Effects of Thread-Local Task Trees in the CHT-MPI C++ Programming Library for the Chunks and Tasks Programming Model

Joseph Rogers
Abstract

Effects of Thread-Local Task Trees in the CHT-MPI C++ Programming Library for the Chunks and Tasks Programming Model

Joseph Rogers

Programming efficiently in extremely high thread count environments can be both challenging and tedious for the modern programmer. Task based programming models provide an abstraction of the low-level system details by instead allowing a program to be expressed as a set of tasks to be executed by the system. However, these abstractions often have an associated performance overhead that can cut into the benefits of running with many threads. The Chunks and Tasks programming model is one such task based abstraction model. The Chunks and Tasks CHT-MPI C++ library, developed at Uppsala University, follows this programming model. The CHT-MPI library works by first spawning a set of worker processes. Tasks are then assigned to the workers in a dynamic fashion so that those who complete their tasks faster than others are automatically assigned additional tasks. This is done to help with load balancing across the system. Workers may also steal tasks from each other. Each worker process may have many threads.

In this thesis, the CHT--MPI library is modified to reduce inter-thread communication within individual worker processes. Currently, each worker process has a single master list of all tasks to be executed by that worker. This list is shared between all threads within a single worker. This list sharing can create a performance bottleneck when the tasks are relatively short and the list is being modified often. Adding support to the library for thread-local task trees reduces the number of times that threads must synchronize with the master list. With this new feature, a worker process with twenty threads that was assigned a set of highly granular tasks can see a two times boost in performance when compared to the old version of the library. This level of performance improvement was observed for dense matrix-matrix multiplication and integration benchmarks. This is an important result as being able to effectively scale programs of a fixed problem size across an ever increasing number of threads necessitates being able to efficiently execute them in smaller and smaller parts.
Glossary

Overview of shorthand terms used in the thesis.

- CHT-MPI: Chunks and Tasks C++ Programming Library using MPI.
- MPI: Message Passing Interface
- Pthreads: POSIX Threads
- HPC: High Performance Computing
- CPU: Central Processing Unit
- GPU: Graphics Processing Unit
Contents

Glossary ................................................. i

1 Introduction ................................. 1
   1.1 Challenges Associated with Highly Parallel Programming .. 1
   1.2 Why Parallel Programming is Needed ..................... 1
   1.3 Aim of the Thesis .................................. 1

2 Background ................................. 2
   2.1 Conventional Parallel Programming ................... 2
   2.2 Task based Programming ............................. 3
   2.3 The CHT-MPI C++ Library ............................ 3

3 Methods ................................. 5
   3.1 Software Tools Used ................................ 5
   3.2 The current state of Task Management in the CHT-MPI Library 5
   3.3 Explanation of Thread–Local Task Trees ............ 5
   3.4 When to Start and When to Stop Using Thread–Local Task Trees 6
      3.4.1 When to Start .................................. 6
      3.4.2 When to Stop .................................. 7
   3.5 Testing–Methodology ................................. 9

4 Results .......................................... 9
   4.1 System Specs ....................................... 9
   4.2 Benchmarking Results .............................. 10
      4.2.1 When to Create Thread–Local Task Trees .... 10
      4.2.2 When to Synchronize Thread–Local Task Trees .. 11
      4.2.3 Speedup from using Thread–Local Trees ....... 12

5 Discussion ................................. 12
   5.1 When to Create Thread–Local Task Trees ............ 12
   5.2 When to Synchronize Thread–Local Task Trees ...... 13
   5.3 Speedup from using Thread–Local Trees ............ 13

6 Related Work .............................. 14

7 Conclusions and Future Work .......... 14

References ......................................... 15
1 Introduction

The growing proliferation of high thread count cluster computers within the High Performance Computing (HPC) space demands easy to use high level tools that enable programmers more to quickly and efficiently make use of available hardware. When writing code for large systems it is easy to get slowed down by the details of programming in such a parallel environment instead of working on the actual project [1].

1.1 Challenges Associated with Highly Parallel Programming

There are many challenges that programmers face when programming for highly parallel environments. Sharing data and synchronizing threads makes it extremely easy to introduce bugs and nondeterministic behavior. Tracking down these issues and fixing them can be both challenging and time consuming. Programmers must also be wary of data distribution patterns and make choices for this based both on the algorithm being implemented and the hardware it will be running on.

When programming with many threads, communication can become extremely expensive. Large numbers of active threads can exacerbate small inefficiencies that might not be problematic in smaller environments. All of these issues and more create a high barrier to entry for effective programming in the HPC space.

1.2 Why Parallel Programming is Needed

Though parallel programming is difficult, it is a necessary part of today’s computing needs and will continue to be so for the foreseeable future. A modern smartphone has more processing power than the world’s fastest supercomputers from the mid 1990s. The Adreno 530 mobile Graphics Processing Unit (GPU), developed by Qualcomm and released in late 2015, yields over 500 GFlop/s of performance [2]. By comparison, the world’s fastest computer in June of 1996, the SR2201/1024 at the University of Tokyo, only produced about 300 GFlop/s of performance [3]. As of November of 2017, the reported fastest supercomputer in the world is the Sunway TaihuLight in China. It yields a peak 125,435.9 Tflop/s of performance [4]. This is more than a 400,000 times increase in performance in about 20 years. Despite these incredible leaps in performance, the demands for even more compute power have not diminished. Some major areas of HPC today include artificial intelligence, bioinformatics, computational chemistry, climate simulations, self-driving cars, various medical applications, and many others not listed here.

All of these different HPC spaces demand increasingly fast computers. Though it is still possible to create smaller transistors, shrinking them no longer saves energy in the way that it used to [5]. This means that it is now very difficult to create silicon based processors with higher and higher clock speeds, see Figure 1. If programmers want to have their code keep getting faster year to year, then one of their best options is to program for multi-core. Some effects of this can be clearly seen with the wide-spread adoption of using GPUs for certain HPC applications. Though it has its difficulties, parallel programming is already a major part of computing today and will become even more dominant in the future.

1.3 Aim of the Thesis

This thesis focuses on extending certain parts of the CHT-MPI Chunks and Tasks C++ programming library [7]. The goal is to implement thread-local task trees within the CHT-MPI worker processes, and then to measure how performance of the library is affected across a variety of programs. Ideally, having thread-local task trees will enable the library to spend more time doing useful work and reduce the overhead performance cost of running with many threads.
Figure 1: The performance increase of CPUs over time. Up until about the year 2005, computer chips were mostly single core but had rapid increases in clock frequency year to year. Starting in about 2005, many chips started having multiple cores to provide increased performance instead of increasing their clock frequencies. Data adapted from: [6]

2 Background

There exist many ways for a developer to approach parallel programming, some much easier than others. Older, more traditional methods can produce very efficient code, but require that explicit attention is given to low-level system details. More recent higher level abstractions allow for faster development time, but sometimes at the cost of some performance.

2.1 Conventional Parallel Programming

Two popular low-level solutions for programmers wishing to use multiple threads in their code are the POSIX Threads (Pthreads) and Message Passing Interface (MPI) programming libraries. Pthreads tends to focus on parallel programming for computers that have a shared memory system. MPI often focuses on larger clusters using a distributed memory system. Sometimes, programmers will actually use a combination of both Pthreads and MPI together in order to implement one of their projects.

One of the major weaknesses of these technologies is their ease of use. While they can be quite computationally efficient, both of these technologies require the programmer to explicitly manage both the use of memory and distribution of data between all threads. Threads must be manually synchronized, and data must be handled carefully to prevent race conditions. In the case of MPI, when one thread tries to send data to another, the receiving thread must know to expect the incoming data. Threads being out of sync with send and receive requests for data can result in a deadlock and the program will be forever stuck. Often, certain pieces of critical data need to be shared and modified between multiple threads simultaneously. When this happens, the piece of critical data usually needs to be protected with a lock or mutex. This arbitrates access to the data, ensuring that while multiple threads may be able to read the data, only one thread is allowed to write to it at any given time. This keeps the data safe so that it is not corrupted by multiple simultaneous overwrites.

The low-level programmer also needs to take into account load balancing issues. These may occur when calculation times are unpredictable or if the performance of individual
threads varies. All of these issues and more can make traditional parallel programming seem very daunting for new users, and it can still present challenges for the more experienced.

### 2.2 Task based Programming

Pthreads and MPI are both widely used across the industry and can offer excellent performance. They can however present major challenges when learning to use them for the first time. In some cases it can be beneficial to use an abstracted programming library that while potentially slower than Pthreads or MPI, can offer a greatly improved ease of use and faster development time. Task based programming models aim to ease some of the burden of parallel programming while simultaneously optimizing certain coding pitfalls common to parallel programming.

Task based programming models require the user to specify their program as a set of logical tasks that are then mapped to the computer's available resources. The program is finished once all of the specified tasks are complete. This abstracted method of programming aims to help the developer think on a higher level about how to effectively express and expose the parallelism within their code, without worrying as much about the exact low-level details. The Chunks and Tasks programming model aims to do exactly this.

### 2.3 The CHT-MPI C++ Library

The Chunks and Tasks programming model was created at Uppsala University primarily by Emanuel H. Rbensson and Elias Rudberg. In the Chunks and Tasks model, pieces of data and calculations from traditional programs are instead broken up into groups of "chunks" and "tasks" respectively. Tasks represent different types of calculations that must be performed. A simple example is a matrix-matrix multiplication task. Chunks represent the input and output data used for each task [8]. In this case there would be two input chunks representing the two matrices being multiplied. Once the matrix multiplication task is completed it will produce a new output chunk containing the product of the two matrices. This chunk will then be available to any subsequent tasks that might require it.

When a Chunks and Tasks task executes it will often create sub-tasks that must also be completed. These sub-tasks can go on to create additional sub-tasks. Any task that creates sub-tasks is not complete until all of its sub-tasks are also complete. There are no fundamental differences between a task versus a sub-task. It is often not known beforehand if a task will create additional tasks, or if it does, how many. Tasks spawning sub-tasks that go on to spawn even more tasks, creates the idea of the task tree, see Figure 2. Each worker process in the CHT-MPI library has its own task tree. All tasks are mapped one at a time to different threads in the system. This means that a single task runs only on a single thread. Parallelism is exposed by having many tasks.

One of the strengths of the CHT-MPI library is its use of work stealing. Pending tasks that are in the work queue, but not yet started, may be picked up by an idle worker process and executed immediately. This is very powerful because it helps to provide automatic load balancing on systems of any size. This enables programs to be highly scalable so that even systems with thousands of threads can still utilize as much of their resources as possible.

The Chunks and Tasks model also offers an advantage over traditional parallel programming methods by its decentralized nature. In classical parallel programs there is typically a "master thread" that delegates specific work to all of the other threads in the system. An example of this would be one thread that goes through a for loop of all of the other
Figure 2: An example task tree that could result from a program using the Chunks and Tasks model. All programs start with a single mother task that then creates sub-tasks. Individual tasks can create their own child tasks independently from each other. When a task has no children it can be thought of as a "leaf" of the tree. Tasks are ordered for execution in a depth first pattern. Tasks are not marked as completed until all of their child tasks are also completed. The program is finished once the mother task is completed.

threads and assigns them work to do on a specific piece of data, often based on their thread ID. Once the work is completed, the slave threads signal back to the master thread that they are done and that the newly processed data is now ready. Often this works just fine, but many threads sending signals to a single master thread can become a major communication bottleneck in systems with hundreds or even thousands of threads. The severity of this bottleneck depends on how quickly work is being completed and how often slave threads are communicating with the master thread.

By contrast, the Chunks and Tasks programming model has no master–slave relationship. All worker processes run by the library are on equal footing. Individual workers are only aware of the other workers who directly communicate with them. In this way, there is never a single worker that is talking to all the others. Instead, communication is much more scattered and evenly distributed around the system so as not to overload a single pathway.

When using the Chunks and Tasks model, there is also no need for the programmer to directly work with the distribution of data between worker processes on the system. Instead, the runtime library assigns tasks and sends chunks dynamically. This enhances the portability of code, enabling it to run on a variety of scales of systems with no changes to the source code.

The main goal of the Chunks and Tasks programming model is to define a standard that makes parallel programming easier. It provides an abstracted framework that aims to simplify parallel code production. The CHT-MPI C++ implementation of the model automates the details of threads, memory, and communication. Improvements to this programming library are the focus of this thesis.
3 Methods

This section describes the process for how the thread-local task trees were implemented, and some of the choices that were made about when to actually activate them. It also makes note of some of the tools used in the creation of this thesis.

3.1 Software Tools Used

All programming changes to the source of CHT-MPI library were done using the Sublime 3 text editor. The programming of bash scripts used in the submission of jobs to the Uppmax Rackham Cluster was done using Vim. Jobs themselves were submitted using the Slurm Workload Manager. All of the benchmarking results were processed into graphical figures with Matlab. Writing was done in Emacs using the TeXstudio IDE. Version control was handled using Git and hosted on GitHub.

3.2 The current state of Task Management in the CHT-MPI Library

When a program is run using CHT-MPI, a Task-Manager service is started simultaneously. The Task-Manager keeps a list of the currently running, pending, and completed tasks. There is one list for each worker process. Each worker process' list is locally accessible by all the worker's threads. However, this also creates a potential bottleneck in the code depending on the number of threads within a worker process, and how often they are modifying the list.

To ensure the safety of the data, the list is protected by a mutex. This guarantees that only one thread can be modifying the list at a time. Currently while one thread has control of the mutex, other threads may have to idle. Furthermore, there is currently no kind of queue based locking system implemented. This means that all threads not currently controlling the mutex will be constantly trying to access it instead of doing other useful work or going to sleep. Multiple threads competing for a single mutex this way can induce false sharing cache misses. This can produce additional overhead from the library separate from thread idling.

During initial performance runs of the original CHT-MPI library, a simple test was run to try to see whether or not mutex competition really was a problem in programs with highly granular tasks. A quick change to the source code forcing all threads accessing the tasks list to lock and then unlock the mutex an extra time before actually changing the data was immediately visible when looking at the execution time results. This gives credence to the theory that mutex accessing were noticeably contributing to the overhead of the library.

3.3 Explanation of Thread–Local Task Trees

One approach to try and reduce idling as well as mutex competition is to have special sub-lists of tasks that exist on the thread level within each worker process. These sub-lists are referred to as thread-local task trees. The aim of the thread–local task trees is to reduce inter-thread communication as much as possible with regards to the worker level task list.

Local task trees can be useful when a thread is given a task that in turn spawns additional child tasks. When using a local task tree, details of tasks in the tree are not immediately communicated back to the worker level task list. As a result, tasks that exist in local trees are not visible to other threads on the same worker process. Using thread-local task trees allows threads to immediately execute any child tasks they spawn without the need to synchronize with the worker level task list until it is desirable to do so.
There exist certain downsides however. Keeping some tasks local to specific threads could create a load balancing issue under certain conditions. If one thread finishes faster than the others then it may have to idle until the worker level list gets updated. This effect could be severely exacerbated if the number of spawned tasks in each local task tree is very unpredictable from thread to thread. To guard against this, the local trees have been programmed to only activate under certain conditions. They have also been programmed to automatically synchronize with the worker level task list when the pool of tasks available to other threads is low.

3.4 When to Start and When to Stop Using Thread–Local Task Trees

When using the thread–local task trees, two major decisions have to be made about them. At what point does it make sense to start a local tree, and at what point should the local tree be committed back to the worker level list.

3.4.1 When to Start

To control the creation of thread–local trees, a class member variable ancestorDepth was created. The variable ancestorDepth is a measurement of how deep a task is in the context of the worker level task tree. With this variable, each time a thread starts work on a task it can inspect the task’s ancestorDepth, and make a decision about whether or not to create a local task tree. The ancestorDepth threshold can be tuned to an ideal value for programs with different shapes of task trees and for different computer systems with different thread counts.

It is important to consider how to pick a good value for the ancestorDepth variable. It is desirable to take advantage of the positive aspects of the thread–local trees for as long as possible during program execution. However, if they are activated too early, then they risk limiting the parallelism of the program.

Consider a mother task that creates two sub–tasks as shown in Figure 3. In this example, each task always creates two child tasks until the leaf tasks are eventually reached. The two sub–tasks spawned by the mother task have an ancestorDepth of 1 since they are a distance of 1 from the mother task. If these two initial sub–tasks, spawned by the mother task, were chosen as starting points for local trees, then the parallelism of the program would be limited to only two threads. This is because all of the new tasks spawned would then be invisible to the other threads running on the worker process. If the worker process had four threads, then it would be better to wait until an ancestorDepth of at least 2 before beginning to use thread–local task trees.

Different programs may have different shapes of task trees. Figure 4 shows a task tree where each task always spawns four sub–tasks instead of only two as in Figure 3. When creating more sub–tasks at a time, a lower value for ancestorDepth may be used without limiting parallelism.

Only considering task depth may not work well in all situations however. Reliably picking a good value for ancestorDepth requires some prior knowledge about the task tree’s expected shape. Additionally, if the overall shape of the task tree is asymmetrical as in Figure 5, then there may be poor load balancing between the different threads. A more general way to decide when to create local trees is desirable, one that can handle asymmetrical task trees and that does not require any prior knowledge of the program to be executed.
Figure 3: In this task tree each task always spawns two additional sub-tasks until the leaf tasks are eventually reached. With a predictable and symmetrical task tree like this, it is easy to pick out good values for the `ancestorDepth` variable when deciding when to start using thread-local task trees.

Figure 4: This task tree shows tasks that always spawn four sub-tasks. Since it is symmetrical and predictable it is easy to pick a good value for `ancestorDepth`. However, the value for ancestor depth will not be the same as for the task tree show in Figure 3. Since more sub-tasks are spawned at a time in this tree, a lower value for `ancestorDepth` can be used to achieve full parallelism.

To address this, a second class member variable `numPendingTasks` was created that could be used instead of `ancestorDepth`. The variable `numPendingTasks` keeps track of how many pending tasks exist within the worker level task list. Pending tasks are tasks which are ready to execute, but have not been assigned to a thread yet. If `numPendingTasks` is high, it indicates high task granularity and the potential for some performance gains using thread-local task trees.

There are some interesting differences between the two methods. Using the `ancestorDepth` variable enables thread-local task trees to be created in a deterministic way. However as mentioned before, picking an ideal value for `ancestorDepth` requires some knowledge about the topology of the task tree. Alternatively, a good threshold value for `numPendingTasks` is topologically independent. Knowledge of where local trees will get created is lost however. Variation in thread performance introduces uncertainty about exactly which tasks will be chosen as the starting points for local trees.

### 3.4.2 When to Stop

Once a local tree has been created, a decision then has to be made about when to terminate it and synchronize with the worker level task list. The simplest method is to allow each local
Figure 5: Certain task trees may be asymmetrical and have a difficult to predict structure, especially if the tree shape depends on the input data. In cases such as these it may be difficult to choose a good value for `ancestorDepth`. It may be instead easier to use the `numPendingTasks` value to determine the best time to start using thread-local task trees. Once a good value for `numPendingTasks` is chosen, it is task tree independent, and can work well for many different shapes and sizes.

...tree to execute all the way to completion. This method leads to very little time dedicated to synchronization, but causes problems with good load balancing and opportunities for work stealing. Ideally, synchronization should happen as little as possible, but waiting too long may cause other threads within the worker process to idle due to a lack of available work within the worker level task list. If there exists a thread–local task tree with a large number of pending tasks, but not many pending tasks in the worker level task tree, then it may be a good time to synchronize the thread–local tree. Doing this will allow all of the pending tasks to become visible to the other threads.

To do this, the value of `numPendingTasks` is regularly inspected when executing inside a thread–local task tree. When the value of `numPendingTasks` goes below some value, the thread–local tree will automatically synchronize with the worker level task tree. When this happens, all pending tasks that existed within the local task list are added to the worker level task list. This enables other threads within the worker process a chance to get more work. There are some downsides to this however. Exiting local trees early does increase the amount of inter–thread communication, and some threshold value must be chosen for `numPendingTasks` that determines when to synchronize.

As a general rule of thumb, the threshold value when using `numPendingTasks` to leave thread–local trees, should always be greater than or equal to the value used when entering the local trees in the first place. If this is not done then the local trees are liable to exit immediately before they have a chance to be of any benefit.
3.5 Testing–Methodology

To test all of these different options for controlling how to use thread-local task trees, a series of experiments were run. These experiments aimed to address the following points.

1. When to enter a local tree.
2. When to leave a local tree.
3. Effects of using \texttt{ancestorDepth} vs \texttt{numPendingTasks}.
4. When using thread-local task trees, what level of speedup can be obtained, if any, when compared to the default CHT–MPI C++ programming library.

Two different programs were chosen to be run for the benchmarks, matrix–matrix multiplication and integration. These programs were chosen because they have different shaped task trees and offer easy flexibility in choosing the granularity of their tasks. In the case of matrix–matrix multiplication task granularity can be controlled by changing the blocking size. For the integration program it can be controlled by changing the discretization of the domain.

For each of the two programs, several different benchmarks were run. First, to decide when best to start using thread–local task trees, a series of tests were run to probe which values for both \texttt{ancestorDepth} and \texttt{numPendingTasks} yielded the best performance, see Figures 6 and 7. Second, for deciding when to synchronize thread–local task trees with the worker level task tree, another series of tests probing for the best \texttt{numPendingTasks} threshold were run, see Figure 8. Finally, a series of speedup tests were run comparing the general performance impact of using thread–local task trees compared to the original unmodified CHT–MPI library, see Figures 9 and 10. Each experiment was run on a range of different task granularities. It was hypothesized that scenarios with high task granularity would benefit more from thread–local tasks trees than experiments with low task granularity. Each test was run three times to generate an approximate average time taken.

4 Results

This section shows the results obtained from running the benchmarks described in section 3.5. All benchmarks were run using the Uppsala University Uppmax Rackham Cluster Computer.

4.1 System Specs

\textbf{Hardware Specifications:}
Uppmax Rackham Cluster, February 2017
CPU: 2 x 10-core Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz per node.
Memory: 128 GB ECC 2400MHz DRAM per node.
6080 cores, 304 nodes
Lustre file system, 1PB storage
Infiniband FDR interconnect, 56Gb/s, latency 0.7 microseconds.

\textbf{Software Specifications:}
Operating System: CentOS 7
MPI Version: Open MPI 2.1.1
Compiler: G++ 7.1.0
Compiler Flags: \texttt{-O2 -std=c++11}
4.2 Benchmarking Results

4.2.1 When to Create Thread–Local Task Trees

![Graph showing effects of using ancestorDepth to start thread-local task trees.](image)

Figure 6: Results of how choosing when to create thread-local task trees affects program execution time. Results are shown for both the matrix-matrix multiplication and integration programs. In this test all local trees were executed to completion before being synchronized. Timings have been normalized so that tests fit on the same scale. The figure shows the creation of local trees based on the ancestorDepth variable. There exists an area of ideal performance at around 2–5 for the value of ancestorDepth threshold when running the matrix multiplication program. In the case of the integration program however, the ideal threshold is from about 7–18.

The execution time of the code changes significantly depending on what values are chosen for either ancestorDepth or numPendingTasks. Choosing too low or too high of a value results in suboptimal performance, see Figure 7. Results shown have been normalized to fit on the same time scale for each of the two program types.
Figure 7: Results of how choosing when to use thread-local task trees affects program execution time. Timings have been normalized so that tests fit on the same scale. Results are shown for both the matrix-matrix multiplication and integration programs. In this test all local trees were executed to completion before being synchronized. The figure shows the creation of local trees based on the `numPendingTasks` variable. There exists an area of ideal performance at around 20–25 for the value of `numPendingTasks`.

### 4.2.2 When to Synchronize Thread-Local Task Trees

Figure 8: Choosing when to synchronize thread-local task trees has a significant effect on performance. Timings have been normalized so that tests fit on the same scale. For both the matrix-matrix multiplication and integration benchmarks, synchronizing when the value for `numPendingTasks` falls below the number of threads being used appears to be ideal. For this test data on the x-axis start at 19 because 19 threads were being used to execute the program. Tests run with with a leaving threshold less than the thread count causes the thread-local task trees to always be synchronized immediately. This is because tasks that are currently executing also contribute to the `numPendingTasks` value.
Choosing how long to wait before deciding to synchronize a thread–local task tree with the worker level task tree can also have a significant effect on performance, see Figure 8.

4.2.3 Speedup from using Thread–Local Trees

![Matrix Multiplication Parallel Speedup Diagram]

Figure 9: Performance scaling of the matrix multiplication program across several sets of rules for entering and exiting thread–local task trees. For choosing when to enter local trees, an anzcestord Depth threshold of 4 was used, and a numPendingTasks threshold of 19 was used. Not using any local trees causes performance scaling flatten out starting at about 10 threads. Under the right conditions, using local trees causes performance to continue scaling all the way to the maximum thread count available on the machine. Note that in this test a different threshold value than that used in Figure 10 for anzcestord Depth had to be manually chosen in order to get good performance, but that the threshold for numPendingTasks did not have to be changed.

Without using any local trees, parallel performance is not able to scale positively across all threads within a single worker process. Scaling actually starts to trend negative as more threads are added. Once local trees are turned on with good usage rules however, performance scaling continues all the way to 19 threads, the maximum allocated for the worker, see Figures 9 and 10.

5 Discussion

5.1 When to Create Thread–Local Task Trees

Figures 6 and 7 demonstrate how it is important to create thread–local task trees only under certain conditions. In these tests, local task trees were created only once a chosen threshold had been met for either the anzcestord Depth or numPendingTasks values. For both values, choosing too low of a threshold yielded worse performance than no local trees at all. This is due to a limit in achievable parallelism that happens when local trees are created so early. Tasks in a local tree keep all of their child tasks assigned to the same thread instead of being spread out around the system. Alternatively, choosing too high of a value caused the local trees to never be created at all. In this case, the performance was the same as the original CHT–MPI library, with no support for thread–local task trees.
Figure 10: Performance scaling of the integration program across several sets of rules for entering and exiting thread-local task trees. For choosing when to enter local trees, an ancestorDepth threshold of 8 was used, and a numPendingTasks threshold of 19 was used. Not using any local trees causes performance scaling flattens out starting at about 5 threads. Under the right conditions, using local trees causes performance to continue scaling all the way to the maximum thread count available on the worker. Note that in this test a different threshold value than that used in Figure 9 for ancestorDepth had to be manually chosen in order to get good performance, but that the threshold for numPendingTasks did not have to be changed.

Choosing an ideal value for the thresholds can be a bit tricky. In the case of ancestorDepth the ideal value for each of the two programs is not the same due to their differing task tree topologies. In the case of numPendingTasks however they are about the same. It was recognized that there exists a relationship between the ideal threshold for numPendingTasks and the number of threads that each worker process is using. It was discovered that the threshold chosen for numPendingTasks should be at minimum at least as high as the number of threads being run inside the worker process. This is a useful observation because it allows the programmer to setup the thread-local task trees in an optimal way without prior knowledge of what type of program is going to be run.

5.2 When to Synchronize Thread–Local Task Trees

Figure 8 shows how choosing when to synchronize thread–local task trees affects performance. For both the matrix multiplication and integration programs, choosing higher and higher values for the synchronization threshold leads to increasingly worse performance. In both cases it appears that the best value for synchronization is actually the same as the worker process’ thread count. This means that the value for entering and exiting thread–local task trees for numPendingTasks is actually the same. This makes a certain amount of sense. If local trees are not worth using until a certain value for numPendingTasks has been reached, then simply keep using them until the value goes lower than that threshold. Once the value is lower, synchronize and stop using local trees.

5.3 Speedup from using Thread–Local Trees

Figures 9 and 10 show how using thread–local task trees is able to dramatically improve the multi–threaded performance scaling of both programs. While before, both programs
were not able to properly scale up to the total thread count of the worker, with the addition of thread-local task trees, both programs scale to the total thread count of 19, and appear to be able to continue scaling further if more threads were available.

When good threshold values are chosen, using ancestorDepth or numPendingTasks for the threshold yield about the same performance. This is only true however when numPendingTasks is allowed to synchronize early instead of executing the local tree to completion. This is suspected to be due to variation in thread performance as mentioned in section 3.4.1. When running the speedup test multiple times for numPendingTasks without leaving, the performance times can be quite different, and sometimes just as good as the "with leaving" test.

6 Related Work

There are many other projects that focus on mitigating the challenges of parallel programming. Many of them are libraries or frameworks that take a different approach than those of Chunks and Tasks.

The DAGuE framework focuses on architecture aware task scheduling for problems that can be represented as a Direct Acyclic Graph [9]. This means that the programmer should not have to consider the exact number of cores that their code will be running on. As long as they've specified it correctly, the DAGuE framework will automatically distribute tasks and data around the system in an optimal way. The DAGuE framework also has a feature that is similar to Chunks and Tasks' thread-local task trees. In the DAGuE framework there is a distributed task scheduling system where numerous related tasks may be scheduled at a one time to a single process. This is because tasks in DAGuE may have both a predecessor and successor, similar to mother and child tasks in Chunks and Tasks. If the DAGuE scheduler detects that one of these related tasks is executable using data that is already locally available on the process, then it can safely be scheduled using only local information. This helps to avoid the master-slave communication bottleneck as described in Section 2.3.

The Threading Building Blocks (TBB) C++ library developed by Intel also has some similarities to the CHT-MPI library. TBB breaks programs down into logical tasks and dynamically assigns them to the system's threads so as to be both scalable and architecture independent. TBB also implements work stealing functionality. These features can sometimes come at a cost however. A 2008 study at Princeton University showed that under certain circumstances overhead from the library itself accounted for 47% of the total execution time on only a 32 core system [10]. This would likely become even larger on a system with an even greater thread count.

While many technologies focus on general purpose parallel programming, there are some that specialize for specific types of operations. Boost is a set of C++ libraries that offer great parallel programming support for many tasks related specifically to linear algebra [11].

7 Conclusions and Future Work

Barring some massive technological breakthrough, computers are going to continue increasing their performance by adding more cores instead of faster cores. Being able to take advantage of this hardware requires being able to efficiently break programs up into very small parts that can be distributed across a system. The ever increasing demand for high
performance parallel computation will continue to drive the development of new, efficient, and easy to use tools to assist developers with parallel programming. The CHT–MPI C++ Programming Library is one such tool.

The work done in this project implemented thread-local tasks trees as an added feature to the CHT–MPI library. This reduced the amount of inter-thread communication required for programs with a high task granularity. For some of the tests conducted, this new feature was able to provide a more than two times speedup in the execution time of the code.

There still exist many potential extensions and optimizations that could benefit the CHT–MPI library, as well as improvements that could be made to its general usability. Future work on the library could include optimizations to how the worker level task lists are accessed. Though the number of accesses to the worker level list has been dramatically reduced, further improvements could be made by the addition of a queue based locking system.

Using abstracted programming libraries to increase accessibility and enable faster development times has a number of advantages. While there often exists some performance overhead, working with an extra layer of abstraction allows the programmer to think on a higher level about the code they implementing. It also helps open up the HPC space to a greater number of developers, enabling them to take advantage of large systems without the need to understand all of the low-level details.

References


