REUSABLE TEMPLATE LIBRARY FOR PARALLEL PATTERNS

By

CHI-KIN WONG

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

UNIVERSITY OF FLORIDA

2002
To my mother, who always supported me and encouraged me.
ACKNOWLEDGMENTS

I would like to thank Dr. Beverly A. Sanders for all her support and motivation. She has always been willing to give her precious time to help me with her great ideas and expertise whenever I asked for her advice. She also provided me an office space and a dual processor machine for testing and analysis. She is a very responsible and kind educator who has the desire to help students to succeed. This thesis would never have been possible without her great guidance. I would also like to thank Dr. Doug D. Dankel and Dr. Joseph N. Wilson for agreeing to serve on my committee and to review my work, and Mr. Sullivan Beck, Computer Information Science and Engineering Department system administrator, for teaching me how to install and to configure the self-managed Linux machine. Last, I would like to thank all my friends, schoolmates, and professors who have always been with me through my college year.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>x</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 BACKGROUND OF OPENMP</td>
<td>4</td>
</tr>
<tr>
<td>Compiler Internal</td>
<td>4</td>
</tr>
<tr>
<td>Programming in OpenMP</td>
<td>7</td>
</tr>
<tr>
<td>3 REUSABILITY OF OPENMP</td>
<td>13</td>
</tr>
<tr>
<td>OpenMP and Other Classical Thread Program</td>
<td>14</td>
</tr>
<tr>
<td>Syntax Constraints</td>
<td>19</td>
</tr>
<tr>
<td>4 THE PARALLEL PATTERNS</td>
<td>27</td>
</tr>
<tr>
<td>Embarrassingly Parallel</td>
<td>27</td>
</tr>
<tr>
<td>Divide And Conquer</td>
<td>28</td>
</tr>
<tr>
<td>Pipeline Processing</td>
<td>29</td>
</tr>
<tr>
<td>Separable Dependencies</td>
<td>30</td>
</tr>
<tr>
<td>5 DESIGN, IMPLEMENTATION, AND ANALYSIS</td>
<td>32</td>
</tr>
<tr>
<td>Embarrassingly Parallel</td>
<td>32</td>
</tr>
<tr>
<td>Design</td>
<td>32</td>
</tr>
<tr>
<td>Function Parameter</td>
<td>32</td>
</tr>
<tr>
<td>Template Library Implementation</td>
<td>33</td>
</tr>
<tr>
<td>Examples</td>
<td>35</td>
</tr>
<tr>
<td>Pipeline Processing</td>
<td>37</td>
</tr>
<tr>
<td>Design</td>
<td>37</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Override Functions for Divide And Conquer</td>
<td>48</td>
</tr>
<tr>
<td>2 Constructs</td>
<td>68</td>
</tr>
<tr>
<td>3 Data-Sharing Attribute Clauses</td>
<td>70</td>
</tr>
<tr>
<td>4 Run-time Library Functions</td>
<td>71</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel C++ Compiler 6.0</td>
<td>5</td>
</tr>
<tr>
<td>Parallel Region Internal</td>
<td>6</td>
</tr>
<tr>
<td>OpenMP Example 1</td>
<td>8</td>
</tr>
<tr>
<td>OpenMP Example 2</td>
<td>8</td>
</tr>
<tr>
<td>OpenMP Example 3</td>
<td>9</td>
</tr>
<tr>
<td>OpenMP Example 4</td>
<td>9</td>
</tr>
<tr>
<td>OpenMP Example 5</td>
<td>10</td>
</tr>
<tr>
<td>OpenMP Example 6</td>
<td>10</td>
</tr>
<tr>
<td>OpenMP Example 7</td>
<td>11</td>
</tr>
<tr>
<td>OpenMP Example 8</td>
<td>12</td>
</tr>
<tr>
<td>Embarrassingly Parallel</td>
<td>27</td>
</tr>
<tr>
<td>Divide And Conquer</td>
<td>29</td>
</tr>
<tr>
<td>Pipeline Processing</td>
<td>30</td>
</tr>
<tr>
<td>Separable Dependencies</td>
<td>31</td>
</tr>
<tr>
<td>Embarrassingly Parallel</td>
<td>34</td>
</tr>
<tr>
<td>Embarrassingly Parallel Loop Base Example</td>
<td>36</td>
</tr>
<tr>
<td>Embarrassingly Parallel Object Base Example</td>
<td>37</td>
</tr>
<tr>
<td>Pipeline Processing Programming Structure</td>
<td>38</td>
</tr>
</tbody>
</table>
The high performance parallel computing is designed for complex and intensive scientific computation. OpenMP is the latest industry standard for multiprogramming in shared-memory system. Unlike classical thread programming, OpenMP embeds fork-join parallelism to create a high level abstraction for parallel programming. In this project, I applied OpenMP parallel library in C++ to build a reusable template library for parallel programming patterns and discovered the design and implementation, reusability, and scalability of OpenMP. Embarrassingly Parallel, Pipeline Processing, Divide-And-Conquer, and Separable Dependencies are the well-known design patterns I picked for this project. Since vendors have their own designs for OpenMP implementation, we have to choose one. The design and implementation of this project are based on Intel architecture with a special Intel C++ compiler that supports OpenMP on Linux environment.
CHAPTER 1
INTRODUCTION

The use of design patterns has emerged as an effective way to help programmers design high quality software. To be useful, patterns that work together to solve design problems are collected into structured hierarchical catalogs called pattern languages. A pattern language helps guide programmers through the whole process of application design and development. Within the pattern language, one of the design areas is concerned with structuring the algorithm to take advantage of potential concurrency. Patterns in this space describe overall strategies for exploiting concurrency.

Besides the design pattern approach on parallel computation, the improvements on compiler and hardware technology also make parallel execution more convenient. In recent years, a new industry standard of parallel programming for the shared-memory multiprocessor architecture has been developed. It is OpenMP. The “MP” in OpenMP stands for multiprogramming. OpenMP is a specification for a set of compiler directives, libraries, routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. In 1998, hardware and software vendors started designing an open standard on multiprogramming that provides the promise of single source portability for shared-memory parallelism. This standardization became widely accepted by public when the OpenMP Specification Version 2.0 [1] was released in March 2002. OpenMP is a high-level language extension that encapsulates the lower level parallel programming hazard from the programmer. All the optimizations for parallelization and vectorization are handled in the compiler; application developers can
leave the scalability problem to the compiler vendors. As mentioned before, OpenMP is an open standard designed by vendors of computer software and hardware and they all have their own implementation on compiler that supports OpenMP. In Chapter 2, the implementation detail of OpenMP in the Intel Linux C++ Compiler 6.0 [2], and the OpenMP directive for C++ are described. The reusability of OpenMP is also discussed in the same chapter.

The goal of this thesis is to explore the reusability and flexibility of OpenMP in design patterns. Since OpenMP is a parallel library, I attempt to apply OpenMP on four parallel algorithms from Dr. Sanders’ pattern languages: Embarrassingly Parallel, Divide-And-Conquer, Pipeline Processing, and Separable Dependence [3]. Chapter 2 gives background knowledge of OpenMP. It describes the internal of the Intel C++ compiler and the syntax of OpenMP. Many programming examples are given to the reader to understand the basic concept of OpenMP programming. Chapter 3 is the most important for this thesis. It discovers the reusability and flexibility of OpenMP on fitting into design patterns. Chapter 4 briefly reviews the four parallel algorithms and their usages. Making the parallel algorithms and structures reusable for the developer can enhance object-oriented design in software and reduce the development time. To explore the reusability of OpenMP, we try to make some parallel algorithms of this pattern language into a reusable template library with a new parallel programming standard, OpenMP. The users of the library must understand the parallel algorithms so that they use them correctly for their parallel computation needs. The purpose of this template library is to guide users to design parallel solution for their problem without taking care the lower level programming of OpenMP. Chapter 5 describes the design, implementation,
examples, and result analysis of the reusable template library for each parallel algorithm in detail. Chapter 6 talks about the related research in the industry. I hope this thesis can help readers to understand more about the reusability of OpenMP.
CHAPTER 2
BACKGROUND OF OPENMP

The rapid and widespread acceptance of shared-memory multiprocessor architecture has created a pressing demand for an efficient way to program these systems. At the same time, developers of technical and scientific applications in industry and in government laboratories find the need to parallelize huge volumes of code in a portable fashion. The OpenMP Application Program Interface supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architecture, including UNIX platforms and Windows platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer. It consists of a set of compiler directives and library routines that extend Fortran, C, and C++ codes to express shared-memory parallelism.

Compiler Internal

OpenMP’s programming model uses fork-join parallelism: master thread spawns a team of threads as needed. Parallelism is added incrementally. Hence, we do not have to parallelize the whole program at once. OpenMP is usually used to parallelize loops. A user realizes that most his time is spent in loops, so he splits them up between threads. The compiler we used for the implementation is Intel C++ compiler 6.0 for Linux, which supports OpenMP directive for parallelization optimizations. Besides existing
parallelization techniques, additional OpenMP parallelization component is included in the Intel compiler.

The process of code parallelization is as follows:

1) pre-pass that transforms OpenMP parallel sections and worksharing sections into a parallel loop and worksharing loop, respectively.
2) A work-region graph builder that builds region hierarchical graph based on the OpenMP-aware control-flow graph.
3) A loop analysis phase for building the loop structure that consist of loop control variable, loop lower-bound, loop upper-bound, loop-pre-header, loop header, and control expression
4) A variable classification phase that performs analysis of shared and private variables
5) A multithreaded code generator that generate multithread code at the compiler intermediate code level based on Guide, a multithreaded run-time library API that is provided by the Intel KAI Software Laboratory (KSL)
6) A privatizer that performs privatization to handle firstprivate, private, lastprivate, and reduction variable
7) A post-pass that generates code to cache in thread local storage for handling threadprivate variable.

Besides the existing parallelizing optimization techniques, Intel C++ compiler initiated a new compiler technology called Multi-Entry Threading (MET). The rationale
behind MET is that the compiler does not create a separate compilation unit for a parallel region or loop. Instead, the compiler generates a threaded entry and a threaded return for a given parallel region and loop. Based on this idea, three new graph nodes in the Region-based graph are introduced, built on top of the control-flow graph. These graph nodes are T-entry (thread entry), T-ret (thread return), and T-region (thread code region). T-entry indicates the entry point of a multithreaded code region and has a list of `firstprivate`, `lastprivate`, `shared`, and/or `reduction` clause variable for communication among the threads in a team.

T-ret indicates the exit point of a multithreaded code region and guides the lower-level target machine code generator to adjust stack offset properly and give the control to the caller inside the runtime library. T-region represents a multithreaded code region that is attached inside the original user routine.

The T-region represents the OpenMP parallel sections and the second T-region represent the OpenMP parallel loop.
Programming in OpenMP

Programming OpenMP directive in C++ makes parallel programming easier and more productive. The user takes advantage of leaving most of the parallelization and synchronization work to the compiler. In this chapter, the basics of OpenMP program is described with examples. Some simple examples explain some basic technique of programming OpenMP in C++. At the end of this section, a translation of a OpenMP C++ to the pseudo intermediate language will show how the compile translates the OpenMP region to a fork-join program. More details and complex examples of OpenMP will be found in the OpenMP Specification posted in www.openmp.org [1]. The simplified version of the OpenMP Application Program Interface is in Appendix A.

Example 1. The `omp_set_num_threads` method is a runtime library function that allows the user to set the default number of threads to run the subsequent parallel region. OpenMP provides a rich runtime library functions to control the thread creation, lock synchronization, and the operating system environment for execution. Any runtime library function can be either called inside or outside of the parallel region. In this example, the default number of threads is set to four. The “private(z)” is a data-sharing attribute clauses that allow the user to control the sharing attributes of variable for the duration of the parallel region. Different OpenMP directive provides its own data-sharing clauses. In the following program, the computation of z and the call of subprocedure will be executed by four threads in a team in parallel. The number of thread should be explicitly set; otherwise, the default number should be one in most environments.
omp_set_num_threads(4);
#pragma omp parallel private(z)
{
    z = x * y / 2;
    subprocedure(z);
}

Figure 2-3: OpenMP Example 1

**Example 2.** The two parallel loops bind inside the parallel region will be executed in sequence and each loop is independently executed in parallel. The *nowait* clause in the first loop indicates the threads within the team can immediately execute the second loop while some threads in the team may try finishing up the work of first loop. The *nowait* clause avoids the implicit barrier at the end of the first loop directive. OpenMP provides four kinds of scheduling to specify how iteration of the for-loop are divided among threads of the team. One of the kinds is static scheduling that divides the iterations into chunks of a size specified by the chunk size (1 is the size in this case). The chunks are statically assigned to threads in the team in round-robin fashion.

#pragma omp parallel schedule(static,1)
{
    #pragma omp for nowait
    for (i=1; i<s; i++)
        b[i] = (a[i] + a[i+1]) / 8;
    #pragma omp for
    for (i=0; i<t; i++)
        y[i] = subprocedure(z[i]);
}

Figure 2-4: OpenMP Example 2

**Example 3.** The *parallel sections* directive identifies a noniterative work-sharing construct. Each section within the sections construct will be executed once by a thread in the team. The *shared* clause contains an object that is shared among threads within the parallel region. In this example, subprocedure_X and subprocedure_Y will be executed once concurrently and get access to the same storage area of the object *a*. 
Example 4. OpenMP also provide shortcuts to abbreviate the parallel construct and the work-sharing directives by combining them into a single construct. The parallel-for construct can contain a single for directive. The lastprivate clause makes variable z to remain its original value after the parallel region. The private clause declare the variable i to be private for each of the thread in the team.

Example 5. OpenMP supports nested parallelism that allows a parallel directive dynamically inside another parallel directive. For each parallel directive, the compiler will establish a new team of threads for its execution. To benefit from this OpenMP feature, the user must enable the nested parallelism by calling the runtime library function or environment variable. Some rules may restrict dynamic directive nesting and the user should refer to the OpenMP specification for more detail. If two nested for directive bind to the same parallel directive, the incorrect nesting of work-sharing directive is incompliant. The following is a correct example of nested parallelism that each for directive binds to its own parallel directive.
Example 6. Lock plays an important role in data synchronization. OpenMP provides regular lock functions, nested lock functions, and lock variables to ensure data synchronization within the parallel region. In the following example, the `omp_set_lock` function will cause the threads to be idle while waiting for the lock to enter the critical section. The `omp_test_lock` function obtains the lock if it’s available. The difference between the two functions is that the `omp_set_lock` function blocks, but the `omp_test_lock` function does not.

Example 7. In the beginning of this chapter the internal of the Intel C++ compiler is reviewed. The compiler translates the OpenMP parallel region into fork-join parallelism. It is a high-level parallel language that the compiler handles the low-level
parallelization and threads synchronization, and hides the implementation detail from the user. This abstraction makes programming parallel code easier and more productive. In the end of the section, we describe how the Intel C++ compiler translates the OpenMP C++ code into a fork-join program in pseudo intermediate language internally. The following example code is found in Intel Technology Journal Volume 6 Issue 1 (February 2002) Page 4. This example contains two main OpenMP directives, parallel loop and parallel sections. Each of these directive and bind with its own parallel directive.

```c
void parfoo()
{
    int m, y, x[5000];
    int w, z[3000];
    #pragma omp parallel sections shared(w,z,y,x)
    {
        w = floatpoint_foo(z, 3000)
        #pragma omp section
        y = myinteget_goo(x, 5000)
    }
    #pragma omp parallel for private(m) shared(y, z, w) schedule(guided)
    for (m=0; m<3000; m++) {
        z[m] = z[m] * w * y;
    }
}
```

Figure 2-9: OpenMP Example 7

During the code transformation, the `__kmpc_fork_call` is inserted for thread invocation by the multithreaded code generator. This function takes the T-entry point and data environment for the parallel loop, parallel section, and parallel region, and transforms the serial loop, section, or region to a multithreaded loop, sections, or region. In this example, the parallel section is transformed to a parallel loop. Then, the code generator localizes the bounds of the loop, data variable, other runtime initialization, and synchronization code with `__kmpe_static_init` and the `__kmpe_static_fini` functions. T-entry and T-ret mark the entry and exit point of the T-region.
In the second part of the example, the parallel loop in the above OpenMP code is scheduled with type guided. The multithreaded code generator generates a runtime dispatch and initialization function (\texttt{__kmpec\_dispatch\_init}). This function takes similar information for the parallel region and the runtime system. The generator generates an enclosing while loop to dispatch loop-chunk at runtime through the \texttt{__kmpec\_dispatch\_next} function in the library.

```c
R-entry void parfoo( ){
  int m, y, x[5000];
  float w, z[3000];
  __kmpec_fork_call(loc, 4, T-entry(_parfoo_psection_0), &w, z, x, &y)
  goto L1:
  T-entry _parfoo_psection_0(loc, tid, *w, z[], *y, x[]) {
    lower_pid = 0; upper_pid = 1;
    __kmpec_static_init(loc, tid, STATIC, &lower_pid, &upper_pid ...);
    for (pid=lower_pid, pid<=upper_pid; pid++) {
      if (pid == 0) {
        *w = floatpoint_foo(z, 3000);
      } else if (pid == 1)
        *y = myinteger_goo(x, 5000); } // end of for loop
    __kmpec_static_fini(loc, tid);
    T-ret;
  }
L1:
  __kmpec_fork_call(loc, 3, T-entry(_parfoo_ploop_1), &w, z, &y);
  goto L2:
  T-entry _parfoo_ploop_1(loc, tid, *w, z[], *y) {
    lower = 0; upper = 3000;
    __kmpec_dispatch_init(loc, tid, GUIDED, &lower, &upper, ...);
    while (__kmpec_dispatch_next(loc, tid, &lower, &upper, ...)) {
      for (prv_m=lower; prv< upper; prv_m++) {
        z[prv_m] = z[prv_m] * (*w) * (*y);
      }
    } T-ret;
  }
L2:
  R-return; }
```

Figure 2-10: OpenMP Example 8
CHAPTER 3
REUSABILITY OF OPENMP

Code reusability is always an ongoing research topic in computer science. Despite many researches and studies done on this topic, programmers are still writing code that they cannot be able to reuse. The goal of code reusability is to avoid repeating the same or similar code fragment in different place by writing it once again. Of course, the advantage of reusable code is beyond preventing rewriting the same code twice. It is also about efficiency, robustness, flexibility, correctness, clarity, safety, generality, ease of use, and component management.

The reusability of a code fragment is a concern with how well it can be reused in the different program implementation. There is no single way to measure the reusability of a code fragment since there is a variety of factors that determine how well the code fragment is being reused. In many case, one fragment may be more reusable by one factor but less reusable by the others. These situations have trade-offs that the programmer must make a decision of whether using the reusable module.

OpenMP is an extended parallel library in Fortran, C/C++. Some researchers are trying to implement the OpenMP interface in Java language as well. When we talk about the reusability of OpenMP, it is better for us to look at its compatibility with C++ syntax[4] and parallel programming style, and OpenMP syntax. The reusability of OpenMP must trace back to beginning of the development of OpenMP back in 1997. Before a parallel language standard existed, computer venders implemented their own parallel library for their products, and parallel code was not portable from one vendors’
machine to the others. Software and application vendors first wanted to standardize one
multiprogramming language. Therefore, the OpenMP specification for Fortran was first
released in late 1997. Fortran has been the fastest commercial computer language for
arithmetic and scientific computation. The initial idea of creating this new standard was
for scientific application and high performance computation. Afterward, the
development for OpenMP C/C++ API followed the footstep of Fortran original
specification. Unlike C++, Fortran is a structured language for fast scientific and
arithmetic computation. The idea of object-oriented language was not applied to Fortran,
and hence OpenMP did not acquire this paradigm. Therefore, this led to the difficulty to
develop object-oriented component with OpenMP. In this chapter, we discuss the
OpenMP reusability in higher abstraction and then its syntax constraint to C++
programming. The reusability of OpenMP parallel patterns and how well it fits into
parallel patterns and object-oriented programming (OOP) are discussed in Chapter 4.

OpenMP and Other Classical Thread Program

Before getting deeper into reusable issues, the user must know the goal of
OpenMP library. This can help us understand why the designer of OpenMP API
implemented with compiler directive, instead of using the most common technique of
function package and procedure call. Pthreads, MPI, and HPF are the popular open
standard libraries in the parallel computing market for different hardware architectures.
By comparing them, the design purpose of OpenMP will emerge.

Pthreads. Like OpenMP, it runs on shared-memory environment. However,
Pthread have never been targeted toward the technical/high performance computing
(HPC) market. This is reflected in the minimal Fortran support, and its lack of support for
data parallelism. Even for C applications, pthreads requires programming at a level lower than most technical developers would prefer.

**MPI.** Unlike OpenMP, MPI (Message Passing Interface) works on a cluster of machines with separate processes and memory space. Message-passing has become accepted as a portable style of parallel programming, but has several significant weaknesses that limit its effectiveness and scalability. Message-passing in general is difficult to program and does not support incremental parallelization of an existing sequential program. Message-Passing was initially defined for client/server applications running across a network, and so includes costly semantics (including message queuing and selection and the assumption of wholly separate memories) that are often not required by tightly-coded scientific applications running on modern scalable systems with globally addressable and cache coherent distributed memories. The performance of MPI and OpenMP are similar in many benchmark tests.

**HPF.** HPF has never really gained wide acceptance among parallel application developers or hardware vendors. Some applications written in HPF perform well, but others find that limitations resulting from the HPF language itself or the compiler implementations lead to disappointing performance. HPF's focus on data parallelism has also limited its appeal.

Let us compare the parallel programming style of OpenMP, Pthread, and MPI. MPI and Pthreads have a totally different program concept from OpenMP. Like most of the library, they use function calling methods that programmer can invoke any member functions and assign value to library attributes after the package is imported. MPI program is designed for running on a cluster of closely coupled machines. Each machine
in the cluster has a copy of a program and executes it independently. The result passing
and distributed process communication is handled by using MPI library functions. The
MPI programmer has to handle the data partition and task distribution. Since it has no
shared data between processes, copies of data structure and data partition is required for
the distributed execution environment. The partition operation on distributed data or
array can be messy as the scalar of machine in the cluster increase. In some cases, the
program may need modification or repartition for data distribution. The following
pseudo MPI program shows the structure of most MPI program.

OpenMP doesn’t have the data and tasks partitioning difficulty that lead to other
reusability issues. The OpenMP program runs either on a single shared-memory system
or on distributed shared memory architecture that provides a single memory address
shared between tasks. This avoids the multiple copies of process necessary in standard
message passing implementation. Since OpenMP is a high-level language, fork-join
parallelism and thread creations are handled by the compiler that supports OpenMP,
instead of handled by the programmer and OS library.

```c
#include "mpi.h"
#include <stdio.h>

int main(argc, argv) int argc; char **argv;
{
    MPI_Request requests[large];
    MPI_Status statuses[large];
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    if (rank == 0) {
        // rank 0 is main machine that the user currently execute the program.
        // process 0 in machine 0 wait for the result from machine 1 and 2.
        // when receive the results, do some computation with them.
        // send the result to the rest of machine for more computation.
        // Set barrier to wait for all machine to finish computation.
    }
}
```

Figure 3-1: Simple MPI Program
// reduce all results from the rest of the machine.
}

else if(rank == 1) {
    // rank 1 is machine 1 in the cluster.
    // process 1 in machine 1 to this computation.
    // when it’s done, send the result back to machine 0.
}

else if(rank == 2) {
    // rank 2 is machine 2 in the cluster.
    // process 2 in machine 2 to this computation.
    // when it’s done, send the result back to machine 0.
}

else {
    // wait for data from machine 0.
    // when receive the data, do computation.
    // when it’s done, send the result back to machine 0 for reduction.
}

MPI_Finalize();
return 0;
}

Figure 3-1: Simple MPI Program (continue)

OpenMP has one important reusability feature. It uses C++ pragma directive

design that makes the legacy-code or already written programs easily to adopt OpenMP
parallelization. The pragma directives are not taken into account by the compilers which
do not recognize them. The OpenMP directives can be easily added on the existing
sequential program to make rapid and error-free parallelization. Therefore, with these
two factors, the sequential and parallel version of a program is the same. With this
coding mechanism, the idea of this new standard encourages partially and incrementally
parallelism on existing sequential program. MPI is all-or-nothing parallelism that the
program must be parallelized entirely as a whole. OpenMP is different than MPI, it
allows program to parallelize a bit at a time or any place in the serial code. Data-sharing
attribute clauses, like private and reduction clauses, frequently used in parallel program
could be added to enhance the parallel functionality in the sequential program. This
convenience and the reusability of OpenMP do not appear in other classical thread
programming or MPI. The following is an example of parallelizing a sequential program with OpenMP directive and clauses.

```c
#pragma omp parallel for private(i) reduce(+: sum)
// For Parallelization
for(i = 0; i < n; i++) {
    sum = sum + a[i] + b[i];
}
```

Figure 3-2: OpenMP Parallel FOR

However, this easy deployment of OpenMP pragma directives on sequential code has a drawback. Its lack of flexibility syntax that leads up to the idea of OpenMP is more suitable for data reusability and data parallelism than for object oriented program and task parallelism. Unlike a package of function calls, the structure of pragma directives are restricted to certain programming order, which make the program design less object-oriented. For example, the `parallel for` and `parallel sections` directives must be followed by a for-loop statement and section directive respectively. This issue will be reviewed in the later section of OpenMP and C++ syntax. Comparing with other classical parallel programming tool, OpenMP has a trade-off on structure reusability. Unlike Pthreads and MPI, OpenMP is designed for scientific application or regular algorithm computation. The design of OpenMP is specialized for a chunk of sequential algorithm placed inside the parallel region and let the compiler handle the parallelization. It does not provide mechanism for finer granularity of parallelism. For example, in Pthreads, the barriers or the critical sections may concern only some of the threads. This barrier mechanism may be useful for domain decomposition, where the threads need to synchronize only with their neighbors and not with all the threads. This is difficult to achieve in OpenMP where the barrier directives are bound to all the threads in the team. This can be done in MPI and Pthreads.
// synchronization point for only two threads.
if(thread_id ==0 || thread_id == 2){
    barrier(2);
}

Figure 3-3: Pthreads Barrier

**Syntax Constraints**

In the second part of this chapter, we describe how the syntax constraints of OpenMP limit the development of object-oriented program. We discuss the reusability of OpenMP by exploring the compatibility between OpenMP and C++ syntax. As we know C++ is a very successful object-oriented language and most of its existing standard and commercial libraries are highly compatible with C++ in developing OOP. The library usage and code reusability have a direct relation. This section explains in detail about how OpenMP syntax problem limits the reusability and extensibility in C++. The syntax is briefly explained. For more information on OpenMP syntax, please refer to the OpenMP Specification or the Appendix A.

In C++ object-oriented programming, scope rule for data object is stricter than other languages. Unlike Java that create new object with dynamic allocation, C++ allows the creation of local or dynamic allocate object. Either way of object creation, functions take pass-by-reference parameter to mutate the attribute of an object. Also, creating objects and calling member functions are some of the important mechanisms of increasing code reusability in object-oriented programming. To use OpenMP as a parallelization module, passing a reference of an object to a function, where the parallel region that shares the object for computation, can enhance the reusability of OpenMP and encourage parallelizing sequential programs. However, the parameter of all data-shared clause is restricted to variables only, any pointer or references of a data are not accepted.
The data type and the variable cannot be predetermined before compile time. If the parallel region is in a method and the type of the data is not determined, data-shared clause can not be used since the program depends on the input that provides the variable of the shared data. If the method takes a pass-by-value parameter of the shared data, a new copy of the shared data is created and the functionality of the data-shared clause is lost. The following example is not possible since a pointer of integer is passed into the shared clause.

```c
int *i, j = 20;
*i = j;
#pragma omp parallel for shared(j) //Correct! j i s a variable
#pragma omp parallel for shared(i) //Incorrect! i i s a pointer
```

Figure 3-4: Incorrect OpenMP Example For Data Sharing Clause

Reduction is an operation that uses an associative binary operator to combine a set of values into a single result. For example, a reduction using the plus operation will return the summation of the value in the set. The OpenMP reduction clause accepts only hard coded arithmetic operators and no overloaded operator. This restriction will defeat many of the C++ operator-overloading operation between objects. The operator parameter must be hard coded to the reduction clause body. In some cases, the operator for the reduction operation may not be determined before the compile time; therefore, the user is unable to hard code that operator in advance. Developers must implement different version of reduction operations with a specified operator for every individual cases.

```c
#pragma omp parallel for reduce(+ : sum) // Correct
#pragma omp parallel for reduce(- : sub) // Correct
Char op = ‘+’;
// Incorrect: op is a character
#pragma omp parallel for reduce(op : sum)
Object sum;  // Incorrect: operator+ is an overloaded operator in sum
#pragma omp parallel for reduce(operator+ : sum)
```

Figure 3-5: Incorrect OpenMP Reduction Program
Pthreads allows users to dynamically create any number of threads at runtime and execute function blocks as threads independently. However, unlike Pthreads, OpenMP does not allow dynamic creation of thread at runtime. Similarly to Pthreads, the OpenMP *sections* directive is the only way to assign individual task into a thread. However, the number of threads in the *sections* directive must be defined in programming time.

The following figure shows how the *section* directives must be declared and placed inside the *sections* directive. These restrictions make the code less flexible for object-oriented programming and reduce the flexibility of dynamic creation of parallel *sections* directive. Also, the code block of the *section* directive must be implemented; the clients can only either override defined method in the section or pass function pointer into the *section* directive. In most of the programming cases, the number of threads that execute independent tasks are dynamically created at runtime. The *sections* directive code is hardly reused as a module due to its inflexible construct design of *sections* directive.

```c
// Correct structure of SECTIONS directive.
// The SECTION directive must declare inside the SECTIONS directive.
// The work1() to work3() are executed once individually by a team of // thread.
#pragma omp parallel sections
{
    #pragma omp section
    work1();
    #pragma omp section
    work2();
    #pragma omp section
    work3();
}

// Incorrect: section directive is bound by sections directive.
// The section directive must place inside the sections directive.
```

Figure 3-6: Incorrect OpenMP Sections Directive
Figure 3-6: Incorrect OpenMP Sections Directive (continue)

Similar to the syntax problem of sections directive, the for-statement must be placed right after the `parallel for` directive. This OpenMP design creates the same problem as the `sections` directive. The syntax of OpenMP does not allow the for-statement implemented in any other way. This is another syntax inflexibility similar to the `sections` directive above. The following is the example code.

```c
#pragma omp parallel sections
{  
    subprocedure();
}
void subprocedure(){
    #pragma omp section
        work();
}
```

Figure 3-7: Incorrect OpenMP For Directive

Design Patterns

The third part of this chapter discusses the reusability of OpenMP in terms of the design patterns. Let us briefly talk about what a design pattern is. Professor Christopher Alexander, at U.C. Berkeley, said “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to the problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” The Design Patterns book [5] has a more refined
explanation, it said that a design patterns systematically names, motivates, and explains a
general design that addresses a recurring design problem in object-oriented systems. It
describes the problem, the solution, when to apply the solution, and its consequences. It
also gives implementation hints and examples. The solution is a general arrangement of
object and classes that solve the problem. The solution is customized and implemented
to solve the problem in a particular context. The research on design pattern and object-
oriented programming is going on intensively. OpenMP is a stand-alone directive
library that provides parallel region but not a class that can provide abstraction and
interface. When measuring the reusability of OpenMP in term of design pattern, there are
three questions we can ask: what reusable pattern is suitable for the problem, how is the
design pattern fit for the problem as solution or as a part of the solution pattern, and how
the OpenMP directives fit into a particular design pattern. The first question is out of the
scope of this thesis. The programmer must find different pattern solutions for his specific
needs. When OpenMP is used in a design pattern for parallel computation, the API
design of the reusable module using OpenMP and the compatibility of OpenMP and the
component itself will affect the reusability of entire system. The questions of how
suitable OpenMP for parallel computation component and how to design this component
that would fit for the problem as a part of the solution pattern are discussed in this
section.

Most of us understand the concepts of object, interfaces, classes, and inheritance.
The challenge lies in applying those concepts to build flexible and reusable software.
Inheritance and composition are two most common techniques for reusing functionality
in object-oriented programming. Class inheritance lets the client define the
implementation of one class in terms of another’s. We understand the motivation of the reusable model is to provide some parallel patterns functionality. Inheritance may not be suitable for building this reusable library because it is providing parallelization functionality, instead of class properties and object behavior. Class inheritance can cause dependency on the parent class when the subclass is reused. A library module should be a stand-alone entity in the system that has no implementation dependency for other module in the design. Object composition is an alternative to class inheritance. A new functionality can be obtained by assembling or composing objects to get more complex functionality. Most of the software library adopts object composition as the reusable technique. Object composition is a black-box reuse technique that no internal details of object are visible to the client. It keeps each class and module encapsulated and focused on one task. Client reuses the composition object through its well-defined interfaces. As a conclusion, object composition is in favor for building our reusable library for parallel patterns.

Once object composition is picked for the design, we have to think of how to provide the functionality for reusing. Class inheritance provides default implementation for operations and let subclass override them. Object composition provides reuse functionality through parameterized types, also known as templates. Templates are unspecified types supplied as function parameters at the point of use. This generic design gives client more flexibility to reuse library module through its interface with any data type.

OpenMP directives represent a parallel region in the program. As mentioned in the previous chapter, the code must be placed within a function of a module in some
form. Designed as object composition, the user must use the module’s API to pass the object as a template value and its task blocks as function pointers into the parallel region for parallel computation. Therefore, the design of the module’s API has a direct relation of how the OpenMP code is used in the program. Like classical thread programming, a task for a thread is designed as a function block. The advantage is the independence between the programming of the task and the thread. The client can easily make changes on the task without modifying the part of initialization and invocation of the thread.

Unlike other classical threading, OpenMP cannot create any threads, like Pthreads, and process, like MPI. To treat OpenMP parallel region as an individual module in the design pattern, we pass a list of parallel tasks into the module API, so that OpenMP can either use *parallel for* directive or *parallel sections* directive to execute all tasks in parallel. This also gives the same advantage of independence and mobility to the design pattern.

The other concern in design pattern is the viscosity, the degree of stickiness. It comes in two forms: viscosity of the design and viscosity of the environment. The syntax of OpenMP directives has less viscosity to the design pattern. Its user friendly design is used to extend sequential program to parallel program without hacking the design. The design is preserved and less erroneous in either adding or removing OpenMP directives. Since OpenMP is a standard for multiprogramming and many commercial compilers support this directive design, OpenMP code has no machine dependence to any environment. Since it even compiles in a non-OpenMP-supported compiler, the programmers have no need to modify the code and the program is executed sequentially.
A module should be open for extension but closed for modification. This may be the most important principle of object-oriented design. A module containing OpenMP directives has no problem for open extension. OpenMP parallel region is a basic block structure that has a parallel entry point and exit point. It is a totally independent block that has no interference to and from other part of the program. An OpenMP module should not have problem for extension, even if one OpenMP module is nested within another OpenMP module. It’s because OpenMP supports nested parallelism as mentioned in the previous chapter.
Embarrassingly Parallel

This pattern is used to describe concurrent execution by a collection of independent tasks. Parallel Algorithms that use this pattern are called embarrassingly parallel because once the tasks have been defined the potential concurrency is obvious. The challenge is to organize the computation so that all the units of execution finish their work at about the same time. Therefore, the computation load is balanced among processors.

![Diagram of 6 independent tasks assigned to 4 UEs with good load balance]

Figure 4-1: Embarrassingly Parallel

This pattern should automatically and dynamically balance the load as necessary.

With this pattern, faster or less-loaded UEs automatically do more work. When the
amount of work required for each task cannot be predicted ahead of time, this pattern produces a statistically optimal solution.

Embarrassingly Parallel is used when the problem consists of tasks that are known to be independent; that is, there are no data dependencies between tasks. This pattern can be particularly effective when the startup cost for initiating a task is much less than the cost of task itself. It will obtain a good load balance factor when number of tasks is much greater than the number of processor to be used in the parallel computation. Also, the task processing time may vary unpredictably during the runtime but Embarrassingly parallel with OpenMP can easy distribute the workloads equally over threads.

**Divide And Conquer**

This pattern is used for parallel applications based on the well-known divide-and-conquer strategy; concurrency is obtained by solving concurrently the subproblems into which the strategy splits the problem. With this pattern, a problem is solved by splitting it into subproblems, solving them independently, and merging their solutions into a solution for the whole problem. The subproblems can be solved directly, or they can in turn be solved using the same divide-and-conquer strategy, leading to an overall recursive program structure.
Figure 4-2: Divide And Conquer

Divide-And-Conquer strategy is used when the problem can be split recursively into many independent subproblem. The base case can be solved independently and the subsolution can be merged recursively back to one complete result. 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 and when the split produces subproblems of roughly equal size.

**Pipeline Processing**

Pipeline Processing is used for algorithms in which data flows through a sequence of tasks or stages. It represents a “pipelined” form of concurrency. The basic idea of this pattern is much like the idea of an assembly line: To perform a sequence of essentially identical calculations, each which can be broken down into the same sequence of steps, we set up “pipeline,” one stage for each step, with all stages potentially executing concurrently. Each of the sequence of calculations is performed by having the
first stage of the pipeline perform the first step and pass the result from first stage to the next pipeline stage.

![Diagram of Pipeline Processing]

**Figure 4-3: Pipeline Processing**

We use the Pipeline Processing pattern 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, such that for each input the calculations must be done in order, but it is possible to overlap computation of different stages for different inputs as indicated in the figures in the Motivation section. This pattern is particularly effective when the number of calculations is large compared to the number of stages. It is possible to dedicate a processor to each element, or at least each stage, of the pipeline.

**Separable Dependencies**

Separable Dependencies is used for task-based decompositions in which the dependencies between tasks can be eliminated as follows: Necessary global data is replicated for different tasks. The computations are solved locally in the independent tasks. The (partial) results are stored in local data structures. Global results are then obtained by reducing (combining) results from the individual tasks.
In general, task-based algorithms present two distinct challenges to the software designer: allocating the tasks among the processors so the computational load is evenly distributed; and managing the dependencies between tasks so that if multiple tasks update the same data structure, these updates do not interfere with each other. This pattern represents an important class of problem in which these two issues can be separated. In the problem, dependencies can be pulled outside the set of concurrent tasks, allowing the tasks to proceed independently. In a shared-memory environment, it is possible for all tasks that do not modify the global data structure to share a single copy.

Figure 4-4: Separable Dependencies

Separable Dependencies is used when the problem is represented as a collection of concurrent tasks. Dependencies between the tasks must satisfy two restrictions. Only one task modifies the object, and other tasks need only its initial value. The object can thus be replicated. The result from the independent tasks can be combined (reduce) into a single result with a specified operator to become a final global result.
Embarrassingly Parallel

Design

Embarrassingly Parallel is one of the most common parallel patterns used today. There are two kinds of Embarrassingly Parallel APIs for object function base and loop base. The user must define a number of functions with independent tasks inside for object function base API. For the loop base API, a single function that contains a for-loop that does a similar task on each iteration.

There are some restrictions the user must follow:

- The parallel functions must take no argument.
- The return type of the parallel functions must be void type.
- The number specified in the first argument of the run method must be correspond to the same number of function pointers passed as arguments.
- The parallel function pointer must be the member function of the same object that the user passed as the second parameter.
- The “embarrassingly.h” must be included as header file.

Function Parameter

Object base. The following is the API for the object base function:

\[
\text{template<class> void objectBase(const int& methods ...).}
\]

The objectBase method in the template class first takes a number of parallel function point the user will pass as arguments later. Then, the “…” is the unspecified number of arguments that the user can pass as many function pointer into this part of parameter. The function pointer must point to a member function of a same object that instanced the EmbarrassinglyParallel template object.
**Loop base.** The following is the API for the loop base function:

```
template<class> void loopBase(const int & methods, pFuncInt task).
```

The loopBase method in the template class first takes a number of that the function pointer will be called in the parallel region. It would be the number of iterations of the for-loop inside the task function or the number of threads needs to be forked. The second parameter is the function pointer that takes only an integer as the parameter.

**Template Library Implementation**

**Loop base.** The method will first decide how many threads will run in the parallel section. The loop index variable is set to private so that each thread can have a copy of it. The nowait clause means there is no implicit barrier for the threads that can terminate itself when the share of its job is done. The schedule clause is set to static so that each iteration are divided into a specific chunk size. The chucks of task are assigned to threads in a round-robin fashion. Therefore, each thread should have about the same amount of work to achieve the load balance.

**Object base.** The method will first parse the unspecified number of arguments and put the data type into the array. The data type is supposed to be member function pointer of template class T. The integer passed in the first argument must match the number of function pointers passed to this method. Since the function pointer type is not a C++ plain-old-data data type, the array of holding all the function pointer must be initialized with a fixed size. If the user does not explicitly set the number of thread for the parallel section, the default number of thread will be 6.
In the parallel section, the instance \( a \) of the template object \( T \) will be shared among all threads. This design allows the parallel functions to share the same set of global data within object \( a \).

```cpp
#if !defined embarrassing_h
#define embarrassing_h
#include <iostream.h>
#ifndef omp_h
#define omp_h
#include <omp.h>
#endif

template<class T> class EmbarrassinglyParallel
{
public:
    typedef void (T::*pFunc)(void); // type define function pointer
    typedef void (T::*pFuncInt)(int);
    T* a;
    int omp_threads; // number of threads will run
    bool user_set;

    EmbarrassinglyParallel(T* obj)
    {
        a = obj;
        omp_threads = 10;
        user_set = false;
    }

    void set_Thread(int num)
    {
        omp_threads = num;
        user_set = true;
    }

    void objectBase(const int& methods ...);
    void loopBase(const int& methods, pFuncInt task);
};

template<class T>
void EmbarrassinglyParallel<T>::loopBase(const int& methods, pFuncInt task)
{
    int i = 0;
    omp_set_num_threads(omp_threads);
    #pragma omp parallel
    {
        #pragma omp for private(i) nowait schedule(static,1)

        Figure 5-1: Embarrassingly Parallel
```
for(i = 0; i < methods; i++)
{
    (a->*task)(i);
}
}
}

template<class T>
void EmbarrassinglyParallel<T>::objectBase(const int& methods ...)
{
    int omp_threads = 1, i = 0;
    pFunc hold[100];
    va_list ap;
    va_start(ap, methods);
    for(i=0; i<methods; i++){
        hold[i] = va_arg(ap,pFunc);
        if(hold[i] == 0)
            break;
    }
    va_end(ap);
    if(user_set == false){
        if(methods > 10){
            omp_threads = 6;
        }
        else{
            omp_threads = methods;
        }
    }
    cout << "inside embarrassingly\n";
    omp_set_num_threads(omp_threads);
    #pragma omp parallel shared(a)
    {
        cout << "hello\n";
        #pragma omp for private(i) nowait schedule(static,1)
        for(i=0; i < methods; i++){
            cout << omp_get_thread_num() << " ";
            T* pa = a;
            (pa->*hold[i])();
        }
        cout << "done embarrassing\n";
    }
    #endif

Figure 5-1: Embarrassingly Parallel (continue)

Examples

Loop base. Here are two tiny examples to demonstrate how the user can
implement function for the loop-base Embarrassingly Parallel library. Both examples
will do $c[i] = a[i] \times b[i]$ for one thousands iterations. The first example, oneTask, will equally distribute the job into 4 tasks and run within 4 threads. The second example, oneTask_2, will distribute the 1000 iterations into 10 threads in the round-robin fashion.

```cpp
#include <iostream.h>
#include "embarrassing.h"

#define size 1000

class Example_1 {
  private:
    int c[size], a[size], b[size], task;
  public:
    Example_1(){
      int *pc = &c[0], *pa = &a[0], *pb = &b[0], task = 4;
      for(int i=0;i<size;i++){ *pc++ = 0; *pa++ = i; *pb++ = i; }
    }
    void setTask(int n){
      task = n;
    }
    void oneTask(int i){
      int start = i * (size/task);
      int end = i * (size/task) + (size/task);
      for(start; i < end; i++)
        c[i] = a[i] * b[i];
    }
    void oneTask_2(int i){
      c[i] = a[i] * b[i];
    }
};

int main(int argc, char *argv[])
{
  int task = 2, thread = 4;
  Example_1 r;
  EmbarrassinglyParallel<Example_1> elp(&r);
  r.setTask(task);
  elp.set_Thread(thread);
  elp.loopBase(task,&Example_1::oneTask);
  elp.set_Thread(10);
  elp.loopBase(1000,&Example_1::oneTask_2);
  return 0;
}
```

Figure 5-2: Embarrassingly Parallel Loop Base Example

**Object base.** The object-base design is very different from the loop-base. We assume all the four tasks for this example are not similar to one another. Different
functionalities are implemented into four “independent” functions below. In the main method, the object of the template class of EmbarrassinglyParallel is created with the instance of class Ex2. Then, the objectBase method of the EmbarrassinglyParallel class is called with all four independent functions as parameter.

```cpp
#include "embarrassing.h"

class Ex2{
    void independent1(){
        // Independent Job 1
    }
    void independent2(){
        // Independent Job 2
    }
    void independent3(){
        // Independent Job 3
    }
    void independent4(){
        // Independent Job 4
    }
};

int main(int argc, char* argv[])
{
    Ex2 r;
    EmbarrassinglyParallel<Ex2> elp(&r);
    elp.objectBase(4,&Ex2::independent1,&Ex2::independent2,
                   &Ex2::independent3,&Ex2::independent4);
}
```

Figure 5-3: Embarrassingly Parallel Object Base Example

**Pipeline Processing**

**Design**

The message passing between stages in pipeline processing makes the reusability on pipeline parallel pattern low in the OpenMP structure. The straight of OpenMP is work sharing between a team of threads. The work sharing *parallel for* and *parallel sections* constructs is not designed for inter-process communication between processes. Data is difficult to passed between those two OpenMP constructs. The solution here is to use the MPI programming style that define if-and-else statements and thread-ID to divide the work between a team of threads within a sequential program.
In the above example, the number of threads will be set equal to the number of if-statements inside the parallel region. In this case, 3 threads are created for 3 blocks of if-statement. Each thread will be assigned to execute a block of code according to its thread ID. If the number of threads is not specified explicitly to the same number of if statements needed, the master thread may only execute the first block of if statement and, the parallel region can still be compiled and generate incorrect output in the runtime.

In the pipeline processing, the data is passed along the pipeline from one stage to the next. The synchronization and the data concurrency must be handled by the user himself. This design gives the flexibility on how and where the concurrency should apply and allows user to implement any data type as the output data between stages. The OpenMP locks are designed for OpenMP parallel region for data synchronization. A data buffer should be used to hold data temporarily between stages. The access of the buffer must be synchronized between two stages to prevent out-of-order pipeline and incorrect data passing.

**Function Parameter**

The following is the function parameter for Embarrassingly Parallel:

```cpp
template<class T> void PipelineProcessing(int num, T a ...)
```
The PipelineProcessing method here is like the one in EmbarrassingParallel. Both function parameters will take an integer, and a list of unspecified number of argument. The detail was mentioned above.

**Template Library Implementation**

Instead of doing the same way as the EmbarrassinglyParallel, the PipelineProcessing method is not a member function of a template class. It is a stand-alone function in the header file. The function will do about the same thing, as the run method in the EmbarrassingParallel, except the parallel section is different. A parallel for work-sharing directive was used in EmbarrassingParallel. Here, we use the if-and-else statement to parallelize the pipeline stages.

```cpp
#include <iostream.h>

#ifndef omp_h
#define omp_h
#include <omp.h>
#endif

template<class T> void PipelineProcessing(const int& methods, T *a...)
{
    int omp_threads = 1, i = 0, tn = 1;
    typedef void (T::*pFunc)(void);
    pFunc hold[6];

    va_list ap;
    va_start(ap,a);
    for(i = 0; i<methods; i++){
        hold[i] = va_arg(ap,pFunc);
        if(hold[i] == 0)
            break;
    }
    va_end(ap);

    if(methods <= 6){
        omp_threads = methods;
       omp_set_num_threads(omp_threads);
    }
}
```

Figure 5-5: Pipeline Processing
#pragma omp parallel shared(a) private(tn) if(methods <= 6)
{
    tn = omp_get_thread_num();
    switch(tn)
    {
    case 0: (a->*hold[tn])(); break;
    case 1: (a->*hold[tn])(); break;
    case 2: (a->*hold[tn])(); break;
    case 3: (a->*hold[tn])(); break;
    case 4: (a->*hold[tn])(); break;
    case 5: (a->*hold[tn])(); break;
    }
    cout << "done pipeline\n";
}
else
    cerr << "Number of stage is too many\n";
#endif

Figure 5-5: Pipeline Processing (continue)

Examples

This example simulates an out-of-order pipeline that reads a value from a buffer that between the current stage and the previous stage, computes a new result with that value, and puts the result into the buffer that between the current stage and the next stage. All buffers between stages are set to zero initially and the source buffer that the first stage read the data from contain the shared data to the pipeline. One thread is assigned to each stage in the pipeline; therefore, the number of threads and the number stages are identical. Since the thread scheduling is handled dynamically during the runtime, the suspension time and the time slices of a thread are unknown. To preserve the procedure order of each task, the next stage cannot read from the read buffer if the value inside is zero, the user must implement OpenMP locks to synchronize the access to the shared buffer between stages. The task would finish in a different order than they are listed in the source buffer.
```
#include "node.h"

Node::Node()
{
    int* p = &source[0];
    *p++ = 4; *p++ = 5; *p++ = 6; *p++ = 7; *p++ = 8; *p++ = 9;
    *p++ = 10; *p++ = 11; *p++ = 12; *p++ = 13;
    p = &result[0];
    for(i = 0; i<SIZE; i++)
        *p++ = 0;
    p = &buffer1[0];
    for(i = 0; i<SIZE; i++)
        *p++ = 0;
    p = &buffer2[0];
    for(i = 0; i<SIZE; i++)
        *p++ = 0;
    omp_init_lock(&lock);
}

void Node::stage1()
{
    int countA = 0;
    while(countA < 10){
       omp_set_lock(&lock);
        buffer1[countA] = 0;
        buffer1[countA] += source[countA] * 10 + 10 * 100 /100;
        countA++;
        omp_unset_lock(&lock);
    }
}

void Node::stage2()
{
    int countB = 0;
    while(countB < 10){
        omp_set_lock(&lock);
        if(buffer1[countB] != 0){
            buffer2[countB] += buffer1[countB] * 10 + 100;
            countB++;
        }
        omp_unset_lock(&lock);
    }
}

void Node::stage3()
{
    int countC = 0;
    while(countC < 10){
        omp_set_lock(&lock);
        if(buffer2[countC] != 0){
```

Figure 5-6: Pipeline Processing Example
result[countC] += buffer2[countC] / 100;
    countC++;
  }
  omp_unset_lock(&lock);
}

int main()
{
    Node n;
    PipelineProcessing<Node>(3,&n,&Node::stage1,&Node::stage2,&Node::stage3);
}

Figure 5-6: Pipeline Processing Example (continue)

Separable Dependencies

Design

As described in the Section 4.3, Separable Dependencies pattern is similar to
Embarrassingly Parallel, except it has replication and reduction procedures in the
beginning and the end of the pattern respectively. Therefore, the design for this pattern
adds those two procedures on top of Embarrassingly Parallel. It also has two kind of
implementations: loopBase and objectBase.

Function Parameter

Object base. The following function parameter is for the Separable
Independence:

\[
\text{template<class T, class C> void objectBase(const int& methods, T *a ...)}
\]

The objectBase methods in the template class will first take a number of pointer-to-
functions that will be placed in the unspecified number of argument section. The second
parameter is a point to the object that corresponds to the pointer-to-functions. There must
be at least three pointer-to-functions (one replication, one task, and one reduction) passed
into the unspecified number of argument.

Loop base. The following function parameter is for the Separable Independence
object base:
template<class T, class C) void loopBase(const int& methods, T* a, char op, C c, pFunc replicate, pFuncInt tasks, pFuncInt reduce)

The LoopBase method in the template class is very different from the ObjectBase method previously mentioned. All the arguments in the function are presetted. The first integer parameter is the number of iterations of the loop. The second one is the pointer of the object that corresponds to the last three function pointers. The third character argument is the reduction operator that the parallel section will perform and those character can only be one of the following binary operators: “+”, “-”, “*”, “/”, “&&”, and “||”. The fourth argument as Class C is the reduction scalar variable that OpenMP required for its reduction clause. The last three function pointers are for replication, tasks, and reduction.

Template Library Implementation

Object base. First the objectBase method will parse the unspecified number of arguments, if it does not have at least three function pointers and the order of the replication, tasks, and reduction functions are not in the correct position, the objectBase method will simply crash without doing anything. Users can explicitly set the number of thread to execute the parallel region. If the user does not set that number, the default setting of thread is 10. The replication function will be executed once by a single thread in the beginning of the parallel section. The tasks functions will be placed inside a work-sharing parallel for directive for independent executions. Then, an implicit barrier will wait for all task functions to finish before starting the reduction function. The reduction function will be also executed by a single thread at the end of the parallel sections, and that single thread may not be the master thread.

Loop base. The loopBase method is tightly designed for using OpenMP directive and special clause. This design can take the maximum advantage of the OpenMP parallel
loop directive in loop parallel optimization. There are two kinds of function pointers involve:

- **pFunc** – function pointer with void as return type and function parameter.
  ```cpp
typedef void (T::*pFunc)(void);
```

- **pFuncInt** – function pointer with void as return type and one integer argument.
  ```cpp
typedef void (T::*pFuncInt)(int);
```

The replication function is **pFunc** pointer with no argument taken. The task and reduction functions are **pFuncInt** pointer with one integer argument taken. The integer argument is the number of particular iteration in the loop. The user must specify an identical computation in the task function for each loop iteration.

There are two problems when using the reduction clause in OpenMP. First, the reduction clause does not take overloaded operators; therefore, the binary operator sign must be hard coded to it. I have only implemented the parallel section for “+” operation. I spent so much time and could not find another way of doing it. When the user passes the character of a operator, the program will switch a correspondent code for executing a particular operator. Also, the reduction scalar variable cannot be other type but a variable type. The variable reference and variable pointer cannot be used in this field even though I dereferenced the pointer and the reference.

```cpp
#ifdef separable_h
#define separable_h
#include <iostream.h>
#ifndef omp_h
#define omp_h
#include <omp.h>
#endif
#endif

template<class T, class C = int> class SeparableIndependent
{
  typedef void (T::*pFunc)(void);
  typedef void (T::*pFuncInt)(int);
  int omp_threads;
  T *a;
}
```

Figure 5-7: Separable Independence
bool set;
public:
SeparableIndependent(T *tmp): a(tmp), omp_threads(10), set(false){};
void set_Thread(int thread){ omp_threads = thread; set = true; }
void objectBase(const int methods ...);
void loopBase(const int& methods, char op, C c, pFunc replicate,
pFuncInt tasks, pFuncInt reduce);
};

template<class T,class C>
void SeparableIndependent<T,C>::objectBase(const int methods ...)
{
  int i = 0;
  pFunc hold[10];
  va_list ap;
  va_start(ap,methods);
pFunc replicate = (pFunc)(va_arg(ap,pFunc));
  for(i=0; i<methods-2; i++){
    hold[i] = (pFunc)(va_arg(ap,pFunc));
  }
pFunc reduce = (pFunc)(va_arg(ap,pFunc));
  va_end(ap);
  if(set == false)
    omp_threads = methods;
  omp_set_num_threads(omp_threads);
  #pragma omp parallel
  {
    #pragma omp single
    {
      (a->*replicate)(); // replicate
    }
    #pragma omp barrier
    #pragma omp for private(i)
    for(i = 0 ; i < methods-2; i++)
    {
      (a->*hold[i])();// independent tasks
    }
    #pragma omp barrier
    #pragma omp single
    {
      (a->*reduce)(); // reduce
    }
  }
}

template<class T,class C>
void SeparableIndependent<T,C>::loopBase(const int& methods,
char op, C c, pFunc replicate, pFuncInt tasks, pFuncInt reduce)
{
  int i = 0;
  omp_set_num_threads(omp_threads);
  if(op == '+')
Figure 5-7: Separable Independence (continue)
Examples

This following example uses the Separable Independence template library to compute the PI. Since we are running this in a shared memory system, the replication and reduction method is eliminated for performance purpose.

```cpp
#include "problem.h"
#include "separable.h"
#include <iostream.h>

class separableTest
{
  public:
    int task, iterations;
    double w, sum, pi, f, a;
    double PI25DT;

    separableTest(int n, int t)
    {
      iterations = n;
      task = t;
      pi = 0.0;
      PI25DT = 3.141592653589793238462643;
      w = 1.0 / (double) n;
    }
    void replicate(){};
    void tasks(int i)
    {
      double x = w * (((double) i) - 0.5);
    }
};
```

Figure 5-8: Separable Independence Example
sum = sum + (4.0 / (1.0 + x * x));
}
void reduce(int task)
{
}
);

int main(int argc, char *argv[])
{
    int n = atoi(argv[1]);
    int t = atoi(argv[2]);
    double start, end;

    separableTest st(n,t);
    SeparableIndependent<separableTest,double> spit(&st);
    start = omp_get_wtime();
    spit.loopBase(n,'+',st.sum,&separableTest::replicate,
        &separableTest::tasks, &separableTest::reduce);
    end = omp_get_wtime();
    st.pi = st.sum * st.w;
    cout << st.pi << "parallel " << (end - start) << endl;
}

Figure 5-8: Separable Independence Example (continue)

Divide And Conquer (DAC)

Design

The merge sort is the most well-known divide-and-conquer algorithm. It simply explains the basic idea of the divide-and-conquer strategy on solving a variety of problems in many different science fields. How to split the problem into sub problems and when to stop splitting recursively are the two essential questions before the user can apply the divide-and-conquer strategy to any problem. Most of the divide-and-conquer algorithm use either two-way or three-way strategy to divide the problem to solve two or three independent sub problems. The implementation here will only focus on two-way split strategy like merge sort.

The divide-and-conquer is the hardest parallel pattern for making reusable framework as a general solution for all existing algorithms. The problem and the algorithm of DAC are closely related and it’s difficult to separate the problem from the
algorithm. The idea of this reusable pattern for DAC is different from the previous design that only requires passing parameters of functions pointers. The user must override the following virtual functions that predefined by the library itself:

Table 5-1: Override Functions for Divide And Conquer

<table>
<thead>
<tr>
<th>Function to Implement:</th>
<th>Design Issue:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;class T, class C, class D&gt; bool condition(C first, C last, D* other);</code></td>
<td>The condition function will return a Boolean value that determines the termination of the recursive call.</td>
</tr>
<tr>
<td><code>template&lt;class T, class C, class D&gt; template&lt;class C&gt; splitLeft(C first, C last, D* other);</code></td>
<td>This split function will return the left margin of the first sub problem of the data structure.</td>
</tr>
<tr>
<td><code>template&lt;class T, class C, class D&gt; template&lt;class C&gt; splitRight(C first, C last, D* other);</code></td>
<td>This split function will return the right margin of the second sub problem of the data structure.</td>
</tr>
<tr>
<td><code>template&lt;class T, class C, class D&gt; void merge(C first, C middle_L, C middle_R, C last, D* other);</code></td>
<td>The merge function will merge all the previous results from the returns of the two-way recursion. All the computation work will be done here.</td>
</tr>
</tbody>
</table>

**Function Parameter**

All shared data used by the algorithm should be stored in either array or linked list. The way to split the sub problem is based on the location of the data listed in the data structure. A set of data will be divided into two subsets. The parameter “first” of type C points to the first data of the list and so on. The last parameter defined as a pointer of type D is a user define type in case there is more information needed in the algorithm.

```
bool condition(C first, C last, D* other);
template<class C> splitLeft(C first, C last, D* other);
template<class C> splitLeft(C first, C last, D* other);
void merge(C first, C middle_L, C middle_R, C last, D* other);
```

Figure 5-9: Divide-And-Conquer Override Functions

```
<table>
<thead>
<tr>
<th>First</th>
<th>middle_L</th>
<th>middle_R</th>
<th>last</th>
</tr>
</thead>
</table>
```

Figure 5-10: Array or Linked List of data type C
Template Library Implementation

The constructor of DivideAndConquer will set the number of threads and the number of thread for nested parallel to 10 as default. The scheduling of the parallel execution set to dynamic and handle in the runtime. The user can explicitly set the number of thread with the set_Thread method. In the divideConquer method, first, the condition for further recursive split is checked with the condition method provided by the user. The divideConquer method will stop recursive calling when the condition method returns false. Otherwise, the data will be split for two sub problems and the split methods return two variable of type C, which is usually an index of an array or a position in a linked list.

After data is split into two subsets, the divideConquer will recursively call itself in the parallel region with the parallel sections directive of OpenMP. Each block of parallel section will be assigned a thread to run it exactly once. Since the threads are dynamically assign to parallel sections, there is no document that tells how the compiler will interpret the nested parallelism in OpenMP.

DAC library takes the advantage of OpenMP nested parallelism here. Many computation problems have an outer-level of coarse grained parallelism, where the number of tasks is few, but where each task contains a large amount of work. Each such out-level task might itself be a parallel task of more fine grained parallelism DAC takes the advantage of OpenMP nested parallelism. Problem like this invites to use of multi-level parallelism, or nested parallelism. The two-way recursive nested parallelism can speed up some DAC problem that requires shallow trace of recursion. Otherwise, the performance degrades as the recursion goes deeper.
Figure 5-11: Divide And Conquer

```cpp
#include <iostream.h>

#define divconq_h
#endif

#define omp_h
#endif

template<class T, class C = int, class D = int> class DivideAndConquer
{
  T* root;
  C* index;
  D* other;
public:
  DivideAndConquer(T *a): root(a) {
    omp_set_num_threads(10);
    omp_set_dynamic(0);
    omp_set_nested(10);
  };
  void set_Thread(int thread) {
    omp_set_num_threads(thread);
    omp_set_nested(thread);
  }
  void divideConquer(C first, C last, D* other) {
  }
};

template<class T, class C, class D>
void DivideAndConquer<T,C,D>::divideConquer(C first, C last, D* other) {
  if(root->condition(first, last, other)) {
    return;
  } else {
    C middle_L = root->splitLeft(first, last, other);
    C middle_R = root->splitRight(first, last, other);
    #pragma omp parallel sections
    {
      #pragma omp section
      {
        divideConquer(first, middle_L, other);
      }
      #pragma omp section
      {
        divideConquer(middle_R, last, other);
      }
    }
    root->merge(first, middle_L, middle_R, last, other);
  }
}
#endif
```
Example

The following example for divide-and-conquer is a simple merge sort algorithm, which sort a sequence of integers from 0 to 9999. To use the DAC library, user must implement four predefined functions used by the library. The condition function first takes the left and right bound as parameter and check if the recursion gets to its base condition to stop further recursion. It will return true for further recursive call or false to stop recursion. The splitLeft and splitRight functions do similar task to find the two inclusive points in the shared data structure for further recursive call if the condition function returns false. The data type of these points are either primitive data type or user define type. The merge function does all the actual necessary work when the recursive calls return. The main method shows how to use the DAC library with the class of mergesort.

```cpp
#include "divconq.h"

class mergesort
{
public:
    int length;
    int a[10000];

    // initialize the elements in the array first in constructor
    mergesort()
    {
        length = 10000;
        for(int i = 9999; i>=0; i--)
            a[i] = i;
    }

    // this function is to check the recursive condition
    bool condition(int low, int high, int* non)
    {
        cout << "condition " << low << " " << high << endl;
        if(low == high) return true;
        else return false;
    }
};
```

Figure 5-12: Divide-And-Conquer Example
// this function is for the left side split
int splitLeft(int low, int high, int* non)
{
    return (low + high) / 2;
}

// this function is for the right side split
int splitRight(int low, int high, int* non)
{
    return ((low + high) / 2) + 1;
}

// this function help to merge the left and right side partition
void merge(int low, int pivot, int pivot_r, int high, int* non)
{
    int length = high - low + 1;
    cout << "merge " << low << " " << pivot << " " << pivot_r << " " << high << endl;
    int working[length];
    for(int i = 0; i < length; i++)
        working[i] = a[low+i];

    int m1 = 0;
    int m2 = pivot - low + 1;
    for(int i=0; i<length; i++)
    {
        if(m2 <= high-low)
            if(m1 <= pivot-low)
                if(working[m1] > working[m2])
                    a[i+low] = working[m2++];
                else
                    a[i+low] = working[m1++];
            else
                a[i+low] = working[m2++];
        else
            a[i+low] = working[m1++];
    }
}

int main()
{
    int non = 0;
    mergesort mgs;
    DivideAndConquer<mergesort, int> dac(&mgs);
    dac.divideConquer(0, 9999, &non);
}
CHAPTER 6
FINDING AND ANALYSIS

Shared-memory system for high performance computing is not a new idea. However, in the past decade, the distributed processing system and the message-passing paradigm is gaining more popularity in the world of parallel computing. Hundred of parallel libraries are built on message-passing technique for distributed systems. In contrast, Parallelization application on shared memory architecture is not as popular since special architecture hardware is required.

Why Is It Not Popular Yet?

OpenMP was developed under many hardware vender supports. The first release of OpenMP Fortran was in late 1997. Even this multiprogramming standard is approved by software venders, many company did not take advantage of the standardization to implement this new parallel feature in their compiler. A few companies just added OpenMP in their Fortran compiler. Even the OpenMP C/C++ Specification was released in October 1998, not until recently, Intel and Sun Microsystems added OpenMP feature into the C/C++ compiler Version 5.2 and Version 6.0, and Sun ONE Studio 7 Compiler Collection, respectively.

The reason why the computer industry is not very interested in OpenMP is shared-memory system for parallel computing is not as flexible as message passing parallelism. Building a cluster of desktop computers is much affordable than purchasing one multiprocessors machine. A cluster of computers is more favorable for parallel computation to most of the academic and scientific institutes. OpenMP is designed for
shared memory system. Its programs can either run on a single multiprocessor machine or a closely coupled network of machines with a virtual shared-memory operating system. Many performance benchmarks show the speed of MPI faster than OpenMP in a cluster of machine. Even the difference of their performances is not significant, about 15%; OpenMP only limits itself to a set of geographically nearby machines. MPI program can run on a distributed system. OpenMP cannot take the advantage of the global distributed network since the shared-memory mechanism limits the distance of machines for sharing the same memory space.

**Basic Performance Analysis**

Before getting deeper on the analysis of the pattern language, let us examine the performance of OpenMP on the dual processor machine provided by Dr. Sanders. Before having the dual processors machine, I was provided a single processor machine that has a Pentium II 355 MHz processor and 128MB RAM. All the OpenMP test programs preformed much poorer than the same version of sequential program. A single Pentium II processor machine cannot take the advantage of the speedup by adding OpenMP directive into a sequential program. Dr. Sanders understood the situation. A new machine was used for testing and analysis. This machine is built by DELL and has basic features of two Pentium II 365MHz processors, 256MB RAM, and one SCIS hard disk. Since Intel is one of a few company that provide a C/C++ compiler with OpenMP support and the dual processors machine I have is i836 architecture, I decided to install the free version of Intel C/C++ compiler 6.0 for RedHat Linux 7.1 or 7.2.

Before examining any code for testing, we have to understand the strength of OpenMP. It allows us to parallelize a sequential program simply by adding pragma
directives. The most powerful feature is loop parallelization and data parallelism.

OpenMP is not good at task parallelism since it does not provide any synchronization and condition function. The most simple loop parallelization test I could think of is PI calculation. The following is the code I wrote for the sequential and parallel version of the PI computation.

```c
#include <iostream.h>
#include <math.h>
#include <omp.h>

int main(int argc, char* argv[]) {  
    double w, x, sum, pi, f, a;  
    double PI25DT = 3.141592653589793238462643;

    int i = 0, n = 1000000000;  
    cout << "Enter number of intervals: ";  
    cin >> n;

    double start = omp_get_wtime(); // Get the Start Time  
    w = 1.0 / (double) n;  
    sum = 0.0;

    if(atoi(argv[1]) == 1) // Parallel Version of PI  
    {  
        int thread = 1;  
        if(argc > 2)  
        {  
            thread = atoi(argv[2]);  
        }  
        omp_set_num_threads(thread); // Set Number of Thread  
        #pragma omp parallel for private(x,i) shared(w) reduction(+:sum)  
        for(i=0; i<n;i++)  
        {  
            x = w * (((double) i) - 0.5);  
            sum = sum + (4.0 / (1.0 + x * x));  
        }
    }
    else // Sequential Version of PI  
    {  
        for(i=0; i<n;i++)  
        {  
            x = w * (((double) i) - 0.5);  
            sum = sum + (4.0 / (1.0 + x * x));  
        }
    }  
    pi = w * sum;  
    cout << pi << endl;  
    double end = omp_get_wtime(); // Get the Finish Time  
    printf("computed pi = %.24f",pi);  
    printf("\nError is %.24f \n", fabs(pi - PI25DT));  
    cout << "time: " << end-start << endl;
}
```

Figure 6-1: PI Calculation
OpenMP loop parallelization divides the loop into chunks of work and assigns them into a team of threads. The overheads of the thread creation and of switching thread during execution are high. Therefore, the small number of iterations cannot overcome this disadvantage. With a big number of iteration, the OpenMP can show its speedup in multiprocessor machine. The following chart shows the performances in computing the PI with different number of iterations.

![Figure 6-2: 10,000 iterations](image)

Figure 6-1 shows that OpenMP does not speedup the PI calculation with 10,000 iterations in any number of threads. The process executed with one thread means the process is a sequential program. Therefore, we always compare the speed between the parallel version and the sequential version. The OpenMP does not overcome the overhead of thread creation in a small number of iterations.
As we can see from Figure 6-2 and 6-3, the large number of iterations makes better performance and also overcome the overhead disadvantages. These two charts also show that the processes perform the best with 2-thread execution. As the number of
thread increase, the overhead problem worsens the speedup performance. This may have relation with the number of processor in the machine. However, I could not do farther testing on this issue since I had no access to other multiprocessor machine. My assumption is that the user should not set the number of threads for parallel loop execution and let the machine determine the default number of thread in runtime.

**Performance Analysis on Parallel Patterns Template Library**

The goal of this thesis is to discover the reusability of OpenMP through implementing the template library for four parallel patterns. As I mentioned in Chapter 3, Reusability of OpenMP, OpenMP is designed for data parallelism and not for task parallelism. The parallel patterns we picked for this experiment is for task parallelism.

**Embarrassingly Parallel.** This pattern is good for task parallelism and data parallelism. I implemented two different versions for loop base and object base. The object base implementation is absolutely not practical since it cannot take the advantage of OpenMP loop parallelization. The performance is definitely slower than the sequential version of the program. On the other hand, the loop base design takes the advantage of OpenMP loop parallelization and speedup the performance up to about 15%. I found out that the default number of thread determined by the runtime environment makes the optimal number of thread for the parallel execution. As a result, I suggest user not to set the OpenMP thread number. The following is the performance analysis of example program from Chapter 5. I tried to test the program with different number of threads and different number of tasks.
The performance is better with 2-Tasks than 3-Tasks. In these first two figures, the format legend shows “2(3),” which means the program is divided into two tasks and running with 3 threads. For example, 2-Tasks finished the 900,000 iterations in 0.071 second and 3-Tasks in about 0.097 second. Fortunately, OpenMP template library is faster than the sequential program in both experiments. In both case, OpenMP is about 30% faster than sequential program. Also, the more the number of thread, the worse the performance we have.
Figure 6-6: Embarrassingly Parallel Performance with 3-Tasks

**Pipeline Processing.** It is very difficult to simulate pipeline processing with OpenMP. As I mentioned in Chapter 5, we can only use the *sections* directive to assign each pipeline stage to a thread. Unfortunately, OpenMP does not provide synchronization and condition functions. OpenMP is not design for blocking concurrency that suspends a certain thread in a team while allowing other thread in the same team to keep running. If the user needs to synchronize the pipeline, he must implement the condition method or semaphore with OpenMP lock variables. I should say pipeline processing is another idea of task parallelism that breaks down sequential computation into distinct stages. Each stage performs a particular task or access certain set of data. OpenMP is not designed for breaking computation into pieces of tasks. Even we try to use OpenMP *sections* directive like classical thread programming to define the stages, the performance of the parallel pipeline patterns is not any better than the
sequential version of the computation. This is because sections directive is just a feature in OpenMP but it does not provide as good performance as loop parallelization.

**Separable Independence.** In a distributed processing system, like MPI, it will be reasonable to make replica of the data for each individual process. However, duplicate of the data for each thread in shared memory system can only cause inefficient use of computer resource. Using OpenMP, we can eliminate the data replication step. In the example program shown in Chapter 5, the task procedure and the reduction procedure is combined into only one method. This is because OpenMP encourage block parallelization that we put all the code in one parallel region.

In the basic performance analysis, we test the performance of the dual processor machine with the PI calculation program. The speedup of the parallel version is almost 100% faster than the sequential version. However, when I tried to use the template library for this computation, the performance is not as good as the inline OpenMP version. The only explanation for this phenomenon is inline OpenMP version does not need to call the task function. The entire calling and referencing procedures will degrade the program performance a big time. The inline OpenMP version allows the compiler optimize the program better. The following analysis show the library is about 17% faster than the sequential version and 30% slower than the inline OpenMP version.
**Divide And Conquer.** This pattern is difficult to implement a reusable template library. As I describe the design issue in Chapter 5, the only way to implement this template library is to override some predefined functions and pass those functions into the library API. However, the Divide-And-Conquer algorithm is perfect for data parallelization. The problem is split into two smaller subproblems but they still shared the same set of data. OpenMP supports nested parallelism that makes this recursive divide-and-conquer algorithm possible. The number of thread for parallel execution should be chosen by the runtime environment itself, which is the default value. Using a user-specified number of threads, the performance may not be as optimal as the default number chosen by the runtime environment. Figure shows the performance between using the Divide-And-Conquer template library and the sequential program of merge sort. As the number of element increase, the performance of the template library gets better gradually than the sequential program.
Figure 6-8: Divide And Conquer Performance
CHAPTER 7
RELATED WORK

Many parallel libraries, tools, and utilities are developed to be integrated with user programs or used as standalone parallel problem solving systems. Some ambitious projects cover a very broad spectrum of problems. Most of them are developed by universities and research institutions for scientific application and simulation purposes and built for distributed cluster architecture. The message passing technique is mostly used to build distributed parallel library. The degree of openness and integration vary in different library system. This chapter talks about some successful products of commercial and open-source parallel libraries.

NAG. Numerical Algorithms Group has developed a commercial MPI Parallel Library. This is one of a few software vendors that produce commercial parallel solutions. The library contains 183 routines that have been specifically developed for use on distributed memory systems and clusters of workstations and PCs. The interfaces are kept as close as possible to the Fortran Library routines to ensure smoother integration and the user does not, in general, need knowledge of MPI. The library is structured to hide the detail of message passing. This library allows the user to make use of the performance of truly parallel machines or networks of workstations behaving as if they were a single parallel machine. It offers greater speed of execution over conventional sequential numerical software and, particularly on networks of workstations, allows problems to be solved, which may be beyond the memory capacity of a single machine. It makes use of a logical grid of processors, which are then
allocated, to available physical processors. Subsequent calls to Library routines execute on each logical processor and cooperate to solve the problem.

**P-Suite.** It is the latest open-source parallel computing tool that built and maintained by thousands of developers in the open-source community. P-Suite is a collection of scientific programs that run in parallel using the Message Passing Interface standard. They all solve common (and not so common) computing-intensive problems found in many fields of science (including computer science!), such as fractal rendering, N-Body problems, cryptanalysis, and so on. All P-Suite programs use the P-Suite lib, which is a framework for developing parallel MPI applications.

**Globus.** It is the most successful distributed parallel computing tool has ever built. The Globus project is developing the fundamental technology that is needed to build computational grids, execution environments that enable an application to integrate geographically-distributed instruments, displays, and computational and information resources. Such computations may link tens or hundreds of these resources. The Globus Toolkit is the software tools and services necessary to build a computational grid infrastructure and to develop applications that can exploit the advanced capabilities of the grid. Using the basic services provided by the toolkit, researchers may build a range of higher-level capabilities. For example, Globus provides a complete implementation of the Message Passing Interface that can run across heterogeneous collections of computers.

**KAP/Pro Toolset.** The Intel KAP/Pro Toolset for OpenMP combines a complete OpenMP implementation with unique supporting development tools to make it easy to add parallel threading to existing software. OpenMP is the industry standard approach to
shared-memory parallelism for compute-intensive applications, and KAI is leading the way with the industry's most complete OpenMP development solution.

**ScaLAPACK.** Scalable Computing Laboratory at US Department of Energy, ScaLAPACK is a library of parallelized linear algebra routines, which operates on clusters using PVM or MPI. ScaLAPACK requires an installation of the LAPACK linear algebra routines and the BLACS library for communication in linear algebra programs. These separate pieces take a bit of work to configure and install (pre-built libraries are available for a few platforms), but ScaLAPACK could save a lot of time and effort if it helps avoid rewriting old code or writing new parallelized code.

**PAQMSG.** PAQMSG is an MPI-based communication library for the parallelization of air quality models on structured grids. It consists of distribution, gathering and repartitioning routines for XY and HV domain decomposition implementing a master-worker strategy. The library is architecture and application independent and includes optimization strategies for different types of architectures.
APPENDIX
OPENMP C AND C++ PROGRAMMING INTERFACE

Directives

Directives are based on #pragma directives defined in the C and C++ standards.

Compilers that support the OpenMP C and C++ API will include a command-line option that activates and allows interpretation of all OpenMP compiler directives.

```
#pragma omp directive-name [clause[,... clause]] new-line
```

Figure A-1: Syntax of an OpenMP directive [1]

<table>
<thead>
<tr>
<th>#pragma omp parallel</th>
<th>defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel. This is the fundamental construct that starts parallel execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp for</td>
<td>defines an iterative working-sharing construct that specifies that iteration of the associated loop will be executed in parallel. The iterations of the for loop are distributed across threads that already exist in the team executing the parallel construct to which it binds.</td>
</tr>
<tr>
<td>#pragma omp sections</td>
<td>defines a non-iterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team.</td>
</tr>
<tr>
<td>#pragma omp single</td>
<td>define a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread).</td>
</tr>
<tr>
<td>#pragma omp parallel for</td>
<td>define a shortcut for parallel region that contains only a single for directive.</td>
</tr>
<tr>
<td>#pragma omp parallel sections</td>
<td>define a shortcut form for specifying a parallel region containing only a single sections directive. The semantics are identical to explicitly specifying a parallel directive immediately followed by a sections directive.</td>
</tr>
<tr>
<td>#pragma omp master</td>
<td>define a construct that specifies a structured block that is executed by the master thread of the team. Other</td>
</tr>
</tbody>
</table>

Table A-1: Constructs [1]
threads in the team do not execute the associated structured block. There is no implied barrier either on entry to or exit from the construct.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp critical</td>
<td>defines a construct that restricts execution of the associated structured block to a single thread at a time.</td>
</tr>
<tr>
<td>#pragma omp barrier</td>
<td>synchronizes all the threads in a team. When encountered, each thread in the team waits until all of the others have reached this point. After all threads in the team have encountered the barrier, each thread in the team begins executing the statements after this directive.</td>
</tr>
<tr>
<td>#pragma omp atomic</td>
<td>The atomic directive ensures that a specific memory location is updated atomically, rather than exposing the possibility of multiple, simultaneous writing threads.</td>
</tr>
<tr>
<td>#pragma omp flush</td>
<td>specifies a “cross-thread” sequence point at which the implementation is required to ensure that all threads in a team have a consistent view of certain objects (specified below) in memory. This means that previous evaluations of expression that reference those objects are complete and subsequent evaluations have not yet begun. For example, compilers must restore the values of the object from registers to memory, and hardware may need to flush write buffers to memory and reload the values of the object from memory.</td>
</tr>
<tr>
<td>#pragma omp order</td>
<td>this directive must be within the dynamic extent of a for or parallel for construct. The for or parallel for directive to which the ordered construct binds must have an ordered clause specified. The ordered constructs are executed strictly in the order in which they would be executed in a sequential execution of the loop.</td>
</tr>
<tr>
<td>#pragma omp threadprivate</td>
<td>this directive makes the named file-scope, namespace-scope, or static block-scope variable specified in the variable-list private to a thread. The variable-list is a common-separated list of variable that do not have an incomplete type.</td>
</tr>
</tbody>
</table>

**Data-Sharing Attribute Clauses**

Several directives accept clauses that allow a user to control the sharing attributes of variable for the duration of the parallel region. Sharing attribute clauses apply only to variables in the lexical extent of the directive on which the clause appears. Not all of the
following clauses are allowed on all directives. The list of clauses that are valid on a particular directive are described in the OpenMP Specification in detail.

Table A-2: Data-Sharing Attribute Clauses [1]

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>declares the variables in variable-list to be private to each thread in a team.</td>
</tr>
<tr>
<td>Firstprivate</td>
<td>this clause provides a superset of the functionality provided by the private clause. For this clause on a work-sharing construct, the initial value of the new private object for each thread that executes the work-sharing construct is the value of the original object that exists prior to the point in time that the same thread encounters the work-sharing construct.</td>
</tr>
<tr>
<td>Lastprivate</td>
<td>this clause provides a superset of the functionality provided by the private clause. For this clause on a work-sharing construct, the value of each lastprivate variable from the sequentially last iteration of the associated loop, or the lexically last section directive, is assigned to the variable’s original object.</td>
</tr>
<tr>
<td>Shared</td>
<td>this clause shares variables that appear in the variable-list among all the threads in a team. All threads within a team access the same storage area for shared variable. Default – it allows the user to affect the data-sharing attributes of variables. The default behavior is the same as default(shared) were specified.</td>
</tr>
<tr>
<td>reduction</td>
<td>performs a reduction on the scalar variable that appear in variable-list, with the operator op, like reduction(op : variable).</td>
</tr>
<tr>
<td>Copyin</td>
<td>this clause provides a mechanism to assign the same value to threadprivate variables for each thread in the team executing the parallel region. For each variable specified in a copyin clause, the value of the variable in the master thread of the team is copied, as if by assignment, to the thread-private copies at the beginning of the parallel region.</td>
</tr>
<tr>
<td>Copyprivate</td>
<td>this clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It is an alternative to using a shared variable for a value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The copyprivate clause can only appear on the single directive.</td>
</tr>
</tbody>
</table>

### Run-time Library Functions

This section describes the OpenMP C and C++ run-time library functions. The header `<omp.h>` declares two types, several functions that can be used to control and
query the parallel execution environment, and lock functions that can be used to synchronize access to data.

Table A-3: Run-time Library Functions [1]

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void omp_set_num_threads(int)</td>
<td>sets the default number of threads to use for subsequent parallel regions that do not specify a num_threads clause.</td>
</tr>
<tr>
<td>int omp_get_num_threads(void)</td>
<td>returns the number of threads currently in the team executing the parallel region from which it is called.</td>
</tr>
<tr>
<td>int omp_get_max_threads(void)</td>
<td>returns an integer that is guaranteed to be at least as large as the number of threads that would be used to form a team if a parallel region without a num_threads clause were to be encountered at that point in the code.</td>
</tr>
<tr>
<td>int omp_get_thread_num(void)</td>
<td>returns the thread number, within its team of the thread executing the function. The thread number lies between 0 and omp_get_num_threads() – 1, inclusive. The master thread of the team is thread 0.</td>
</tr>
<tr>
<td>int omp_get_num_procs(void)</td>
<td>returns the number of processors that are available to the program at the time the function is called.</td>
</tr>
<tr>
<td>int omp_in_parallel(void)</td>
<td>returns a non-zero value if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0.</td>
</tr>
<tr>
<td>void omp_set_dynamic(int)</td>
<td>it enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.</td>
</tr>
<tr>
<td>int omp_get_dynamic(void)</td>
<td>returns a non-zero value if dynamic adjustment of threads is enabled, and returns 0 otherwise.</td>
</tr>
<tr>
<td>void omp_set_nested(int)</td>
<td>it enables or disables nested parallelism.</td>
</tr>
<tr>
<td>int omp_get_nested(void)</td>
<td>returns a non-zero value if nested parallelism is enabled and 0 if it is disabled.</td>
</tr>
</tbody>
</table>
LIST OF REFERENCES


BIOGRAPHICAL SKETCH

Chi-Kin Wong was born in Hong Kong. He grew up there and came to the United States of America for higher education after finishing high school. He joined the Department of Computer and Information Science and Engineering at the University of Florida in the spring 1999. He received his bachelor’s degree in computer science in May 2001. Afterward, he continued his graduate study at the University of Florida. He received his Master of Engineering degree in December 2002. He was the treasurer and vice-president of the UF Badminton Club for three years and a member of the UF Hong Kong Student Association. His research interests include design patterns, parallel computing, and compiler.