EXPLORING SHARED MEMORY AND HYBRID PARALLELIZATION STRATEGIES FOR IMAGE PROCESSING

A Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Vishwanath Venkatesan
August 2008
EXPLORING SHARED MEMORY AND
HYBRID PARALLELIZATION STRATEGIES
FOR IMAGE PROCESSING

Vishwanath Venkatesan

APPROVED:

Dr. Edgar Gabriel
Dept. of Computer Science, University of Houston

Dr. Shishir Shah
Dept. of Computer Science, University of Houston

Dr. Stefan Birmanns
School of Health Information Sciences, UTHSC

Dean, College of Natural Sciences and Mathematics
Acknowledgments

My sincere thanks to Dr. Edgar Gabriel, for his untiring support and assistance throughout this endeavor. I would also like to thank Dr. Shishir Shah, for providing the image segmentation application. My special appreciation to Dr. Oscar Hernandez, for helping me with the profiling of the *OpenMP* parallel codes. Finally, my earnest gratitude goes to my parents and my brother for their constant support and encouragement in achieving this target.
EXPLORING SHARED MEMORY AND HYBRID PARALLELIZATION STRATEGIES FOR IMAGE PROCESSING

An Abstract of a Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Vishwanath Venkatesan
August 2008
Abstract

The introduction of multi-core architectures has lead shared memory programming to mainstream parallel computing for scientific applications. Since its introduction OpenMP has been a wide spread paradigm for shared memory programming. With its inherent compatibility to other parallel programming paradigms, OpenMP facilitates easier use of the hybrid parallelization for clustered multi-core platforms.

This thesis evaluates parallelization strategies for image processing applications. Among the investigated applications are: a texture-based segmentation code for thyroid Fine Needle Aspiration Cytology (FNAC) images, a Partial Differential Equation (PDE)-based application for time independent problems and a PDE solver for time-dependent problems. The necessity for parallelizing image processing algorithm stems from the fact that the resolution of images has increased over the years and the computational complexity of image processing algorithms is high. These applications were parallelized with OpenMP for shared memory architecture and using a hybrid parallelization approach using MPI and OpenMP for cluster of SMP’s.

All OpenMP parallelized applications were evaluated on an eight processor dual-core Opteron system. The hybrid applications were evaluated on a 24 node dual-core AMD Opteron cluster connected by a 4x InfiniBand network interconnect. In most cases the parallelization resulted in good performance and scalability. Furthermore, the thesis presents the experiences in parallelizing the applications with both parallelization strategies.
Contents

1 Introduction 1
  1.0.1 Architecture ........................................... 2
  1.0.2 Parallel Programming Models ............................... 5
  1.1 Related Work ............................................. 6
    1.1.1 Image Processing and Parallel Computing ................. 7
  1.2 Goal of this thesis ....................................... 9

2 Fundamental Concepts 10
  2.1 OpenMP .................................................. 10
    2.1.1 Worksharing and Memory Model ............................. 13
    2.1.2 Synchronization in OpenMP ................................ 14
    2.1.3 Sample OpenMP code ................................... 15
  2.2 POSIX threads ........................................... 16
    2.2.1 Pthreads API .......................................... 17
    2.2.2 Sample POSIX thread code ................................ 18
  2.3 Threading Building Blocks ................................. 21
  2.4 Comparison of models ................................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>MPI and Hybrid Parallelization</td>
<td>26</td>
</tr>
<tr>
<td>2.5.1</td>
<td>MPI</td>
<td>26</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Hybrid Parallel Programming</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>OpenMP parallelization for Image Segmentation</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>OpenMP Parallelization Strategies</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Convolution and FFTW</td>
<td>32</td>
</tr>
<tr>
<td>3.1.2</td>
<td>kmeans - Clustering</td>
<td>38</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Smoothing</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>Results and Evaluation</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>Parallelization of Partial Differential Equations(PDE)</td>
<td>50</td>
</tr>
<tr>
<td>4.1</td>
<td>ADCL Testing Framework(ATF)</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Heat Equation Solver</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Results and Analysis</td>
<td>57</td>
</tr>
<tr>
<td>4.3.1</td>
<td>ATF- OpenMP parallelization</td>
<td>58</td>
</tr>
<tr>
<td>4.3.2</td>
<td>ATF - Hybrid parallelization</td>
<td>60</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Hybrid Parallel Heat-equation solver</td>
<td>63</td>
</tr>
<tr>
<td>5</td>
<td>Summary</td>
<td>64</td>
</tr>
</tbody>
</table>

Bibliography 66
List of Figures

1.1 An example of a shared memory architecture. 3
1.2 An example of a Distributed Memory architecture. 5
2.1 An Example of the fork-join model. 12
3.1 The Testing and Analysis Utilities (TAU) - 3D profiling for the image segmentation with convolution. 37
3.2 Pseudo code for kmeans algorithm. 39
3.3 Performance analysis of the complete application. 45
3.4 Performance analysis of the convolution module. 46
3.5 Performance analysis of the kmeans-clustering module. 47
3.6 Performance analysis of the Smoothing module. 48
4.1 The Scalability of both the MPI and OpenMP versions of ATF for a given set of problem sizes. 59
4.2 The execution time of both the MPI and OpenMP versions of ATF for a constant problem size. 60
4.3 The Scalability of both the MPI and Hybrid versions of ATF for a given set of problem sizes. 61
4.4 The execution time of both the MPI and Hybrid versions of ATF for a constant problem size. ......................... 62

4.5 The execution time of both the MPI and Hybrid versions of Heat-equation solver for a constant problem size. ................. 63
List of Tables

3.1 The Flat-profile GRPOF output of the sequential image segmentation code. ................................................................. 34

3.2 Time consumption of the various methods of parallelizing the Kmeans-algorithm, for \( \text{image with } 512 \times 512 \text{ pixels and } 31 \text{ channels and } 8 \text{ threads} \). ................................................................. 42

4.1 The Flat-profile GRPOF output of the sequential code for ATF- solver. 53

4.2 The Flat-profile GRPOF output of the sequential code for Image segmentation. ................................................................. 56
Chapter 1

Introduction

Parallel computing is a form of computation in which multiple instructions are carried out simultaneously [1]. The main concept of parallel computing is to divide a problem into smaller chunks and solve them concurrently. The primary reasons for using parallel computing are to solve large problems and save time (wall-clock). Parallelism can be exploited at multiple levels such as,

- Instruction level,

- Task level, and

- Data level.

Instruction-level parallelism is mostly carried out at the hardware-level, task and data parallelism are typically carried out at the software level. Parallel computing is being used in many different applications [2] which involve complex numerical simulations like weather prediction, chemical analysis, and simulation of nuclear reactions. Parallel processing is also increasingly applied for applications that involve
handling large amounts of data such as databases, data mining, computer-aided
diagnosis in medicine, advanced graphics and virtual reality.

1.0.1 Architecture

Parallel computing on a larger extent depends on the underlying computing re-
source and a relevant parallel programming paradigm. With the birth of multi-core
processors, computing entered mainstream computing, since every PC is a parallel
computer. Parallel computer architectures on a broader perspective can be classified
based on their memory organization into two types;

- Shared memory architectures and

- Distributed memory architectures.

**Shared Memory** parallel computers are those, in which processors have the
capability to access the memory as a global address space. Processors can operate
independently and the changes made on a data item by one processor are visible to
all other processors. The propinquity of the processors facilitates faster data sharing.
Memory access in a shared memory system can be classified into two types, Uniform
Memory Access (UMA) and Non Uniform Memory Access (NUMA). In UMA,
the access time to memory from all processors is the same and all the processors in
the system are symmetric and identical.

NUMA is non-symmetric as the access time to memory for all processors is differ-
ent. Mostly the NUMA shared memory system is formed by linking more than one
Symmetric Multi Processor (SMP) together. In this case, the memory access across
the SMP’s is comparatively slow. Since in a shared memory system the access is global, there is a need for coherence among the processors to ensure every processor is aware of all the updates. Thus, both UMA and NUMA have cache coherent variants (cc-UMA and cc-NUMA). The main problem with this type of architecture is its scalability; the larger the number of processors in the system, the larger is the traffic to access the memory. Moreover, there is additional overhead for cache-coherence memory management with cc-NUMA and cc-UMA systems. A simple example of shared memory architecture is given in Figure 1.1. Shared memory systems started coming into mainstream programming with the introduction of multi-core architectures. A multi-core architecture is one which combines two or more compute cores into one single die. This necessitated improvements to shared memory programming which are discussed in the forthcoming sections.
Distributed Memory parallel computers are those in which each processor has its own local memory. This eliminates the requirement for cache coherence. If there is a need for data exchange between the processors or for synchronization with another processor, the programmer has to specify it explicitly. A basic distributed memory architecture design [3] is shown in Figure 1.2. Distributed memory architectures are more prevalent compared to shared memory architectures as they can be formed with the help of commodity based off-the-shelf components. The performance of the cluster largely depends on the type of connectivity used, the simplest form being ethernet based networks. Distributed memory architectures are also used in distributed computing such as grid computing, where many different resources are loosely connected by a network (usually the Internet), to solve large-scale computational problems [4]. The main drawback of the distributed memory system is that it requires significant modifications of the sequential code in order to fully utilize the resources.

Shared memory architectures can also be emulated on a distributed memory system, wherein software provides a virtual global address space to the users, thereby hiding the underlying distributed memory architecture completely. Typically, this is not as efficient as a shared memory system due to the overhead in retrieving information associated with a different processor. Some software systems that provide this capability are Delphi [5], strings [6], NanosDSM [7].
1.0.2 Parallel Programming Models

A parallel programming model is a software technology which is used to match the application with the underlying parallel system hardware [2] and, can be made available to the user either as a library or new language features. Parallel programming models are classified broadly based on the two types of architectural styles (distributed-memory or shared-memory). The programming models designed chiefly for distributed memory systems are,

- Message Passing Interface (MPI),
- Parallel Virtual Machine (PVM).
Both models are library based. Of the two models mentioned above MPI is the dominant-model used in High-performance computing today. It is the *de-facto* standard for parallel programming on distributed memory systems. The major programming models in the case of the shared memory systems are:

- POSIX threads (Pthreads),
- Open Multi-Processing (OpenMP), and
- Intel’s Threading Building Blocks (TBB).

Pthreads was the earliest standard for thread based shared memory programming. The first version of OpenMP was published in 1997 and is usually preferred to Pthreads by end-users developing scientific applications, as it provides a more flexible and user-friendly interface for developing parallel applications. The Threading Building Block (TBB) is a C++ template library developed by Intel to take advantage of multi-core processors and is based on the concept of tasks. This particular programming model has been kept proprietary for a long time and has been made open-source in 2007. These shared memory models are discussed in detail in Chapter 2.

### 1.1 Related Work

Grand Challenge Applications are computational problems in science or engineering that go beyond the capability of the largest supercomputers available today. There is a variety of grand challenge applications, for example, computational fluid dynamics [8], electronic structure calculations [9], and quantum chromodynamics. Image
processing [10] is considered to be one such complex application. The forthcoming subsections discuss some applications of parallel computing in image analysis tasks [10].

1.1.1 Image Processing and Parallel Computing

Due to the large amounts of data involved, image processing can be computationally demanding. In most image processing algorithms, different portions of an image can be processed in parallel with different processors or threads. As an example, Ray tracing [10] in computer graphics is the technique of tracing the individual rays of light as it travels through visual scene. The computerized representation of the visual scene consists of a mathematical description of the shapes and surface characteristics of each object. The ray tracing rendering method works by following individual rays backwards from the viewer’s eye, through the two dimensional image plane. The most time consuming step is to determine if a ray intersects a particular object, which may require 25 floating point operations per pixel. For an image consisting of 2000 $\times$ 2000 pixels and a visual scene with 100 objects the total number of floating point operations required is $2000 \times 2000 \times 100 \times 25 = 10^{10}$. To avoid distortion of the image, due to aliasing effects, as many as ten rays must be used for each pixel, thereby increasing the number of operations to $10^{11}$. For an animated sequence of 15 minutes using 100 frames/second, the computation involved is $10^{16}$ FLOPs. An application performing these ray-tracing operations achieving on a particular system a performance of 1 GFLOPs, the total time consumed will be around 4 months [10]. There has also been work involved in parallelizing ray tracing algorithm.
In [11] the authors have performed a pixel-oriented parallelization of the ray tracing algorithm on a hybrid parallel architecture with processor farm model [12]. In this parallel implementation, the image space is partitioned to each processor by using efficient load-balancing methods. The work specifies that the parallel implementation provided a speedup of 3.89 with four processing elements.

Parallel programming has also been used to improve the performance of reading JPEG 2000 images and MPEG-4 VTC videos on SMPs using OpenMP. JPEG 2000 and MPEG-4 Visual Texture Coding (VTC) have been accepted as a standard for still image coding and are both wavelet-based [13]. The work described in [14] improves the performance of both the algorithms by exploiting the parallelism available in two main coding stages, which are wavelet-lifting and code-block processing part for the JPEG 2000 and the convolution based wavelet-filtering and zero-tree coding in MPEG-4 VTC.

In [15] the authors have parallelized the edge detection algorithm using a domain decomposition based method called Uniform Distribution on a cluster of Sun workstations. In this method, a large image is divided equally into multiple smaller sub-images. The data for one or more sub-images is distributed among processors at run-time using load balancing, which ensures nearly equal distribution of computation among processors. In recent times, image processing techniques using Partial Differential Equations (PDE) [16] have emerged. A PDE based image processing is formulated as a filtering or diffusion process or an optimization problem like segmentation and registration, both of which can be formulated using partial differential equations. The Mumford-Shah [17] segmentation model also uses an energy function
to solve a defined segmentation problem. Similarly, variational PDE models are also being used for image segmentation as described by Chan [18].

1.2 Goal of this thesis

This thesis deals with exploring shared memory and hybrid parallelization strategies for image processing applications. The applications used in this study are: a texture-based segmentation of thyroid Fine Needle Aspiration Cytology (FNAC) images [19] and PDE-based applications for time independent [20] and time-dependent [21] problems. The texture-based segmentation application was parallelized using the shared memory parallel programming paradigm OpenMP. The other two applications were parallelized using OpenMP and hybrid parallelization methodologies (OpenMP + MPI). This thesis discusses the experiences in parallelizing aforementioned applications and compares the ease of programming with the different programming models. Furthermore, the performance of each parallel code version is evaluated and compared to the sequential and MPI parallel code versions.

In this thesis, Chapter 2 gives an overview of the main concepts of the most widespread parallel programming paradigms. Chapter 3 introduces the texture-based image segmentation application, explains the parallelization strategies used and discusses the results obtained. Chapter 4 presents hybrid parallelization strategies of two Partial Differential Equation (PDE) based image processing applications and presents performance results for both scenarios. Finally, Chapter 5 summarizes the thesis and presents future research directions and improvements.
Chapter 2

Fundamental Concepts

This chapter discusses the most popular programming models available for parallel computing and evaluates their advantages and disadvantages. The programming models are divided into two main broad categories namely, distributed memory and shared memory models. OpenMP, Pthreads and Threading building blocks (TBB) are shared memory models while Message Passing Interface (MPI) is a distributed memory model.

2.1 OpenMP

OpenMP (Open multi-processing) [22] [23] is an Application Programming Interface (API) that has been developed to enable portable and scalable shared memory parallel programming. It supports parallelization of FORTRAN, C and C++ applications by defining a specified set of compiler directives, environment variables, and library routines. OpenMP can be used for parallelizing applications for platforms
ranging from desktops to supercomputers. The API permits incremental parallelization; wherein, portions of a program can be parallelized in steps.

OpenMP maintains the specification of a sequential program, and builds a block of work that is in-turn executed cooperatively by a collection of threads. A thread is a run-time entity that executes a stream of instructions independently. From the operating system perspective, resources are allocated either to processes or threads. A process can comprise more than one thread to execute the program, in which case the resources will be shared. Every individual thread will require a program counter, processor register, processor status, signal masks, and stack for saving its own variables. Among the resources managed on a process level and, therefore, shared by multiple threads are the address space, file handles, signal handlers, and timers. In general, every computationally intense thread should be executed on a separate core or processor. In the case there are more threads than cores available, the operating system performs simultaneous multi-threading, i.e., multiple threads have to share the same core/CPU through time-slicing.

OpenMP uses the fork-join model [2]. Figure. 2.1 shows an example of this model, in which the program starts execution with a single thread. This thread is referred to as the initial thread. A region of code enclosed within an OpenMP parallel construct is called a parallel region. Whenever an executing thread encounters a parallel region, it forks a team of threads to execute that parallel block. The thread that forks becomes the master thread and coordinates with the other threads. Once the parallel region ends, only the master thread stays and all the other threads terminate(join).
OpenMP directives are specially formatted comments or pragma statement, which is generally applied to the code beneath it. A pragma statement in general affects only the thread it encounters.Pragma statements can be ignored by the compiler, if the code is not compiled with the appropriate OpenMP compiler flag. The user is provided with the ability to spawn threads, specify the means to share workload among threads, perform synchronization (in case certain work needs to be done exclusively by one thread), and to declare either private or shared variables. The forking of threads is performed when a `PARALLEL` directive is encountered. Additional information can be provided along with the parallel directive. At the end of a parallel region, there is an implicit barrier synchronization which ensures that all threads have reached that point before proceeding. If a thread in a team of threads encounters a parallel region, it in turn spawns new threads, for which it would be the master (nested parallelism).

### 2.1.1 Worksharing and Memory Model

Work-sharing is very important in OpenMP since, it specifies how the work is shared among the threads. The choice of work-sharing may have a considerable influence on
the performance. In scientific computing most of the work is performed in loops, for e.g., a ‘for’ loop in C/C++ and a ‘do’ construct in FORTRAN. The appropriate OpenMP directive is added before each loop within the parallel region. OpenMP has certain restrictions on the kind of loops it can parallelize. The main restriction within this context is that the number of iterations must be known at compile time, in order for OpenMP to distribute the iterations among the threads. For example, a ‘while’ loop in C cannot be parallelized with OpenMP. In such cases, the user should try to convert the ‘while’ loop into a ‘for’ loop.

Since OpenMP is based on the shared-memory model, all data involved is shared and visible to all threads. However, there are situations in which the data needs to be thread-specific, where the data is potentially different and private to each thread. For example, in the case of parallelizing a ‘for’ loop, the loop counter variable is private to each thread. In OpenMP, on specifying the corresponding work-sharing construct, the compiler enforces the iteration variable to be private. In situations where variables are thread-specific, they have to be declared private. Private variables are also often beneficial in improving the performance of the application by reducing the number of updates made to a shared memory region. This helps to avoid memory contention and reduces the need for synchronization for cc-NUMA platforms, which can become expensive when large numbers of threads are involved.

2.1.2 Synchronization in OpenMP

Thread synchronization is required for coordinating simultaneous threads to complete a task with correct run-time order and avoid unexpected race conditions. Different
applications require different levels of synchronization. OpenMP provides a small set of synchronization methods, which reduce the likelihood of synchronization errors. To make life easier for the programmer, it also provides implicit synchronization. OpenMP, by default, provides BARRIER synchronization, wherein each thread waits at the end of a work-sharing construct or a parallel region, for all the other threads in the team to complete the execution. For situations when only one thread at a time has to update a piece of code in a parallel region, the other threads can either wait (busy-wait) or work on other parts (blocking). The choice between blocking and busy-wait depends on the application. The main set of synchronization mechanisms provided by OpenMP are CRITICAL, ORDERED, ATOMIC, FLUSH, and BARRIER. The CRITICAL directive enforces, that only one thread can execute a code sequence at a time. The ORDERED directive is used to impose a serial order on the execution of a code section. The ATOMIC directive is used to update a memory location in an uninterrupted fashion. The FLUSH directive is used to ensure that all threads in a team have a consistent view of memory. The MASTER directive forces execution only by the master thread. The BARRIER directive forces all team members to gather at a particular point in code. Note that the BARRIER directive cannot be used inside a parallel region or a work-sharing construct, as it can result in a potential deadlock. The main problem with OpenMP in the current specification is that synchronizing a subset of threads is difficult as there is no explicit support for it.
2.1.3 Sample OpenMP code

Below is a sample code that performs a simple vector addition. The OpenMP part of the code is as simple as two lines; one to start the parallel region and the other to specify the work-sharing construct. Apart from these, the `omp.h` header file has to be included. The parallel region triggers the specified number of threads and the work-sharing construct distributes the work among the threads. Here, the two variables `i` and `tid` are made private, so that all threads in the team will have a private copy of these variables. The rest of the variables which do not modify the result are un-categorized and so by default remain shared. The total number of iterations is distributed among the available threads. In this case, 100 iterations are distributed across say 4 threads, and then each thread will have 25 iterations to work with. The function, `omp_get_num_threads` is used to retrieve the total number of threads involved and the function, `omp_get_thread_num` retrieves the particular thread number. These are OpenMP library functions.

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define SIZE 100
int main ()
{
    int nthreads, tid, i, chunkSize;
    float a[SIZE], b[SIZE], c[SIZE];
    /* Some initializations */
    for (i=0; i < SIZE; i++)
```
a[i] = b[i] = i * 1.0;
/* parallel region starts */
#pragma omp parallel private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    #pragma omp for
    for (i=0; i<N; i++)
    {
        c[i] = a[i] + b[i];
        printf("\text{result_vector}[%d]= %f\n", i, c[i]);
    }
} /* end of parallel region */

2.2 POSIX threads

POSIX threads (pthreads) [24] have been defined in 1995 in the IEEE POSIX 1003.1c standard as an API for creating and manipulating threads. Though pthreads is commonly used on Unix-like operating systems such as Solaris or Linux, windows implementations do exist as well. Most hardware vendors provide Pthreads in addition to
their proprietary thread API’s.

2.2.1 Pthreads API

The Pthreads functions are implemented in C. The subroutines present in the pthreads API can be roughly categorized [2] into three types,

- **Organizing Thread Subroutines:** These functions perform actions like creating, detaching and joining threads. The various functions that pthreads provides for this are, `pthread_create()` (to create a thread), `pthread_exit()` (to terminate current thread), `pthread_cancel()` (to cancel execution of another thread), `pthread_join()` (to block current thread until another one terminates), `pthread_attr_init()` (to initialize thread attributes), `pthread_attr_setdetachstate()` (to set the detach state attribute), `pthread_attr_getdetachstate()` (to get the detach state attribute), `pthread_attr_destroy()` (to destroy thread attributes), `pthread_kill()` (to send a signal to a thread).

- **Synchronization Subroutines:** These functions are also called ”mutex functions”. Mutex is an abbreviation for Mutual exclusion. The various pthreads mutex functions are, `pthread_mutex_init()` (to initialize mutex lock), `pthread_mutex_destroy()` to (destroy a mutex), `pthread_mutex_lock()` (to acquire blocking mutex lock), `pthread_mutex_trylock()` (to acquire non-blocking mutex lock), `pthread_mutex_unlock()`(to release mutex lock).

- **Communication Subroutines:** These functions take care of the communication between the threads that share the mutex. These are based on the
programmer specified conditions (condition variables). The set of pthreads functions provided are `pthread_cond_init()` (to initialize a condition variable), `pthread_cond_destroy()` (to destroy a condition variable), `pthread_cond_signal()` (to signal a condition variable), `pthread_cond_wait()` (to wait on a condition).

### 2.2.2 Sample POSIX thread code

Below is a sample pthreads code [2] for a simple sum calculation code with loops. The global sum is maintained by a mutex variable. The array is decomposed by distributing the loop iteration.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NTHREADS 4
#define ARRAYSIZE 1000000
#define ITERATIONS ARRAYSIZE / NTHREADS

double sum=0.0, a[ARRAYSIZE];
pthread_mutex_t sum_mutex;

void *do_work(void *tid)
{
    int i, start, *mytid, end;
    double mysum=0.0;

    /* Initialize my part of the global array and keep local sum */
```
mytid = (int *) tid;
start = (*mytid * ITERATIONS);
end = start + ITERATIONS;
printf ("Thread %d doing iterations %d to %d\n",*mytid,start,end-1);
for (i=start; i < end ; i++) {
    a[i] = i * 1.0;
    mysum = mysum + a[i];
}
/* Lock the mutex and update the global sum, then exit */
pthread_mutex_lock (&sum_mutex);
sum = sum + mysum;
pthread_mutex_unlock (&sum_mutex);
pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    int i, start, tids[NTHREADS];
    pthread_t threads[NTHREADS];
    pthread_attr_t attr;

    /* Pthreads setup: initialize mutex and explicitly create threads in a
     * joinable state (for portability). Pass each thread its loop offset */
    pthread_mutex_init(&sum_mutex, NULL);
    pthread_attr_init(&attr);
    pthread_mutex_init(&sum_mutex, NULL);
    pthread_attr_init(&attr);
In the code above the function `pthread_create` spawns the set of threads required for parallelizing the function, `do_work`. In the `do_work` function, the workload
is divided based on the number of threads. The global sum is calculated with syn-
chronization around it, with the help of mutex locking functions. Though pthreads
is quite comprehensive and portable (no special compiler required), its major lim-
itation is the amount of thread-specific code required, which can be seen from the
sample code given above. Pthreads is inherently a low-level API and requires a large
number of operations to perform even simple threading tasks (declaration of thread-
ing structures, individual creation of threads, computation and assignment of loop
bounds for each thread, and termination of threads). The amount of thread-specific
code can further increase substantially with increasing complexity of the application.
Although POSIX threads are still dominant for implementing operating system level
operations like device drivers, developers of scientific applications have increasingly
been looking for a simpler alternative to pthreads.

2.3 Threading Building Blocks

The Threading Building Blocks (TBB) [25] is a library which supports parallel pro-
gramming using standard C++ templates. It does not require the programmer to
have much knowledge on threads, synchronization and load balancing, since it defines
C++ templates for common parallel patterns. The main advantage of TBB is that
it helps to specify parallelism using tasks rather than raw threads. This helps TBB
to specify parallelism more conveniently. TBB supports nested parallelism, which
enables creating larger components from smaller ones. Until July 2007 TBB was a
proprietary Intel product. Since then, it was made an open-source project. The TBB
commercial release 2.0 [26] supports Microsoft Windows (XP or newer) using Visual
C++ compiler (version 7.1 or higher), Mac OS X using Intel C++ Compiler and Linux using Intel C++ compiler or GNU Compiler Collection (gcc). Additionally, the open source builds of TBB support Solaris and FreeBSD.

TBB balances the workload across the available cores using task stealing. In the TBB’s task stealing model, the workload is evenly divided among the available processor cores. If one core completes its work while other cores still have a significant amount of work in their queues, TBB reassigns some of the work from one of the busy cores to the idle core. This dynamic reallocation of workload reduces the overhead associated with the programmer by avoiding changes to be made to the source code or executable. Since TBB uses C++ templates it relies on compile-time polymorphism (function overloading), which is usually considered to be more efficient than dynamic run-time polymorphism (operator overloading).

Some of the main advantages of TBB are:

- Specifying parallelism in terms of tasks instead of threads. Threads are low-level, heavy constructs that are close to the hardware. This can make programming inefficient. TBB converts tasks onto threads such that there is efficient use of processor resources.

- Full compatibility with other threading packages such as OpenMP.

- Emphasizing data-parallel programming. This feature facilitates multiple threads to work on different parts of a collection. Since the collection is divided into smaller pieces, scalability improves.

- Facilitating generic programming with very few constraints.
Some of the main limitations of TBB are:

- As of today, TBB does not support accelerator cards or distributed memory architectures.
- No full protection from race conditions, deadlock and non-deterministic behavior.
- Relatively restricted scope and applicability compared to OpenMP.
- Cannot be used for C or FORTRAN applications as it relies on the class and C++ template features.

Below is a sample code for parallelizing a for loop using TBB taken from [27]. All the code apart from the for loop and the 'function call for foo' are TBB specific. The amount of TBB-specific code is comparatively less than in the POSIX threads example, but can still introduce significant amount of code compared to the sequential version.

```cpp
#include "tbb/blocked_range.h"

class ApplyFoo {
    float *const my_a;

public:
    void operator()( const blocked_range<size_t>& r ) const {
        float *a = my_a;
        /* Original Code*/
        for( size_t i=r.begin(); i!=r.end(); ++i )
            Foo(a[i]);
```
The operator function has a parameter name `blocked_range<size_t>`, which is a template class provided by the library. It describes a one-dimensional iteration space over T. The `parallel_for` function does require a copy constructor in the body object, which can be invoked to create separate copies for each worker thread. Since the object is copied, it should not be modified by the operator function. This is a reason that operator function has been defined constant.

### 2.4 Comparison of models

As discussed previously POSIX threads are the only low-level shared memory parallelization API among the three models presented here. Therefore, it provides many functionalities, flexibility, and freedom for programming. This freedom becomes a constraint as it requires a lot of thread specific code. It becomes very complex for huge applications, and it is hard to debug. Moreover, thread-safety also needs to be ensured by the user, which makes programming with Pthreads all the more difficult.

Shared memory programming has become mainstream parallel computing, only with the birth of multi-cores. This has necessitated decades of legacy sequential applications to be parallelized. A common technique working with POSIX threads
is to create threading pools, in which a number of threads are created at program start up and workload is distributed among them. However, this approach requires considerable thread-specific code and there is no guarantee that it will scale optimally with the number of available processors.

On the other hand, OpenMP is minimally invasive in the sequential code. A code parallelized with OpenMP can also be executed sequentially by mere removal of the associated compiler flag. This can be useful in certain situations like debugging. Unlike Pthreads, OpenMP does not require the parallelization of the entire application at once (incremental parallelization). A code can be parallelized using as little as two additional lines as shown in the section 2.1.3. Hybrid parallelization becomes very easy with OpenMP. One main drawback of OpenMP is that it is compiler dependent. To execute an OpenMP code the system must have a compiler that supports OpenMP.

The Intel Threading Building Blocks provides abstraction for parallelization using C++. This can be used only with C++ code as discussed in section 2.3. The main advantage of TBB is that it can specify parallelization based on tasks and not threads. Though OpenMP holds the monopoly with C and FORTRAN, they have very minimal support for C++. TBB may be advantageous for parallelization using C++, but the user effort involved is also comparatively high as seen from the sample code in the section 2.3. With the proposal of including tasking into the upcoming OpenMP 3.0 specification, many of the advantages of TBB compared to OpenMP might vanish.
2.5 MPI and Hybrid Parallelization

As discussed in Chapter 1, different parallel programming paradigms have evolved depending on the hardware architecture. The following section discusses the most prominent model for distributed memory architectures, MPI, and a hybrid parallelization strategy based on MPI and OpenMP.

2.5.1 MPI

MPI (Message Passing Interface) [28] is the de-facto standard for programming distributed memory architectures. It has been in existence since 1990. The main concept in MPI is an explicit data exchange between two processes using Send/Receive functions. Since MPI is a library based model, it does not require any special compiler and can run on any platform which has an MPI implementation installed. MPI is independent from the machine architecture, operating system, and the network. MPI defines the Application Programming Interface (API) for communication routines, and does not represent a network protocol. As of today, it has language bindings defined for C, C++ and FORTRAN.

Although MPI provides a lot of flexibility to the user on how to parallelize a code, creating an MPI parallel code largely depends on users efforts. Some of the major advantages of MPI are,

- Notion of data types, for both pre-defined types such as an int or a float and user defined structures;
• Support for synchronous, asynchronous and one-sided communication operations;

• Support for collective communication and group constructors.

MPI has been shown to scale well to very large number of processors. Due to the explicit parallelization approach suggested by MPI, its application typically makes good use of caches through local memory. On the other hand, some of its drawbacks are the enormous efforts required to develop MPI code, since every aspect such as load-balancing has to be implemented manually by the user. Furthermore, debugging MPI code can be time consuming and difficult.

2.5.2 Hybrid Parallel Programming

MPI and OpenMP are two completely different models for different types of architectures, namely distributed and shared memory architectures. While MPI can be used for shared memory machines with a performance compromise and OpenMP can used for distributed memory architectures using softwares such as Treadmarks [29] and SCASH [30], both of them are sub-optimal on architectures for which they have not been originally been designed. The next-generation parallel architectures will comprise multi-core processors which contain two or more independent cores per CPU, connected by a network interconnect. This architecture suggests the merging of the two styles of parallel programming. In such machines, a programmer can write a combination of MPI and OpenMP codes; wherein, MPI takes care of the inter-node communication in the SMPs and OpenMP for intra-node parallelization.

In most hybrid codes, OpenMP is used for the computational parts leaving all
MPI communication outside the parallel regions. In this case, all the threads except
the master thread will be inactive when the MPI routines are executed. This is
the most suitable model when the communication is less significant compared to the
computation. A simple hybrid parallel code is given below.

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

int main (int argc, char *argv[])
{
    int id, np;
    int threads, numthreads;
    char name[MPI_MAX_PROCESSOR_NAME];
    int namelen;
    int i;

    MPI_Init (&argc, &argv);

    MPI_Comm_size (MPI_COMM_WORLD, &np);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Get_processor_name (name, &namelen);

    #pragma omp parallel private(threads,numthreads)
    {
        thread = omp_get_thread_num();
```
numthreads = omp_get_num_threads();
printf ("This is Process %2d
and thread %d of %d threads out of %d running on host %s
", id, np, name, threads, numthreads);
}
MPI_Finalize ();

return (0);
}

In the case of overlapping communication and computation, the MPI communication can be either executed by a single thread (thread-funneled approach) or by multiple threads (thread-multiple approach) [31]. The thread-funneled approach is not efficient as it can cause load-imbalance, and it becomes imperative that the thread with the least workload perform the MPI communication. Similarly, in the thread-multiple approach, multiple threads take care of the MPI communication. While this approach improves the load balancing among the threads, it requires the communication library to be thread-safe. The type of hybrid parallelization leading to the best overall performance depends on the application’s requirements. In cases where the computation is highly dominant, the first method often leads to good results. But in cases where communication contributes equally to computation or worse, the latter methods specified could be more advantageous.
Chapter 3

OpenMP parallelization for Image Segmentation

Cancer is one of the major reasons for premature death in the human population. Thyroid nodule is a type of cancer, common in 20% of the world’s population and approximately 50% of 60-year-old persons [19] [32]. Compared to various screening and detection methods, the fine needle aspiration cytology (FNAC) is considered to be less invasive, safe and inexpensive [33]. The staining procedure in FNAC changes the color of the cells and tissues. This makes the cytological evaluation under a standard microscope with moderate magnification feasible. Image segmentation is one of the wide-spread and critical problems, since it forms the first step in identifying cells and tissue structures relevant for subsequent analysis. The unpredictable nature of images makes the process of segmentation even more challenging. More recently, textural features have been exploited for cell segmentation. As stated in [19], texture can be obtained using a multitude of methods ranging from gray-level
co-occurrence matrices (GLCM), fractal measures, Law’s texture measures, gradient structure tensors (GST), and Gabor Filters.

In recent times, multi-spectral microscopes capable of acquiring spectral images under transmitted illumination have been used to digitize and analyze cell smears. Spectral imaging allows simultaneous measurement of spectral and spatial information of a sample. This facilitates measurement of the spectral response at any pixel of a two-dimensional image. A spectral image consists of a series of images and each image is acquired under a narrow band wavelength of light. Studies have shown that biological tissue exhibits unique spectra in transmission. By exploring the spectral differences in tissue pathology, many chemical and physical characteristics which are not revealed by traditional imaging systems, can be realized and used to improve the analysis efforts.

The application employed in this section is based on a sequential code written by Prof. Shishir Shah in the Department of Computer Science at the University of Houston. The code uses a bank of Gabor filters, to extract a measure of texture at each pixel followed by a clustering algorithm to group pixels belonging to the same class. Specifically, a bank of self similar filters, through appropriate dilations and translations, are generated on the basis of the Gabor function. Three scales and four orientations are used, resulting in a total of twelve filters in the bank. An average image from the multi-spectral stack is generated, to get an efficient measure of the texture for each pixel in the multi-spectral image. All the twelve filters are applied to the average image and the magnitude responses such as the mean and standard deviation are computed and stored as a feature vector. The absorption parameter
can be computed for each pixel using the Beer-Lambert law \[19\] \[34\]

\[ A = \log(1/T) \] (3.1)

in each channel of the multi-spectral image. This generates a 45-dimensional \[19\] feature vector for each pixel belonging to a spectral image with 31 channels. The extracted features are clustered using the standard k-means algorithm, which results in effective grouping of pixels belonging to the thyroid cells and partitioning of the image. This thesis presents the strategies and experiences with developing the OpenMP parallel version of the code, and discusses the performance improvements of the same.

### 3.1 OpenMP Parallelization Strategies

Since the application discussed in section 3 involves images of 8–50 GB, parallelization was mandatory in order to obtain any result for these images. The sequential code used was implemented in C and consists of three large modules: convolution and FFT, k-means clustering and a smoothing algorithm. In the following, we discuss the parallelization of each module individually.

#### 3.1.1 Convolution and FFTW

The module, which performs the image padding, convolution, de-convolution and filtering, is referred here as convolution module. Most of the computations in this module are done using the routines from the FFTW library [35]. FFTW is a library containing a set of C-routines, which is used for Discrete Fourier Transform
operations. This library supports complex, real and parallel transforms of one or two dimensions. Some of the main advantages of FFTW are its portability and its self-optimizing feature with which it can tune the performance of the routines based on the type of the machine, cache, size of memory, the number of registers and all other factors that influence the performance of the code.

The working of FFTW is divided into two steps. In the first step, the ‘FFTW-planner’ determines the best way to compute the transform on a particular machine for a given problem. This information is stored as an execution plan. FFTW execution routines with its input parameters compute the transform as dictated by the plan. The plan can be reused many times. Though the planning time becomes small in-case of typical high-performance applications where a plan is reused many times, it can be significant in case of smaller computations. The FFTW library supports different algorithms for the planning step, applying either exhaustive testing of the available options (FFTW_MEASURE) or a heuristic, delivering an estimate on the best performing plan (FFTW_ESTIMATE).

The application discussed in this section, uses the multi-dimensional complex transforms. The current release of the code relies on FFTW version 2.1.5, for both sequential and OpenMP version of the code. The FFTW-2.1.5 has direct support for shared memory parallel codes using POSIX threads, with OpenMP being an added feature. Although version 3.1.2 of FFTW has direct support for OpenMP, the library version used in this thesis is 2.1.5 due to incompatibilities between the two versions.
The first step in parallelizing any code using OpenMP is to find the most time-consuming modules. This is required in order to exploit the incremental parallelization capabilities of OpenMP. To determine the profile of the application and get an idea about the source code, the GPROF tool is used in this analysis. This tool gives an idea of the routines in the source code that consume most of the time, during execution. The profile file which was generated for the sequential version of the application is shown in Table 3.1. In Table 3.1

<table>
<thead>
<tr>
<th>% time</th>
<th>Cumulative time (s)</th>
<th>Self (s)</th>
<th>no. of calls</th>
<th>self (s/call)</th>
<th>total (s/call)</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.61</td>
<td>3.62</td>
<td>3.62</td>
<td>1</td>
<td>3.62</td>
<td>3.62</td>
<td>kmeans</td>
</tr>
<tr>
<td>21.45</td>
<td>5.93</td>
<td>2.31</td>
<td></td>
<td></td>
<td></td>
<td>fftw_twiddle_rader</td>
</tr>
<tr>
<td>10.21</td>
<td>7.03</td>
<td>1.10</td>
<td></td>
<td></td>
<td></td>
<td>fftw_twiddle_rader</td>
</tr>
<tr>
<td>9.42</td>
<td>8.05</td>
<td>1.02</td>
<td></td>
<td></td>
<td></td>
<td>fftw_no_twiddle_10</td>
</tr>
<tr>
<td>8.64</td>
<td>8.98</td>
<td>0.93</td>
<td></td>
<td></td>
<td></td>
<td>fftw_no_twiddle_13</td>
</tr>
<tr>
<td>3.16</td>
<td>9.32</td>
<td>0.34</td>
<td></td>
<td></td>
<td></td>
<td>fftw_no_twiddle</td>
</tr>
<tr>
<td>2.60</td>
<td>9.60</td>
<td>0.28</td>
<td></td>
<td></td>
<td></td>
<td>fftw_twiddle_4</td>
</tr>
<tr>
<td>2.14</td>
<td>9.83</td>
<td>0.23</td>
<td></td>
<td></td>
<td></td>
<td>fftw_strided_copy</td>
</tr>
<tr>
<td>1.86</td>
<td>10.03</td>
<td>0.20</td>
<td></td>
<td></td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>1.86</td>
<td>10.23</td>
<td>0.20</td>
<td>12</td>
<td>0.02</td>
<td>0.03</td>
<td>cconv2d</td>
</tr>
<tr>
<td>1.49</td>
<td>10.39</td>
<td>0.16</td>
<td></td>
<td></td>
<td></td>
<td>fftw_twiddle_4</td>
</tr>
<tr>
<td>1.02</td>
<td>10.50</td>
<td>0.11</td>
<td>1</td>
<td>0.11</td>
<td>0.11</td>
<td>gray_erode</td>
</tr>
<tr>
<td>0.84</td>
<td>10.59</td>
<td>0.09</td>
<td>12</td>
<td>0.01</td>
<td>0.01</td>
<td>pad_image</td>
</tr>
<tr>
<td>0.74</td>
<td>10.67</td>
<td>0.08</td>
<td>12</td>
<td>0.01</td>
<td>0.01</td>
<td>pad_filter</td>
</tr>
<tr>
<td>0.51</td>
<td>10.72</td>
<td>0.06</td>
<td></td>
<td></td>
<td></td>
<td>fftw_executor_simple</td>
</tr>
<tr>
<td>0.23</td>
<td>10.75</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
<td>executor_many</td>
</tr>
<tr>
<td>0.19</td>
<td>10.77</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td>planner_normal</td>
</tr>
<tr>
<td>0.05</td>
<td>10.77</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
<td>fftw_no_twiddle_9</td>
</tr>
<tr>
<td>0.00</td>
<td>10.77</td>
<td>0.00</td>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
<td>gaborbank</td>
</tr>
</tbody>
</table>

Table 3.1: The Flat-profile GRPOF output of the sequential image segmentation code.

- %-time refers to the percentage of the total time consumed by the function.
• **Cumulative seconds** refers to sum of the number of seconds accounted for by this function and those listed above it.

• **Self seconds** refers to the number of seconds accounted for by this function alone.

• **Calls** refers to the number of times this function was invoked. This value is obtained only if the function is profiled, or it is left blank.

• **Self s/call** refers to the average number of seconds spent in this function per call. This value is blank if the function is not profiled.

• **Total ms** refers to the average number of seconds spent in this function and its descendants per call. This value is blank if the function is not profiled.

• **Name** refers to the name of the function.

As seen in Table 3.1 the FFT-modules and the k-means clustering function consume most time. As discussed in the previous paragraph, following the incremental parallelization strategy of OpenMP, the convolution module is parallelized first.

The parallelization of the convolution module required two main steps:

1. Initializing the threads - This is required for FFTW to inform its routines, that it is a multi-threaded execution.

2. Substituting the sequential functions with their appropriate parallel counterparts - This was quite simple as there was only one additional parameter required for those functions, namely the number of threads.
Although these two steps were quite straightforward, as they were related to usage of the FFTW library, there were also pitfalls in this approach. Since the FFTW routines are a black-box for the rest of the code, parallelizing all modules leads to nested parallelism within the code. This degrades the performance hugely due to the large number of thread creations. This problem was debugged with the help of the performance analysis tool TAU [36], by checking on the number of cache misses between threads and the number of invalidations in the convolution loop. Removing the OpenMP directives, which lead to the nested parallelism, improved the performance of the code significantly. To further improve the performance, the estimate heuristic of the planner routines has been used instead of the originally used measure flag. Further, the planning was done inside the convolution module, which was expensive as it was done for all iterations of the main loop, instead of reusing it for all iterations. Thus, the planning and the padding of image were moved to the main function outside of the convolution routine. Moreover, the FFTW_IN_PLACE option, which produces a plan for which the output array is same as the input array, has been set in order to improve the memory consumption of the library. This is imperative while handling huge images. The parallelization of the convolution module was profiled again with TAU to make sure that it was well distributed.

The profile shown in Figure 3.1 depicts the profile of the convolution part of the image segmentation code, after performing the optimizations outlined above. As seen in the figure, the different colors depict the different intensity in terms of time. The profile is a 3D- visualization across the time, threads and functions. As shown in the profile, the work has been distributed largely among all threads, except for two
Figure 3.1: The Testing and Analysis Utilities (TAU) - 3D profiling for the image segmentation with convolution.
peaks at the master thread. These peaks represent the convolution function, where
the threads are forked and joined later. Since the profiling tool is capable of showing
loop-level details, two peaks are shown: one for the loop itself and one for the function
holding it. The other significant time consumption in that module was writing the
result of the convolution to a file. Unfortunately, this could not be parallelized as
there is no option for parallel-I/O in OpenMP in the current specification.

3.1.2 kmeans - Clustering

The kmeans clustering is an algorithm to cluster or group objects into K groups,
based on attributes, where K is a positive integer. The grouping is done by mini-
mizing the sum of the squares of the distances between data and the corresponding
cluster centroid. This can be represented by the [37] equation 3.2.

\[
V = \sum_{i=1}^{k} \sum_{x_j \in S_i} (x_j - \mu_i)^2
\]  

(3.2)

The main steps of the algorithm can be listed as:

- The k points are placed in the space of the objects that are being clustered,
  which form the initial centroid.

- Each object is then assigned to the group that has the closest centroid.

- When the objects have been assigned, the position of the centroid gets recal-
  culated.

- The steps 2 and 3 are carried out until there is no change in the centroid.
The pseudo code for the implementation of the Kmeans-algorithm has been given in Figure 3.2.

The OpenMP parallelization for Kmeans put forth many challenges. As seen in the pseudo code in Figure 3.2, the determination of the centroids is performed across an indefinite ‘while’ loop. Since OpenMP is compiler-based API, it does not support parallelization of loops with undeterminable number of iterations at compile time. So the parallelization was done at the level of individual pixels. This, in turn, posed one more challenge in parallelizing the arrays used for determining the number of pixels assigned to a cluster and the weight of each cluster across all threads, as both the arrays had to be shared. This again became an issue, as OpenMP did not
support reduction for arrays in C. Different approaches were explored to overcome this problem.

**Critical Section** The first and the most obvious method to solve the problem was to add a critical construct to the code associated with shared data modification. The critical construct provides a way to ensure, that multiple threads do not attempt to update the same shared data simultaneously. The critical construct in OpenMP is implemented with the help of `pragma omp critical` directive. The immediate part of the code, following this directive will be under a critical section. Since the number of clusters was small compared to the pixels, the number of memory locations accessed was also relatively small. Thus, the contention for access among the threads was high. This degraded the performance with increasing number of threads. Though this approach ensured the data correctness, there was no performance improvement.

**Array of Locks** The array of locks is the substitute for critical section suggested by OpenMP experts to improve performance. The main difference is, critical section blocks the access to the complete array from other threads. Using an array of locks blocks only a particular value which is being updated from the other threads. Theoretically, it reduces the contention among threads and thereby improves the performance. But in this situation, it did not improve the performance as the number of clusters was small compared to the pixels. So the thread contention still existed. Although, it improved the performance compared to the previous implementation, it was still not satisfactory.

**Additional Thread-ID Parameter** Since the thread contention was the problem, the third method was devised by having the thread-ID, as the second and third
dimension for the cluster and the weight of each cluster arrays, respectively. This removes the necessity for any critical section or locks for the shared-data, as there will be no thread contention. Every thread will update its own data. Finally, to aggregate the data, a manual reduction can be done across the threads, the clusters and the pixels, to reduce the 3-d and 2-d arrays to 2-d and 1-d respectively. This theoretically seemed to be a potent idea. But, since the number of clusters still remains low, there were lot of cache misses for the threads, and a lot of time was spent in cache coherence updates. Moreover, having a third dimension added additional burden on to the compiler. Though this version improved the performance of the application compared to the previous two methods, it was still not convincing.

**Atomic construct** One of the other effective alternatives for a critical region construct is an atomic construct. This is an in-built construct, which is provided by OpenMP to remove contention among threads, while accessing shared data. It is specified using `pragma omp atomic`. This, unlike the other constructs, has to be applied to every shared data update separately. The reason it is applied to just one assignment statement is that it protects updates to an individual memory location. Fortunately, this was the same case in the kmeans code present. Although the atomic construct provided the best performance among the four, it did not show any speedup.

**Privatized local update** This is the final version, leading to the best performance. In this approach the different clustering arrays were privatized and then parallelized. This enabled every thread to have a private copy of the arrays, thereby completely eliminating the contention. The element-wise reduction was performed
on the private arrays of each thread, which was outside the scope of \textit{parallel-for} but within the parallel region. This method produced excellent performance. Although it did not scale for smaller problem sizes, which is understandable from the lack of computations, it scaled for larger images. This was the version which was used for the final measurements.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Time(s) ($512 \times 512$ image 31 channels and 8 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical Section</td>
<td>36.04</td>
</tr>
<tr>
<td>Array of Locks</td>
<td>23.43</td>
</tr>
<tr>
<td>Additional Thread-id Parameter</td>
<td>15.32</td>
</tr>
<tr>
<td>Atomic construct</td>
<td>10.73</td>
</tr>
<tr>
<td>Privatized local update</td>
<td>1.034</td>
</tr>
</tbody>
</table>

Table 3.2: Time consumption of the various methods of parallelizing the Kmeans-algorithm, for \textit{(image with 512 \times 512 pixels and 31 channels and 8 threads)}.

To compare these methods more effectively, the time consumption was measured for every implementation. All the implementations are tested with an image consisting of 512x512 pixels and 31 channels. The code was executed with eight threads in all cases. The results have been shown in Table 3.1.2, and support the comments on the performance of the various versions in the previous paragraphs.

3.1.3 Smoothing

In the smoothing step, the code compares the cluster to which a pixel has been assigned, with the cluster, of all neighboring pixels. In case all neighboring pixels belong to the same cluster, which is, however, different from the cluster of the pixel currently analyzed, the pixel gets reassigned to the cluster of the neighboring pixels. This algorithm helps to smooth the image and avoid isolated pixels.
The smoothing function consists of a set of nested for-loops, and an additional loop for assigning the values of the smoothed labels to the original image array. There are three different shared-data arrays for storing the temporary labels, the smoothed labels and the original image. Of these three arrays, the temporary label array has to be made private to a thread to improve parallelization and avoid thread contention. The other two arrays are shared, as their array-index is such that there would be no contention between the threads.

First, the outermost loop controlling the number of smoothing steps to be performed was parallelized. However, it was found that the iteration loop always has a constant upper limit of five for any input size, and thus parallelizing this loop did not improve the performance. Next, the nearest inner-loop was parallelized. The loop for storing the output back to the main image file had to be made critical inside the parallel region, to avoid data races. But using critical section affected the performance. Since it was the part performing the reduction, it made sense to use just one thread to perform that operation. This was accomplished with the help of the single construct. The single construct is associated with the structured block of code immediately following it and specifies that this block should be executed by only one thread. It does not state which thread should execute the code block. The thread chosen could vary from one run to another. As long as the selected thread completes the operation, the other threads wait at a barrier. This construct worked well from the performance perspective.

To further improve the performance, different schedules were experimented and
the dynamic schedule for the parallel for work-share construct provided the highest performance and scalability. Similarly in the single construct, it was unnecessary to hold all the threads till the completion of the gathering process. So a no wait clause was added to it, to improve resource utilization.

3.2 Results and Evaluation

The OpenMP code was executed on zeola, a shared memory system consisting of eight dual core 2.6 GHz AMD Opteron processors with 64GB of main memory. The compiler used is the gcc-4.2.0. The tests were performed on images of size from 512x512 pixels with 31 spectral channels, to 1024x1024, 2048x2048, 4096x4096 and 8192x8192 pixels, all using 21 spectral channels. The images used in these experiments are in 'raw' uncompressed format. So the size of the largest image used was 1.5GB. Three tests were performed for every problem size. The results obtained across the three runs were consistent. Thus, the minimum time over the three runs has been used in the following analysis. Some of the few performance metrics for evaluating any application are:

- **Speedup:** The speedup is a measure that specifies, how much faster a parallel algorithm is, when executing it with \( p \) processes/threads for the same problem size and environment compared to its sequential counterpart. *Linear speedup* is achieved when the performance of an algorithm doubles with doubling the number of processes/threads.
• **Parallel Efficiency:** Parallel efficiency is defined as the speedup for the processes to the number of processes involved. It is given by $\frac{\text{Speedup}}{\text{number of processes}}$.

• **Scalability:** The scalability in parallel computing is defined as the capability of a parallel system to gracefully handle increasing workload. A parallel system whose performance also improves proportionally with increase in hardware resource is called a scalable system.

As described in the section 3.1, the image segmentation application has been classified into three main modules. The results are analyzed based on the three modules, and the application on the whole. The results have been extracted for both independent modules and the application on the whole.

![Figure 3.3: Performance analysis of the complete application.](image)

45
Overall the OpenMP code scales well for all the problem sizes. For smaller problem sizes, it does not scale a lot for more than four threads. The largest image showed a speedup of 5.9 for 8 threads and 9.7 for 16 threads. The convolution module has scaled well compared to the kmeans module. This problem of scalability rises mainly from the overhead introduced by the runtime environment when managing an OpenMP application. This behavior is also observed when the performance of a sequential application is compared with its OpenMP version using a single thread.

![Figure 3.4: Performance analysis of the convolution module.](image)

This overhead is caused because of the compiler. When a particular loop is parallelized using OpenMP, the compiler removes all the optimizations within the parallel region. So, when executing the OpenMP code using a single thread, it is less optimized compared to the sequential code. This results in the performance
deficiency. The graph depicting the overall performance of the application for all the problem sizes has been given in Figure 3.3.

![Graph showing overall performance](image)

**Figure 3.5:** Performance analysis of the kmeans-clustering module.

The convolution is the second most time consuming module of the application; kmeans-clustering being the first. The results have shown that the convolution has scaled well. For smaller problem sizes, this code section shows a speedup of around 5.9 for eight threads. This limited speedup is probably due to under utilization of processing resources for the small image. The largest image analyzed in this section shows a speedup of around 14.2 for 16 threads for the convolution. The executions for the convolution module have been depicted in the graph shown in Figure 3.4.

The kmeans clustering algorithm was the most time consuming module of the application. The limitations faced with this code section might, to a certain extent, be
a result of the current abilities of OpenMP. With the upcoming new OpenMP specification in version 3.0, which has support for task-based parallelism like in TBB [25] and Cilk [39], we would expect better performance for this code section. For the current implementation, for the largest image it provides a speedup of around 9.5 for 16 threads. The performance analysis of this module has been plotted in a graph as shown in Figure 3.5.

The smoothing module does not consume as much time as the other modules. So, it does not really affect the overall performance of the application. It shows a similar behavior as the other modules with a speedup of around 5.9 for 8 threads and a speedup of around 9.1 for 16 threads for the largest problem size. The main reason this module does not scale further is that there is not enough work for 16 threads
even for the largest image analyzed. The performance analysis for the smoothing module has been given in Figure 3.6.
Chapter 4

Parallelization of Partial Differential Equations (PDE)

Image processing as discussed in Chapter 1, is one of the major computationally complex applications of this computing era. In the last decade, there has been a new domain of image processing that has emerged, the so-called PDE-based image processing techniques. As per the authors in [16], the major difference between conventional image processing and PDE-based image processing is that, in PDE-based method the processing is formulated as a diffusion process (filtering) or as an optimization problem (segmentation and registration) both of which, can be solved with PDE techniques. This allows formulation of the outcome of the processing and the processing itself, which can lead to new possibilities to obtain the end results. Examples include isotropic diffusion of images, edge-detection, and image in-painting algorithms [40]. The Mumford-Shah [17] segmentation model used in [17] also uses an energy function to solve a defined segmentation problem. Similarly, variational
PDE models are also being used for image segmentation as described by Chan in [18].

In order to support parallelization of these image processing applications, this section investigates the OpenMP and hybrid parallelization of two types of PDE solvers in general: a time-independent PDE solver, and a simple solver for time-dependent problems such as the heat-equation.

### 4.1 ADCL Testing Framework (ATF)

This application solves a system of linear equations, which is the result of the central difference scheme used to discretize a time-independent partial differential equation. The system solved in this code is given by the equation 4.1

\[- \triangle u + 1000 \frac{\partial u}{\partial x} = f \quad \text{on} \quad \Omega = (0, 1) \times (0, 1) \times (0, 1)\]  

with Dirichlet boundary conditions

\[u = 0 \quad \text{for} \quad \partial \Omega.\]  

The right hand side is chosen such that the solution of the PDE is

\[u(x, y, z) = e^{(xyz)} \cdot \sin(\phi x) \cdot \sin(\phi y) \cdot \sin(\phi z)\]  

Although this code only represents a small benchmark, it is exemplary for many finite difference codes by solving the resulting system of linear equations using an iterative solver. The code uses the Transpose-Free Quasi-Minimal Residual method (TFQMR) algorithm. The computationally most expensive part in that solver is matrix-vector multiplications.
ATF has been originally developed as part of the Abstract Data and Communication Library (ADCL) [41], which is an application level communication library used for selecting the best performing implementation of a particular pattern for a given environment at run-time. The ADCL testing framework (ATF) provides a realistic test case for ADCL [20] to compare performance of different implementations of neighborhood communication on various computational environments.

The ATF parallelization has been done in two forms. The first being the OpenMP-based parallelization and the other being the Hybrid parallelization. The MPI version of the ATF application has been provided from the work done by (Parallel Software Technologies Lab) PSTL’s Alumni, Shuo Huang on his master’s thesis [20]. The same has been used here for generating the Hybrid parallel version.

The first step to create an OpenMP parallel version as discussed in Chapter 3 is to determine the time-consumption of individual modules. This helps us to get an insight of where the computation is performed, thereby helping to determine which module to start parallelizing. To determine this, the GPROF provided by gcc is used.

Gprof generates a profile for the sequential code, showing where the computation is high. The gpof flat-profile for the ATF-solver is given in Table 4.1. As seen in the profile, the matrix multiplication is the module that consumes maximum time. Though matrix multiplication performs the major computation part for the TMFQR algorithm, there are other computations which consume comparatively negligible time.

Thus, the main focus is on parallelizing the matrix multiplication module. This
<table>
<thead>
<tr>
<th>% time</th>
<th>Cumulative time(s)</th>
<th>Self (s)</th>
<th>no of calls</th>
<th>self s/call</th>
<th>total s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>76.44</td>
<td>0.97</td>
<td>0.97</td>
<td></td>
<td></td>
<td></td>
<td>ATF_.Matmul_blocking</td>
</tr>
<tr>
<td>7.09</td>
<td>1.06</td>
<td>0.09</td>
<td></td>
<td></td>
<td></td>
<td>ATF_.tfqmr_Cal_A_EQ_B_plus_z_mul_C</td>
</tr>
<tr>
<td>3.15</td>
<td>1.10</td>
<td>0.04</td>
<td></td>
<td></td>
<td></td>
<td>ATF_.tfqmr_Cal_A_EQ_c_mul_B</td>
</tr>
<tr>
<td>2.36</td>
<td>1.13</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
<td>ATF_.tfqmr_Cal_A_EQ_B_plus_C</td>
</tr>
<tr>
<td>2.36</td>
<td>1.16</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
<td>ATF_.tfqmr_Cal_A_EQ_B_sub_C</td>
</tr>
<tr>
<td>1.58</td>
<td>1.18</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td>ATF_.tfqmr_Cal_A_EQ_B_sub_z_mul_C</td>
</tr>
<tr>
<td>1.58</td>
<td>1.20</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td>ATF_.tfqmr_Cal_c_EQ_c_plus_A_mul_B</td>
</tr>
<tr>
<td>1.58</td>
<td>1.22</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td>ATF_.tfqmr_Cal_dq_EQ_dq_plus_z_mul_A</td>
</tr>
<tr>
<td>0.79</td>
<td>1.23</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
<td>ATF_Precon_Matset</td>
</tr>
<tr>
<td>0.79</td>
<td>1.24</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
<td>ATF_Reset</td>
</tr>
<tr>
<td>0.79</td>
<td>1.25</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
<td>ATF_Set</td>
</tr>
<tr>
<td>0.79</td>
<td>1.26</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
<td>ATF_Set_rhs</td>
</tr>
<tr>
<td>0.79</td>
<td>1.27</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
<td>ATF_.tfqmr_Cal_A_EQ_B</td>
</tr>
</tbody>
</table>

Table 4.1: The Flat-profile GRPOF output of the sequential code for ATF- solver.

The module has a set of matrix-vector multiplication operations defined with the help of a set of ‘for’ loops. Since the main matrix is hepta-diagonal, it is stored in seven independent arrays, each of them having five dimensions. Thus, the matrix-vector multiplication consists of seven independent ‘for’ loops for each diagonal, each loop running over all five dimensions of each diagonal, leading to nested ‘for’ loops. This means that the sequential order should be maintained and individual matrix-vector multiplications have to be parallelized.
As OpenMP allows incremental parallelization, the first step was to parallelize individual ‘for’ loops with ‘pragma omp parallel for’ directives. These directives spawn threads on their own and do not require a parallel region. However, this version did not perform as expected, since the overhead due to spawning and joining threads were too high.

As a second step a parallel region was created and the ‘pragma omp for’ workshare construct was used. This made all the ‘for’ loops to be parallelized with the team of threads created at the parallel region. This strategy improved the performance compared to the previous strategy. Since all the ‘for’ loops had the same boundary conditions, the operations were merged into one ‘for’ loop and the order of execution was maintained. This reduced the overhead of distributing threads across many workshare constructs. There was no real contention of shared data among the threads, so the arrays involved in the matrix-vector multiplication were kept shared. The iteration counters were privatized to perk up the performance. When different schedules like STATIC(default), DYNAMIC and GUIDED were applied to the ‘omp for’ workshare construct, the STATIC schedule provided the maximum performance comparatively. Since the other modules individually contribute to almost ten percent of the overall time, parallelizing them will not improve the overall performance. This is because the cost involved for parallelizing them will be more than the performance improvement obtained. So as a trade-off, those modules are kept sequential.

The hybrid version of this ATF-solver was created by merging the OpenMP version with the MPI version. The MPI version was developed in the work presented
in [20]. The thread-funnelled approach discussed in Chapter 2 is used for generating the hybrid version of this application, since most MPI libraries as of today are not completely thread safe. Thus, only one thread is handling the MPI communication.

### 4.2 Heat Equation Solver

The second PDE solver parallelized in this thesis solves the heat-equation [21]. The main goal within this thesis was to generate hybrid version, since an MPI version already existed. The Heat equation [21] also known as the diffusion equation is a representative partial differential equation of parabolic type. With properly chosen boundary conditions, the heat equation looks as shown in equation 4.4, where \( u \) is a function of \( x, y, z \) and \( t \). The name of the equation comes from the fact that it describes the propagation of temperature in a three-dimensional body at position \((x, y, z)\) at time \(t\). Thus, \( u \) is a real valued function of four variables \( x, y, z, t \).

\[
\begin{align*}
    u_{xx} + u_{yy} + u_{zz} &= u_t \\
    \text{where } u_t &= \frac{\partial u}{\partial t} \\
    u_{xx} &= \frac{\partial^2 u}{\partial x^2} \\
    u_{yy} &= \frac{\partial^2 u}{\partial y^2} \\
    u_{zz} &= \frac{\partial^2 u}{\partial z^2}
\end{align*}
\]

The heat-equation solver used in this thesis uses an explicit method, which is based on an equation derived using a Finite-Difference approach.

Like the other applications, similar steps were applied for this application as well. First, to get a clear picture of the maximum time consuming modules, the flat profile
needs to be generated. Through the GPROF tool the flat profile was generated as shown in Table 4.2.

<table>
<thead>
<tr>
<th>% time</th>
<th>Cumulative time (s)</th>
<th>Self (s)</th>
<th>no. of calls</th>
<th>self (s/call)</th>
<th>total (s/call)</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>92.00</td>
<td>1.25</td>
<td>1.25</td>
<td>3949</td>
<td>0.00</td>
<td>0.00</td>
<td>update_interior</td>
</tr>
<tr>
<td>4.42</td>
<td>1.31</td>
<td>0.06</td>
<td>5702356</td>
<td>0.00</td>
<td>0.00</td>
<td>central_diff</td>
</tr>
<tr>
<td>2.94</td>
<td>1.35</td>
<td>0.04</td>
<td>3949</td>
<td>0.00</td>
<td>0.00</td>
<td>update_faces</td>
</tr>
<tr>
<td>0.74</td>
<td>1.36</td>
<td>0.01</td>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
<td>get_mesh_mem</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>3949</td>
<td>0.00</td>
<td>0.00</td>
<td>check_done</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>3948</td>
<td>0.00</td>
<td>0.00</td>
<td>switch_steps</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>apply_bc</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>find_bcnodes</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>get_bcnodes_mem</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>get_coords</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>get_datatypes</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>get_dimlist</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>get_domain</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>get_offset</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>get_time_mem</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>read_input</td>
</tr>
<tr>
<td>0.00</td>
<td>1.36</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>1.35</td>
<td>update_solution</td>
</tr>
</tbody>
</table>

Table 4.2: The Flat-profile GPROF output of the sequential code for Image segmentation.

As shown in the profile above in Table 4.2 the `update_interior` function of the heat equation solver consumes most of the time. This naturally means that it is the module that needs parallelization. The `update_solution` function invokes the `update_interior` function. Though the major part of the computation in `update_solution` was carried out with a `do` loop; since the `update_interior` function is a separate module, it can be parallelized with OpenMP. The computations of the `update_interior` function are collected together in a set of nested
'for' loops. For parallelizing with OpenMP, a parallel region was created and the first 'for' loop in the set of nested 'for' loops were parallelized with \texttt{pragma omp for} workshare construct. This could have been done directly with the \texttt{pragma omp parallel for} directive, but both will behave the same way in this case, as there is only one work-share construct inside the parallel region. The data organization (shared or private) has to be very specific to ensure data consistency. By looking at the computations performed within the nested 'for' loops, certain variables were completely independent and had no chance of creating thread contention. These variables were made private to each thread. The boundary conditions of the loops were calculated ahead of the parallel region. In this case, these variables needed to be made \texttt{firstprivate}. Variables declared as \texttt{firstprivate} \cite{22} are private variables which are pre-initialize with the value of the variable with the same name before the construct. Thus, this allows making the boundary conditions private to each thread and computed ahead of the parallel region. Since this application was already MPI parallelized, adding OpenMP parallelization to it made it a Hybrid-parallelized code. Even in this application, the thread-funneled approach was employed for generating the hybrid-parallel application. So, MPI does the data parallelism for this application and OpenMP performs the task parallelism.

4.3 Results and Analysis

The results of both the applications introduced above have been discussed in this section. The OpenMP and MPI version of ATF-solver was executed on \textit{zeola}, a shared memory system consisting of eight dual core 2.6 GHz AMD Opteron processors with
64GB of main memory. This OpenMP version of the code used gcc-4.2.0 compiler and the MPI version used MPICH [42]. The MPI version was developed in Parallel Software Technologies Lab (PSTL) is presented in [20]. The hybrid version cannot be executed on zeola, so it was executed shark cluster [43], which consists of 24 single processor, dual core AMD Opteron nodes running at 2.2 GHz, each node equipped with 2GB of main memory. The MPI version of the code was also executed in the same cluster for comparison reasons. The executions for the heat-equation solver were performed on the shark cluster. For the speedup tests, the problem sizes were fixed for both applications.

To verify scalability of the code, the problem size was increased accordingly with increasing number of threads or processes. In the ATF-solver one thread or process executes a problem size of $32 \times 32 \times 32$ and two threads or processes executes $64 \times 32 \times 32$ and four threads or processes executes $64 \times 64 \times 32$ and so on. The problem size used for measuring the execution time was $128 \times 128 \times 128$ for all the versions of ATF. In the case of the heat-equation solver the problem size used for measuring the execution time was $120 \times 120 \times 120$.

### 4.3.1 ATF- OpenMP parallelization

The graph depicting the OpenMP scale-up tests on zeola shared memory system is given in Figure 4.1. As shown in the Figure 4.1, the OpenMP version of the code scales similarly to the MPI version on the same machine. One of the reasons for the behavior could be the inherent sequential nature of the application. As seen in the profile generated for ATF in Table. 4.1, all the computations were boiled down to
Figure 4.1: The Scalability of both the MPI and OpenMP versions of ATF for a given set of problem sizes.

the matrix-vector multiplication module of the application. This is in a way good, as the parallelization becomes simple with all the major computations are collectively present at one place. On the other hand, it eliminates the parallelization of the other modules, as cost of parallelization becomes more than the performance obtained. Figure 4.2 shows the results of the speedup analysis for a constant problem size of $128 \times 64 \times 64$. The code provided a speedup on $zeola$ of around three for four processes/threads. However, for more than four processes/threads, the speedup did not further improve. The main difference between the two parallel versions was that, the execution time consumed for one thread was higher than that consumed for one
4.3.2 ATF - Hybrid parallelization

The hybrid version of the code performed well compared to the OpenMP and the MPI versions. The scalability of the hybrid and MPI versions of the ATF-solver has been depicted in the graph shown in Figure 4.3.

Figure 4.2: The execution time of both the MPI and OpenMP versions of ATF for a constant problem size.
The MPI measurements were made for measuring scalability using the problem sizes specified in the section 4.3 with one process per node and two processes per node. The hybrid measurements were made with two threads per process per node. The hybrid version proved to be highly scalable, as the time consumption remained fairly consistent for all the problem sizes. There was a small difference in the execution time between the one process and the two processes case. This is due to the fact that the communication overhead peeps in. Overall the hybrid version scales well. The execution time consumed for a constant problem size(128 × 64 × 64 ) for both the hybrid and MPI versions has been shown in the graph in Figure 4.4.

Figure 4.3: The Scalability of both the MPI and Hybrid versions of ATF for a given set of problem sizes.
Figure 4.4: The execution time of both the MPI and Hybrid versions of ATF for a constant problem size.

The speedup for the hybrid version was very good, showing a speedup of 1.58 for one process with two threads and around 7.7 for four processes with two threads. This is almost equal to linear speedup. Though the MPI code shows a good speedup of around 3.7 for four processes, the overall time consumption is reduced in the case of the hybrid code. Though the speedup was not good for the OpenMP version, adding the OpenMP statements to the MPI version resulted in a performance improvement.
4.3.3 Hybrid Parallel Heat-equation solver

The heat-equation solver on hybrid parallelization was consistent without any data-races in the result. However, the application on the whole showed an unstable behavior. The time consumption varied drastically for every execution. The execution time for the MPI version of the code, with one process per node, two process per node and hybrid version of the code with two threads per process per node with a problem size of $120 \times 120 \times 120$ has been shown in Figure 4.5. This could be due to process-thread affinity problem, wherein the allocation of the tasks to the threads becomes sub-optimal, which in turn can cause decrease in the performance. Solving the thread/process affinity problem was, however, outside the scope of this thesis.

![Figure 4.5: The execution time of both the MPI and Hybrid versions of Heat-equation solver for a constant problem size.](image-url)
Chapter 5

Summary

This study is about evaluating parallelization strategies for some image processing applications. The main goals of this thesis were to parallelize these applications and evaluate their performance by comparing them with their sequential and MPI counterparts. The different applications that were parallelized are: a texture-based segmentation of thyroid Fine Needle Aspiration Cytology (FNAC) images, PDE-based solver for time independent problems and a PDE solver for time-dependent problems.

The FNAC image segmentation application was parallelized with OpenMP; the other two applications were parallelized with hybrid (MPI+OpenMP) parallelization. The different approaches employed for parallelization were discussed. The results obtained show that the first application resulted in good performance improvement, and to a larger extent scaled well. The second application showed almost linear speedup for the hybrid version. The OpenMP version of the same application lacked performance of around 40% due to compiler overheads.
The third application, PDE-based heat-equation solver did not perform well, as there was some performance related un-stabilities in the application. One of the reasons could have been process/thread affinity problems. Solving the process/thread affinity problem can be a potential area for future research, since it is expected to be highly eminent with increasing number of cores on a CPU. A strategy to overcome this problem will be a good extension for this thesis. A further extension to this work could be in applying parallelization to reading and writing compressed images similar to work described in [14]. This could further help to improve the performance of image analysis applications, since I/O has major factor in handling very large amounts of data. While MPI provides some support for parallel I/O, OpenMP in its current version does not provide any support for these operations.
Bibliography


