Boise State University
ScholarWorks

Computer Science Graduate Projects and Theses | Department of Computer Science

12-1-2012

Object Oriented Implementation of the Parallel Toolkit Library

Sandhya Vinnakota
Boise State University

OBJECT ORIENTED IMPLEMENTATION
OF
THE PARALLEL TOOLKIT LIBRARY

by

Sandhya Vinnakota

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

December 2012

© 2012
Sandhya Vinnakota
ALL RIGHTS RESERVED

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the project submitted by

Sandhya Vinnakota

Project Title: Object Oriented Implementation of the Parallel Toolkit Library

Date of Final Oral Examination: 12 December 2012

The following individuals read and discussed the project submitted by student Sandhya Vinnakota, and they evaluated their presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Amit Jain, Ph.D.	Chair, Supervisory Committee
Jim Buffenbarger, Ph.D.	Member, Supervisory Committee
Jyh-haw Yeh, Ph.D.	Member, Supervisory Committee

The final reading approval of the project was granted by Amit Jain, Ph.D., Chair, Supervisory Committee. The project was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ACKNOWLEDGMENTS

I would like to express gratitude to a lot of people for support during this work. I would like to especially thank my husband and my son Vibodh Ayyapureddi without whose support I would not have finished this project. I also thank my parents for their support and sacrifice .

I would also like to thank my advisor Dr Amit Jain for his wonderful support and patience throughout the project.

ABSTRACT

With manufacturing efficiencies and technological innovation the computing power of commodity machines has been increasing accompanied by decreasing costs. With the very favorable price/performance ratio the computing community has shifted from monolithic machines to networked machines.

This has created the need for software to manage the parallelism of the network. One such work has been the Parallel Toolkit Library. The Parallel Toolkit Library provides support for common design functionalities used throughout parallel programs.

This work extends the PTK C library for C++ parallel programs. The motivation for the current project stems from the need to let parallel programs reap the benefits of a library with an object oriented programming approach. This also fits well with the introduction of C++ bindings in MPI. The library has been structured on object-oriented concepts. The functionality of the PTK-C has been encapsulated into various classes. Individual functionalities have also been split into multiple classes leading to modularity and reusability of code.

Template programming has been used to ensure type safety. The testing results are consistent with expectations in that the PTK-C++ is very much comparable to the PTK-C in terms of performance. In most cases, it would be more efficient to use the toolkit than to rewrite the code to recreate the efficiencies already present in the library.

TABLE OF CONTENTS

ABSTRACT v

LIST OF TABLES ix

LIST OF FIGURES x

1 Introduction 1

1.1 Problem Statement 1

1.2 Prior Work 1

1.2.1 Existing work in object-oriented PTK like libraries 1

1.3 Parallel functionality implemented 2

2 The Tool Kit Implementation 4

2.1 Introduction 4

2.1.1 Template Programming 4

2.1.2 Exception Handling 4

2.1.3 Memory Handling 5

2.1.4 Namespace 5

2.2 Implementation Discussion 6

2.2.1 Init 6

2.2.2 CommonVarHelper 6

2.2.3 Exit 6

2.2.4 Workpool 7
2.2.5 WorkpoolUserInterface 7
2.2.6 DistributedWorkpool 8
2.2.7 SendSets 9
2.2.8 SendTaskSets 9
2.2.9 SendResultSets 9
2.2.10 WorkpoolData 9
2.2.11 DualPassTokenRing 10
2.2.12 CentralizedWorkpool 11
2.2.13 Scatter1d 11
2.2.14 Scatter2d 12
2.2.15 Gather1d 12
2.2.16 Gather2d 12
2.2.17 AllToAll1d 13
2.2.18 AllToAll2d 13
2.2.19 Multicast 14
2.2.20 Filemerge 14

3 Using The Toolkit: Examples 15
3.1 Example using the ShortestPathCentralWorkpool 16
3.2 Example using the ShortestPathDistributedWorkpool 19

4 Testing and Benchmarking 22
4.1 Testing coverage of toolkit classes 22
4.2 Graphs comparing the performance of C vs C++ 23
4.3 Simulation notes 26

5 Conclusions 27
5.1 What have we done so far? 27
5.2 Design Patterns 28
5.3 What have we learned 28
5.4 Potential Future work 28
5.4.1 Complex Datatypes 28
5.4.2 Multi-threading 29
REFERENCES 30
A MPI Datatypes 31
B Installing the PTK Library 32

LIST OF TABLES

3.1 PTK examples and corresponding classes (Part 1) 15

3.2 PTK examples and corresponding classes (Part 2) 16

4.1 Testing coverage of toolkit classes (Part 1) 22

4.2 Testing coverage of toolkit classes(Part 2) 23

LIST OF FIGURES

2.1 Diagram depicting the relationship between the workpool classes 7

4.1 C versus C++:Shortest Path Central (Simple) - with granularity = 1. Vertices in K's 23

4.2 C versus C++:Shortest Path Central:Granularity=200, Vertices per task=100 24

4.3 C versus C++:Shortest Path Distributed (Simple) - with granularity = 1 25

4.4 C versus C++:Shortest Path Distributed:More Efficient,Granularity=200 26

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

With the introduction of the C++ bindings in Message Passing Interface(MPI)-2, and the existing Parallel Tool Kit(PTK) library being implemented in C, there was a need for C++ version of the PTK. This project addresses that need by converting the PTK-C library to PTK-C++.

As this work is an extension of the PTK [1], the scope of the project will be limited by the work done in original PTK C-library [1].

1.2 Prior Work

1.2.1 Existing work in object-oriented PTK like libraries

The most recent work is the work carried out in [1], which implemented the PTK library in C.

A rigorous literature search has failed to unearth any progressive work being done in the area of work pool management for parallel programs. The literature survey is consistently pointing to the work carried out by Sachs and McGough [3]. There is no code or any additional documentation available for [3].

The literature survey has indicated that the most pertinent body of work at present is the MPI Boost Library [4], which provides a C++ friendly interface to the standard MPI. However the library does not have any support for workpool management.

1.3 Parallel functionality implemented

The following common parallel functionalities are implemented in this project.

- *Scatter* describes the sending of data from a central coordinator process to a number of worker processes. Given a data set of size `n`, and a number of processes `p`, the data is generally divided into sets of size `n/p`. The `ith` set is then sent to the `ith` process.

  Main classes : `Scatter1d`, `Scatter2d`

- *Gather* describes an action that is essentially the opposite of scatter. After data is scattered, the processes each perform some application specific computations. The coordinator then gathers the data from each of the worker processes. The coordinator may then perform some computation on the gathered data.

  Main classes : `Gather1d`, `Gather2d`

- *All-to-all* communication happens when all of the processes need to send and receive data to and from all of the other processes.

  Main classes : `AllToAll1d`, `AllToAll2d`

- *Distributed file merge* is a variation of gather. This is necessary when a number of files are distributed over the cluster and also need to be merged and stored centrally at the coordinator process.

Main classes: `FileMerge`

- *Centralized workpool.* In the centralized model, the coordinator process maintains a pool of tasks that need to be performed. It sends each process a task. When the process is finished, it tells the coordinator it is done and is ready for a new task. The coordinator then sends another task. This is repeated until all the tasks are completed.

  Main classes: `CentralizedWorkpool`

- *Distributed workpool.* In the distributed model, each process maintains a pool of tasks that need to be performed. If a process runs out of tasks it can ask other processes for a task. Tasks are passed back and forth until they are all completed.

  Main classes: `DistributedWorkpool`

CHAPTER 2

THE TOOL KIT IMPLEMENTATION

2.1 Introduction

The following subsections deal with the implementation details common to all classes. The classes will be discussed in the subsequent section.

2.1.1 Template Programming

C++ template-style programming has been used in the PTK C++ toolkit. This allows us to use generic datatypes and also ensures typesafety.

Most of the classes in the PTK-C++ use the following style `template <typename TaskType, typename ResultType>` The general convention followed is: `TaskType` refers to the task datatype and `ResultType` refers to the result datatype.

2.1.2 Exception Handling

To provide custom exception handling, a new class `ptkException` has been defined in PTK-C++. All the classes in PTK-C++ make use of `ptkException` class to throw exceptions. When a process throws an exception, it is the responsibility of the user to handle the exception and terminate the program accordingly.

2.1.3 Memory Handling

In PTK C version, memory allocation, in most cases, is managed by the library user. However the PTK-C++ toolkit ended up allocating memory to the resulting data buffers whenever the size of the buffer can be calculated by the library. This is a benefit of using template programming.

2.1.4 Namespace

All the PTK-C++ classes have been added to a new namespace, `ptk`. To access the C++ code, the library user has to either use `using namespace ptk` or use the scope operator :: to access the C++ classes and methods.

2.2 Implementation Discussion

2.2.1 Init

The library user should access the methods `getInstance()` and `initialize()` at the outset, before proceeding, as these methods initialize the variables like group size, current process number and communicator handler. The previously global variables for group size, current process number and communicator handler are no longer global and have been made private members of the `Init` class. This class is a Singleton, to avoid creating multiple copies of all these variables at runtime.

2.2.2 CommonVarHelper

This class is inherited by all the PTK-C++ classes. The `CommonVarHelper` provides a method to access the private variables of the `Init` class by calling their respective get methods.

However, it is the responsibility of the inheriting class to ensure that the values are available in its class by calling the `populatePtkVariables()` method inherited from `CommonVarHelper`.

2.2.3 Exit

The `exitProgram()` method in this class calls `MPI::Barrier` and `MPI::Finalize`. Similar to the `Init` class, only one instance of this class should exist, so the `Exit` class is implemented as a Singleton.

2.2.4 Workpool

This is an abstract class which provides some common variables and an interface for all the workpool implementation classes. This class is inherited by the `CentralizedWorkpool` and `DistributedWorkpool` classes.

The Figure 2.1 depicts the relationship between the various workpool classes which are described in the subsequent paragraphs.

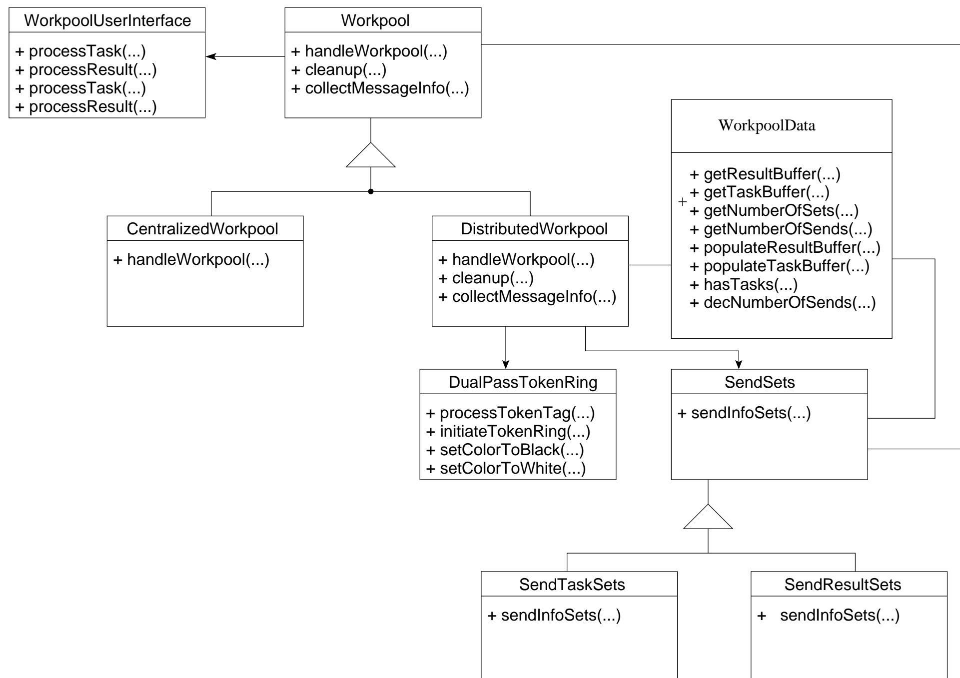

Figure 2.1: Diagram depicting the relationship between the workpool classes

2.2.5 WorkpoolUserInterface

In workpool management, the main pieces of data that are passed around are tasks and results. A workpool needs to only know the task/result datatype and their size in bytes. The workpool classes have no knowledge of how to process these tasks and results. It is the responsibility of the library user to process the tasks and results.

The `WorkpoolUserInterface` is an abstract class that defines methods to process tasks and results. All the methods in the `WorkpoolUserInterface` are implemented to return exceptions.

All the workpool classes expect a reference to the `WorkpoolUserInterface`. Whenever tasks/results are to be processed, the workpool calls the corresponding method using the `WorkpoolUserInterface` reference it received. The library user wanting to use the workpool classes should create a class which inherits from the `WorkpoolUserInterface`. As all the default methods in `WorkpoolUserInterface` throw exceptions, the library user is forced to overide to these methods.

2.2.6 DistributedWorkpool

In the distributed version of the workpool, all the nodes are workers.

Distributed workpool imlementation mainly carries out the following:

- probes the network for tasks/results.
- receives tasks/results and calls the library-user methods to process Results/Tasks
- sends out tasks/results by grouping them based on the receiver
- detects termination

In PTK-C++, the object-oriented concept of abstraction is used. Instead of including all the above listed functionalities in the same class, only the main essence of the distributed workpool is encapsulated in the `DistributedWorkpool` class. The implementation details for termination detection, and sending out of tasks/results, have been moved to separate classes. By doing so, we achieve abstraction and reusability.

2.2.7 SendSets

`SendSets` is an abstract class which provides the common interface to send tasks and results. This class includes some common variables and a method `sendInfoSets`. In PTK-C++, this class is inherited by `SendTaskSets` and `SendResultSets`.

2.2.8 SendTaskSets

The `SendTaskSets` class inherits from the `SendSets` class and implements the `sendInfoSets()` method. This class mainly deals with the implementation details of sending out tasks. `sendInfoSets(...)` loops through the incoming stream of tasks and packs tasks into one buffer until it encounters a new `sendTo` receiver. If the `sendTo` variable equals -1, then the tasks are sent to all the nodes in the group.

2.2.9 SendResultSets

The `SendResultSets` class inherits from the `SendSets` class and implements `sendInfoSets()` method. This class mainly deals with the implementation details of sending out results. `sendInfoSets(...)` loops through the incoming stream of results and packs results into one buffer until it encounters a new `sendTo` receiver. If the `sendTo` variable equals -1, then the results are sent to all the nodes in the group.

2.2.10 WorkpoolData

The `WorkpoolData` class was created to abstract the multiple parameters during the communications between `DistributedWorkpool` and `SendTaskSets/SendResultSets`.

2.2.11 DualPassTokenRing

In PTK-C++, termination detection has been separated from the workpool logic and implemented in a new class, `DualPassTokenRing`. This enables reusability of the termination logic. `DualPassTokenRing` class uses the dual-pass token-ring algorithm, originally developed by Djikstra, Feijen, and van Gasteren [5].

The main methods of this class are described briefly.

- `setColorToWhite()` Sets the color of the token to white.

- `setColorToBlack()` Sets the color of the token to black.

- `processToken(....)` This method decides how to modify the process color and token based on the incoming token value. If the received token is white and the process color is white, it passes on the white token. If the received token is white and the process color is black, it turns the token black and passes it on. It the token is black, it will be passed on as is, regardless of the color of the process. After a process passes on the token, it changes its color to white. If the root receives a white token, then it sets the incoming `type` variable to `DONETAG`. Now it is the responsibility of the calling process to parse the `DONETAG` and decide on the next course of action. In the `DistributedWorkpool` class, the root sees that the type is set to `DONETAG` and sends out `DONETAG` to all the workers.

- `initiateTokenRing()` The root from the workpool calls this method to start the token ring process.

2.2.12 CentralizedWorkpool

`CentralizedWorkpool` is the subclass of the `Workpool` class. The library user wanting to use the centralized workpool, invokes the `handleWorkpool(...)` method of the `CenralizedWorkpool` class.

The implementation logic in the PTK-C++ version has been divided into multiple methods, as opposed to one single function in PTK-C version. The `rootProcess(...)` method takes care of the coordinator‘s functionality.

The coordinators‘ (root process) responsibility is to maintain a pool of tasks, distribute them to the workers, process results, and terminate the function when all the tasks have been completed. Tasks are stored in a vector. The coordinator is given an initial set of tasks to begin with, called the `startingObjects`.

The coordinator then enters a loop, waiting for messages from the workers. The received message may be a request for a task, or a result. If the request is for a task, and the task list is not empty, the coordinator sends the worker a task. If the request is a result, the coordinator calls the `processResults(...)` method.

The workers also enter a loop. They send a request for a task to the coordinator and then receive the task. The `processTask(...)` method is called and the result is sent back to the coordinator. The loop is terminated when the worker receives a `DONETAG` message from the coordinator.

2.2.13 Scatter1d

The `Scatter1d` class is the C++ implementation of the scatter functionality.

The scatter operation in the `Scatter1d` class is mainly carried out by `scatterRootOperation(...)` and `scatterNonRootOperation(...)`.

`scatterRootOperation(...)` sends out data to the receiving nodes. In `scatterNonRootOperation(...)`, the library allocates memory to the receiving buffer and populates it with the data received from the root.

2.2.14 Scatter2d

`Scatter2d` class is the C++ implementation of the scatter2d functionality.

The scatter operation in the `Scatter2d` class is mainly carried out by `scatterRootOperation(...)` and `scatterNonRootOperation(...)`.

In the `scatterRootOperation(...)` the root distributes the data in a two-dimensional array to all the receiving nodes. In the `scatterNonRootOperation(...)` the non-root process allocates memory for an one-dimensional array and populates it with data received from the root.

2.2.15 Gather1d

The gather operation in the `Gather1d` class is mainly carried out by `gatherRootOperation(...)` and `gatherNonRootOperation(...)`.

In `gatherRootOperation(...)` the root gathers data from other process nodes and stores it in a one-dimensional array. In `gatherNonRootOperation(...)` the nodes send an array of size `sendCount`, with the exception of the last process which sends an array of size `lastCount`.

2.2.16 Gather2d

The gather operation in the `Gather2d` class is mainly carried out by `gatherRootOperation(...)` and `gatherNonRootOperation(...)`.

In `gatherRootOperation(...)` the root allocates memory to the two-dimensional array and then fills it up with the data coming from the other nodes. In the resulting two dimensional array, each row corresponds to data from one node. The number of rows in the two-dimensional array at root equal the number of processes in the group. In `gatherNonRootOperation(...)`, the process in the group sends a one-dimensional array to the root.

2.2.17 AllToAll1d

Using the `AllToAll1d` class, each process node can send and receive the same amount of data to and from the other processes.

In the `sendAllToAll(...)` method, each process loops with the number of iterations equal to the group size. At each iteration, each process sends data to the process in the `i`th position to the right, and receives from the process `i`.

2.2.18 AllToAll2d

Using the `AllToAll2d` class, each process node sends and receives a variable amount of data from all the other processes. The communication pattern is the same as that of the `AllToAll1d`. The `sendAllToAll(...)` method implements the main logic for the `AllToAll2d`. This method receives a two dimensional array as input. Each row in this array corresponds to a process in the group. The result from `AllToAll2d` communication is a two-dimensional array. The toolkit allocates memory and populates it with data and returns the two-dimensional array to the user program.

2.2.19 Multicast

Using the `mcast(...)` method of the `Multicast` class, a sending node can send a collection of data to all the other nodes in the group.

2.2.20 Filemerge

The `FileMerge` class merges all the files received from the other processes.

CHAPTER 3

USING THE TOOLKIT: EXAMPLES

Snippets of code pertaining to the shortestpath calculation, using the `CentralizedWo-rkpool` and `DistributedWorkpool`, are presented here. The full code can be found at

`$PTK_HOME/examples/ShortestPathCentralizedImpl.cpp`.

`$PTK_HOME/examples/ShortestPathDistributedImpl.cpp`.

Other examples, which show the usage of the various PTK-C++ classes can be found in `$PTK_HOME/examples`. Please refer to Table 3.1 and Table 3.2 for the list of examples.

Table 3.1: PTK examples and corresponding classes (Part 1)

Toolkit Class	Example/test program
AllToAll1d	AllToAll1dImpl.cpp
AllToAll2d	AllToAll2dImpl.cpp
	BucketSortWithAllToAll2dImpl.cpp
CentralizedWorkpool	ShortestPathCentralImpl.cpp
	ShortestPathCentralMoreEfficientImpl.cpp
	SumOfSquaresTest.cpp
CommonVarHelper	all programs
DistributedWorkpool	ShortestPathDistributedImpl.cpp
	ShortestPathDistributedMoreEfficientImpl.cpp

Table 3.2: PTK examples and corresponding classes (Part 2)

DualPassTokenRing	ShortestPathDistributedImpl
	ShortestPathDistributedMoreEfficientImpl.cpp
Exit	all programs
FileMerge	FileMerge.cpp
Gather1d	Gather1dTest.cpp
Gather2d	Gather2dTest.cpp
Init	all programs
Multicast	ShortestPathCentralImpl.cpp
	ShortestPathCentralMoreEfficientImpl.cpp
Scatter1d	ShortestPathDistributedImpl.cpp
Scatter2d	BucketSortWithScatterImpl.cpp
SendTaskSets	ShortestPathDistributedImpl.cpp
	ShortestPathDistributedMoreEfficientImpl.cpp
SendResultSets	ShortestPathDistributedImpl.cpp
	ShortestPathDistributedMoreEfficientImpl.cpp
Workpool	ShortestPathCentralImpl.cpp
	ShortestPathDistributedImpl.cpp
WorkpoolData	ShortestPathDistributedImpl.cpp
	ShortestPathDistributedMoreEfficientImpl.cpp
WorkpoolUserInterface	ShortestPathCentralImpl.cpp
	ShortestPathDistributedImpl.cpp
ptkException	all programs

3.1 Example using the ShortestPathCentralWorkpool

The shortest path example has been divided into two classes `ShortestPaths` , `ShortestPathImpl1` and one main program `ShortestPathCentralImpl.cpp`.

Shortestpath logic common to both centralized and distributed workpool examples has been included in the `ShortestPaths` class.

The library user should create a class that inherits from the `WorkpoolUserInterf-ace` and provide implementations for the inherited methods, `processTask(...)` and `processResult(...)`. This is illustrated in the snippet below.

```
class ShortestPathImpl1: public WorkpoolUserInterface<int,int> {
 int processTask(int *task, int **ptkResult, int *returnSize);
 int processResult(int *results, int **ptkNewTasks, int *numNewTasks);
 }
```

The task in this example is composed of a vertex, represented as an int, followed by an array of ints, which represent the current best set of distances from the source. The `processTask(...)` method of the `ShortestPathImpl1` class, copies the incoming task values into a local vertex value and a `distances` array. `distances` is a member of the `ShortestPaths` class.

The `processTask(...)` method then traverses the distances array, checking to see if there are any new distances that are better than the current one. If it finds a better distance, it stores the new information. The first value it packs is the array index, which represents the path through which the new distance comes. The second value is the vertex, and the third is the new distance. These can be thought of as sets of results. These results are packed into a global array so that we are not incurring the overhead of frequent memory allocation and deallocation. The function returns values appropriate to whether or not it found any new results.

The `processResult(...)` method of the `ShortestPathImpl1` class, handles the incoming new results. The number of results in the results array is the first element in that array, represented as an integer. This value is extracted and assigned to a value named `size`. The results array is iterated `size` times. At each iteration, the vertex, the vertex the new distance is through (the fromVertex), and the new distance are considered. If the new distance is better than the distance we have stored for this vertex, the new information is stored. If new information is stored, a flag is set to indicate that there is new data for that vertex. Then the array of flags is looped

through, and new tasks for vertices are created if there is new information to handle.

The main pieces of code in `ShortestPathCentralImpl.cpp` :

As `Init` and `Exit` are Singleton classes, `getInstance()` methods need to be invoked.

```
Init &init1 = Init::getInstance();
Exit &exit1 = Exit::getInstance();
```

The `initialize()` method from the Init class initializes the PTK environment. Failure to invoke this method will result in malfunction of the code.

```
init1.initialize(argc, argv);
```

The current process number (`me`) and group size (`gsize`) values need to be retreived in the following manner.

```
me = init1.getMyProcessNumber();
gsize = init1.getGroupSize();
```

An instance of the `CentralizedWorkpool` class is created by passing in a reference to `ShortestPathImpl1`, a subclass of the `WorkpoolUserInterface`.

```
ShortestPathImpl1 *sPathImpl = new ShortestPathImpl1(....);

wpool = new CentralizedWorkpool<int, int>(sPathImpl,
                                          granularity,
```

```
                                              root,
                                              verbose);
```

In order to trigger the centralized workpool logic, the `handleWorkpool(...)` method is called.

```
    try {
           messagesSent = wpool->handleWorkpool(A,
                                                taskSize,
                                                resultSize,
                                                1); // arraylen
    } catch (ptkException* e) {
           cout << "Exception caught" << endl;
           cerr <<e->getMessage() << endl;
    }
```

`exitProgram()` needs to be called at the end of every MPI program as it invokes `MPI::Finalize()`.

```
  exit1.exitProgram();
```

3.2 Example using the ShortestPathDistributedWorkpool

The shortestpath example using the `DistributedWorkpool` is similar to the above example. This section deals with those pieces that differ from the `ShortestPathCentralizedWorkpool` example.

The ShortestPath example has been divided into two classes `ShortestPaths` , `ShortestPathImpl` and one main program `ShortestPathDistributedImpl.cpp`.

The library user should create a class that inherits from the `WorkpoolUserInterface` and provide implementation to the inherited methods, `processTask(...)` and `processResult(...)`. This is illustrated in the snippet below.

```
class ShortestPathImpl: public WorkpoolUserInterface<int,int>
{
    int processTask(int *dataToProcess, int tasksToProcess,
                    int **ptkNewTasks, int *numTasks,
                    int **ptkResults, int *numResults);

    int processResult(int *result);
    ....

 }
```

In this example, each process is responsible for (total vertices / number of processes in group) number of vertices. As each process keeps its own current set of best results, the entire distance array does not need to be sent as part of the task.

The `processTask(...)` method picks up the vertex, path, and distance information from the incoming task. It iterates through its current best distances array to see if the new vertex information creates any new tasks. As it finds new tasks it stores them up. Each new task contains information about the process to send the new task to, the vertex, the path, and the new distance.

The `processResult(...)` method is triggered at each node and it iterates through the new distances received, compares them to the nodes' own current best data, and replaces any values that are better than the current one.

Main pieces of code in `ShortestPathDistributedImpl.cpp` :
An instance of the `DistributedWorkpool` class is created by passing in a reference to `ShortestPathImpl`, a subclass of the `WorkpoolUserInterface`.

```
ShortestPathImpl *sPathImpl = new ShortestPathImpl(....);

Workpool<int> *wpool = new DistributedWorkpool<int, int>(sPathImpl,
                                                        granularity,
                                                               root,
                                                            verbose);
```

In order to trigger the distributed workpool logic, the `handleWorkpool(...)` method is called.

```
    try {
           messagesSent = wpool->handleWorkpool(A,
                                                taskSize,
                                                resultSize,
                                                1); // arraylen
    } catch (ptkException* e) {
           cout << "Exception caught" << endl;
           cerr <<e->getMessage() << endl;
    }
```

CHAPTER 4

TESTING AND BENCHMARKING

4.1 Testing coverage of toolkit classes

The tables 4.1 and 4.2 list out the various classes and their corresponding test programs.

Table 4.1: Testing coverage of toolkit classes (Part 1)

Toolkit Class	Example/test program
AllToAll1d	AllToAll1dImpl.cpp
AllToAll2d	AllToAll2dImpl.cpp
	BucketSortWithAllToAll2dImpl.cpp
CentralizedWorkpool	ShortestPathCentralImpl.cpp
	ShortestPathCentralMoreEfficientImpl.cpp
	SumOfSquaresTest.cpp
CommonVarHelper	all programs
DistributedWorkpool	ShortestPathDistributedImpl.cpp
	ShortestPathDistributedMoreEfficientImpl.cpp
DualPassTokenRing	ShortestPathDistributedImpl
	ShortestPathDistributedMoreEfficientImpl.cpp
Exit	all programs
FileMerge	FileMerge.cpp
Gather1d	Gather1dTest.cpp
Gather2d	Gather2dTest.cpp
Init	all programs

Table 4.2: Testing coverage of toolkit classes(Part 2)

Multicast	ShortestPathCentralImpl.cpp
	ShortestPathCentralMoreEfficientImpl.cpp
Scatter1d	ShortestPathDistributedImpl.cpp
Scatter2d	BucketSortWithScatterImpl.cpp
SendTaskSets	ShortestPathDistributedImpl.cpp
	ShortestPathDistributedMoreEfficientImpl.cpp
SendResultSets	ShortestPathDistributedImpl.cpp
	ShortestPathDistributedMoreEfficientImpl.cpp
Workpool	ShortestPathCentralImpl.cpp
	ShortestPathDistributedImpl.cpp
WorkpoolData	ShortestPathDistributedImpl.cpp
	ShortestPathDistributedMoreEfficientImpl.cpp
WorkpoolUserInterface	ShortestPathCentralImpl.cpp
	ShortestPathDistributedImpl.cpp
ptkException	all programs

4.2 Graphs comparing the performance of C vs C++

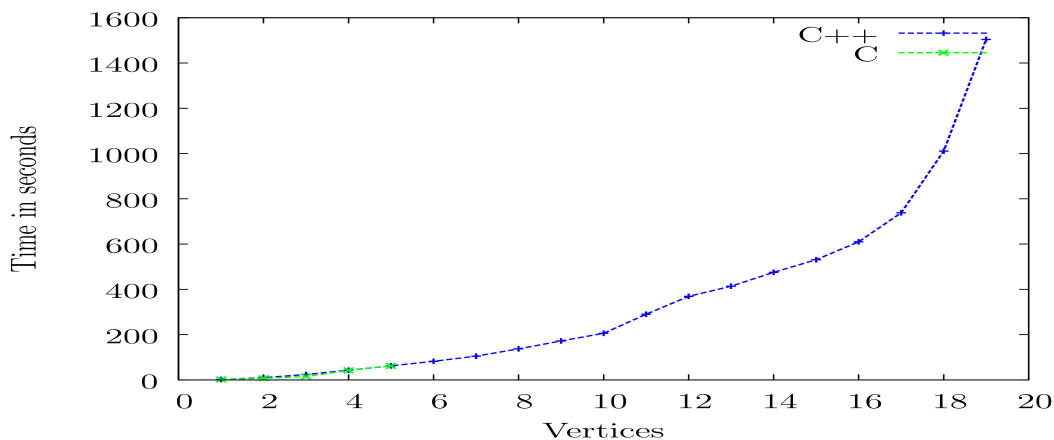

Figure 4.1: C versus C++:Shortest Path Central (Simple) - with granularity = 1. Vertices in K‘s

Figure 4.1 illustrates the difference in performance between the C and the C++ version. The performance is comparable and shows that there is not much overhead

from the object oriented approach. The C version of the PTK stops functioning after 6000 vertices. This is an artifact of `malloc()` which is used in the memory allocation. The C version uses a growable array which doubles its size, on the fly, whenever the original array runs out of memory. `malloc()` has a limitation where it cannot allocate memory beyond a certain value depending on the compiler. On onyx the limit is 2Gig. Once the program gets to a point where more than 2Gig needs to be allocated, `malloc()` returns a NULL pointer and the program exits.

This can be surmounted in other ways, like using linked lists. This, however, resulted in significant performance degradation and was not very practical when implemented. The C++ version uses vectors, which are available in C++. The memory allocation is handled by C++ libraries. As a result, the C++ version was able to run upto 20000 vertices.

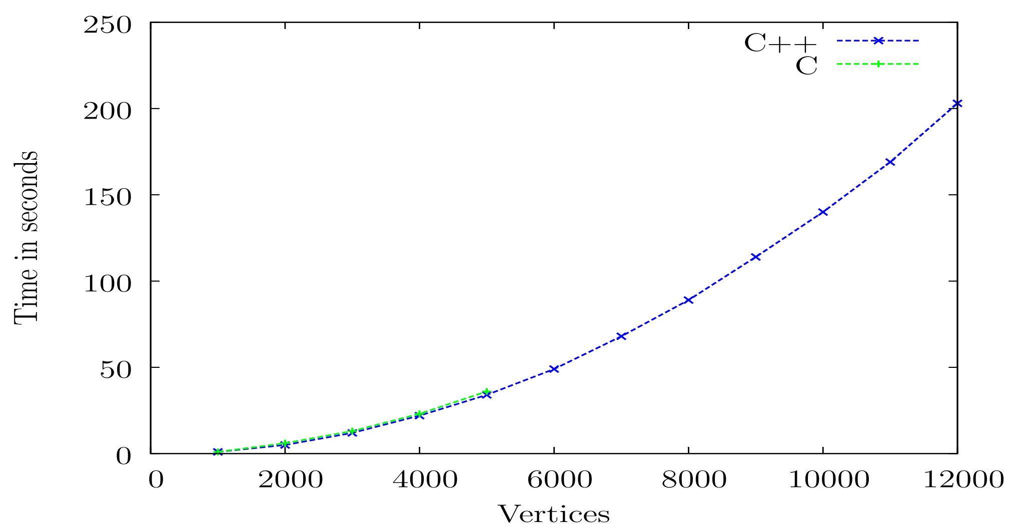

Figure 4.2: C versus C++:Shortest Path Central:Granularity=200, Vertices per task=100

Figure 4.2 illustrates the performance of the C and C++ versions for the more efficient case of the ShortestPathCentralized example. As in the previous graph, the C version exhibits the limitation of `malloc()` and stops functioning at 6000 vertices.

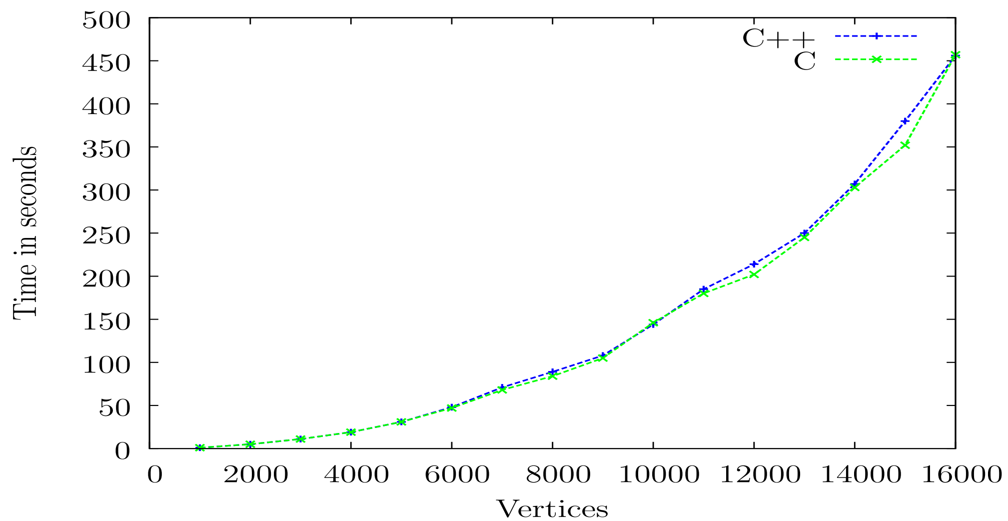

Figure 4.3: C versus C++:Shortest Path Distributed (Simple) - with granularity = 1

Figure 4.3 illustrates the performance of the C and C++ versions for the simple case of the ShortestPath example, which uses the Distributed Workpool. The performance is very similar. The memory allocation constraint for distributed workpool is much more relaxed than the centralized workpool, as memory is allocated across multiple machines.

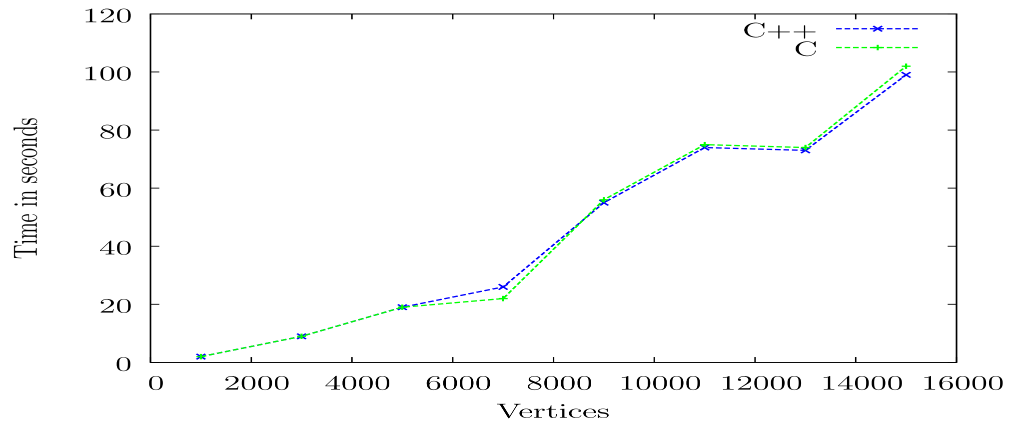

Figure 4.4: C versus C++:Shortest Path Distributed:More Efficient,Granularity=200

Figure 4.4 illustrates the performance of the C and C++ versions for the more efficient case of the ShortestPath example, which uses the Distributed Workpool. The performance is very similar.

i

4.3 Simulation notes

All the graphs were generated with the number of processors = 20.

Testing was performed on two department clusters, onyx and beowulf.

CHAPTER 5

CONCLUSIONS

5.1 What have we done so far?

PTK-C++ provides an object-oriented version of PTK-C. PTK-C++ now allows the user to fully reap the benefits of the MPI C++ bindings.

PTK-C++ provides support for common design tasks used throughout parallel programs. The examples given help a user understand how to use the library classes.

The data-sharing patterns of gather, scatter, and all-to-all allow users the flexibility of having odd amounts of data that are not evenly divisible by the number of processes. The two-dimensional versions allow the user to share "ragged" arrays of data. These elements are not provided by MPI. The file-merging functionality automates a common cluster task.

The workpools remove a significant layer of detail from writing workpool code. The user of the workpool needs to provide the library with functions for processing tasks and results. The library takes care of sending and receiving tasks and results. Most importantly it handles termination detection, which can be quite cumbersome to design and write.

The testing results are consistent with expectations in that PTK-C++ is comparable to PTK-C in terms of performance.

In most cases it would be more efficient to use the toolkit than to rewrite the code to incorporate the efficiencies already present in PTK.

5.2 Design Patterns

The Singleton pattern has been used for imlementing the `Init` and `Exit` classes as they fit the criterion for the Singleton.

The project has a few instances where the code was implemented based on inspiration from the Strategy pattern. For example, the Workpool abstract class defines two(`CenralizedWorkpool` and `DistributedWorkpool`) sets of algorithms. Each of the set encapsulates its own algorithm and the user is given the choice of picking the algorithm at run-time.

5.3 What have we learned

- Object Oriented Design: A project cannot be started with the preconceived notion of using Design patterns. It involved thinking in terms of classes and object oriented approach. The design started of by investigating the required functionality rather than by translating the C code into C++.
- Had to learn the details of MPI-C++ to use them in the PTK.

5.4 Potential Future work

5.4.1 Complex Datatypes

The PTK-C and PTK-C++ implementations function with simple datatypes. The PTK-C++ classes could be easily enhanced to work for complex datatypes.

5.4.2 Multi-threading

Experimenting with multi-threading, particularly in the centralized workpool, would be worth while doing. There may be a bottleneck in the root being able to hand out tasks quickly enough. Creating separate threads for communication and computation might solve this problem. The decision was made to avoid any threading, to make the code as universal and portable as possible.

REFERENCES

[1] Kirsten Allison. *PTK: A PARALLEL TOOLKIT LIBRARY* M.S. Thesis, Computer science department, Boise State university, 2007

[2] William Gropp, Ewing Lusk, Anthony Skjellum. *Using MPI: Portable Paral- lel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, Massachusetts, 1999.

[3] Sachs, N, McGough, J. A Hybrid Process Farm/Work Pool Implementation in a Distributed Environment Using MPI. *Conference Proceedings, Midwest Instructional Computing Symposium* , Duluth, MN, April 2003

[4] Douglas Gregor, Matthias Troyer. *Boost.MPI.* `http://www.boost.org/doc/libs/1_46_1/doc/html/mpi.html#stylefiles`

[5] Edsger W. Dijkstra, W.H.J Feijen, A.J.M van Gasteren. "Derivation of a Termination Detection Algorithm for a Distributed Computation." *Information Processing Letters*, vol. 16(5), pp. 217-219.

[6] Jason Man and Amit Jain. *PRAND: A Parallel Random Number Generator*. Oct 2008. `http://cs.boisestate.edu/~amit/research/prand`.

APPENDIX A

MPI DATATYPES

The datatypes supported in the MPI version of the PTK-C++ toolkit are defined as follows:

```
MPI::CHAR
MPI::SHORT
MPI::UNSIGNED_SHORT
MPI::INT
MPI::UNSIGNED
MPI::LONG
MPI::UNSIGNED_LONG
MPI::FLOAT
MPI::DOUBLE
MPI::LONG_DOUBLE
MPI::LONG_LONG_INT
```

APPENDIX B

INSTALLING THE PTK LIBRARY

The library is available to download at `http://cs.boisestate.edu/~amit/research/ptk`. You can either download the library as tarballs or as RPM files. You will need MPICH version 2 (or some other MPI-2 supporting libary). You will also need the prand library [6] to run some of the examples.

To install MPI version of the PTK library from the tarball, use the following commands:

```
tar xzvf ptk-mpi.tar.gz
cd ptk-mpi/src
su
make system_install
```