ex) free(data);*/
}

void algorithm3(){
/*= DATA TRANSFER 1  ================================*/
 /*= the data which will be used by this algorithm in local process    =*/
 /*= More of this block can be added on needed basis                   =*/

 int* data; /* Modify the data type*/
 /* receive the size of data to receive */
 MPI_Recv(&sizeOfData,oneElm,MPI_INT,ROOT,
              tag,MPI_COMM_WORLD,&status);

 /* dynamic memory allocation */
 data = (int*/* Modify the data type*/ )
        malloc(sizeOfData*sizeof(int/* Modify the data type*/));

 /* Receive the input data  */
 MPI_Recv(&data,sizeOfData,
           MPI_INT,     /*= <- Modify the data type  =*/
           ROOT,tag,MPI_COMM_WORLD,&status);
/*=                                                              =*/
/*=================================================*/

/*=IMPLEMENTATION1  ================================*/
/*= code for this algorithm should be implemented                =*/
/*=================================================*/

 /* Memory deallocation ex) free(data);*/}

/*************************/
/* main method           */
/*************************/
void main( argc, argv )
    int argc;
    char **argv;
{
 int myRank;
 int mySize;

 int i;
 int j;
```

```
MPI_Init(&argc, &argv);

/*find local rank*/
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

/*find out last rank by using size of rank*/
MPI_Comm_size(MPI_COMM_WORLD,&mySize);

 /*************************************************************/
 /* This process distributes tasks to other processing elements */
 /*************************************************************/

if(myRank == 0)
 {/* code for root process.
      This root process receives messages from other processes when they are idle.
      Then this process removes an algorithm name for a task from task name queue
      and send it back to sender of the  message.
      If the task queue is empty, this process sends back an exit message */

  /*=MODIFICATION1==================================*/
  /*= the data for each algorithm                   =*/
  /*= the data type should be modified              =*/
  int* algo1Data;
  int* algo2Data;
  int* algo3Data;
  int* algo4Data;
  int* algo5Data;
  /*=                                               =*/
  /*================================================*/

  int numOfSentExit = 0;

  /* This array is a task queue.  */
  queue =  (int *) malloc(SIZE_OF_Q*sizeof(int));

  /*   */
  for(i=0;i<NUM_OF_ALGO;i++)
      {
       numOfTask[i] = mySize;
      }
  for(i=0;i<NUM_OF_ALGO;i++)
      {
       for(j=0;j<numOfTask[0];j++)
      {
```

```
            insert(i);
        }
        }

/* Receive message from other processes */
while(moreExitMessage)
        {
         int destination= ROOT;
         /* Receive message from any process */
         MPI_Recv(&algoNameRecv,oneElm,MPI_INT,MPI_ANY_SOURCE,
                  MPI_ANY_TAG,MPI_COMM_WORLD,&status);
         destination = algoNameRecv;

         if(!isEmpty())
         {
           /* If the task queue is not empty, send back an algorihtm name for a task
            to the sender of received message */
           algoNameSend = queue[remove()];
           MPI_Send(&algoNameSend,oneElm,MPI_INT,
                       destination,tag,MPI_COMM_WORLD);

           switch(algoNameSend)
                {
                case 1:
                 /*=IMPLEMENTATION2============================*/
                 /*= code for calculating the size of data to send  and          =*/
                 /*= finding the starting address of data to send                =*/
                 /*= should be implemented                                       =*/
                 /*============================================*/

                 /*=DATA TRANSFER2============================*/
                 /*= More of this block can be added on needed basis             =*/
                 /*=                                                             =*/
                 MPI_Send(&sizeOfData,oneElm,MPI_INT,
                             destination,tag,MPI_COMM_WORLD);
                 MPI_Send(
                          &algo1Data, /*= <- Modify the initial address of data =*/
                          sizeOfData,
                          MPI_INT    /*= <- Modify the data type =*/
                          ,destination,tag,MPI_COMM_WORLD);
                 /*=                                                             =*/
                 /*============================================*/

                 break;
                case 2:
                 /*=IMPLEMENTATION2============================*/
```

```
        /*= code for calculating the size of data to send  and          =*/
        /*= finding the starting address of data to send                =*/
        /*= should be implemented                                       =*/
        /*==============================================*/

        /*=DATA TRANSFER2==============================*/
        /*= More of this block can be added on needed basis             =*/
        /*=                                                             =*/
        MPI_Send(&sizeOfData,oneElm,MPI_INT,
                 destination,tag,MPI_COMM_WORLD);
        MPI_Send(
                 &algo1Data, /*= <- Modify the initial address of data =*/
                 sizeOfData,
                 MPI_INT     /*= <- Modify the data type =*/
                 ,destination,tag,MPI_COMM_WORLD);
        /*=                                                             =*/
        /*==============================================*/
        break;
      case 3:
        /*=IMPLEMENTATION2=============================*/
        /*= code for calculating the size of data to send  and          =*/
        /*= finding the starting address of data to send                =*/
        /*= should be implemented                                       =*/
        /*==============================================*/

        /*=DATA TRANSFER2==============================*/
        /*= More of this block can be added on needed basis             =*/
        /*=                                                             =*/
        MPI_Send(&sizeOfData,oneElm,MPI_INT,
                 destination,tag,MPI_COMM_WORLD);
        MPI_Send(
                 &algo1Data, /*= <- Modify the initial address of data =*/
                 sizeOfData,
                 MPI_INT     /*= <- Modify the data type =*/
                 ,destination,tag,MPI_COMM_WORLD);
        /*=                                                             =*/
        /*==============================================*/

        break;
      default:
        break;
      }
  }

else
  { /*If the task queue is empty, send exit message */
```

```
            algoNameSend = EXIT;
            MPI_Send(&algoNameSend,oneElm,MPI_INT,
                        destination,tag,MPI_COMM_WORLD);

            /* keep tracking the number of exit message sent.
              If the number of exit messages is same with the number of processes,
              root process will not receive any more message requesting tasks.*/

            numOfSentExit++;
            if(numOfSentExit==mySize-1)
                {
                 moreExitMessage=FALSE;
                }
          }

         /* computation and/or message passing for combining subsolution can be
            added here */
       }
  }

/*****************************************************************/
/* Code for other processes, not root. These processes send message */
/* requesting next task to execute to root process when they are idle. */
/* These processes execute tasks received from root */
/*****************************************************************/
else
  {
   idleProc = myRank;
   /*Send message to root process, send buffer contains the rank of
       sender process whenever it is idle*/
   MPI_Send(&idleProc,oneElm,MPI_INT,ROOT,tag,MPI_COMM_WORLD);
   while(isAlgorithmName)  /*while message contains an algorihtm name */
        {
         /* Receive message from root which contains the name of task
            to execute in this process */

         MPI_Recv(&algoNameRecv,oneElm,MPI_INT,ROOT,tag,
                  MPI_COMM_WORLD,&status);


         /* each process executes tasks using the algorihtm name
            and data received from root */
         switch(algoNameRecv)
        {  /* More case statements should be added or removed, according to
              the number of tasks */
           case EXIT : isAlgorithmName = FALSE;
```

```
            break;
          case 1:
            algorithm1();
            break;
          case 2:
            algorithm2();
            break;
          case 3:
            algorithm3();
            break;
          case 4:
            algorithm4();
            break;
          case 5:
            algorithm5();
            break;
          default:
            break;
        }
         if(algoNameRecv != EXIT)
        {
            /*Send message to root process, send buffer contains the rank of
                sender process whenever it is idle*/
            idleProc = myRank;
            MPI_Send(&idleProc,oneElm,MPI_INT,ROOT,tag,MPI_COMM_WORLD);
        }
        }
   }

 /*==============================================*/
 /*= Codes for collecting subsolution and       =*/
 /*= computing final solution can be added here.=*/
 /*==============================================*/
 MPI_Finalize();
}
```

3.3.5 Usage Example

The steps to reuse this Implementation Example are as follows:

- Implement the blocks labeled as IMPLEMENTATION1. Each block should contain codes for each algorithm.

- Implement the block labeled as IMPLEMENTATION2. What should be implemented are codes for calculating the initial address and the number of elements of the send buffer for each algorithm.

- Modify the data type of the variables and data type parameters which are in the blocks labeled as DATA TRANSFER 1. The data type of each variable should match with the data type of the input data for each algorithm. The purpose of this "send" function is to send input data to the destination process. More of this block can be added on needed basis.

- Modify the data type and data type parameters which are in the block labeled as DATA TRANSFER2. Each data type parameter must be one of MPI data type which matches with the type of the data to receive. The purpose of these "receive" function is to receive input data for the algorithm execution on local process. More of this block can be added on needed basis

Assume that there is a parallel program design such that the amount of computation for each task is unknown. If some of the tasks share same algorithm, these tasks should be implemented as one algorithm function in one of the block labeled as IMPLMENTATION1. To execute the tasks, the same number of algorithm name with the number of tasks that share same algorithm should be put in the queue.

3.4 Implementation of Pipeline Processing

3.4.1 Intent

This is a solution to the problem of how to implement the Pipeline Processing pattern in the MPI and C environment.

3.4.2 Applicability

This pattern can be used after the program is designed using patterns of finding concurrency design space and patterns of the algorithm structure design space and the resulted design is a form of the Pipeline Processing pattern.

3.4.3 Implementation

Figure3.1 illustrates the usage of message passing to schedule tasks in the Implementation Example. The arrows represent the blocking synchronous send and receive in MPI. The squares labeled C1, C2, etc., in Figure3.1, represent the elements of calculations to be performed which can overlap each others. The calculation elements are

implemented as functions in the Implementation Example. The calculation elements (functions) should be filled up to do computation by the user of this Implementation Example. Adding more functions for more calculation is trivial because message passing calls inside functions are very similar between functions.

Each Stage (Stage1, Stage2, and etc.), in Figure3.1, corresponds to each process. In the Implementation Example, each Stage calls single function (calculation element) sequentially.

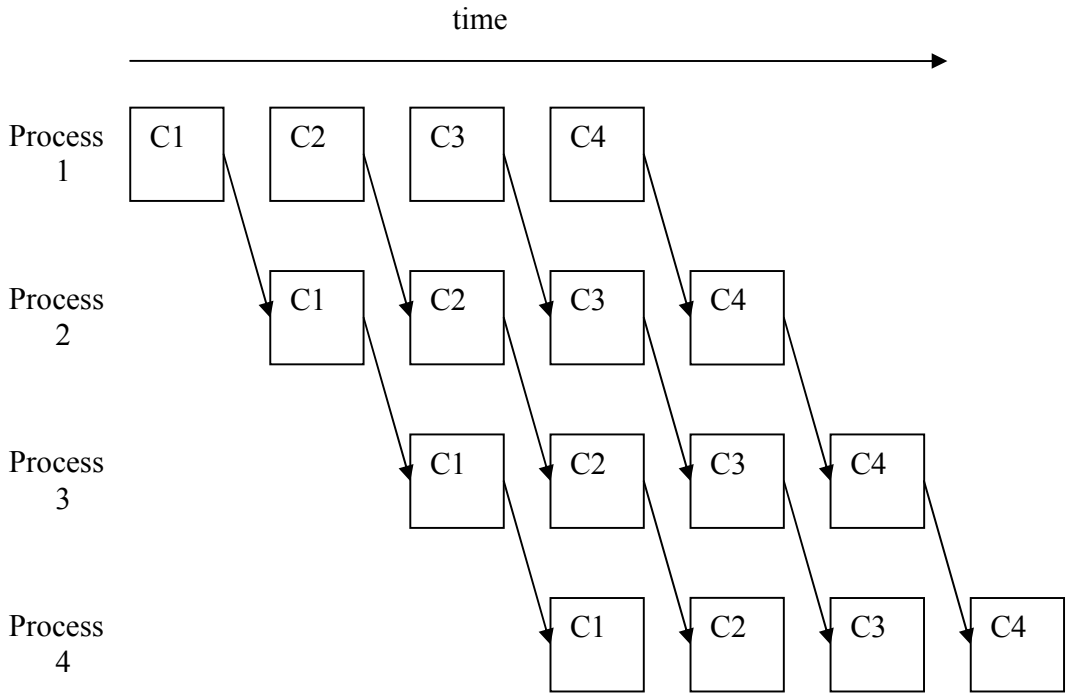

Figure 3-2 Usage of Message Passing

One point to note is that the first pipeline stage do not receive message and last element do not send message for scheduling of tasks.

Scheduling of tasks is achieved by blocking and synchronous mode point-to-point communications (MPI_SSEND, MPI_RECEIVE).

In MPI, if the Send mode is blocking and synchronous and the Receive mode is blocking, then the Send can complete only if the matching Receive has started. The statements after the matching receive will not be executed until the receive operation is finished: MPI provides synchronous communication semantics.

A duplicate of MPI_COMM_WORLD (initial intra-communicator of all processes) is used to provide separate communication space and safe communication.

If we change the duplicate of MPI_COMM_WORLD into appropriate communicator for a group of processes then, this structure can be used for a portion of a program where the Pipeline Processing pattern can be used.

3.4.4 Implementation Example

```
#include <mpi.h>

#include <stdlib.h>
#include <stdio.h>

int numOfCalElem = 4;
char startMsg[7];
int myRank;
int mySize;
int tag = 1;

char* sendBuf, recvBuf;
int sendBufSize, recvBufSize;

MPI_Status status;

/* Only four pipeline stages are implemented in this Implementation Example.
   More elements can be added or removed, according to the design
   of the parallel program */

/******************/
/*First Pipeline Stage */
/******************/
void firstPipelineStage(MPI_Comm myComm)
{
 /*=IMPLEMENTATION1========================================*/
 /*=                                                       =*/
```

```
 /*= code for this Pipeline Stage should be implemented here                =*/
 /*=                                                                        =*/
 /*=========================================================================*/

 /* send message to the next Pipeline Stage of process with myRank+1 */
 MPI_Ssend(startMsg,strlen(startMsg),MPI_CHAR
          ,myRank+1,tag,MPI_COMM_WORLD);


/*=DATA TRANSFER 1=========================================*/
/*= More send function, which has same structure, can be added  =*/
/*= to transfer data                                            =*/
 MPI_Ssend(            /*=  Modify the following parameters     =*/
          sendBuf,     /* <- starting address of buffer         =*/
          sendBufSize, /* <- the number of elements in buffer   =*/
          MPI_CHAR,    /* <- the data type                      =*/
                       /*=======================================*/
          myRank+1,tag,MPI_COMM_WORLD);
}

/********************/
/* Pipeline Stage 2      */
/********************/
void pipelineStage2(MPI_Comm myComm)
{
 /* Receive message from the previous Pipeline Stage of process with myRank-1 */
 MPI_Recv(startMsg,strlen(startMsg),MPI_CHAR,myRank-1,
          tag, MPI_COMM_WORLD,&status);

 /*=DATA TRANSFER 2=========================================*/
 /*= More receive functions, which has same structure, can be added  =*/
 /*= to transfer data                                                =*/
 MPI_Recv(             /*=  Modify the following parameters     =*/
          recvBuf,     /*= <- starting address of the buffer    =*/
          recvBufSize, /*= <- number of elements to receive     =*/
          MPI_CHAR,    /*= <- data type                        =*/
                       /*=======================================*/
          myRank-1,tag, MPI_COMM_WORLD,&status);


 /*=IMPLEMENTATION1=========================================*/
 /*=                                                        =*/
 /*= code for this Pipeline Stage should be implemented here =*/
 /*=                                                        =*/
 /*==========================================================*/
```

```
  /* send message to the next Pipeline Stage of process with myRank+1 */
  MPI_Ssend(startMsg,strlen(startMsg),MPI_CHAR,myRank+1,tag,
            MPI_COMM_WORLD);

/*=DATA TRANSFER 1=========================================*/
/*= More send function, which has same structure, can be added  =*/
/*= to transfer data                                            =*/
  MPI_Ssend(              /*=  Modify the following parameters  =*/
            sendBuf,      /* <- starting address of buffer      =*/
            sendBufSize,  /* <- the number of elements in buffer =*/
            MPI_CHAR,     /* <- the data type                   =*/
                          /*=====================================*/
            myRank+1,tag,MPI_COMM_WORLD);
}

/********************/
/* Pipeline Stage 3       */
/********************/
void pipelineStage3(MPI_Comm myComm)
{
  /* Receive message from the previous Pipeline Stage of process with myRank-1 */
  MPI_Recv(startMsg,strlen(startMsg),MPI_CHAR,myRank-1,
            tag, MPI_COMM_WORLD,&status);

  /*=DATA TRANSFER 2=========================================*/
  /*= More receive functions, which has same structure, can be added  =*/
  /*= to transfer data                                                =*/
  MPI_Recv(                 /*=  Modify the following parameters  =*/
            recvBuf,        /*= <- starting address of the buffer =*/
            recvBufSize,    /*= <- number of elements to receive  =*/
            MPI_CHAR,       /*= <- data type                      =*/
                            /*=====================================*/
            myRank-1,tag, MPI_COMM_WORLD,&status);

  /*=IMPLEMENTATION1=========================================*/
  /*=                                                        =*/
  /*= code for this Pipeline Stage should be implemented here =*/
  /*=                                                        =*/
  /*==========================================================*/

  /* send message to the next Pipeline Stage of process with myRank+1 */
  MPI_Ssend(startMsg,strlen(startMsg),MPI_CHAR,myRank+1,tag,

            MPI_COMM_WORLD);

/*=DATA TRANSFER 1=========================================*/
```

```
/*= More send function, which has same structure, can be added          =*/
/*= to transfer data                                                    =*/
  MPI_Ssend(          /*=  Modify the following parameters              =*/
          sendBuf,    /* <- starting address of buffer                  =*/
          sendBufSize, /* <- the number of elements in buffer           =*/
          MPI_CHAR,  /* <- the data type                                =*/
                      /*=======================================*/
          myRank+1,tag,MPI_COMM_WORLD);
}

/********************/
/* Pipeline Stage 4      */
/********************/
void pipelineStage4(MPI_Comm myComm)

{
  /* Receive message from the previous Pipeline Stage of process with myRank-1 */
  MPI_Recv(startMsg,strlen(startMsg),MPI_CHAR,myRank-1,
          tag, MPI_COMM_WORLD,&status);

  /*=DATA TRANSFER 2=======================================*/
  /*= More receive functions, which has same structure, can be added   =*/
  /*= to transfer data                                                 =*/
  MPI_Recv(                /*=  Modify the following parameters       =*/
          recvBuf,         /*= <- starting address of the buffer      =*/
          recvBufSize,     /*= <- number of elements to receive       =*/
          MPI_CHAR,        /*= <- data type                           =*/
                           /*=================================*/
          myRank-1,tag, MPI_COMM_WORLD,&status);

  /*=IMPLEMENTATION1=======================================*/
  /*=                                                                  =*/
  /*= code for this Pipeline Stage should be implemented here          =*/
  /*=                                                                  =*/
  /*==================================================================*/

  /* send message to the next Pipeline Stage of process with myRank+1 */
  MPI_Ssend(startMsg,strlen(startMsg),MPI_CHAR,myRank+1,tag,
          MPI_COMM_WORLD);

/*=DATA TRANSFER 1=======================================*/
/*= More send function, which has same structure, can be added          =*/
/*= to transfer data                                                    =*/
  MPI_Ssend(          /*=  Modify the following parameters              =*/
          sendBuf,    /* <- starting address of buffer                  =*/
          sendBufSize, /* <- the number of elements in buffer           =*/
```

```
            MPI_CHAR,  /* <- the data type                                  =*/
                       /*=====================================*/
            myRank+1,tag,MPI_COMM_WORLD);
}

/*******************/
/* Pipeline Stage 5        */
/*******************/
void PipelineStage5(MPI_Comm myComm)
{
 /* Receive message from the previous Pipeline Stage of process with myRank-1 */
 MPI_Recv(startMsg,strlen(startMsg),MPI_CHAR,myRank-1,
          tag, MPI_COMM_WORLD,&status);

/*=DATA TRANSFER 2=====================================*/
 /*= More receive functions, which has same structure, can be added     =*/
 /*= to transfer data                                                   =*/
 MPI_Recv(                /*=  Modify the following parameters         =*/
          recvBuf,        /*= <- starting address of the buffer        =*/
          recvBufSize,    /*= <- number of elements to receive         =*/
          MPI_CHAR,       /*= <- data type                             =*/
                          /*=====================================*/
          myRank-1,tag, MPI_COMM_WORLD,&status);



 /*=IMPLEMENTATION1=========================================*/
 /*=                                                        =*/
 /*= code for this Pipeline Stage should be implemented here =*/
 /*=                                                        =*/
 /*==========================================================*/

 /* send message to the next Pipeline Stage of process with myRank+1 */
 MPI_Ssend(startMsg,strlen(startMsg),MPI_CHAR,myRank+1,tag,
          MPI_COMM_WORLD);

/*=DATA TRANSFER 1=====================================*/
/*= More send functions, which has same structure, can be added        =*/
/*= to transfer data                                                   =*/
 MPI_Ssend(              /*=  Modify the following parameters         =*/
          sendBuf,       /* <- starting address of buffer             =*/
          sendBufSize,   /* <- the number of elements in buffer       =*/
          MPI_CHAR,      /* <- the data type                          =*/
                         /*=====================================*/
          myRank+1,tag,MPI_COMM_WORLD);
}
```

```
/*******************/
/* Last Pipeline Stage    */
/*******************/

void lastPipelineStage(MPI_Comm myComm)
{
 /* Receive message from previous Pipeline Stage of process with myRank-1 */
 MPI_Recv(startMsg,strlen(startMsg),MPI_CHAR,myRank-1,
          tag, MPI_COMM_WORLD,&status);

 /*=DATA TRANSFER 2=====================================*/
 /*= More receive functions, which has same structure, can be added     =*/
 /*= to transfer data                                                   =*/
 MPI_Recv(                 /*=  Modify the following parameters        =*/
          recvBuf,         /*= <- starting address of the buffer       =*/
          recvBufSize,     /*= <- number of elements to receive        =*/
          MPI_CHAR,        /*= <- data type                            =*/
                           /*=================================*/
          myRank-1,tag, MPI_COMM_WORLD,&status);

 /*=IMPLEMENTATION1=========================================*/
 /*=                                                        =*/
 /*= code for this Pipeline Stage should be implemented here =*/
 /*=                                                        =*/
 /*==========================================================*/
}

void main(argc,argv )
    int argc;
    char **argv;
{
 int i = 0;
 MPI_Comm myComm;

 MPI_Init(&argc, &argv);

 MPI_Comm_dup(MPI_COMM_WORLD, &myComm);

 /*find rank of this process*/
 MPI_Comm_rank(myComm, &myRank);

 /*find out rank of last process by using size of rank*/
 MPI_Comm_size(myComm,&mySize);

 strcpy(startMsg,"start");
```

```
 switch(myRank)
  {
  case 0 :
   for(i=0;i<numOfCalElem;i++)
        firstPipelineStage(myComm);
   break;
  case 1 :
   for(i=0;i<numOfCalElem;i++)
        pipelineStage2(myComm);
   break;
  case 2 :
   for(i=0;i<numOfCalElem;i++)
        pipelineStage3(myComm);
   break;
  case 3 :
   for(i=0;i<numOfCalElem;i++)
        pipelineStage4(myComm);
   break;
  case 4 :
   for(i=0;i<numOfCalElem;i++)
        pipelineStage5(myComm);
   break;

   /*=MODIFICATION2=======================*/
   /*= More case statement can be added or removed    =*/
   /*=  on needed basis.                              =*/
   /*=======================================*/

  case 5 :
   for(i=0;i<numOfStage;i++)
        lastPipelineStage(myComm);
   break;
  default:
   break;
  }

 MPI_Finalize();
}
```

3.4.5 Usage Example

The steps to use this frame are as follows:

- Add or remove the pipeline stage functions according to the number of pipeline stages of the program design. Each pipeline stage function has same structure of message passing, so it can be coped and pasted to implement more pipeline stages. But the first and last pipeline stage should not be removed because the first pipeline stage does not receive messages and the last pipeline stage does not send messages.

- Add or remove the case statements, in the block labeled as MODIFICATION2, according to the number of pipeline stages of the program design.
- Implement the blocks labeled as IMPLEMENTATION1. Each block should contain codes for each calculation element.
- Modify the initial address of send buffer, the number of elements in send buffer, and the data type parameters which are in the block labeled as DATA TRANSFER1. The data to be sent is the input for the next calculation element. More send functions, which have same structure, can be added according to the need.
- Modify the initial address of receive buffer, the number of elements in receive buffer, and the data type parameters which are in the block labeled as DATA TRANSFER2. The data to be received is the input for this calculation element. More receive functions, which have same structure, can be added according to the need.

Consider following problem as an example problem to parallelize: There are four SAT scores of four schools. Find out standard deviation for each class. But the main memory of one processor barely contains the scores of one school.

To solve this problem, the pipeline stages can be defined as follows: The first pipeline stage computes the sum and average of SAT scores of each school. The second pipeline stage computes the differences between each individual score and the average. The third pipeline stage computes squares of each difference. Forth pipeline stage computes the sum of the computed squares.

This problem can be easily implemented by following the above steps. The implementation of this is provided in appendix.

3.5 Implementation of Asynchronous-Composition

3.5.1 Intent

This is a solution of the problem how to implement the parallel algorithm which resulted from the Asynchronous-Composition pattern in the algorithm structure design space in MPI (Message Passing Interface)?

3.5.2 Applicability

Problems are represented as a collection of semi-independent entities interacting in an irregular way.

3.5.3 Implementation

Three points that need to be defined to implement the algorithm resulting from Asynchronous-Composition pattern using MPI are tasks/entities, events, and task scheduling.

A task/entity, which generates an event and processes them, can be represented as a process in an implementation of an Algorithm using MPI. An event corresponds to a message sent from the event generating task to the event processing task. All tasks can be executed concurrently.

In this Implementation Example, each case block in a switch statement should contain code for each semi-independent entity (process) which will be executed concurrently.

An event corresponds to a message sent from the event generating task (process) to the event processing task (process) in MPI and C environment. Therefore, event handling is a message handling. For safe message passing among processes, the creation of groups of processes which need to communicate is necessary. Creating communicators for the groups in MPI is also necessary. Because of that, we added groups and communicators creations in this Implementation Example.

This Implementation Example is also an example of event handling in MPI environment. In a situation that an entity (process) receives irregular events (messages sent) from other known entities (processes), we can implement it using defined constant of MPI, MPI_ANY_SOURCE and MPI_ANY_TAG.

Assume that process A, B and C sends messages in an irregular way to process D and the process D handles (processes) events (messages). It is same situation as the car/driver example of the Asynchronous-Composition pattern. To use the MPI_ANY_SOURCE and MPI_ANY_TAG in the receive routine of MPI, we need to create an dedicated group and communicator for these entities (processes) to prevent entity (process) D receives messages from other entities (processes) that is not intended to send message to the entity (process) D.

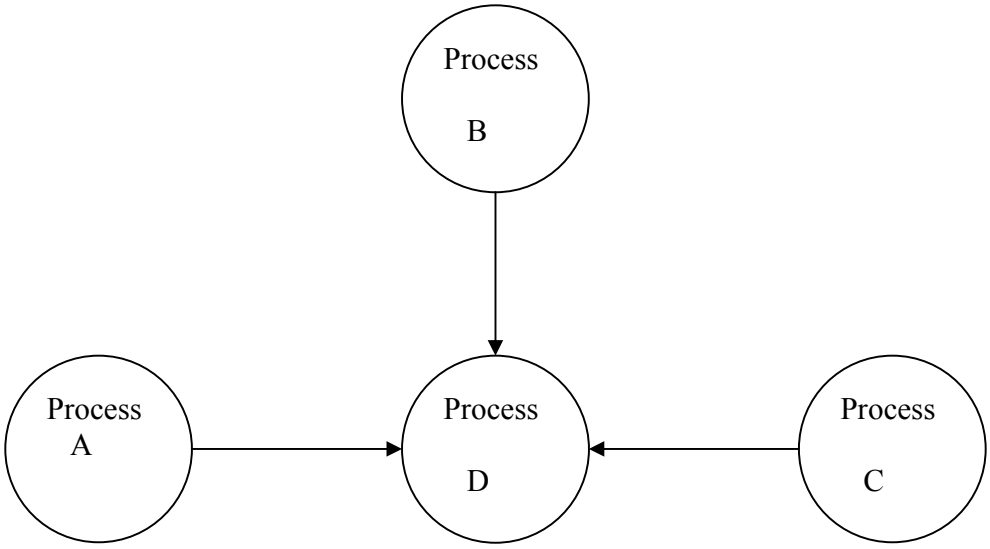

Figure 3-3 Irregular Message Handling

3.5.4 Implementation Example

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>


main(argc, argv)
    int argc;
    char **argv;
{

    /* More group variables can be added or removed on need basis*/
```

```
MPI_Group MPI_GROUP_WORLD,group_A,group_B;

/*More communicator variables can be added or removed on need basis */
MPI_Comm comm_A,comm_B;

/*This varilable will hold the rank for each process in MPI default communicator
   MPI_COMM_WORLD */
int rank_in_world;

/*This variable will hold the rank for each process in group A.
  Variables can be added or removed on need basis.*/
int rank_in_group_A;
int rank_in_group_B;

/* ranks of processes which will be used to create subgroups.
   More array of ranks can be added or removed on need basis.*/
int ranks_a[]={1,2,3,4};
int ranks_b[]={1,2,3,4};

MPI_Init(&argc, &argv);

/*Create a group of processes and communicator for a group .
  More groups and communicator can be created or removed on need basis*/

MPI_Comm_group(MPI_COMM_WORLD,&MPI_GROUP_WORLD);

/*Create group and communicator */
MPI_Group_incl(MPI_GROUP_WORLD,4,ranks_a,&group_A);
MPI_Comm_create(MPI_COMM_WORLD,group_A,&comm_A);

/*Create group and communicator */
MPI_Group_incl(MPI_GROUP_WORLD,4,ranks_b,&group_B);
MPI_Comm_create(MPI_COMM_WORLD,group_B,&comm_B);

MPI_Comm_rank(MPI_COMM_WORLD, &rank_in_world);

switch(rank_in_world)
{

   /*This case 1 contains codes to be executed in process 0 */
   case 0:
      {
         /* events can be generated or processed */

         /*  work */
         break;
```

```
    }

/*This case contains codes to be executed in process 1 */
case 1:
    {
        char sendBuffer[20];
        int isOn=1;
        int i=0;

        while(isOn)
        {
            /* works that need to be done before generating event */
            strcpy(sendBuffer,"event");/* an example */

            /* Generate Event (message passing) */
            MPI_Send(sendBuffer,strlen(sendBuffer),MPI_CHAR,3,1,comm_B);
            printf("sent message");
            i++;

            /* break loop */
            if(i==10) /* this should be changed according to the problem */
            {
                isOn = 0;
            }
        }

        break;

    }

/*This case contains codes to be executed in process 2 */
case 2:
    {
        char sendBuffer[20];
        int isOn=1;
        int i=0;

        while(isOn)
        {
            /* works that need to be done before generating event */
            strcpy(sendBuffer,"event");/* an example */

            /* Generate Event (message) */
            MPI_Send(sendBuffer,strlen(sendBuffer),MPI_CHAR,3,1,comm_B);
            i++;
```

```
            /* break loop */
            if(i==10) /* this should be changed according to the problem */
            {
                isOn = 0;
            }
        }
        break;
    }

/*This case contains codes to be executed in process 3 */
case 3:
    {
        char sendBuffer[20];
        int isOn=1;
        int i=0;

        while(isOn)
        {
            /* works that need to be done before generating event */
            strcpy(sendBuffer,"event"); /* an example */

            /* Generate Event (message) */
            MPI_Send(sendBuffer,strlen(sendBuffer),MPI_CHAR,3,1,comm_B);
            i++;

            /* break loop */
            if(i==10) /* this should be changed according to the problem */
            {
                isOn = 0;
            }
        }
        break;
    }

/*This case contains codes to be executed in process 4 */
case 4:
    {
        char receiveBuffer[20];
        int isOn = 1;
        int messageCount=0;
        MPI_Status status;

        while(isOn)
        {
            MPI_Recv(receiveBuffer,20,MPI_CHAR,MPI_ANY_SOURCE,
                        MPI_ANY_TAG,comm_B,&status);
```

```
                messageCount++;
                if(0==strncmp(receiveBuffer,"event",3))
                {
                    /* work to process the event (message) */
                    printf("\nreceived an event at process 4");

                if(messageCount==30)
                {
                    isOn = 0;
                }
            }
            break;
        }
    /*
        more cases(processes) can be added or removed.
    */
    default: break;
 }
MPI_Finalize();
}
```

3.6 Implementation of Divide and Conquer

3.6.1 Intent

For the completeness of the parallel pattern language, it would be beneficial for a programmer to show an implementation example of a top-level program structure for the Divide-and-Conquer pattern in the Algorithm Design space.

3.6.2 Motivation

The top-level program structure of the divide-and-conquer strategy that is stated in the Divide-and-Conquer pattern is as follows

```
Solution solve(Problem P){
  if (baseCase (P))
    return baseSolve(P);
  else {
    Problem subProblems[N];
    Solution  subSolutions[N];
    subProblems = split(P);
    for (int i=0;i<N;i++)
      subSolutions[i] = solve(subProblems[i]);
    return merge(subSolutions);
  }
```

}

This structure can be mapped onto a design in terms of tasks by defining one task for each invocation of the solve function.

The common MPI implementations have a static process allocation at initialization time of the program. We need a mechanism to invoke the solve function at another processor whenever the solve function calls the split function, and the split function splits the problem into sub-problems so that subproblems can be solved concurrently.

3.6.3 Applicability

This pattern can be used after the program is designed using patterns of the parallel pattern language and the resulting design is a form of the Divide-and-Conquer pattern and if we want to implement the design in an MPI and C environment. This structure can be used as an example of how to implement a top-level program structure in an MPI and C environment or directly adopting this structure as an implementation of the program by adjusting control parameters and adding a computational part of the program.

3.6.4 Implementation

We are trying to implement a top-level program structure of the divide-and-conquer strategy in MPI and C so that parallel program design resulting from the Divide-and-Conquer pattern can be implemented by filling up each function or/and adjusting the structure. The basic idea is using message passing to invoke the solve functions at other processing elements when needed. In a standard MPI and C environment, each processing element has the same copy of the program with other processing elements, and same program executes at each processing element communicating with each other on a needed basis. When the problem splits into subproblems, we need to call the solve

function at other processes so that subproblems can be solved concurrently. To call the solve functions, the split function send messages to other processing elements and the blocking MPI_Receive function receives messages before the first solve function at each process. This message passing can contain data/tasks divided by the split function. In this structure, every solve function calls the merge function to merge subsolutions.

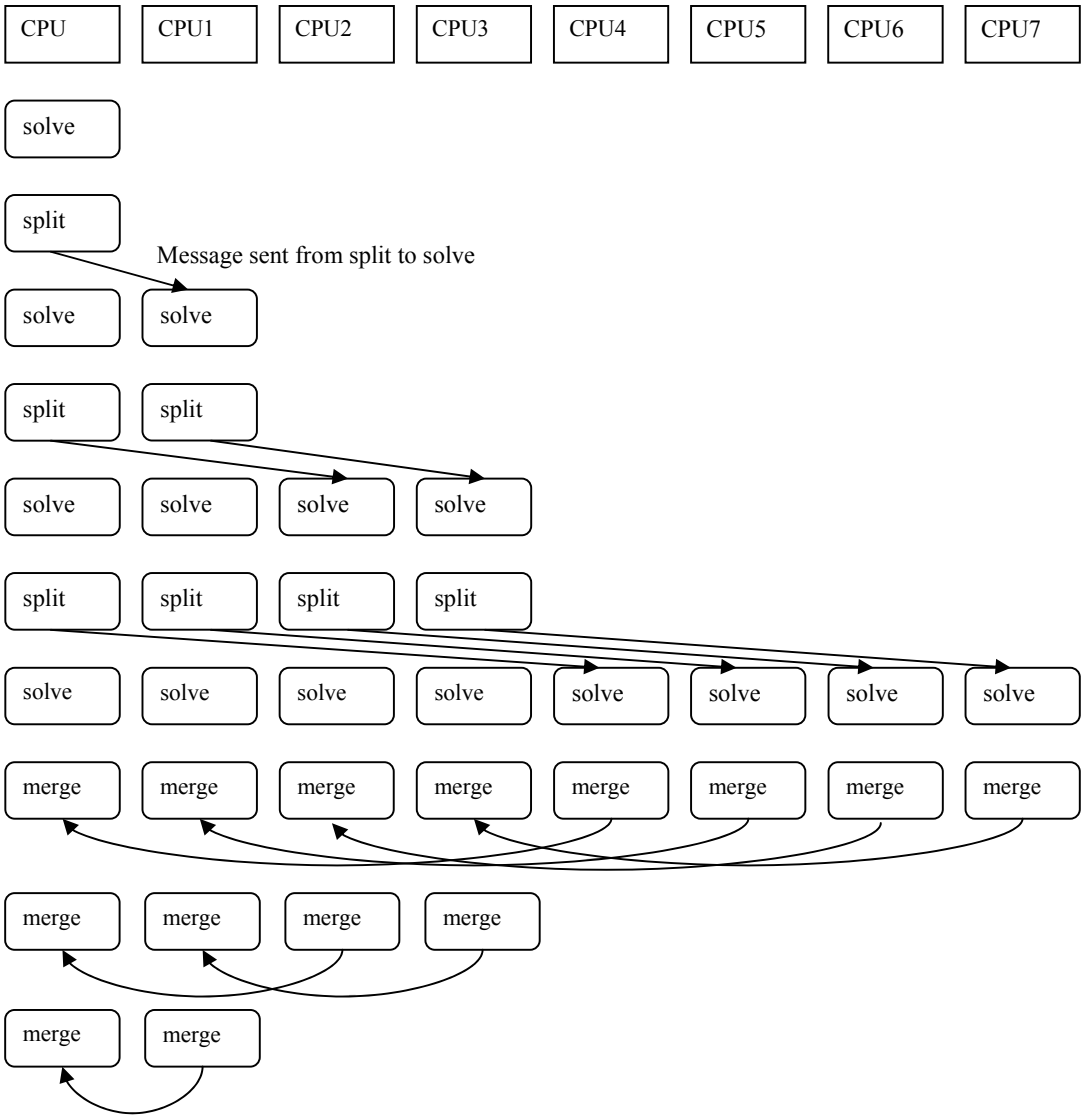

Figure 3-4 Message Passing for Invocation of the Solve and Merge Functions

For simplicity, we split a problem into two subproblems and used problem size to determine whether or not the subproblem is a base case. Figure3.3 shows the sequence of

function calls in each process and message passing to invoke the solve function at remote process for new sub-problem and to merge sub-solutions.

3.6.5 Implementation Example

```
#include <mpi.h>

#include <stdlib.h>
#include <stdio.h>

#define DATA_TYPE  int
int  numOfProc; /*number of available processes*/

int my_rank;
int ctrlMsgSend;
int ctrlMsgRecv;
int* localData;
int dataSizeSend;
int dataSizeRecv;

/********************/
/* Solve a problem        */
/********************/
void solve(int numOfProcLeft)
{
 if(baseCase(numOfProcLeft))
   {
    baseSolve(numOfProcLeft);
    merge(numOfProcLeft);
   }
 else
   {
    split(numOfProcLeft);

    if(numOfProcLeft!=numOfProc)
         {
          merge(numOfProcLeft);
         }
   }
}

/*****************************************************/
/* split a problem into two subproblems                          */
/*****************************************************/
int split(int numOfProcLeft)
```

```
{
 /*=IMPLEMENTATION2 ================================*/
 /*= Code for splitting a problem into two subproblems  =*/
 /*= should be implemented                              =*/
 /*=                                                    =*/
 /*====================================================*/

 ctrlMsgSend = numOfProcLeft/2;
 /* invoke a solve function at the remote process  */
 MPI_Send(&ctrlMsgSend,1,MPI_INT,
          my_rank+numOfProc/numOfProcLeft,
          0,MPI_COMM_WORLD);

 /*=DATA TRANSFER 1 ================================*/
 /*= More of this block can be added on needed basis =*/
 MPI_Send(&dataSizeSend,1,MPI_INT,
          my_rank+numOfProc/numOfProcLeft,
          0, MPI_COMM_WORLD);
 MPI_Send(
          &localData[dataSizeLeft], /*<- modify address of data*/
          dataSizeSend,
          MPI_INT, /*<-modify data type*/
          my_rank+numOfProc/numOfProcLeft,
          0, MPI_COMM_WORLD);
 /*=                                                  =*/
 /*====================================================*/
 /* invoke a solve function at a local process */
 solve(numOfProcLeft/2);
 return 0;
}

/*****************************************/
/* Merge two subsolutions into a solution */
/*****************************************/
void merge(int numOfProcLeft)
{
 if(my_rank >= numOfProc/(numOfProcLeft*2))
   {
    ctrlMsgSend = 1;

    /* Send a subsolution to the process from which this process got the subproblem  */
    MPI_Send(&ctrlMsgSend,1,MPI_INT,
             my_rank -  numOfProc/(numOfProcLeft*2),
             0,MPI_COMM_WORLD);

    /*=DATA TRANSFER 3================================*/
```

```
    /*= More of this block can be added on needed basis              =*/
    MPI_Send(&dataSizeSend,1,MPI_INT,
            my_rank -  numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD);
    MPI_Send(
            localData,          /*<-modify address of data */
            dataSizeSend,
            MPI_INT,            /*<-modify data type */
            my_rank -  numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD);
    /*=                                                              =*/
    /*==============================================*/
  }
 else
  {
    MPI_Status status;
    /* Receive a subsolution from the process which was invoked by this process */
    MPI_Recv(&ctrlMsgRecv,1,MPI_INT,
            my_rank+numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD,&status);

    /*=DATA TRANSFER 3==================================*/
    /*= More of this block can be added on needed basis              =*/
    MPI_Recv(&dataSizeRecv,1,MPI_INT,
            my_rank+numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD,&status);
    MPI_Recv(
            &localData[dataSizeLeft],  /*<- modify address of data */
            dataSizeRecv,
            MPI_INT,  /*<- modify data type*/
            my_rank+numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD,&status);
    /*=                                                              =*/
    /*==============================================*/

    /*=IMPLEMENTATION3==========================*/
    /*= Code for merging two subsolutions into one solution      =*/
    /*=  should be implemented                                   =*/

    /*=   …                                                      =*/
    /*==============================================*/


  }
}
```

```
/*********************************************/
/* Decide whether a problem is a "base case"                */
/*that can be solved without further splitting                */
/*********************************************/
int baseCase(int numOfProcLeft)
{
 if(numOfProcLeft == 1)
   return 1;
 else
   return 0;
}

/*****************************/
/* Solve a base-case problem         */
/*****************************/
int baseSolve(int numOfProcLeft)
{
 /*=IMPLEMENTATION1==================================*/
 /*== Code for solving base case problem should be implemented       ==*/
 /*==  …                                                             ==*/
 /*====================================================*/
 return 0;
}


main(argc, argv)
    int argc;
    char **argv;
{
 int i;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
 MPI_Comm_size(MPI_COMM_WORLD, &numOfProc);

 count = (int *)malloc((maxInt+1)*sizeof(int));
 if(my_rank == 0){
   ctrlMsgSend=numOfProc;  /* number of processes must be power of 2 */
   }


   ctrlMsgRecv = numOfProc;
   solve(ctrlMsgRecv);
 }
 else{
   /* Every process waits a message before calling solve function   */
```

```
  MPI_Recv(&ctrlMsgRecv,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
          MPI_COMM_WORLD,&status);

  /*=DATA TRANSFER 2==================================*/
  /*= More of this block can be added on needed basis  =*/
  MPI_Recv(&dataSizeRecv,1,MPI_INT,MPI_ANY_SOURCE,
          MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  MPI_Recv(
          localData,          /*<- modify address of data */
          dataSizeRecv,
          MPI_INT,            /*<- modify data type*/
          MPI_ANY_SOURCE,MPI_ANY_TAG,
          MPI_COMM_WORLD,&status);
  /*=                                                  =*/
  /*==================================================*/

  dataSizeLeft = dataSizeRecv;
  solve(ctrlMsgRecv);
 }
 if(my_rank==0){

  for(i=0;i<numOfIntToSort;i++)
    {
       printf(" %d",localData[i]);
    }
 }

 MPI_Finalize();
}
```

3.6.6 Usage Example

To use the previous framework, the number of available processes must be power of 2.

The steps to use this Implementation Example Code are as follows:

- Implement the block labeled as IMPLEMENTATION1. This block should contain codes for “baseSolve” algorithm.
- Implement the block labeled as IMPLEMENTAION2. This block should contain codes for “split” algorithm.
- Implement the block labeled as IMPLEMENTATION3. This block should contain codes for “merge” algorithm.
- Find out and modify the initial address of send buffer, the number of elements in send buffer, and the data type parameters which are in the block labeled as DATA

TRANSFER1. The data to be sent is the input for the solve function of remote process. More of this block can be added on needed basis.

- Find out and modify the initial address of receive buffer, the number of elements to receive, and the data type parameters which are in the block labeled as DATA TRANSFER2. The data to receive is the input for the solve function of local process. More of this block can be added on needed basis.

- Find out and modify the initial address of receive buffer, the number of elements to receive, and the data type parameters which are in the block labeled as DATA TRANSFER3. The data to receive and send are the input for the merge function of local process. More of this block can be added on needed basis.

One point to note about using the Implementation Example is that the "baseCase" of the Implementation Example is when there are no more available processes.

Merge sort algorithm is well known problem which uses divide and conquer strategy. If we consider the merge sort over N integers as the problem to parallelize, it can be easily implemented using the Implementation Example. To use the Implementation Example, the merge sort algorithm should be modified as follows: The "baseSolve" is a counting sort over the local data (an array of integers). The "split" algorihtm divides received data (an array of integers) into two contiguous subarrays. The "merge" algorithm merges two sorted array into one sorted array. The "BaseCase" of this problem is when there are no more available processes. The problem can be implemented by following the implementation steps above. The implementation of this problem is provided in appendix.

CHAPTER 4
KERNEL IS OF NAS BENCHMARK

The Numerical Aerodynamic Simulation (NAS) Program, which is based at NASA Ames Research Center, developed a set of benchmarks for the performance evaluation of highly parallel supercomputers. These benchmarks, NPB 1 (NAS Parallel Benchmarks 1), consist of five parallel kernels and three simulated application benchmarks. The principle distinguishing feature of these benchmarks is their "pencil and paper" specification. All details of these benchmarks are specified only algorithmically. The Kernel IS (Parallel Sort over small integers) is one of the kernel benchmark set.

4.1 Brief Statement of Problem

Sorting N integer keys in parallel is the problem. The number of keys is 2^{23} for class A and 2^{25} for class B. The range of keys is $[0, B_{\max})$ where $B_{\max}$ is 2^{19} for class A and 2^{21} for class B. The keys must be generated by the key Generation algorithm of the NAS benchmark set. The initial distribution of the key must follow the specification of memory mapping of keys. Even though the problem is sorting, what will be timed is the time needed to rank every key, and the permutation of keys will not be timed.

4.2 Key Generation and Memory Mapping

The Keys must be generated using the pseudorandom number generator of the NAS Parallel Benchmark. The numbers generated by this pseudorandom number generator will have range (0, 1) and have very nearly uniform distribution of the unit interval.[11-12] The keys will be generated using this number in the following way. Let r_f

be a random fraction uniformly distributed in the range (0,1), and let K_i be the i^{th} key. The value of K_i is determined as $K_i \leftarrow \lfloor B_{\max}(r_{4i+0} + r_{4i+1} + r_{4i+2} + r_{4i+3})/4 \rfloor$ for $i = 0,1,...,N-1$. K_i must be an integer and $\lfloor \cdot \rfloor$ indicates truncation. $B_{\max}$ is 2^{19} for class A and is 2^{21} for class B. For the distribution of keys among memory, all keys initially must be stored in the main memory units and each memory unit must have same amount of keys in a contiguous address space. If the keys cannot be evenly divisible, the last memory unit can have a different amount of keys, but it must follow specification of NAS Benchmark. See Appendix A for details.

4.3 Procedure and Timing

The implementation of Kernel IS must follow this procedure. The partial verification of this procedure tests the ranks of five unique keys where each key has unique rank. The full verification rearranges the sequence of keys using the computed rank of each key and tests that the keys are sorted.

1. In a scalar sequential manner and using the key generation algorithm described above, generate the sequence of N keys.
2. Using the appropriate memory mapping described above, load the N keys into the memory system.
3. Begin timing.
4. Do, for $i = 1$ to $I_{\max}$

(a) Modify the sequence of keys by making the following two changes:

$$K_i \leftarrow i$$

$$K_{i+I_{\max}} \leftarrow (B_{\max} - i)$$

(b) Compute the rank of each key.

(c) Perform the partial verification test described above.

5. End timing.

6. Perform full verification test described previously.

Table 4-1: Parameter Values to Be Used for Benchmark

Parameter	Class A	Class B
N	2^{23}	2^{25}
B_{max}	2^{19}	2^{21}
seed	314159265	314159265
I_{max}	10	10

CHAPTER 5
PARALLEL PATTERNS USED TO IMPLEMENT KERNEL IS

In this chapter, we will explain how we used the parallel pattern language to design the parallel program and implement it in an MPI and C environment.

5.1 Finding Concurrency

5.1.1 Getting Started

As advised by this pattern, we have to understand the problem we are trying to solve. According to the specification of Kernel IS of NAS Parallel Benchmark, the range of the keys to be sorted is $[0,2^{19})$ for class A and a range of $[0,2^{21})$ for class B. Because of this range, bucket sort, radix sort, and counting sort can sort the keys in O(n) time.[13] These sequential sorting algorithms are good candidate algorithms to parallelize because of their speeds. Because of following reasons, Counting Sort is selected as a target algorithm to parallelize. According to the specification of the benchmark, what will be timed is the time needed to find out the rank of each keys, and the actual permutation of keys occurs after the timing. This means that it is important to minimize the time needed to find out rank of each key. One interesting characteristic of the three candidate sorting algorithms is following. The Radix Sort and Bucket Sort make permutations of keys to find out rank of each key, in other words, the keys should be sorted to know the rank of every key. But, the Counting Sort does not need to be sorted to find out the rank of every key. This means that the Counting Sort takes less time in ranking every key than the others. Because of this reason, the Counting Sort is selected. Another reason is the simplicity of the Counting Sort algorithm.

The basic idea of the Counting Sort algorithm is to determine, for each element x, the number of elements less than x. This information can be used to place element x directly into its position in the output array and the number of elements less than x is $rank(x)-1$.

Counting Sort Algorithm is as follows:

Counting-Sort(A, B ,k) // A is an input array. B is an output array. C is an intermediate //array. k is an biggest key in the input array.

for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** $length[A]$
  do $C[A[j]] \leftarrow C[A[j]]+1$
// $C[i]$ now contains the number of elements equal to i
for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i]+C[i-1]$
// $C[j]$ now contains the number of elements less than or equal to i
for $j \leftarrow length[A]$ **down to** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]]-1$

Algorithm 4-1 Counting Sort

5.1.2 Decomposition Strategy

The decision to be made in this pattern is whether or not to follow data decomposition or task decomposition pattern. The key data structure of the Counting Sort algorithm is an array of integer keys, and the most compute-intensive part of the algorithm is counting each occurrence of same keys in the integer keys array and summing up the counts to determine the rank of each key. The task-based decomposition is a good starting point if it is natural to think about the problem in terms of a collection of independent (or nearly independent) tasks. We can think of the problem in terms of tasks that count each occurrence of the same keys in key array. So we followed the task decomposition pattern.

5.1.3 Task Decomposition

The Task Decomposition pattern suggests finding tasks in functions, loop, and updates on data. We tried to find tasks in one of the four loops in the Counting Sort algorithm. We found, using this pattern, that there are large enough independent iterations in the second *for loop* of the counting sort algorithm. These iterations can be divided into many enough relatively independent tasks which find out rank of each key. We can think that each task concurrently counts each occurrence of the same keys on *array A*. The *array A* is shared among tasks right now and it is not divided yet.

Because of dependency among tasks, this pattern recommends using Dependency Analysis pattern.

5.1.4 Dependency Analysis

Using this pattern, *data sharing dependencies* are found on *array A*. Each task is sharing the *array A* because the *array A* is accessed by each task to read a number in each cell of the array. If one *task* has read and counted the integer number of a cell, that number should not be read and counted again.

Array C is shared, too, because each task access this array to accumulate the count for each number, and one cell must not be accessed or updated concurrently by two or more tasks at the same time. We follow Data-Sharing Pattern since data sharing dependencies are found.

5.1.5 Data Sharing Pattern

Array A can be considered *read only* data among the three categories of shared data according to this pattern, because it is not updated. *Array A* can be partitioned into subsets, each of which is accessed by only one of the tasks. Therefore *Array A* falls into *effectively-local* category.

The shared data, *array C,* is one of the special cases of read/write data. It is *an accumulate* case because this array is used to accumulate the counts of each occurrence of the same keys. For this case, this pattern suggests that each task has a separate copy so that the accumulations occur in these local copies. The local copies are then accumulated into a single global copy as a final step at the end of the computation.

What has been done until this point is a parallelization of *second loop* of Counting Sort algorithm. The fourth for loop can be parallelized as follows: Each task has its own ranks of keys in its own data set so the keys can be sorted in each process. To sort these locally sorted keys, redistribute keys after finding out the range of keys for each task so that each task has approximately same amount of keys and the ranges of keys in ascending order among tasks.

Then the tasks of sorting redistributed keys have a dependency on locally sorted keys. But this dependency is read only and effectively local so the locally sorted keys can be redistributed among processes without complication, according to the ranges, because those are already sorted. Then each task can merge the sorted keys in its own range with out dependency using the global count.

Therefore, the final design of parallel sorting (the implementation of Kernel IS) follows. First, each task counts each occurrence of the same keys on its own subset data, accumulates the results on its own output array, and then reduces them into one output array. Second, each task redistributes the keys into processes, according to the range of keys of each process, and merges the redistributed keys.

5.1.6 Design Evaluation

Our target platform is Ethernet-connected Unix workstations. It is a distributed memory system. This Design Evaluation pattern says that it usually makes it easier to

keep all the processing elements busy by having many more tasks than processing elements. But using more than one UE per PE is not recommended in this pattern when the target system does not provide efficient support for multiple UEs per PEs (Processing Element. Generic term used to reference a hardware element in a parallel computer that executes a stream of instructions), and the Design can not make good use of multiple UEs per PE, which is our case because of reduction of output array (if we have more local output array than the number of PEs, then the time needed to reduce to one output will take much longer). So our program adjusts the number of tasks into the same number of workstations.

This pattern questions simplicity, flexibility, and efficiency of the design. Our design is simple because each generated task will find out rank of each key on its own subset data (keys) and then reduce them into global ranks. The next steps of sorting locally and redistribution of keys and merging the redistributed local key arrays are also simple because we already know the global ranks. It is also flexible and efficient because the number of tasks is adjustable and the computational load is evenly balanced.

5.2 Algorithm Structure Design Space

5.2.1 Choose Structure

The Algorithm-Structure decision tree of this pattern is used to arrive at an appropriate Algorithm-Structure for our problem.

The major organizing principle of our design is organization by tasks because we used the loop-splitting technique of the Task-Decomposition pattern, so we arrived at the Organized-by-Tasks branch point. At that branch point, we take a partitioning branch because our ranking tasks can be gathered into a set linear in any number of dimensions.

There are dependencies between tasks and it is an associative accumulation into shared data structures. Therefore we arrived at the Separable-Dependencies pattern.

5.2.2 Separable Dependencies

A set of tasks that will be executed concurrently in each processing unit corresponds to iterations of a second *for loop* and third *for loop* of Algorithm 4.1. Each task will be independent of each other because each task will use its own data. This pattern recommends balancing the load at each PE. The size of each data is equal because of data distribution specification of Kernel IS so the load is balanced at each PE. The specification of keys distribution of Kernel IS is also satisfied. The fact that all the tasks are independent leads to the Embarrassingly Parallel Pattern.

5.2.3 Embarrassingly Parallel

Each task of finding all the ranks of distinct key array can be represented as a process. Because each task will have almost same amount of keys, each task will have same amount of computation. So the static scheduling of the tasks will be effective as advised by this pattern. For the correctness considerations of this pattern, each tasks solve the subproblem independently, solve each subproblem exactly once, correctly save subsolutions, and correctly combine subsolutions.

5.3 Using Implementation Example

The design of the parallel integer sort problem satisfies the condition of simplest form of parallel program as follows: All the tasks are independent. All the tasks must be completed. Each task executes same algorithm on a distinct section of data. Therefore the resulted design is a simplest form of Embarrassingly Parallel Pattern. The Implementation example is provided in Chapter 3. Using the Implementation Example of the Embarrassingly Parallel pattern, the Kernel IS of NAS Parallel Benchmark set can be

implemented. The basic idea of Implementation Example of Embarrassingly Parallel pattern is that of scattering data to each process, compute subsolutions, and then combine subsolutions into a solution for the problem. If we apply the Implementation Example two times, one time for finding rank of each key and one for sorting the keys in local processes, the Kernel IS can be impelemented.

Another method of implementation is to use Implementation Example of Divide Conquer pattern. But this Implementation Example is not chosen because it is hard to measure the elapsed time for ranking all the keys.

5.4 Algorithm for Parallel Implementation of Kernel IS

The following algorithm illustrates the parallel design for implementation of Kernel IS (parallel sort over small integer) that has been obtained through the parallel design patterns.

1. Generate the sequence of N keys using key generation algorithm of NAS Parallel Benchmarks.

2. Divide the keys by the number of PEs and distribute to each memory of PEs.

3. Begin timing.

4. Do, for $i = 1$ to $I_{\max}$

   (a) Modify the sequence of keys by making the following two changes:

   $$K_i \leftarrow i$$

   $$K_{i+I_{\max}} \leftarrow (B_{\max} - i)$$

   (b) Each task (process) finds out the rank of each key in its own data set (keys) using ranking algorithm of counting sort.

   (c) Reduce the arrays of ranks in its own local data into an array of ranks in global data.

   (d) Do partial verification.

5. End timing.

6. Perform permutation of keys according to the ranks.

    (a) Each task sorts local keys according to the ranks among its local keys.

    (b) Compute the ranges of keys each process will have so that each process will have nearly same amount of keys and the ranges of keys are in ascending order

    (c) Redistribute keys among processes according to the range of each process.

    (d) Each task (process) merges its redistributed key arrays.

7. Perform full verification.

CHAPTER 6
PERFORMANCE RESULTS AND DISCUSSIONS

6.1 Performance Expectation

An ideal execution time for a parallel program can be expressed as *T/N* where *T* is the total time taken with one processing element and *N* is the number of processing Elements used. Our implementation of Kernel IS of NAS Parallel Benchmarks will not have an ideal execution time because of overhead that comes from several sources. A source of overhead comes from the computations needed to reduce local arrays of ranks into one array of global ranks. The more local arrays of ranks and processing elements we use, the more computation time will be needed. The gap between the ideal execution time and the actual execution time will be increased. Another source of overhead will be the communication because MPI uses message transfer, which typically involves both overhead due to kernel calls and latency due to the time it takes the message to travel over the network.

6.2 Performance Results

The Kernel IS implementation was executed on top of Ethernet-connected workstations and LAM 6.3.2, which is an implementation of MPI.[14] These workstations are Sun Blade 100 with 500-MHz UltraSPARC-IIe cpu, 256-KB L2 External Cache, 256-MB DIMM memory, Ethernet/Fast Ethernet, and twisted pair standard (10BASE-T and 100BASE-T) self-sensing network. Figure 6.1 shows the performance results for class A and B. The rows show the total number of processing elements (workstations) used to

execute Kernel IS implementation. The columns show the execution time of Kernel IS implementation and an ideal execution time in milliseconds for each class A and B. The performance result of NPB2.2 (NAS Parallel Benchmark 2.2. These are MPI-based source-code implementations written and distributed by NAS. They are intended to be run with little or no tuning, and they approximate the performance a typical user can expect to obtain for a portable parallel program. They supplement, rather than replace, NPB 1. The NAS solicits performance results from all sources) for class A and B are shown for comparison purpose.[15]

Table 6-1 Performance Results for Class A and B

Number of Processing Elements	Actual Execution Time for class A	Ideal Execution Time for Class A	Execution Time of NPB2.2 for Class A	Actual Execution Time for Class B	Ideal Execution Time for Class B	Execution Time of NPB2.2 for ClassB
1	30940	30940	20830	147220	147220	N/A
2	16500	15470	25720	76100	73610	1446270
3	12160	10313	N/A	55640	49073	N/A
4	10600	7735	16870	46500	36805	103110
5	9200	6188	N/A	41150	29444	N/A
6	8150	5156	N/A	36550	24536	N/A
7	7600	4420	N/A	33350	21031	N/A
8	7080	3867	9720	30200	18402	42760
9	7470	3437	N/A	31790	16357	N/A
10	7040	3094	N/A	30860	14722	N/A
11	6820	2812	N/A	28740	13383	N/A
12	6750	2578	N/A	27900	12268	N/A
13	6690	2380	N/A	27580	11324	N/A
14	6320	2210	N/A	26630	10515	N/A
15	6000	2062	N/A	25560	9814	N/A
16	5780	1933	16680	25230	9201	62080

The execution times for Class B when the number of processing elements is 1 and 2 are not shown in Figure 6.2 because execution time for Class B is too big to show in Figure 6.2. The reason for that long execution time is that NPB2.2 consumes much more

memory than the physical memory, which leads to many I/O between the hard disk and main memory of workstations.

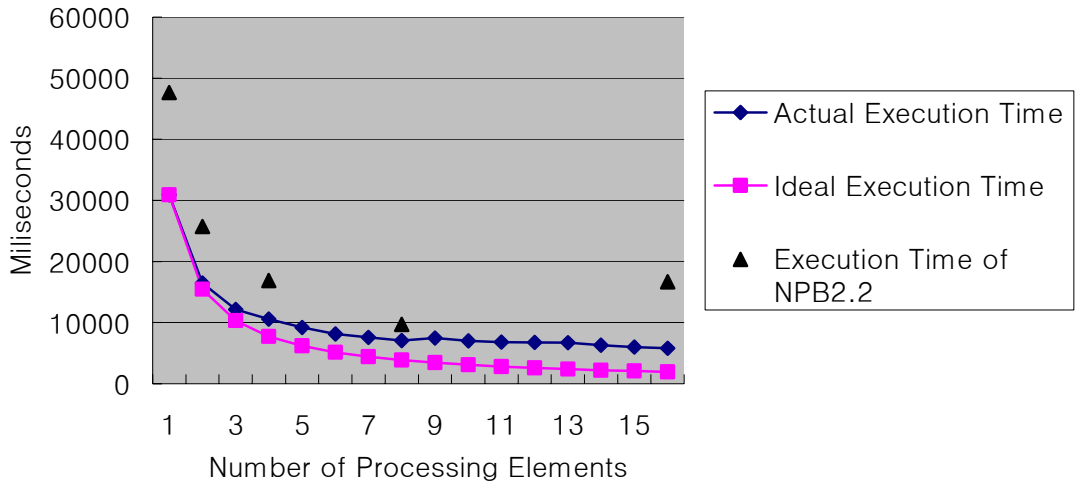

Figure 6-1 Execution Time Comparison for Class A

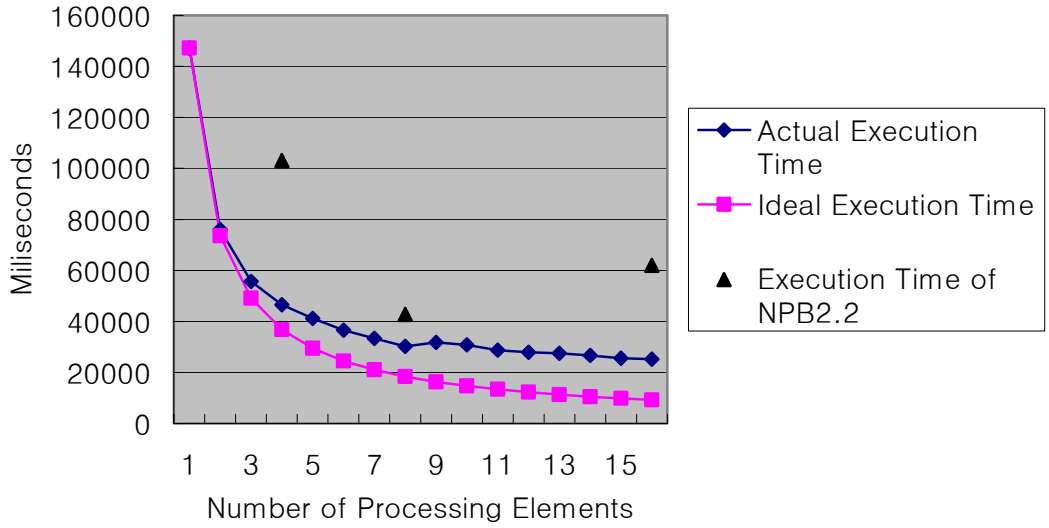

Figure 6-2 Execution Time Comparison for Class B

6.3 Discussions

As seen from the previous graphs, the following conclusions can be made about the performance of the parallel implementation of Kernel IS of NAS Parallel Benchmarks. The most increase in performance is achieved by using one or two additional processing

elements, that is when the total number of processing elements is 2 or 3 respectively and there are smaller improvements by using more processing elements. The more processing elements we use for computation, the more overhead was created and the gap between the ideal execution time and the actual execution time increased, as we expected. The overall performance is acceptable because it has better performance compared to the performance of NPB2.2

CHAPTER 7
RELATED WORK AND CONCLUSIONS AND FUTURE WORK

7.1 Related work

7.1.1 Aids for Parallel Programming

Considerable research has been done to ease the tasks of designing and implementing efficient parallel program. The related work can be categorized into a program skeleton, a program framework, and design patterns.

An algorithmic skeleton was introduced by M. Cole as a part of his proposed parallel programming systems (language) for parallel machines.[16] He presented four independent algorithmic skeletons which are "fixed degree divide and conquer," "iterative combination," "clustering," and "task queues." Each of skeletons describes the structure of a particular style of algorithm in terms of abstraction. These skeletons capture very high-level patterns and can be used as an overall program structure. The user of this proposed parallel programming system must choose one of these skeletons to describe a solution to a problem as an instance of the appropriate skeleton. Because of this restriction, these skeletons can not be applied to every problem. These skeletons are similar to the patterns in the algorithm structure design space of parallel pattern language in a sense that both are an overall program structure and provide algorithmic frameworks. But these skeletons provide less guidance for an inexperienced parallel programmer about how to arrive at one of these skeletons in comparison with parallel pattern language.

Program Frameworks, which can address overall program structure but are more detailed and domain specific, are widely used in many areas of computing.[17] In a parallel

computing area, Parallel Object Oriented Methods and Applications (POOMA) and Parallel Object-oriented Environment and Toolkit (POET) are examples of frameworks from Los Alamos and Sandia, respectively.[18-19]

The POOMA is an object-oriented framework for a data-parallel programming of scientific applications. It is a library of C++ classes designed to represent common abstractions in these applications. Application programmers can use and derive from these classes to express the fundamental scientific content and/or numerical methods of their problem. The objects are layered to represent a data-parallel programming interface at the highest abstraction layer whereas lower, implementation layers encapsulate distribution and communication of data among processors.

The POET is a framework for encapsulating a set of user-written components. The POET framework is an object model, implemented in C++. Each user-written component must follow the POET template interface. The POET provides services, such as starting the parallel application, running components in a specified order, and distributing data among processors.

In recent years, Launay and Pazat developed a framework for parallel programming using Java.[20] This framework provides a parallel programming model embedded in Java by a framework and intended to separate computations from control and synchronizations between parallel tasks. Doug Lea provided design principles to build concurrent applications using the Java parallel programming facilities.[21]

Design patterns that can address many levels of design problem have been widely used with object oriented sequential programming. Recent work of Douglas C. Schmidt, et al

addresses issues associated with concurrency and networking but mostly at a fairly low level.[22]

Design pattern, frameworks, and skeleton share the same intent of easing parallel programming by providing libraries, classes, or a design pattern. In comparison with parallel pattern language, which provides a systemic design patterns and implementation patterns for parallel program design and implementation, frameworks and skeletons might have more efficient implementations or better performance in their specialized applicable problem domain. But parallel pattern language could be more helpful for inexperienced parallel programmer in designing and solving more general application problems because of its systematic structure in exploiting concurrency and providing frameworks and libraries down to implementation level.

7.1.2 Parallel Sorting

Parallel sorting is one of the most widely studied problems because of its importance in wide variety of practical applications. Various parallel sorting algorithms have also been proposed in the literature. Guy E. Blelloch, et al. analyzed and evaluated many of the parallel sorting algorithms proposed in the literature to implement as fast a general purpose sorting algorithm as possible on the Connection Machine Supercomputer model CM-2.[23] After the evaluation and analysis, the researchers selected the three most promising alternatives for implementation: bitonic sort that is a parallel merge sort, parallel version of counting-based radix sort, and a theoretically efficient randomized algorithm, sample sort. According to their experiments, sample sort was the fastest on large data sets.

Andrea C. Dusseau, et al. analyzed these three parallel sorting algorithms and column sort using a LogP model, which characterizes the performance of modern parallel

machines with a small set of parameters: the communication latency, overhead, bandwidth, and the number of processes.[24] They also compared performances of Split-C implementation of the four sorting algorithms on message passing, distributed memory, massively parallel machine, CM-5. In their comparison, radix and sample sort was faster than others on a large data set.

To understand the performance of parallel sorting on hardware cache-coherent shared address space (CC-SAS) multiprocessors, H. Shan, et al. investigated the performance of two parallel sorting algorithms (radix, sample sort) under three major programming models (a load-store CC-SAS, message passing, and the segmented SHMEM model) on a 64 processor SGI Origin2000 (A scalable, hardware-supported, cache-coherent, non-uniform memory access machine).[25] In their investigation, the researchers found that sample sort is generally better than radix sort up to 64k integers per processor, and radix sort is better after that point. The best combination of algorithm and programming models are sample sort under the CC-SAS for smaller data sets and radix sort under SHMEM for larger data sets, according to their investigation.

Communication is fundamentally required in a parallel sorting problem as well as computation. Because of this characteristic and the importance of sorting in applications, parallel sorting problem has been selected as one of kernel benchmarks for the performance evaluation of various parallel computing environments.[26] The Kernel IS of NAS Parallel Benchmark set has been implemented and reported its performance on various parallel supercomputers by its vendors.[27-28]

7.2 Conclusions

This thesis shows how the parallel design patterns were used to develop and implement a parallel algorithm for Kernel IS (Parallel sort over large integers) of NAS

Parallel Benchmarks as a case study. And it presented reusable frameworks and examples for implementations of patterns in the algorithm structure design space of parallel pattern language.

Chapter 6 showed the performance results for Kernel IS. As the result shows, the parallel design patterns help developing a parallel program with relative ease, and it is also helpful in reasonably improving the performance.

7.3 Future Work

Parallel pattern language is an ongoing project. Mapping design patterns with various parallel computing environments and developing frameworks for object oriented programming systems can be considered as future research. Testing the resulted implementation of Kernel IS using parallel pattern language on various supercomputers, comparing this algorithm as a full sorting, not just ranking, and more case studies of the parallel application program using parallel pattern language can also be included in the future work.

APPENDIX A
KERNEL IS OF THE NAS PARALLEL BENCHMAKRK

A.1 Brief Statement of Problem

Sort N keys in parallel. The keys are generated by the sequential key generation algorithm given below and initially must be uniformly distributed in memory. The initial distribution of the keys can have a great impact on the performance of this benchmark, and the required distribution is discussed in detail below.

A.2 Definitions

A sequence of keys, $\{K_i \mid i = 0,1,...,N-1\}$, will be said to be sorted if it is arranged in non-descending order, i.e., $K_i \le K_{i+1} \le K_{i+2} \ldots$. The rank of a particular key in a sequence is the index value I that the key would have if the sequence of keys were sorted. *Ranking*, then, is the process of arriving at a rank for all the keys in a sequence. *Sorting* is the process of permuting the keys in a sequence to produce a sorted sequence. If an initially unsorted sequence, $K_0, K_1, ..., K_{N-1}$ has ranks $r(0), r(1), ...r(N-1)$, the sequence becomes sorted when it is rearranged in the order $K_{r(0)}, K_{r(1)}, ..., K_{r(N-1)}$. Sorting is said to be stable if equal keys retain their original relative order. In other words, a sort is stable only if $r(i) < r(j)$ whenever $K_{r(i)} = K_{r(j)}$ and $i < j$. Stable sorting is not required for this benchmark.

A.3 Memory Mapping

The benchmark requires ranking an unsorted sequence of N keys. The initial sequence of keys will be generated in an unambiguous sequential manner as described below. This

sequence must be mapped into the memory of parallel processor in one of the following ways depending on the type of memory system. In all cases, one key will map to one word of memory. Word size must be no less than 32 bits. Once the keys are loaded onto the memory system, they are not to be removed or modified except as required by the procedure described in the Procedure subsection.

A.4 Shared Global Memory

All N keys initially must be stored in a contiguous address space. If A_i is used to denote the address of the i^{th} word of memory, then the address space must be $[A_i, A_{i+N-1}]$. The sequence of keys, $K_0, K_1, ..., K_{N-1}$, initially must map to this address space as

$$A_{i+j} \leftarrow MEM(K_j) \text{ for } j = 0,1,..., N-1 \qquad \text{(A.1)}$$

where $MEM(K_j)$ refers to the address of K_j.

A.5 Distributed Memory

In a distributed memory system with p distinct memory units, each memory unit initially must store N_p keys in a contiguous address space, where

$$N_p = N / p \qquad \text{(A.2)}$$

If A_i is used to denote the address of the i^{th} word in a memory unit, and if P_j is used to denote the j^{th} memory unit, then $P_j \bigcap A_i$ will denote the address of the i^{th} word in the j^{th} memory unit. Some initial addressing (or "ordering") of memory units must be assumed and adhered to throughout the benchmark. Note that the addressing of the memory units is left completely arbitrary. If N is not evenly divisible by p, then

memory units $\{P_j \mid j = 0,1,..., p-2\}$ will store N_p keys, and memory unit P_{p-1} will store N_{pp} keys, where now

$$N_p = \lfloor N/p + 0.5 \rfloor$$

$$N_{pp} = N - (p-1)N_p$$

In some cases (in particular if p is large) this mapping may result in a poor initial load balance with $N_{pp} >> N_p$. In such cases it may be desirable to use p' memory units to store the keys, where $p' < p$. This is allowed, but the storage of the keys still must follow either equation 2.2 or equation 2.3 with p' replacing p. In the following we will assume N is evenly divisible by p. The address space in an individual memory unit must be $\lfloor A_i, A_{i+N_p-1} \rfloor$. If memory units are individually hierarchical, then N_p keys must be stored in a contiguous address space belonging to a single memory hierarchy and A_i then denotes the address of the i^{th} word in that hierarchy. The keys cannot be distributed among different memory hierarchies until after timing begins. The sequence of keys, $K_0, K_1, ..., K_{N-1}$, initially must map to this distributed memory as

$$P_k \cap A_{i+j} \leftarrow MEM(K_{kN_p+j}) \text{ for } j = 0,1,..., N_p - 1 \text{ and } \quad k = 0,1,..., p-1$$

where $MEM(K_{kN_p+j})$ refers to the address of K_{kN_p+j}. If N is not evenly divisible by p, then the mapping given above must be modified for the case where $k = p-1$ as

$$P_{p-1} \cap A_{i+j} \leftarrow MEM(K_{(p-1)N_p+j}) \quad \text{for} \quad j = 0,1,..., N_{pp} - 1. \qquad (A.3)$$

A.6 Hierarchical Memory

All N keys initially must be stored in an address space belonging to a single memory hierarchy which will here be referred to as the *main memory*. Note that any memory in

the hierarchy which can store all N keys may be used for the initial storage of the keys, and the use of the term "main memory" in the description of this benchmark should not be confused with the more general definition of this term in section 2.2.1. The keys cannot be distributed among different memory hierarchies until after timing begins. The mapping of the keys to the main memory must follow one of either the shared global memory or the distributed memory mappings described above.

The benchmark requires computing the rank of each key in the sequence. The mappings described above define the initial ordering of the keys. For shared global and hierarchical memory systems, the same mapping must be applied to determine the correct ranking. For the case of a distributed memory system, it is permissible for the mapping of keys to memory at the end of the ranking to differ from the initial mapping only in the following manner: The number of keys mapped to a memory units at the end of the ranking may differ from the initial value, N_p. It is expected, in a distributed memory machine, that good load balancing of the problem will require changing the initial mapping of the keys and for this reason a different mapping may be used at the end of the ranking. If N_{p_k} is the number of keys in memory unit P_k at the end of ranking, then the mapping which must be used to determine the correct ranking is given by $P_k \bigcap A_{i+j} \leftarrow MEM(r(kN_{p_k} + j))$ for $j = 0,1,...,N_{p_k} - 1$ and $k = 0,1,...,p-1$ where $r(kN_{p_k} + j)$ refers to the rank of key $K_{kN_{pk}+j}$. Note, however, this does not imply that the keys, once loaded into memory, may be moved. Copies of the keys may be maid and moved, but the original sequence must remain intact such that each time the ranking process is repeated (Step 4 of Procedure) the original sequence of keys exist (except for the two modification of Step 4a) and the same algorithm for ranking is applied.

Specifically, knowledge obtainable from the communications pattern carried out in the first ranking cannot be used to speed up subsequent rankings and each iteration of Step 4 should be completely independent of the previous iteration.

A.7 Key Generation Algorithm

The algorithm for generating the keys makes use of the pseudorandom number generator described in section 2.2. The keys will be in the range $[0, B_{\max})$. Let r_f be a random fraction uniformly distributed in the range $[0,1]$, and let K_i be the i^{th} key. The value of K_i is determined as

$$K_i \leftarrow \lfloor B_{\max}(r_{4i+0} + r_{4i+1} + r_{4i+2} + r_{4i+3})/4 \rfloor \quad \text{for } i = 0,1,...,N-1. \qquad \text{(A.4)}$$

Note that K_i must be an integer and $\lfloor \cdot \rfloor$ indicates truncation. Four consecutive pseudorandom numbers from the pseudorandom number generator must be used for generating each key. All operations before the truncation must be performed in 64-bit double precision. The random number generator must be initialized with $s = 314159265$ as a starting seed.

A.8 Partial Verification Test

Partial verification is conducted for each ranking performed. Partial verification consists of comparing a particular subset of ranks with the reference values. The subset of ranks and the reference values are given in table 2.1.

Note that the subset of ranks is selected to be invariant to the ranking algorithm (recall that stability is not required in the benchmark). This is accomplished by selecting for verification only the ranks of unique keys. If a key is unique in the sequence (i.e., there is no other equal key), then it will have a unique rank despite an unstable ranking algorithm. The memory mapping described in the Memory Mapping subsection must be applied.

Table A-1 : Values to be used for partial verification

Rank (full)	Full scale	Rank (sample)	Sample code
$r(2112377)$	$104+i$	$r(48427)$	$0+i$
$r(662041)$	$17523+i$	$r(17148)$	$18+i$
$r(5336171)$	$123928+i$	$r(23627)$	$346+i$
$r(3642833)$	$8288932-i$	$r(62548)$	$64917-i$
$r(4250760)$	$8388264-i$	$r(4431)$	$65463-i$

A.9 Full Verification Test

Full verification is conducted after the last ranking is performed. Full verification requires the following:

1. Rearrange the sequence of keys, $\{K_i \mid i = 0,1,...,N-1\}$, in the order $\{K_j \mid j = r(0),r(1),...,r(N-1)\}$, where $r(0),r(1),...,r(N-1)$ is the last computed sequence of ranks.

2. For every K_i from $i = 0...N-2$ test that $K_i \le K_{i+1}$

If the result of this test is true, then the keys are in sorted order. The memory mapping described in the Memory Mapping subsection must be applied.

A.10 Procedure

1. In a scalar sequential manner and using key generation algorithm described above, generate the sequence of N keys.

2. Using the appropriate memory mapping described above, load the N keys into the memory system.

3. Begin timing.

4. Do, for $i = 1$ to $I_{\max}$

(a) Modify the sequence of keys by making the following two changes:

$$K_i \leftarrow i$$

$$K_{i+I_{\max}} \leftarrow (B_{\max} - i)$$

(b) Compute the rank of each key.
(c) Perform the partial verification test described above.

5. End timing.

6. Perform full verification test described above.

Table A-2: Parameter values to be used for benchmark

Parameter	Class A	Class B
N	2^{23}	2^{25}
B_{max}	2^{19}	2^{21}
seed	314159265	314159265
I_{max}	10	10

A.11 Specifications

The specifications given in table A.2 shall be used in the benchmark. Two sects of values are given, one for Class A and one for Class B.

APPENDIX B
A PSEUDORANDOM NUMBER GENERATOR FOR THE PARALLEL NAS KERNELS

B.1 Pseudorandom Number Generator

Suppose that n uniform pseudorandom numbers are to be generated. Set $a = 5^{13}$ and let $x_0 = s$ be a specified initial "seed," i.e., an integer in the range $0 < s < 2^{46}$. Generate the integers x_k for $1 \le k \le n$ using the linear congruential recursion

$$x_{k+1} = a x_k \,(\mathrm{mod}\, 2^{46})$$

and return $r_k = 2^{-46} x_k$ as the result. Thus $0 < r_k < 1$, and the r_k are very nearly uniformly distributed distribution on the unit interval. See [2], beginning on page 9 for further discussion of this type of pseudorandom number generator.

Note that any particular value x_k of the sequence can be computed directly from the initial seed s by using the binary algorithm for exponentiation, taking remainders modulo 2^{46} after each multiplication. To be specific, let m be the smallest integer such that $2^m > k$, set $b = s$ and $t = a$. Then repeat the following for i from 1 to m:

$j \leftarrow k/2$

$b \leftarrow bt \,(\mathrm{mod}\, 2^{46})$ if $2j \neq k$

$t \leftarrow t^2 \,(\mathrm{mod}\, 2^{46})$

$k \leftarrow j$

The final value of b is $x_k = a^k s \pmod{2^{46}}$. See [2] for further discussion of the binary algorithm for exponentiation.

The operation of multiplying two large integers modulo 2^{46} can be implemented using 64 bit floating point arithmetic by splitting the arguments into two words with 23 bits each. To be specific, suppose one wishes to compute $c = ab \pmod{2^{46}}$. Then perform the following steps, where int denotes the greatest integer:

$$a_1 \leftarrow \text{int}(2^{-23} a)$$

$$a_2 \leftarrow a - 2^{23} a_1$$

$$b_1 \leftarrow \text{int}(2^{-23} b)$$

$$b_2 \leftarrow b - 2^{23} b_1$$

$$t_1 \leftarrow a_1 b_2 + a_2 b_1$$

$$t_2 \leftarrow \text{int}(2^{-23} t_1)$$

$$t_3 \leftarrow t_1 - 2^{23} t_2$$

$$t_4 \leftarrow 2^{23} t_3 + a_2 b_2$$

$$t_5 \leftarrow \text{int}(2^{-46} t_4)$$

$$c \leftarrow t_4 - 2^{46} t_5$$

An implementation of the complete pseudorandom number generator algorithm using this scheme produces the same sequence of results on any system that satisfies the following requirements:

The input multiplier a and initial seed s, as well as the constants 2^{23}, 2^{-23}, 2^{46} and 2^{-46}, can be represented exactly as 64 bit floating point constants.

The truncation of a nonnegative 64 bit floating point value less than 2^{24} is exact.

The addition, subtraction and multiplication of 64 bit floating point values, where the arguments and results are nonnegative whole numbers less than 2^{47}, produce exact results.

The multiplication of a 64 bit floating point value, which is a nonnegative whole number less than 2^{47}, by the 64 bit floating point value 2^{-m}, $0 \le m \le 46$, reduces and exact result.

These requirements are met by virtually all scientific computers in use today. Any system based on the IEEE-754 floating point arithmetic standard [1] easily meets these requirements using double precision. However, it should be noted that obtaining an exact power of two constant on some systems requires a loop rather than merely an assignment statement with.

B.2 Other Features

The period of this pseudorandom number generator is $2^{44} = 1.76 * 10^{13}$, and it passes all reasonable statistical tests.

This calculation can be vectorized on vector computers by generating results in batches of size equal to the hardware vector length.

By using the scheme described above for computing x_k directly, the starting seed of a particular segment of the sequence can be quickly and independently determined. Thus numerous separate segments can be generated on separate processors of a multiprocessor system.

Once the IEEE-754 floating point arithmetic standard gains universal acceptance among scientific computers, the radix 2^{46} can be safely increased to 2^{52}, although the

scheme described above for multiplying two such numbers must be correspondingly changed. This will increase the period of the pseudorandom sequence by a factor of 64 to approximately.

APPENDIX C

SOURCE CODE OF THE KERNEL IS IMPLEMENTATION

```
/*
  This program is an implementation of Kernel IS(sorting over small integers
  of NAS Parallel benchmark set.
*/
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <mpi.h>
#define  MAX_KEY  524288    /*maximum value of key */
#define  NUM_KEYS  8388608  /*number of keys to be sorted */
#define  TEST_ARRAY_SIZE 5

int partialVerifyVals[TEST_ARRAY_SIZE];
int testIndexArray[TEST_ARRAY_SIZE] =
                                        {2112377,662041,5336171,3642833,4250760};
int testRankArray[TEST_ARRAY_SIZE] ={104,17523,123928,8288932,8388264};

int passedVerification = 0;

/* the number of keys each processing element has */
int numOfLocalKeys;

/* rank of each key */
int* countGlobal;

/* rank of each key in local key subset */
int count[MAX_KEY+1];

/* a subset of keys that each process has*/
int* localKeyArray;
```

```
/* Sorted subset keys before redistribution */
int* tempResultArray;

/* Final sored keys */
int* finalResultArray;

/* rank of  each process */
int myRank;
/* total number of processes */
int mySize;

/************************************/
/*  RANKING                         */
/************************************/
void ranking(){
  long i;
  long j;
  for(i=0;i<MAX_KEY+1;i++){
    count[i]=0;
  }
  for(j=0;j<numOfLocalKeys;j++) {
    count[localKeyArray[j]]=count[localKeyArray[j]]+1;
  }
  for(i=1;i<MAX_KEY+1;i++){
    count[i]=count[i]+count[i-1];
  }
}

/*******************************
  Full Verification Test
   1. Rearrange the sequence of keys element
      in nondescending order.
      a. sort the keys in each memorys.
      b. redistribute the keys among memorys accoding to thier range
      c. sort the redistributed keys
```

```
  2. Test whether the keys are in sored order.
*******************************/
void full_verify(){
  int i, j;
  int temp;
  /* The number of keys that each process will have at last */
  int lastNumOfKeysLocal;

  /* max value in each process before key redistributtion */
  int maxValueInPE[mySize];

  /* number of keys after key redistribution */
  int newNumOfKeysLocal[mySize];

  /* number of keys that will be send to each process */
  int numOfKeysRedis[mySize];

  /* number of keys that each process will receive from other processes and itself */
  int numOfKeysToRecv[mySize];

  /* Entry i specifies the displacement relative to sendbuffer from which to take
     the outgoing data destined for process j*/
  int senddispls[mySize];

  /* Entry j specifies the displacement relative to receivebuffer at which to place the
     incoming data from process i*/
  int recvdispls[mySize];

  int arrayTemp[mySize];
  int *lastArray;
  int accum[mySize];

  MPI_Bcast(countGlobal,MAX_KEY+1,MPI_INT,0,MPI_COMM_WORLD);

  /* Sorting keys locally in process */
  for(j=numOfLocalKeys-1;j>=0;j--){
```

```
  tempResultArray[count[localKeyArray[j]]-1]=localKeyArray[j];
  count[localKeyArray[j]]=count[localKeyArray[j]]-1;
}

/* KEY REDISTRIBUTION */
/* finding out maximum number and number of keys that each process
   will have after key redistribution */
if(myRank == 0){
  i=0;
  int tempN = 0;
  for(j=1;j<mySize;j++){
    for(i;i<MAX_KEY+1;i++){
        if(countGlobal[i] >= numOfLocalKeys+tempN){
          maxValueInPE[j-1]=i;/* maxValueInPE */
          newNumOfKeysLocal[j-1] = countGlobal[i-1]-tempN;
          tempN = tempN + newNumOfKeysLocal[j-1];
          break;
        }
    }
  }
  maxValueInPE[j-1]= MAX_KEY;
  newNumOfKeysLocal[j-1]=NUM_KEYS - tempN;
  tempN=0;
  for(i=0;i<mySize;i++)
    {
        tempN=newNumOfKeysLocal[i]+tempN;
    }
}

/* Broadcast max value in each process and number of keys after
   key redistribution */
MPI_Bcast(maxValueInPE,mySize,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(newNumOfKeysLocal,mySize,MPI_INT,0,MPI_COMM_WORLD);


finalResultArray = (int *) malloc(newNumOfKeysLocal[myRank]*sizeof(int));
```

```
lastNumOfKeysLocal= newNumOfKeysLocal[myRank];
lastArray = (int *) malloc(lastNumOfKeysLocal*sizeof(int));

/* compute the number of keys to send to each process and
   displacement(starting point of send buffer for each process) */
i=0;
temp=0;
for(i;i<mySize;i++ ) {
  senddispls[i] = temp;
  numOfKeysRedis[i]=count[maxValueInPE[i]]-temp;
  temp = temp + numOfKeysRedis[i];
  printf(" %d",count[maxValueInPE[i]]);
}

for(i=0;i<mySize;i++) {
  numOfKeysToRecv[i]=0;
}
/* compute the number of keys each process will receive and the displacement
   (starting point of receive buffer for each process */
MPI_Alltoall(numOfKeysRedis,1,MPI_INT,numOfKeysToRecv,1,
                  MPI_INT,MPI_COMM_WORLD);
i=0;
temp = 0;
for(i=0;i<mySize;i++) {
  recvdispls[i]=temp;
  temp = temp + numOfKeysToRecv[i];
}


/* Redistribution of keys */
MPI_Alltoallv(tempResultArray,numOfKeysRedis,senddispls,MPI_INT,
                    finalResultArray,numOfKeysToRecv,recvdispls,
                     MPI_INT,MPI_COMM_WORLD);

/* Sorting redistributed keys */
temp =0;
```

```
  for(i=0;i<mySize;i++){
    accum[i]=temp;
    temp = temp+newNumOfKeysLocal[i];
  }
  for(j=lastNumOfKeysLocal-1;j>=0;j--){
    lastArray[countGlobal[finalResultArray[j]]-accum[myRank]]=finalResultArray[j];
    countGlobal[finalResultArray[j]]=countGlobal[finalResultArray[j]]-1;
  }
  /* Test wheterher the keys are in sorted order*/
  j = 0;
  for( i=1; i<lastNumOfKeysLocal; i++ )
    if( lastArray[i-1] > lastArray[i] ){
      j++;
    }
  if( j != 0 ){
    printf( "Full_verify: number of keys out of sort: %d\n", j );
  }
  else
    printf("Full verification was successful\n");
}

/************************************/
/*  Partial Verification Test                */
/************************************/
partialVerify(int iteration,int* countGlobal){
  int i=0;
  int k=0;
  int j=0;
  passedVerification=0;

  for( i=0; i<TEST_ARRAY_SIZE; i++ ){
    k = partialVerifyVals[i];
    /* test vals were put here */
    if( 0 <= k  &&  k <= NUM_KEYS-1 ){
      if( i < 3 ){
          if( countGlobal[k-1] != testRankArray[i]+iteration-1 ){
```

```
            printf( "Failed partial verification: "
                    "iteration %d, test key %d ,\n",
                    iteration, i );
            printf("%d %d\n",countGlobal[k-1],testRankArray[i]+iteration-1);
          }
          else
            passedVerification++;
      }
      else{
          if( countGlobal[k-1] != testRankArray[i]-iteration+1){
            printf( "Failed partial verification: "
                    "iteration %d, test key %d\n",
                    iteration, i );
            printf("%d %d\n",countGlobal[k-1],testRankArray[i]-iteration+1);
          }
          else
            passedVerification++;
      }
      if(passedVerification == 5){
          printf("Partial verification was successful\n");
      }
    }
  }
}

/*****************************************************/
/*  Genterate uniformly distributed random number        */
/*****************************************************/
double randlc(X, A)
    double *X;
    double *A;
{
  static int       KS=0;
  static double  R23, R46, T23, T46;
  double          T1, T2, T3, T4;
  double          A1;
```

```
double          A2;
double          X1;
double          X2;
double          Z;
int          i, j;

if (KS == 0){
  R23 = 1.0;
  R46 = 1.0;
  T23 = 1.0;
  T46 = 1.0;

  for (i=1; i<=23; i++){
    R23 = 0.50 * R23;
    T23 = 2.0 * T23;
  }
  for (i=1; i<=46; i++){
    R46 = 0.50 * R46;
    T46 = 2.0 * T46;
  }
  KS = 1;
}

T1 = R23 * *A;
j  = T1;
A1 = j;
A2 = *A - T23 * A1;

T1 = R23 * *X;
j  = T1;
X1 = j;
X2 = *X - T23 * X1;
T1 = A1 * X2 + A2 * X1;

j  = R23 * T1;
T2 = j;
```

```
  Z = T1 - T23 * T2;
  T3 = T23 * Z + A2 * X2;
  j  = R46 * T3;
  T4 = j;
  *X = T3 - T46 * T4;
  return(R46 * *X);
}

/*****************************************/
/*  Generate a sequence of integer keys  */
/*****************************************/
void create_seq( double seed, double a,int* key_array ){
  double x;
  int    i, j, k;
  k = MAX_KEY/4;
  for (i=0; i<NUM_KEYS; i++){
    x = randlc(&seed, &a);
    x += randlc(&seed, &a);
    x += randlc(&seed, &a);
    x += randlc(&seed, &a);
    key_array[i] = k*x;
  }
}

/*************************************/
/*  MAIN                             */
/*************************************/
void main(int argc,char* argv[]){
  int i = 0;
  int j = 0;

  int* key_array;
  int* resultArray; /* sorted integer */

  clock_t start;
  clock_t end;
```

```
double timecounter;
double maxtime;

int numOfKeyLastNode;
MPI_Status status;
MPI_Request request;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
MPI_Comm_size(MPI_COMM_WORLD,&mySize);

/* compute the number of keys that each process will have
   according to the memory mapping of Kernel IS */
numOfLocalKeys=(int)((double)(NUM_KEYS/mySize) + 0.5);
numOfKeyLastNode= NUM_KEYS-(numOfLocalKeys*(mySize-1));
if(myRank == mySize -1){
  numOfLocalKeys = numOfKeyLastNode;
}
countGlobal= (int *) malloc((MAX_KEY+1)*sizeof(int));
localKeyArray =(int *) malloc(numOfLocalKeys*sizeof(int));
tempResultArray = (int *) malloc(numOfLocalKeys*sizeof(int));

/* Process 0 (root process) will distribute keys to each process */
if(myRank==0){
  int i = 0;
  int j = 0;
  key_array =  (int *) malloc(NUM_KEYS*sizeof(int));
  resultArray= (int *) malloc(NUM_KEYS*sizeof(int));

  /****Generating Keys to be sorted **********************************/
  printf("Generating Random Numbers...\n");
  create_seq( 314159265.00,                /* Random number gen seed */
                  1220703125.00,key_array );
  printf("Created Sequence\n");
}
```

```
/*** SCATTER  ****************************************/
MPI_Scatter(key_array,numOfLocalKeys,MPI_INT,localKeyArray,
                numOfLocalKeys,MPI_INT,0,MPI_COMM_WORLD);
/*****************************************************/


/* Synchronize all processes before timing*/
MPI_Barrier(MPI_COMM_WORLD);
start=clock();

for(i=1;i<11;i++){
 /* Modify the sequence of keys */
 if(myRank==0){
  localKeyArray[i] = i;
  localKeyArray[i+10] = MAX_KEY - i;
  for( j=0; j<TEST_ARRAY_SIZE; j++ )
       partialVerifyVals[j] = key_array[testIndexArray[j]];
 }

 /* Compute the rank of each key */
 ranking();

 /*** REDUCE ***********************************************/
 MPI_Reduce(count,countGlobal,MAX_KEY+1,MPI_INT,MPI_SUM,
                    0,MPI_COMM_WORLD);
 /*** REDUCE ***********************************************/



 /* Partial Verification */
 if(myRank==0)
  partialVerify(i,countGlobal);
}

MPI_Barrier(MPI_COMM_WORLD);
```

```
/* End Timing*/
end = clock();
timecounter = ((double) (end - start)) * 1000/CLOCKS_PER_SEC;

MPI_Reduce( &timecounter,
            &maxtime,
            1,
            MPI_DOUBLE,
            MPI_MAX,
            0,
            MPI_COMM_WORLD );

if(myRank==0){
  printf("start:%ld\n",start);
  printf("end:%ld\n",end);
  printf("Sorting process took %f milliseconds to execute\n",maxtime);
}

/* Full verification */
full_verify() ;

MPI_Finalize();
}
```

APPENDIX D

SOURCE CODE OF PIPELINE EXAMPLE PROBLEM

```
#include <mpi.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

int numOfCalElem = 20;
char startMsg[7];
int myRank;
int mySize;
int tag = 1;

char* sendBuf, recvBuf;
double sendBufSize, recvBufSize;

int numOfStudent = 1000;

MPI_Status status;

/* Only four pipeline stages are implemented in this Example Implementation Code.
   More elements can be added or removed, according to the design
   of the parallel program */

/***********************/
/*First Pipeline Stage     */
/***********************/
void firstPipelineStage(MPI_Comm myComm)
{
 /*=IMPLEMENTATION1==============================*/
 /*=                                              =*/
 /*= code for this Pipeline Stage should be implemented here  =*/
```

```
  double scores[numOfStudent];
int i=0;
double sum =0;
double average = 0;
for(i=0;i<numOfStudent;i++)
  {
   scores[i] = ((int)rand())%100;
  }
for(i=0;i<numOfStudent;i++)
  {
   sum = sum + scores[i];
  }
average = sum/numOfStudent;
/*=                                                                    =*/
/*=====================================================*/

/* send message to the next Pipeline Stage of process with myRank+1 */
MPI_Ssend(startMsg,strlen(startMsg),MPI_CHAR
          ,myRank+1,tag,MPI_COMM_WORLD);
/*=DATA TRANSFER 1=============================*/
/*= More send function, which has same structure, can be added      =*/
/*= to transfer data                                                =*/
MPI_Ssend(          /*=  Modify the following parameters            =*/
          scores,       /* <-starting address of buffer             =*/
          numOfStudent,  /* <-the number of elements in buffer       =*/
          MPI_DOUBLE,  /* <-the data type                           =*/
 /*=========================================================*/
          myRank+1,tag,MPI_COMM_WORLD);

/*=DATA TRANSFER 1=============================*/
/*= More send function, which has same structure, can be added     =*/
/*= to transfer data                                               =*/
MPI_Ssend(              /*=   Modify the following parameters      =*/
          &average,     /*= <- starting address of buffer          =*/
          1,            /* <- the number of elements in buffer     =*/
          MPI_DOUBLE,/* <- the data type                           =*/
```

```
   /*=========================================*/
            myRank+1,tag,MPI_COMM_WORLD);
}

/*******************/
/* Pipeline Stage 2       */
/*******************/
void pipelineStage2(MPI_Comm myComm)
{
 double scores[numOfStudent];
 double average=0;
 /* Receive message from the previous Pipeline Stage of process with myRank-1 */
 MPI_Recv(startMsg,strlen(startMsg),MPI_CHAR,myRank-1,
          tag, MPI_COMM_WORLD,&status);

 /*=DATA TRANSFER 2==================================*/
 /*= More receive functions, which has same structure, can be added  =*/
 /*= to transfer data                                                =*/
 MPI_Recv(                  /*=  Modify the following parameters     =*/
          scores,           /*= <- starting address of the buffer    =*/
          numOfStudent,     /*= <- number of elements to receive     =*/
          MPI_DOUBLE,       /*= <- data type                         =*/
                            /*=============================*/
          myRank-1,tag, MPI_COMM_WORLD,&status);

 /*=DATA TRANSFER 2==================================*/
 /*= More receive functions, which has same structure, can be added  =*/
 /*= to transfer data                                                =*/
 MPI_Recv(                  /*=  Modify the following parameters     =*/
          &average,         /*= <- starting address of the buffer    =*/
          1,                /*= <- number of elements to receive     =*/
          MPI_DOUBLE,       /*= <- data type                         =*/
                            /*=========================*/
          myRank-1,tag, MPI_COMM_WORLD,&status);

 /*=IMPLEMENTATION1==================================*/
```

```
 /*=                                                              =*/
 /*= code for this Pipeline Stage should be implemented here      =*/
 int i=0;
 for(i=0;i<numOfStudent;i++)
  {
   scores[i]=scores[i]-average;
  }
 /*=                                                              =*/
 /*==============================================================*/

 /* send message to the next Pipeline Stage of process with myRank+1 */
 MPI_Ssend(startMsg,strlen(startMsg),MPI_CHAR,myRank+1,tag,
          MPI_COMM_WORLD);

 /*=DATA TRANSFER 1==============================*/
 /*= More send function, which has same structure, can be added =*/
 /*= to transfer data                                           =*/
 MPI_Ssend(                    /*=  Modify the following parameters  =*/
          scores,              /* <- starting address of buffer     =*/
          numOfStudent,        /* <- the number of elements in buffer =*/
          MPI_DOUBLE,          /* <- the data type                  =*/
          /*=======================================*/
          myRank+1,tag,MPI_COMM_WORLD);
}

/********************/
/* Pipeline Stage 3      */
/********************/
void pipelineStage3(MPI_Comm myComm)
{
 double scores[numOfStudent];
 /* Receive message from the previous Pipeline Stage of process with myRank-1 */
 MPI_Recv(startMsg,strlen(startMsg),MPI_CHAR,myRank-1,
          tag, MPI_COMM_WORLD,&status);

 /*=DATA TRANSFER 2==============================*/
```

```
/*= More receive functions, which has same structure, can be added    =*/
/*= to transfer data                                                  =*/
MPI_Recv(                  /*=  Modify the following parameters       =*/
        scores,            /*= <- starting address of the buffer      =*/
        numOfStudent,      /*= <- number of elements to receive       =*/
        MPI_DOUBLE,        /*= <- data type                           =*/
        /*=======================================*/
        myRank-1,tag, MPI_COMM_WORLD,&status);

/*=IMPLEMENTATION1==============================*/
/*=                                                                   =*/
/*= code for this Pipeline Stage should be implemented here           =*/
int i=0;
for(i=0;i<numOfStudent;i++)
  {
   scores[i]=scores[i]*scores[i];
  }
/*=                                                                   =*/
/*=====================================================*/

/* send message to the next Pipeline Stage of process with myRank+1 */
MPI_Ssend(startMsg,strlen(startMsg),MPI_CHAR,myRank+1,tag,
        MPI_COMM_WORLD);

/*=DATA TRANSFER 1==============================*/
/*= More send function, which has same structure, can be added        =*/
/*= to transfer data                                                  =*/
MPI_Ssend(             /*=   Modify the following parameters         =*/
        scores,        /* <- starting address of buffer              =*/
        numOfStudent,  /* <- the number of elements in buffer        =*/
        MPI_DOUBLE, /* <- the data type                               =*/
        /*=======================================*/
        myRank+1,tag,MPI_COMM_WORLD);
}

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

```
/* Last Pipeline Stage     */
/************************/

void lastPipelineStage(MPI_Comm myComm)
{
 double scores[numOfStudent];
 double stdDeviation;
 /* Receive message from previous Pipeline Stage of process with myRank-1 */
 MPI_Recv(startMsg,strlen(startMsg),MPI_CHAR,myRank-1,
          tag, MPI_COMM_WORLD,&status);

 /*=DATA TRANSFER 2====================================*/
 /*= More receive functions, which has same structure, can be added  =*/
 /*= to transfer data                                                =*/
 MPI_Recv(                /*=  Modify the following parameters       =*/
          scores,         /*= <- starting address of the buffer      =*/
          numOfStudent,   /*= <- number of elements to receive       =*/
          MPI_DOUBLE,     /*= <- data type                           =*/
                          /*=================================*/
          myRank-1,tag, MPI_COMM_WORLD,&status);

 /*=IMPLEMENTATION1====================================*/
 /*=                                                   =*/
 /*= code for this Pipeline Stage should be implemented here  =*/
 int i=0;
 double sum = 0;
 for(i=0;i<numOfStudent;i++)
   {
    sum = sum + scores[i];
    }
 stdDeviation = sqrt((double)(sum/numOfStudent));

 printf(" %f",stdDeviation);
 /*=                                                   =*/
 /*=====================================================*/
```

```
}

void main(argc,argv )
    int argc;
    char **argv;
{
 int i = 0;
 MPI_Comm myComm;

 MPI_Init(&argc, &argv);

 MPI_Comm_dup(MPI_COMM_WORLD, &myComm);

 /*find rank of this process*/
 MPI_Comm_rank(myComm, &myRank);

 /*find out rank of last process by using size of rank*/
 MPI_Comm_size(myComm,&mySize);

 strcpy(startMsg,"start");

 switch(myRank)
  {
  case 0 :
   for(i=0;i<numOfCalElem;i++)
        firstPipelineStage(myComm);
   break;
  case 1 :
   for(i=0;i<numOfCalElem;i++)
        pipelineStage2(myComm);
   break;
  case 2 :
   for(i=0;i<numOfCalElem;i++)
        pipelineStage3(myComm);
   break;
  case 3 :
```

```
    for(i=0;i<numOfCalElem;i++)
        lastPipelineStage(myComm);
    break;
  default:
    break;
  }

 MPI_Finalize()
}
```

APPENDIX E

SOURCE CODE OF DIVIDE AND CONQUER EXAMPLE PROBLEM

```
#include <mpi.h>

#include <stdlib.h>
#include <stdio.h>

#define DATA_TYPE  int
int  numOfProc; /*number of available processes*/

int my_rank;
int ctrlMsgSend;
int ctrlMsgRecv;
int* localData;
int mergedInt[200];
int dataSizeSend;
int dataSizeRecv;

int maxInt = 200;
int dataSizeLeft;
int numOfIntToSort =  200;
int* integer

int* count;
int* temp;

/********************/
/* Solve a problem      */
/********************/
void solve(int numOfProcLeft)
{
```

```
  if(baseCase(numOfProcLeft))
    {
     baseSolve(numOfProcLeft);
     merge(numOfProcLeft);
    }
  else
    {
     split(numOfProcLeft);
     if(numOfProcLeft!=numOfProc)
          {
           merge(numOfProcLeft);
          }
    }
}

/*****************************************************/
/* split a problem into two subproblems              */
/*****************************************************/
int split(int numOfProcLeft)
{
  /*=IMPLEMENTATION2 ============================*/
  /*= Code for splitting a problem into two subproblems  =*/
  /*= should be implemented                              =*/
  dataSizeSend = dataSizeRecv/2;
  dataSizeLeft = dataSizeRecv-dataSizeSend;
  dataSizeRecv = dataSizeLeft;
  ctrlMsgSend = numOfProcLeft/2;
  /*=                                                    =*/
  /*==============================================*/

  /* invoke a solve function at the remote process  */
  MPI_Send(&ctrlMsgSend,1,MPI_INT,
           my_rank+numOfProc/numOfProcLeft,
           0,MPI_COMM_WORLD);

  /*=DATA TRANSFER 1 ============================*/
```

```
 /*= More of this block can be added on needed basis                =*/
 MPI_Send(&dataSizeSend,1,MPI_INT,
          my_rank+numOfProc/numOfProcLeft,

          0, MPI_COMM_WORLD);
 MPI_Send(
          &localData[dataSizeLeft], /*<- modify address of data*/
          dataSizeSend,
          MPI_INT, /*<-modify data type*/
          my_rank+numOfProc/numOfProcLeft,
          0, MPI_COMM_WORLD);
 /*=                                                                =*/
 /*=========================================*/
 /* invoke a solve function at a local process*/
 solve(numOfProcLeft/2);
 return 0;
}

/*****************************************/
/* Merge two subsolutions into a solution */
/*****************************************/
void merge(int numOfProcLeft)
{
 if(my_rank >= numOfProc/(numOfProcLeft*2))
   {
    ctrlMsgSend = 1;
    dataSizeSend = dataSizeLeft;

    /* Send a subsolution to the process
          from which this process got the subproblem*/
    MPI_Send(&ctrlMsgSend,1,MPI_INT,
             my_rank -  numOfProc/(numOfProcLeft*2),
             0,MPI_COMM_WORLD);

    /*=DATA TRANSFER 3=======================*/
    /*= More of this block can be added on needed basis  =*/
```

```
   MPI_Send(&dataSizeSend,1,MPI_INT,
            my_rank - numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD);
   MPI_Send(

            localData,          /*<-modify address of data */
            dataSizeSend,
            MPI_INT,            /*<-modify data type */
            my_rank -  numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD);
   /*=                                                    =*/
   /*=====================================*/

  }
else
  {
   MPI_Status status;

   int i1;
   int i2;
   int iResult;
   int t;


   /* Receive a subsolution from the process which was invoked by this process */
   MPI_Recv(&ctrlMsgRecv,1,MPI_INT,
            my_rank+numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD,&status);

   /*=DATA TRANSFER 3============================*/
   /*= More of this block can be added on needed basis          =*/
   MPI_Recv(&dataSizeRecv,1,MPI_INT,
            my_rank+numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD,&status);
   MPI_Recv(
            &localData[dataSizeLeft],  /*<- modify address of data */
```

```
            dataSizeRecv,
            MPI_INT,  /*<- modify data type*/
            my_rank+numOfProc/(numOfProcLeft*2),
            0,MPI_COMM_WORLD,&status);
/*=                                                                    =*/
/*======================================================*/

/*=IMPLEMENTATION3============================*/
/*= Code for merging two subsolutions into one solution              =*/
/*=  should be implemented                                           =*/
/*   mergedInt = (int *) malloc((dataSizeLeft+dataSizeRecv)*sizeof(int)); */
for(i1=0,iResult=0,i2=dataSizeLeft;
      i1<=(dataSizeLeft-1)&& i2<=(dataSizeLeft+dataSizeRecv);
      iResult++){
    if(localData[i1]<=localData[i2])
      {
       mergedInt[iResult] = localData[i1];
       i1++;
      }
    else
      {
       mergedInt[iResult]=localData[i2];
       i2++;
      }
}
if(i1>dataSizeLeft-1){
    for(t=i2;t<=dataSizeLeft+dataSizeRecv-1;t++){
      mergedInt[iResult+t-i2] = localData[t];
    }

}
else{
    for(t=i1;t<dataSizeLeft-1;t++)
      mergedInt[iResult+t-i1]= localData[t];
}
```

```
    for(t=0;t<dataSizeRecv+dataSizeLeft;t++){
        localData[t] = mergedInt[t];
    }
    dataSizeLeft = dataSizeRecv+dataSizeLeft;
    /*=    …                                                    =*/
    /*=========================================*/
   }
}

/***********************************************/
/* Decide whether a problem is a "base case"   */
/*that can be solved without further splitting */
/***********************************************/
int baseCase(int numOfProcLeft)
{
 if(numOfProcLeft == 1)
   return 1;
 else
   return 0;
}

/*****************************/
/* Solve a base-case problem  */
/*****************************/
int baseSolve(int numOfProcLeft)
{
 /*=IMPLEMENTATION1============================*/
 /*== Code for solving base case problem should be implemented  ==*/
 int i = 0;
 int j = 0;

 temp=(int*)malloc(dataSizeLeft*sizeof(int));
 for(i=0;i<maxInt+1;i++){
  count[i]=0;
 }
 for(j=0;j<dataSizeLeft;j++){
```

```
    count[localData[j]]= count[localData[j]]+1;
  }
  for(i=1;i<maxInt+1;i++){
    count[i]=count[i]+count[i-1];
  }

  for(j=dataSizeLeft-1;j>=0;j--){
    temp[count[localData[j]]]=localData[j];
    count[localData[j]]=count[localData[j]]-1;
  }
  for(i=0;i<dataSizeLeft;i++) {
    localData[i]=temp[i];
  }
  /*==  …                                                    ==*/
 /*================================================*/
  return 0;
}


main(argc, argv)
     int argc;
     char **argv;
{
  int i;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numOfProc);

  count = (int *)malloc((maxInt+1)*sizeof(int));
  if(my_rank == 0){
    ctrlMsgSend=numOfProc;  /* number of processes must be power of 2 */
    int i=0;
    integers=(int *)malloc(numOfIntToSort*sizeof(int));
    localData = (int *)malloc(numOfIntToSort*sizeof(int));
    for(i=0;i<numOfIntToSort;i++){
```

```
   localData[i]= ((int)rand())%maxInt;
  }

  /*   dataSizeSend = numOfIntToSort; */
  dataSizeRecv = numOfIntToSort;
  ctrlMsgRecv = numOfProc;

  dataSizeLeft = dataSizeRecv;
  solve(ctrlMsgRecv);
}
else{
  /* Every process waits a message before calling solve function   */
  MPI_Recv(&ctrlMsgRecv,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
            MPI_COMM_WORLD,&status);

  /*=DATA TRANSFER 2==============================*/
  /*= More of this block can be added on needed basis                =*/
  MPI_Recv(&dataSizeRecv,1,MPI_INT,MPI_ANY_SOURCE,
            MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  localData = (int *)malloc(dataSizeRecv*sizeof(int));
  MPI_Recv(
            localData,          /*<- modify address of data */
            dataSizeRecv,
            MPI_INT,            /*<- modify data type */
            MPI_ANY_SOURCE,MPI_ANY_TAG,
            MPI_COMM_WORLD,&status);
  /*=                                                              =*/
  /*==============================================*/

  dataSizeLeft = dataSizeRecv;
  solve(ctrlMsgRecv);
}
if(my_rank==0){

  for(i=0;i<numOfIntToSort;i++)
    {
```

```
            printf(" %d",localData[i]);
        }
    }

    MPI_Finalize();
}
```

LIST OF REFERENCES

1. Massingill BL, Mattson TG, Sanders BA. Patterns for Parallel Application Programs. Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999) http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/massingill/massingill.pdf Accessed August 2002.

2. Massingill BL, Mattson TG, Sanders BA. A Pattern Language for Parallel Application Programs. Proceedings of the Sixth International Euro-Par Conference (Euro-Par 2000) Springer, Heidelberg Germany 2000; pages 678-681

3. Massingill BL, Mattson TG, Sanders BA. Patterns for Finding Concurrency for Parallel Application Program. Proceedings of the Eighth Pattern Languages of Programs Workshop (PLoP 2000) Washington University Technical Report Number WUCS-00-29.

4. Massingill BL, Mattson TG, Sanders BA. More Patterns for Parallel Application Programs. Proceedings of the Eighth Pattern Languages of Programs Workshop (PLoP 2001) http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/PLoP2001/bmassingill0/PLoP2001_bmassingill0_1.pdf Accessed August 2002.

5. Massingill BL, Mattson TG, Sanders BA. Parallel Programming with a Pattern Language. International Journal on Software Tools for Technology Transfer 2001;volume 3 issue 2 pages 217-234.

6. Coplien JO, Schmidt DC, editors. Pattern Languages of Program Design. Addison-Wesley, Reading MA 1995.

7. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading MA1995.

8. Message Passing Interface Forum. A Message Passing Interface Standard. http://www.mpi-forum.org/docs/docs.html Accessed August 2002.

9. Bailey D, Barszcz E, Barton J, Browning D, Carter R, Dagum L, Fatoohi R, Fineberg S, Frederickson P, Lasinski T, Schreiber R, Simon H, Venkatakrishnan V, Weeratunga S. The NAS Parallel Benchmark RNR Technical Report RNR-94-007 1994.

10. Massingill BL, Mattson TG, Sanders BA. A Pattern Language for Parallel Application Program. http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/PDSE99_long.htm Accessed August 2002.

11. Knuth DE. The Art of Computing Programming: volume 2. Addison-Wesley, Reading MA 1981.

12. IEEE Standard for Binary Floating Point Numbers. ANSI/IEEE Standard 754-1985. IEEE New York 1985.

13. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. The MIT Press, Cambridge MA1998.

14. Squyer M, Lumsdaine A, George WL, Hagedorn JG, Devaney JE. The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI. MPI Developer's Conference (MPIDC), Ithaca NY 2000.

15. Saphir W, Wijngaart RV, Woo A, Yarrow M. New Implementations and Results for the NAS Benchmark 2. 8th SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis MN 1997.

16. Cole M. Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge MA1989.

17. Coplien JO, Schmidt DC, editors. Pattern Languages of Program Design. Addison-Wesley, Reading MA1995.

18. Reynders JV, Hinker PJ, Cummings JC, Atlas SU, Banerjee S, Humphrey W, Karmesin SR, Keahey K, Srikant M, Tholburn M. POOMA: A Framework for Scientific Simulation on Parallel Architectures. SuperComputing, San Diago CA 1995.

19. Armstrong R. An Illustrative Example of Frameworks for Parallel Computing. Proceedings Of Parallel Object Oriented Methods and Applications (POOMA) http://www.acl.lanl.gov/Pooma96/abstracts/rob-armstrong/pooma96.htm. Accessed August 2002

20. Launay P, Pazat JL. A Framework for Parallel Programming in Java. Proceedings of the High-Performance Computing and Networking (HPCN Europe) Springer, Heidelberg Germany 1998; pages 628-637

21. Lea D. Concurrent Programming in Java. Addison-Wesley, Reading MA 1996.

22. Douglas D, Schmidt C, Stal M, Rohnert H, Buschmann F. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Wiley & Sons, New York NY 2000.

23. Blelloch GE, Plaxton CG, Leiserson CE, Smith SJ, Zagha M. A Comparison of Sorting Algorithms for the Connection Machine CM-2. ACM Symposium on Parallel Algorithms and Architectures, Hilton Head SC 1991.

24. Dusseau AC, Culler DE, Schauser KE, Martin RP. Fast Parallel Sorting under LogP: Experience with the CM-5. IEEE Transactions on Parallel and Distributed Systems August 1996; pages 791-805

25. Shan H, Singh JP. Parallel Sorting on Cache-Coherent DSM Multiprocessors. Proceedings of the 1999 conference on Supercomputing http://www.supercomp.org/sc99/proceedings/papers/shan.pdf Accessed August 2002.

26. Al AE. A Measure of Transaction Processing Power. Datamation 1985; volume 32 issue 7 pages 112-118

27. Saini S, Bailey DH. NAS Parallel Benchmark (Version 1.0) Results 11-96. Report NAS-96-018 December 1996.

28. Saini S, Bailey DH. NAS. Parallel Benchmark Results 12-95. Report NAS-95-021 December 1995.

BIOGRAPHICAL SKETCH

Eunkee Kim was born 1970 in Dea-Gu, Republic of Korea. He received a B.S in physics at Yeung-Nam Universtiy in Keung-San, Republic of Korea, in 1997. He joined the graduate program of the Computer and Information Science and Engineering Department in 1999 to pursue his master's degree.