A Thread-Based Parallel Programming Library for Numerical Algorithms

by

Maitham Makki Alhubail

Submitted to the School of Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Computation for Design and Optimization

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Signature redacted

Author ............  \\

Signature redacted

Certified by ............  \\

Signature redacted

Accepted by .................

Nicolas Hadjiconstantinou
Co-Director, CDO SM Program
A Thread-Based Parallel Programming Library for Numerical Algorithms

by

Maitham Makki Alhubail

Submitted to the School of Engineering on May 9, 2014, in partial fulfillment of the requirements for the degree of Master of Science in Computation for Design and Optimization

Abstract

This thesis presents a new simple lightweight C++ thread based parallelization library, intended for use in numerical algorithms. It provides simple multitasking and task synchronization functions. The library hides all internal system calls from the developer and utilizes thread pooling to provide better performance and utilization of system time and resources. The library is lightweight and platform independent, and has been tested on Linux, and Windows. Experiments were conducted to verify the proper functionality of the library and to show that parallelized algorithms on a single machine are more efficient than using the Message Passing Interface (MPI) using shared memory. In the opinion of several researchers who have used this library, the parallelized code is more easily understood and debugged than MPI. The results of initial experiments show that algorithms are as efficient or better than those using MPI.

Thesis Supervisor: John R. Williams
Title: Professor
3 Experiments

3.1 Basic Functionality Test ........................................... 31
  3.1.1 Simple Manipulation of Array Elements ...................... 32
  3.1.2 Parallel Merge Sort Implementation .......................... 33

3.2 MPI Comparison ..................................................... 33

3.3 Numerical Experiments ............................................. 36

4 Result Analysis ...................................................... 39

5 Conclusion and Future Work ......................................... 41

  5.1 Conclusion ....................................................... 41
  5.2 Future Work ...................................................... 42

A Tables ................................................................. 43

Bibliography ............................................................. 47
List of Figures

1-1 A typical server application processing requests in parallel using multiple threads. ......................... 12
1-2 A typical structure of parallelized numerical computation code .... 12

2-1 Demonstration of simple use of C preprocessor directive .......... 16
2-2 A typical Sleep Based Thread Pool Implementation ............... 17
2-3 Thread Suspension Process in Windows Platform .................... 19
2-4 Thread Notification Process in Windows Platform .................... 19
2-5 Locking Based Thread Suspension in UNIX-Linux ..................... 21
2-6 Locking Based Thread Notification in UNIX-Linux ................... 21
2-7 Illustration for scheduling a job to be executed by a worker thread .. 22
2-8 Demonstration of how Execution Queues are implemented ......... 23
2-9 Implementation work flow of Prerequisite Job Functionality ........ 24
2-10 Implementation Illustration for Function Locking Management .... 25
2-11 Blocking for A Specific Job ......................................... 26
2-12 Illustration of how the Library Blocks until all jobs are done ...... 28
2-13 Getting The number of Queued and Running Jobs ................. 29

3-1 Dividing an Array into 4 Equal Parts to be Processed by 4 Execution Threads ........................................ 32
3-2 .................................................................................. 33
3-3 Execution Time of Parallel Merge Sort ............................... 34
3-4 Execution Time of Parallel Merge Sort Comparing this library and MPI 35
3-5 Execution Time of Parallel Merge Sort Comparing this library (with Tuning) and MPI ........................................... 35
3-6 Dividing the 3D Unit Cube domain to 4 sub-domains .............. 37
3-7 Comparison of Execution Time for 100 Jacobi Solve Iterations (for 3D Poisson Problem) between MPI and this Library ............. 38
List of Tables

A.1 Processing Time for 80000000 Array Elements ............... 43
A.2 Merge Sort Time for 80000000 Double Array .................. 44
A.3 Merge Sort Time Comparison between this Library and MPI for 20000000 Double Array ........................................ 44
A.4 Merge Sort Time Comparison between this Library and MPI for 80000000 Double Array in a high end server machine ................. 44
A.5 Comparing Time for 100 Jacobi Solve Iterations Time for 3D Poisson Equation between this Library with 4 worker threads and MPI with 4 Processes ........................................ 45
Chapter 1

Introduction and Literature Review

1.1 Introduction

Multiprocessor and multi-core computers used to be owned by big companies and were not easily accessible and available to individuals. In the past, parallel programming was a skill that programmers need to acquire to be able to parallelize their code and make use of multiple processors. The Message Passing Interface (MPI) API made parallel programming and the utilization of multiprocessor machines and computer clusters much easier by providing a set of functionalities for parallel processing, data exchange and synchronization [1]. Even though MPI provides comprehensive parallel computing capabilities, it is too complex to be used by most developers or engineers willing to utilize the multi-core machines and parallelize the whole or a part of an algorithm [2]. The fact that multi-core machines are now available and accessible to everyone is the main driver behind this thesis. Nowadays, even the cell phones that we carry in our pockets are powered by a multi-core processors! This thesis presents a simple, easy to use and platform independent library that simplifies parallel computing for developers and efficiently allows them to utilize multi-core computers.

There are two main categories of parallel computing. The first type is where no synchronization is needed. For example, a server processing users requests. Parallel
A typical server application processing requests in parallel using multiple threads.

A typical structure of parallelized numerical computation code computing here allows the server to create one task per user request and assign it to a processor or a processing core. When the task is done, the core is ready to handle the next task (i.e. the next user request). Figure 1-1 illustrates this type of parallel programming.

The other type of parallel programming is where a task is split into smaller sub-tasks and then synchronization takes place among the subtasks to share or exchange data and further continue processing until the original task is done. This type of parallel programming is common in the numerical computation world. Many numerical algorithms involve tasks that can be broken into subtasks, which then can be executed in parallel in a multi-core machine. Figure 1-2 illustrates this type of parallel programming.

While the library presented here can be used for either type of parallel programming style; the main focus is on parallel computing where synchronization needs to take place. The work done here is mainly intended to be used to simplify the par-
allelization of numerical algorithms. It is designed to give engineers and developers easy access to the many computing cores in their computers. While this library is not intended to be used to parallelization code in computer clusters, it provides excellent parallelization and synchronization functionalities for parallelizing code on a single multi-core machine.

1.2 Literature Review

The main 'workhorse' of traditional High Performance Computing is the Message Passing Interface API. This provides a rich set of message passing and parallelization functionalities that can be used both at the single machine level and at the computer cluster level. However, MPI is complicated and difficult to learn and therefore might not be the right choice for a developer who is planning to code an algorithm and quickly utilize the multi-core CPU. Efficient MPI programming requires good programming and debugging skills [1, 2].

Another excellent set of C++ libraries are the Boost libraries [3]. The Boost libraries are comprehensive and rich, but are not simple enough to be used immediately by a developer who does not have sufficient understanding of the structure of the Boost library and its internal data structures [4].

A third library is the Concurrency and Coordination Runtime (CCR) library by Microsoft [5]. CCR provides functionalities that allow the utilization of multi-core machines by a fairly simple effort. CCR’s main problem is that it is written in .NET and most developers prefer C or C++ for easier wrapping with other languages. Moreover, the CCR is written in .NET and is limited to the Windows platform and does not run on the UNIX based platforms [5]. It was concluded that a good compromise would be to develop a library that is written in C++, such as Boost, with the simplicity of Microsoft’s CCR. Following this architecture, I was able to provide simple function calls for both parallelization and synchronization that provide clean abstractions that are easily understood by developers.
Chapter 2

Technical Approach

The technical details of the parallelization and synchronization features of this library are now described. This library is designed to be used by engineers, who have limited knowledge about how operating systems handle threads, and this highly influenced the interface design of this library.

2.1 Thread Based

Since my aim is to provide parallelization at the single machine level, the choice was made to use execution threads over processes. Threads within a process share the memory space and this eliminates the need for complicated sharing data communication pipes or segments of shared memory, controlled by semaphores. Moreover, threads are much faster to create and spawn than processes. In addition, thread synchronization and life cycle control is simple and does not need complicated coding.

2.2 Platform Independent

One of the main features of this library is that developer are be able to use it regardless of their chosen operating system. The library should be perfectly and efficiently functioning under the Windows or any other UNIX-Linux based operating system. There are two simple technical approaches to achieve this platform independence goal;
the first is to have different files for the code and let the compiler choose which to compile based on the type of the operating system. The other approach is just to use simple C preprocessor directives to compile the section that is relevant to the used operating system. Since the implementation of this framework is simple, I choose to use simple C preprocessor directives that identify the OS and simply include the right set of lines of code. The code of this library is actually very generic and the only part that needed those directives is where the management of the threads is located. Figure 2-1 shows a simple section for creating a thread in Windows and other UNIX-Linux based platforms using C preprocessor directives.

### 2.3 Thread Pooling

This library, just like many other libraries, uses thread pools where threads are created and kept ready to perform tasks. Even though the creation of threads is not as expensive as creating processes, in a high performance computing environment that small amount of time needed to create a thread can be utilized to execute lots of computing instructions. Moreover, thread pooling can help minimizing cache misses if a thread is assigned a set of tasks on the same data; this is true provided that the thread is being executed on the same processing core, which is not necessarily the case.

C++ does not by default have a standard way of creating and maintaining a pool of threads. The concept is eventually the same across high level programming languages but it is really implementation specific and different programmers use different ways to implement thread pooling behavior. Following sections describe how thread pooling is implemented in this library.
As stated earlier, this library is written to be compatible with both the Windows and UNIX platforms. And given that thread creation and handling is different between the two platforms, the following explains how thread pooling is done in each of the two platforms.

In this library, 2 modes of thread pooling are provided; sleep based or suspension based.

### 2.3.1 Sleep Based Thread Pool

The first mode is the direct mode where threads are created asking for tasks to execute. If tasks are provided, then the threads execute these tasks and if no tasks exist, the threads go into a sleeping state for a specific amount of time. After the sleeping state is over, threads ask for tasks to execute again and if none is available they go into the sleep state again and the cycle goes on. So, as long as no tasks exit in the execution queue of the system, the threads spend their lives sleeping and asking for tasks to execute. In this mode, the implementation procedure is almost the same in the Windows and UNIX platforms. Figure 2-2 shows the simple life cycle of the sleeping based thread pooling mode.

The implementation code of this method is very simple. It is nothing but the use of "Sleep in Windows" and "sleep in UNIX" OS provided functions to put a thread from execution state to a system sleep state.
2.3.2 Suspension Based Thread Pool

The second thread pooling mode this library provides is based on threads suspension instead of threads sleep. Various components are needed to successfully implement thread pooling with this technique. Also, the level of implementation difficulty varies a lot between the Windows and UNIX platforms.

Windows Platform Implementation

The Windows platform provides built-in OS functions "SuspendThread and ResumeThread" that simplify this task. However, in the UNIX based platforms a custom created thread-suspension method is needed.

In this library, this mode of thread pooling is implemented as follows. A boolean variable is created per thread as a flag. A flag of True indicates to the scheduling system of the library that the thread is suspended and a flag of False simply means that the thread is running. Because a thread pool will have more than one thread, the scheduling system of the library keeps these flags in a boolean array where each entry on the array belongs to a specific thread. Because of the way this library implements this thread pooling technique, each thread will be setting its entry to True (meaning thread is suspended) in the flags array and the scheduling system will set it back to False (meaning thread is running), the flags array becomes a critical section which needs a locking mechanism to protect it from getting into an inconsistent state. So, a boolean array and a lock are the first components needed. For the Windows platform, the components are almost ready to implement this thread pooling mode. However, for the UNIX platform, additional locks are needed. These lock are equal to the number of the threads in the pool. Again, the implementation of this library keeps these lock in an array of locks for better organization. Now, for the Windows platform, the thread pooling with suspension is created as follows. A thread is initially created and asks for a task to execute. If a task is provided, it gets executed otherwise the thread puts itself into suspension mode by calling SuspendThread with its own thread identifier. But, before that, the thread updates its suspended flag to True in the flags
array. Of course that is done after requesting a lock on the flags array, updating its flag and releasing the lock. Figure 2-3 explains the process so far. So, after a thread is suspended it needs to be notified whenever tasks are available for execution. This task is achieved by the library scheduling system. It basically loops over the flags array and calls ResumeThread for any thread whose flag is set to True. Of course, the scheduling system will request a lock on the flags array before setting flags back to False and calling ResumeThread. Figure 2-4 illustrates the thread notification process.
UNIX-Linux Platform Implementation

For the UNIX platform, the process is similar except that a trick needs to be implemented to mimic the behavior of the SuspendThread and ResumeThread. This is done by the thread assigned lock mentioned earlier. And the process goes like this. After a thread is created it asks for a task to execute and if no tasks are available, it puts itself into suspend state by first obtaining a lock on the flags array, updating its flag to True, releasing the flags array lock, requesting a lock twice on its own assigned lock and finally releasing its own lock. The first lock request will pass whereas the second lock request will be blocked and the last unlocking releasing lock statement will not be execute. So, the thread is now suspended waiting on the second lock request statement. It might be obvious now, that the thread is suspended until the library scheduler thread calls an unlock of this threads lock causing it to resume execution, reset its lock (by calling the last unlocking statement) and going back to the cycle of executing a job or going back to the artificially created locking suspension state. The figures 2-5 and 2-6 will help explaining the idea of the locking based thread suspension method, which is implemented for the UNIX platform in this library. Figure 2-5 shows the steps a thread goes through until suspension and figure 2-6 shows how the scheduling library notifies and releases suspended threads. A final word about the thread pooling modes; these modes are not provided just for choice or convenience. It is found that choosing one mode over the other can highly impact the performance of library. Factors such as operating system type, tasks nature, task average execution time and tasks arrival rate are important factors to consider when choosing a thread pool implementation mode over the other.

2.4 Job Scheduling

Now that the scheduling system have created its thread pool with threads ready to execute tasks, it is now time to explain how tasks get scheduled and executed. To schedule a job, developers need to provide the library with 3 important parameters. Those are a thread work function, a call back function and the data that the supplied
Figure 2-5: Locking Based Thread Suspension in UNIX-Linux

Figure 2-6: Locking Based Thread Notification in UNIX-Linux
work and call back functions will operate on. Both the work and call back function must have a standard signature of accepting a void pointer as an input parameter and returning nothing (void). The void input parameter can be casted to anything inside the work and call back functions. After a successful job scheduling call, the system internally creates what is called a thread work structure. That structure stores the work and call back function pointers and the data pointer. Each created thread work structure is then inserted into an internal queue which holds the tasks to be executed. If the created thread pool is of mode 2 (suspend-notify), any suspended thread gets notified that the jobs queue contains tasks that need to be executed. Note that if all the threads are already executing tasks a notification will not be needed. This is because the threads will never go to sleep or suspension mode unless the job queue is empty. This is obvious from the explanation of the thread pooling section stated earlier. Figure 2-7 below shows the work flow of scheduling a job.

2.5 Job Execution Queues

The idea of the job execution queues is fairly simple. Suppose that you have a pair of work and call back functions that need to be executed for a set of data. In this case, the library provides a shortcut of achieving this rather than calling the job scheduling functions for each element in the data set. The library can create an execution queue for any work and call back functions and provide a simple way of just inserting data element into that queue. When the library creates the execution queue, the queue can
be chosen to be either in auto execution mode or in queue mode. In the auto execution mode, any data inserted to the execution queue will immediately be scheduled and executed by the next available thread. Whereas in the queue mode, the inserted data is accumulated in the execution queue until the developer explicitly sets the queue in execution mode. The choice between creating an execution queue in auto execution mode or queue mode really depends on the application. In some cases, it might be better to fill the execution queue with data (tasks) then ask the library to execute all (put the queue in execution mode). However, in other cases have the jobs executed as soon as they enter the queue might be better. Figure 2-8 below shows the steps needed to create an execution queue and how queued jobs get executed.

### 2.6 Prerequisite Job Functionality

The situations where some tasks cannot start execution before other running tasks complete arise a lot during the development of numerical simulators. An example of that is the need to write output to disk after the last computation task is done. The same case is applicable to the need of out of order execution of tasks. The library allows developers to achieve this functionality by returning a job identifier for each scheduled tasks. This job identifier can later be used to identify a prerequisite job for any job that needs to be scheduled in upcoming job scheduling calls. When a job
is scheduled with a prerequisite, the library scheduler treats it as any other job by placing it in the scheduled jobs queue. The difference comes when working threads ask for a job. Before assigning the next job to the asking thread, the scheduling system first makes sure that the prerequisite job already finished execution, and if not that job is skipped and the next ready to execute job is given to the asking thread. The work flow of job prerequisite functionality is clearly illustrated in figure 2-9.

2.7 Managing Function Locking

When developing code, especially for numerical algorithms, developers face cases where a function cannot be executed concurrently by more than one thread due to the execution of a critical section where a shared data object is accessed or modified. The library allows developers to assign a lock number for both the work and the call back functions when scheduling a job. In fact the library maintains a pool of thread locks that are ready to be used when scheduling any job. The number of thread locks maintained in the pool can be specified when initializing the library, however, the library by default creates a minimum number of thread locks that are ready for use. This number is actually equivalent to the number of worker threads in the initialized thread pool. The way developers deal with thread locks in this library is really simple. Each lock is represented as an integer value that can be used as a lock identifier when scheduling a job. Any scheduled jobs sharing the same lock number will not be allowed for parallel execution. Instead, a lock acquisition is required before going into execution. The concept of locking is applicable to both the work and call back
functions. Of course, developers are allowed to specify different locks for the work and call back functions for a single job. The internal representation of thread locks is nothing but an array of locks that are initialized and kept ready for use. Representing thread locks as integer values hides the details of creating and managing thread locks from the developer, allowing him or her to focus on the computation tasks rather than the implementation part of managing thread locking steps and the differences those steps might have among different OS platforms. Figures 2-10 below illustrate the different implementation details of thread locks in this library.

### 2.8 Synchronization

The need of synchronization is an essential part of any iterative numerical algorithm. It is needed in any numerical simulator between time steps. It is essential to easily give developers the ability to pause the execution of the main thread by either waiting for a specific job or by blocking until all scheduled tasks are done; meaning that, all working threads are back to the thread pool either in sleeping or suspension states.
2.8.1 Blocking for a Single Job

Blocking for a specific task is simply achieved using a function call with the job identifier, the identifier that the library returned when the job was scheduled. Internally, the library keeps a record of all running and queued tasks. So, when blocking for a specific job is requested, the library tries to find the supplied job identifier in the running and queued jobs lists. If it is not found, the main thread resumes, otherwise, the library goes into a loop of finds and sleeps. The loop will break when the required job identifier is finally cleared by a worker thread. Figure 2-11 below illustrated how blocking for a specific job is implemented in this library.

2.8.2 Blocking for All Jobs

Blocking for the execution of all scheduled tasks is achieved in a way that is different than blocking for a single scheduled job. The idea is fairly simple. When developers
request for blocking until all running and queued jobs are done, the library suspends
the execution of the main thread until all job queues are empty and all running threads
are back to the library thread pool. From the description above, it is clear that the
main thread will be accessing the internal jobs queue to make sure that it is empty.
And since, the internal jobs queue is also accessed by the working threads, the main
thread will have to request a lock on that resource before checking if it is empty or
not. It is also clear that the main thread will go into a loop of locking, checking, and
unlocking for the internal jobs queue. The required locking and unlocking for the jobs
queue might be a performance impacting factor for this library. This is because the
main thread might cause delays when the worker threads ask for jobs to execute from
the jobs queue. For this reason, the library provides a tuning parameter to be set by
the developer when requesting for a complete block of execution for synchronization
purposes. That tuning parameter controls the frequency at which the main thread will
be performing the locking and unlocking on the internal jobs queue. Developers can
find the best setting by running different experiments or by their feeling on the rate at
which worker threads are fetching and completing scheduled tasks. Experiments have
shown that this tuning parameter plays a major factor on the overall performance of
this library when it comes to blocking execution for synchronization points. Figure
2-12 illustrate how the library achieves execution blocking for synchronization.

2.9 Getting Statistics

It is important for the library to provide developers with functions that allow them to
get the number of running and queued job at any point of time. The library provides
two simple functions, one to count and return the number of running jobs and another
to count the number of tasks waiting in the jobs queue. The implementation of both
functions is very simple. The library internally maintains a list of job identifiers that
are being worked on by the worker threads. So, getting the number of running jobs
is simply done by counting the number of job identifiers in the running jobs list and
returning that to the developer. Of course, the running jobs list is a critical section
that is accessed by worker threads and the main thread. For that reason, the main thread locks the running jobs list before counting the number of job identifiers in the list. Similarly, counting the number of queued jobs is done by returning the size of the internal jobs queue. Again, since the internal jobs queue is a critical section that will be accessed by the worker threads and the main thread, it needs to be locked before its size is returned to the developer as the number of queued jobs. Figure 2-13a and 2-13b show the simple work flow the library follows to count the number of running and queued jobs.
Figure 2-13: Getting The number of Queued and Running Jobs
Chapter 3

Experiments

After the library was fully implemented and built, some of experiments were done to ensure the proper functionality of all library features. One important thing to be tested was to ensure that utilizing this library to parallelize some code does not add a lot of overhead. This simply means that the library should be lightweight and should not require a lot of system resources to function properly. This is really considered as a basic functionality test. More experiments were conducted to compare the runtime of parallelizing code with this library and doing the same using MPI. The last experiment was numerical and was basically done to compare the runtime of this library and MPI when solving the 3D Poisson equation using Jacobi iterative scheme.

3.1 Basic Functionality Test

Two test cases were developed to measure the success of the basic functionality of this thread based library. The first test case is a simple manipulation of the element of a large double array where synchronization is not needed. The other test case is implementing a parallel version of the famous merge sort algorithm using this library.
3.1.1 Simple Manipulation of Array Elements

In this experiment a simple array of doubles is divided into parts that are as many as the number of the worker threads in the library thread pool. For this test case, a fixed double array of size 80000000 was created and then processing its elements is done with different number of threads. Figure 3-1 clarifies this test case with a scenario where 80000000 elements of the array are assigned to 4 threads. The test was conducted with 1, 2, 4, 8, 12, 16 and 20 threads. Figure 3-2 shows the execution time of each run. The test was executed in a machine that has a single Intel i5-2320 quad core CPU. So, the expected results with this trivial test is that execution time should decrease significantly between 1 and 4 threads and after that it should not vary much or may actually start to increase as the threads start to interleave and the context switching increases. Please note that the work done by this test case may not be that realistic and hardly happens in real numerical algorithms. This is because the problem here is inherently parallel and the computations done to each array element does not depend on any other elements on the array. However, this test can tell something about the reliability and the basic functionality of this library.

Figure 3-2 shows the results of this test case. Notice that the results are exactly as expected. So, running the same test on a machine with more processing cores will produce similar result where the curve in the chart starts to have very small changes in values as the number of threads exceeds the number of physical core in the machine.
3.1.2 Parallel Merge Sort Implementation

Another basic functionality test case was done on the famous merge sort algorithm that was parallelized with this library. This test case is a much better test than the previous test as it involves synchronization points between the various parts of the algorithm. Figure 3-3 shows the execution times in seconds. Again the test was done on the same Intel i5-2320 quad core machine. The results, as expected, show the sharp decline in the execution time until the maximum number of cores is reached and after that, the execution time smoothed.

3.2 MPI Comparison

The next set of experiments were done to compare running an MPI parallelized algorithm (with shared-memory in a single machine) with the same parallelized with this thread based library. The algorithm of choice was the parallel merge sort that was used in the previous experiment. The expected result is that using this library should result in a code that is easier to read, understand and that can be written by any engineer. This MPI comparison test was done without applying much tuning to this thread based library. The only thing that was applied, due to code simplicity,
is the base case at which the sorting process ends and the merging process starts. Figure 3-4 shows the execution time of applying merge sort to a double array of size 20000000. The results above are generated from the same machine that was used to run the previous test cases. Notice that even though the MPI processes where communicating over shared memory, having multiple threads sharing the same memory space of one process is a better choice.

The last test was done to on a server machine to compare a tuned merge sort algorithm, written with this library, with the same MPI based merge sort used in the previous test case. The code was tuned to be able to achieve cache optimality during the sorting process. This basically was controlled by the maximum number of elements that can be sorted with the least number of cache misses for each worker thread. Picking the right number for the sorting base case made a significant impact on the overall performance of sorting process. Adjusting that parameter changes the number of tasks to be conducted by each worker thread. The smaller the base case, the more tasks the worker threads will have. The best value will be a balance that will satisfy the reduction of cache misses as well as the best utilization of worker threads. Figure 3-5 shows the results of the tuned thread based merge sort with the MPI based one.
Figure 3-4: Execution Time of Parallel Merge Sort Comparing this library and MPI

Figure 3-5: Execution Time of Parallel Merge Sort Comparing this library (with Tuning) and MPI
3.3 Numerical Experiments

Now that the previous experiments showed the proper functionality of this thread based library and that using this results in a code that is simpler than using MPI, and also performs better; it is now time to conduct some testing with numerical algorithms. A typical representative choice will be a 3D Elliptic problem. For simplicity, I chose to solve the 3D Poisson equation with Dirichlet boundary conditions applied on all boundaries using Finite Difference. The solution will be calculated using the Jacobi iterative method.

So, the equation to solve is as follows:

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f(x, y, z)
\]

The domain is chosen to be the unit cube \([0, 1] \times [0, 1] \times [0, 1]\). The Dirichlet boundary conditions are as follows:

\[
\begin{align*}
    u(0, y, z) &= u(1, y, z) = 0 \\
    u(x, 0, z) &= u(x, 1, z) = 0 \\
    u(x, y, 0) &= u(x, y, 1) = 0
\end{align*}
\]

In 3D, the finite difference stencil for the approximation of the Laplace operator is given as:

\[
\frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} + \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2} + \frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{\Delta z^2} = f(x, y, z)
\]

Using the Jacobi method and a uniform spacial discretization in the 3 directions, that is:

\[
\Delta x = \Delta y = \Delta z
\]
the iterative solution is of form:

\[ u_{i,j,k}^{n+1} = \alpha \left[ (u_{i-1,j,k}^n + u_{i+1,j,k}^n) + (u_{i,j-1,k}^n + u_{i,j+1,k}^n) + (u_{i,j,k-1}^n + u_{i,j,k+1}^n) - f_{i,j,k} \right] \]

Where:

\[ \alpha = \frac{\Delta x^4}{6\Delta x^2} = \frac{\Delta x^2}{6} \]

So, the computation domain will be a cube that will be partitioned into as many partitions as the number of worker threads if using this library or MPI processes if using MPI. Figure 3-6 shows the partitioning of the unit cube domain into 4 sub-domains. The shaded panels in each partition indicate the ghost regions where the data need to be communicated between the 4 threads (or MPI processes in case of MPI) at the end of each Jacobi solve iteration. After the communication is done, the next iteration can begin. A Jacobi solver to the above Elliptic equation was implemented using this thread based library and MPI. An experiment was run on the same single CPU i5-2320 quad core machine that was used for the earlier experiments. Figure 3-7 shows the average time for 100 Jacobi solve iterations as the number of discretization points increases for both, the thread based and MPI based implementations. The results clearly show how thread based numerical programming is better that using MPI in a single machine. The results might not be the best as they were generated from a typical desktop machine. However, the trend of the graph will be similar even when running the same code in a more powerful server machine with multiple multi-core
100 Jacobi Iterations Solve Time for 3D Poisson Equation

Figure 3-7: Comparison of Execution Time for 100 Jacobi Solve Iterations (for 3D Poisson Problem) between MPI and this Library CPUs.
Chapter 4

Result Analysis

The results of the conducted experiments showed that this thread-based library fully utilizes all the cores in a single machine without much overhead. The results emphasize the fact that computation performance notably increases by increasing the number of worker threads until reaching the total number of physical cores in the hosting machine. The results also suggest that the gain of having a single or multiple threads per core really depends on the algorithm and on the amount and nature of work that is conducted by each task.

The experiments that compared process and thread based parallelization using MPI and this library, show that it is better to choose thread based parallelization whenever possible. It is also important to mention that even though this library is easier to use compared to MPI programming, it does not provide even a small subset of what MPI provides. The aim of building this library is to give engineers the chance to easily write parallel C++ code with minimal effort provided that their code is not intended to be parallelized across different machines or a computer cluster. From the experiments also, one might realize that for large problems where the memory of a single machine is not sufficient, the use of MPI across machines and the use of this thread based library within a single machine gives the best performance. MPI will handle the partitioning of the data and the communication between the machines and this thread based library will handle all the internal computation tasks that are done to the data residing in the memory of each machine. This hybrid parallelization
approach will result in better utilization of the machines compared to a fully MPI based parallelization.
Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis presents a simple lightweight C++ thread based parallelization library. The presented library is intended for use in numerical algorithms. It provides simple multitasking and task synchronization functions. The library hides all internal system calls from the developer and utilizes thread pooling to provide better performance and utilization of system time and resources when running large numerical tasks. Experiments were done to test the basic functionalities of the library and assess its performance with different job sizes and number of worker threads. More experiments were also done to compare thread based parallelization using this library with process based parallelization using MPI for both numerical and non-numerical algorithms. The results showed that using this library over MPI results in a simple code that is easy to read, understand and debug and also performs better with the exception that it runs in a single machine and not in a clustered machine environment. It might be hypothesized from the results that for large numerical computations, a hybrid parallelization approach using MPI across machines and this library within machines gives much better performance than fully using MPI across and within machines. In conclusion, it is important to mention that whatever work done in this library is not something that can be used as an end product. This is nothing but a tool to be used by developers who are trying to solve different problems by using the various
computational algorithms.

5.2 Future Work

The work described here is a starting point for building a comprehensive solution that allows numerical engineers, not only to easily utilize the multi-core CPUs in a single machine, but also to provide the same level of simplicity when utilizing a computer cluster. The solution to be built will be a hybrid solution that makes use of the thread based library created here and MPI.
# Appendix A

## Tables

Table A.1: Processing Time for 80000000 Array Elements

<table>
<thead>
<tr>
<th># of Worker Threads</th>
<th>Execution Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.33</td>
</tr>
<tr>
<td>2</td>
<td>8.67</td>
</tr>
<tr>
<td>4</td>
<td>4.55</td>
</tr>
<tr>
<td>8</td>
<td>4.63</td>
</tr>
<tr>
<td>12</td>
<td>4.59</td>
</tr>
<tr>
<td>16</td>
<td>4.52</td>
</tr>
<tr>
<td>20</td>
<td>4.58</td>
</tr>
</tbody>
</table>
Table A.2: Merge Sort Time for 80000000 Double Array

<table>
<thead>
<tr>
<th># of Worker Threads</th>
<th>Execution Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>54.42</td>
</tr>
<tr>
<td>2</td>
<td>30.81</td>
</tr>
<tr>
<td>4</td>
<td>18.74</td>
</tr>
<tr>
<td>8</td>
<td>10.92</td>
</tr>
<tr>
<td>12</td>
<td>11.03</td>
</tr>
<tr>
<td>16</td>
<td>11.04</td>
</tr>
<tr>
<td>20</td>
<td>11.97</td>
</tr>
</tbody>
</table>

Table A.3: Merge Sort Time Comparison between this Library and MPI for 20000000 Double Array

<table>
<thead>
<tr>
<th># Threads-MPI Processes</th>
<th>Execution Time (in seconds)</th>
<th>Execution Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13.61</td>
<td>18.5</td>
</tr>
<tr>
<td>2</td>
<td>9.14</td>
<td>9.75</td>
</tr>
<tr>
<td>4</td>
<td>4.77</td>
<td>5.79</td>
</tr>
<tr>
<td>8</td>
<td>2.69</td>
<td>6.3</td>
</tr>
<tr>
<td>12</td>
<td>2.8</td>
<td>6.01</td>
</tr>
<tr>
<td>16</td>
<td>2.9</td>
<td>6.11</td>
</tr>
<tr>
<td>20</td>
<td>2.92</td>
<td>5.9</td>
</tr>
</tbody>
</table>

Table A.4: Merge Sort Time Comparison between this Library and MPI for 80000000 Double Array in a high end server machine

<table>
<thead>
<tr>
<th># Threads-MPI Processes</th>
<th>Execution Time (in seconds)</th>
<th>Execution Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29.3</td>
<td>145</td>
</tr>
<tr>
<td>2</td>
<td>11.5</td>
<td>74.8</td>
</tr>
<tr>
<td>4</td>
<td>6.75</td>
<td>41.3</td>
</tr>
<tr>
<td>8</td>
<td>4.36</td>
<td>27.9</td>
</tr>
<tr>
<td>16</td>
<td>3.9</td>
<td>19.2</td>
</tr>
</tbody>
</table>
Table A.5: Comparing Time for 100 Jacobi Solve Iterations Time for 3D Poisson Equation between this Library with 4 worker threads and MPI with 4 Processes

<table>
<thead>
<tr>
<th># of Nodes (NX<em>NY</em>NZ)</th>
<th>Execution Time (in seconds)</th>
<th>Execution Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64000</td>
<td>0.3</td>
<td>1</td>
</tr>
<tr>
<td>512000</td>
<td>1.9</td>
<td>2.7</td>
</tr>
<tr>
<td>1000000</td>
<td>3.5</td>
<td>5</td>
</tr>
<tr>
<td>8000000</td>
<td>29</td>
<td>32</td>
</tr>
<tr>
<td>27000000</td>
<td>98</td>
<td>107</td>
</tr>
<tr>
<td>64000000</td>
<td>224</td>
<td>243</td>
</tr>
</tbody>
</table>
Bibliography

[1] Peter Pacheco *An Introduction to Parallel Programming* 2011: Morgan Kaufmann


