DYNAMIC MACHINE LEVEL RESOURCE ALLOCATION TO IMPROVE TASKING PERFORMANCE ACROSS MULTIPLE PROCESSES

by

Richard Walter Thatcher

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2016

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Richard Walter Thatcher

Thesis Title: Dynamic Machine Level Resource Allocation to Improve Tasking Performance Across Multiple Processes

Date of Final Oral Examination: 28 June 2016

The following individuals read and discussed the thesis submitted by student Richard Walter Thatcher, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

James Buffenbarger, Ph.D.	Chair, Supervisory Committee
Kyle Wheeler, Ph.D.	Member, Supervisory Committee
Amit Jain, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by James Buffenbarger, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by Jodi Chilson, M.F.A., Coordinator of Theses and Dissertations.

Dedicated to my family.

ACKNOWLEDGMENTS

The author wishes to express gratitude to Dr. Kyle Wheeler, Dr. Richard Murphy, and Micron Technology without which this work would not have been possible.

ABSTRACT

Across the landscape of computing, parallelism within applications is increasingly important in order to track advances in hardware capability and meet critical performance metrics. However, writing parallel applications is difficult to do in a scalable way, which has led to the creation of tasking libraries and language extensions like OpenMP, Intel Threading Building Blocks, Qthreads, and more. These tools abstract parallel execution by expressing it in terms of work units (tasks) rather than specific hardware details. This abstraction enables scaling and allows programmers to write software solutions that can leverage whatever level of parallelism is available. However, the typical task scheduler is greedy and naïve. Thus, concurrent parallel processes compete for computational resources, which results in unnecessary context switches, mis-timed synchronization, unnecessary resource contention, and the associated consequences. By providing a mechanism of communication between the task schedulers, processes can cooperate to more effectively utilize hardware and avoid the negative consequences of coarse-grained resource contention. This work uses Qthreads to demonstrate that cooperative allocation of computational resources reduces contention and decreases execution time. The overhead added for the resource allocation is shown to have minimal impact. Using the Unbalanced Tree Search (UTS) and High Performance Conjugate Gradient (HPCG) benchmarks, execution time across concurrent processes shows significant decreases across a range of machines running a variety of hardware resources and software configurations. Tests also indicate that dynamic compute-resource allocation provides a clear performance benefit even when

hardware resources are oversubscribed: when there are more processes than processing units. UTS tests saw an average of 4.98% reduction in execution time in Linux compared to Qthread's yielding option and an 89.32% reduction in execution time in Apple OS X. HPCG resulted in partitioning reducing execution time by an average of 22.31% compared to the default Qthreads configuration across all test platforms.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 The Problem

Parallel computing often results in random distributions of work, creating hot spots of activity, especially when computation is not embarrassingly parallel and includes heterogeneous threads and/or processes. As task-parallel programs continue to increase in prevalence, it is increasingly likely that multiple task-parallel processes live on a single machine, and even CPU/socket, creating contention for compute resources.

Lightweight tasking alleviates some of the contention by abstracting the underlying operating-system-level thread usage to distribute work among a set of worker threads. However, each instance of a lightweight tasking library typically assumes it is the only instance on the machine. Naturally, it makes scheduling decisions to maximize performance based off that assumption. Such decisions are often highly counterproductive in a shared or even oversubscribed environment.

This leaves the operating system to handle the contention between processes. This is not unique to parallel processes since many single-threaded, serial processes will also cause the operating system (OS) scheduler to distribute resources to processes based on the scheduling policy via time-slicing and context switching. However, because most OS schedulers treat all schedulable threads as unrelated, they make decisions that do not fit the needs of parallel programs. Further, because parallel programs

rely on multiple schedulable threads, they are more exposed to the negative impacts of unfortunate OS-scheduler decisions.

1.1.1 Context Switching

Pre-emptive context switching is a standard operating-system tool for sharing computational resources. Processes are scheduled to get a time slice to run and those time slices are scheduled according to the OS's scheduling policy. Pre-emptive context switching is the process by which one process is swapped out of the CPU and another process is swapped in for each time slice. A context switch has direct and indirect costs that can degrade the performance of a process.

The direct cost of a context switch is the time to perform the action: save CPU registers and other process state, flush CPU pipeline and cache, run OS-scheduler code to select another process, then load and begin execution of the newly selected process. Indirectly, there will be cold-cache misses as a result of flushing the cache. Li, Ding, and Shen found the direct cost of a context switch to be 3.8 microseconds, and found the indirect costs to reach as high as over one-thousand microseconds on a server with dual Intel Xeons with a 2.0 Ghz clock rate [10].

1.1.2 Jitter

Operating system jitter, also known as scheduling jitter, is delays in processing due to scheduling of other processes and system interrupts, or, more generally, delays due to context switching [1]. Due to the time-slicing of the OS scheduler, in parallel programs this can cause a compounding execution delay related to synchronization.

If processes A and B need to synchronize, and A gets scheduled by the operating system and reaches the synchronization point, but B does not get scheduled during

that time slice, then A will continue to wait for the rest of its time slice. Depending on the type of synchronization, process A may get scheduled repeatedly, with nothing to do but wait, before process B gets scheduled. When B is run next, it will reach the synchronization point allowing A to continue, but A is now back in the scheduler waiting for another time slice. When A is scheduled next, it will continue past the synchronization point.

Since the impact of jitter depends on the scheduling policy, as well as the dynamic set of other processes contending for time slices, it leads to inconsistency in execution time. Furthermore, because this can potentially happen on every synchronization, and synchronizations can occur between more than two threads, this delay can compound and drastically alter execution time. What's more, jitter in one set of threads creates localized delay, which changes the balance of execution, leading to more jitter effects.

The simple way to reduce the effect of jitter is to yield whenever a synchronization point is reached, allowing the process to immediately relinquish its time slice. If there is synchronization mismatch due to jitter, yielding can reduce time spent waiting for synchronization, by allowing other threads or processes to run. However, this can backfire. If A and B both reach their synchronization point during overlapping time slices, they can proceed as normal, but if the processes yield when they synchronize then both A and B would be waiting until their next time slice to continue, adding delay.

1.2 Qthreads

Qthreads is an open-source, user-level tasking library developed at Sandia National Labs that provides a tasking abstraction of hardware. Qthreads' API includes tasking,

synchronization, data structures, etc., and is a portable, user-space, shared library [19].

When Qthreads is initialized, it spawns a number of worker pthreads, or "workers." The workers may be grouped, based on a user specification or automatically via software that reports the hardware topology. Every group of workers is a shepherd, even if there is only one group. By default, Qthreads attempts to have the number of groups equal to the number of non-uniform memory access (NUMA) domains, as determined by its understanding of the hardware.

Each shepherd has a queue of work that its workers pull from to execute. Because of this, and because NUMA regions frequently map to L3 cache sharing, the grouping to NUMA regions allows workers to leverage cache locality based on the work in the queue. In an effort to balance the work between shepherds, and to keep all workers busy, shepherds can steal work from another shepherd's queue [16].

This model allows a programmer to create as many lightweight threads as memory allows, without incurring the typical problems of oversubscribing the hardware, since the number of OS threads doing the work is matched to the hardware.

Due to the work-stealing mechanism, when workers have no work they wait in a spinlock. A spinlock, as opposed to yielding or blocking, maximizes responsiveness when new work gets added to the queue; the worker will likely have much of its time slice remaining, and can begin processing the work immediately. This is ideal when it is expected that the queue will not be empty for very long or when there is nothing else the computer should be doing. The downside of a spinlock-based wait is that when workers are not processing actual work, they are still using a full time slice and causing the CPU to needlessly execute their spin loop. When there is only a single process, this minimizes response latency at the expense of power.

1.2.1 Oversubscription

Qthreads provides a compile-time option that causes the Qthreads spinlock to yield its time slice. These yield functions end the thread or process's current time slice and move them to the back of the process scheduling queue for their priority. This allows other threads or processes to run. This is a useful behavior in situations where it does not make sense to have a process or thread spinning and wasting time, such as in oversubscribed situations where other threads and processes need the resources. This has a large impact in oversubscribed situations. Since the aim of this work is to improve performance in oversubscribed situations, the yielding feature of Qthreads is the true benchmark to test against.

1.3 Other Tasking Approaches

OpenMP includes a tasking specification that is generally implemented as a set of compiler directives [17]. The compiler directives generally link a shared library for the runtime, which uses environment variables as a way to control and configure execution. OpenMP is incorporated with many popular compilers. Both the compiler directives and environment variables provide a significant level of "tunability" in applications, but in practice typically requires tuning for each machine configuration to get the best performance, which can involve changes to both the source code and the environment variables, and is static for the lifetime of the application's execution. The GNU OpenMP implementation has dynamic teams that limits the number of active threads in an OpenMP application to cooperate with other OpenMP applications to avoid or minimize the effects of oversubscription [18]. Similar to this work, the dynamic teams use shared memory to give each process information about how many

processes are running and the implementation of dynamic teams lies entirely inside the OpenMP shared library. However, dynamic teams uses a constantly polling "watchdog" for monitoring the shared data segment, uses locking mechanisms on the shared data segment, and has additional restrictions imposed due to the OpenMP specification.

Intel Threading Building Blocks (TBB) [6] is a tasking library similar to Qthreads provided by Intel. TBB uses a shared library and associated API to give programmers an abstraction of tasks from threads to provide performance and task scalability. Intel's documentation of TBB lays out the same core problems that this work is aimed at addressing [7]. It implies that TBB does not mitigate compute resource contention across processes and that Intel leaves users of TBB to mitigate these issues either through programming or control of the run-time environment.

1.4 Related Research

The concept of partitioning work across threads or processes is not new, and is central to most parallel programming. Partitioning hardware resources for multiple applications is not as common on a single machine.

Nesbit et al. broach the subject of resource management on multicore machines, but approach it from a system level. They propose a concept of Virtual Private Machines (VPM), where each application has its own VPM providing the application a share of memory, processing, etc. [15]. The VPM concept provides virtual hardware resources based on the application providing a quality of service objective (QoS). The QoS is used generate the resource assignments. This clearly requires operating-system-level implementation, and has significant overhead to generate and manage the

VPMs. One benefit is that the VPMs can provide an additional level of sandboxing for applications, siloing them to prevent system instability as a result of misbehaving applications.

Liu et al. suggest space-time partitioning, which virtualizes partitions then scheduling at the partition level [11]. Similar to the VPM approach, this requires operating-system-level implementation, going so far as to propose all scheduling is done at the partition granularity, even the OS services, and a context switch effectively becomes a partition switch. Tessellation is what they call the kernel that implements partitioning and allocation, as well as provides an API allowing programmers to pin threads to specific cores and specify virtual-memory translations. They describe the possibility of having an application span multiple partitions and providing an inter-partition communication method, though they appear to mean simply that multiple applications can interact across partition boundaries. The concept of scheduling partitions obviously brings a lot of challenges, an overhaul of OS scheduling and design, and an additional programming paradigm.

Iancu et al. argue against partitioning, instead looking at oversubscription to increase throughput [4]. They argue that synchronization granularity is the determining factor of performance in oversubscribed environments: smaller synchronization intervals degrade performance while larger synchronization intervals are less susceptible. However there are some nuances to their claim, one being that they must enforce an even distribution of threads across cores at startup or the operating system load balancer will cause performance degradations. Similarly, their oversubscription tests are for a single application that oversubscribes its threads or, in the case of MPI, processes. Both of these conditions preclude concurrent, heterogeneous, parallel-application oversubscription from fitting into their results.

CHAPTER 2

IMPLEMENTATION

The goals of the implementation are to be as efficient as possible to minimize overhead and maintain Linux and Unix compatibility. Additionally, the implementation keeps in line with Qthreads use as a userspace-only shared library. This is worth mentioning because it would be far easier to implement these features inside the operating system kernel. For example, the Qthreads code could be a single runtime, directly handling the resource allocation instead of relying on inter-process communication. This would be more efficient since shared memory would not be necessary but would reduce the portability of the library. Or there could be an OS service or daemon that handles errors and resource allocation.

2.1 Design Overview

The purpose of this implementation is to improve execution time and more efficiently utilize computation resources when a system is oversubscribed. To do this, the processes need a way to communicate so they can cooperatively partition the available computational resources. When a process initializes Qthreads, the process will open a section of shared memory that contains information about other Qthreads processes running on the system. The new process will notify all the other processes that it is present and then do a partitioning of the computational resources. When a process

receives a notification of a new process, it will repartition based on the updated information in the shared memory. When a process finishes, it will update the shared memory section to remove itself and notify the other processes, which will repartition, allowing them to reclaim the resources used by the process that is finishing.

2.2 Thread Liveness

Central to the concept of friendliness across parallel processes is to have only the threads allocated for a process active. The goal is to maintain the sets of inactive and active threads such that the union of the active threads across processes is all threads, and the intersection of the active threads across processes is the empty set. The only exception being thread zero, which is the parent thread for a Qthreads instance, so it is left active as a safety precaution.

Qthreads has inactive threads that use a spinlock for inactivity. For the general, single process purpose of Qthreads, a spinlock is ideal because it allows for the thread to become active as soon as possible within its scheduled time slice. Since there is little resource contention with a single process, a spinlock using its full time slice and essentially doing nothing harms power usage and little else while giving the benefit of being maximally responsive. However, multiple processes using a spinlock will maintain the negative traits of relying on the process scheduler and context switching.

This implementation eschews the spinlocks in favor of I/O blocking, since it is expected that inactive threads will remain inactive for longer periods of time. So quicker transition from inactive to active is traded for less intrusive inactive thread behavior. Since I/O operations can take a lot of time, threads that make blocking I/O calls are removed from the process scheduler so they no longer receive time

slices. Once the data for the waiting process/thread is available, it signals the kernel that the data is ready and the kernel puts the waiting process/thread back into the process scheduler. Since the process/thread is not being scheduled during the time it is waiting, this reduces context switching and wasted time slices. The reduction in context switching in turn improves performance when it is expected that the process/thread will be waiting for longer periods of time and thus added delay between the signal to the kernel and the process/thread getting its scheduled time slice isn't as important.

In this implementation, a pipe is used for the blocking I/O call. To bring a thread out of the inactive pool, a single byte is written to the pipe ending the blocking read. However, to avoid potential for a locked system where a blocking write is called but the thread is not in the inactive pool, and thus not waiting to read, writes to the pipe are non-blocking calls. Using a mutex would be largely equivalent to using a pipe, since it would also block, remove the thread from the scheduler, and incur a signal upon unlocking, but has portability challenges.

2.3 Process Communication

There are two common ways for processes to communicate: message passing and shared memory [9]. Message passing is very broad and can range from signals to pipes to sockets. This implementation effectively uses both message passing and shared memory as a means to allow asynchronous message passing via signals and the data delivered via shared memory segment.

The Qthread processes use shared memory to communicate necessary information to facilitate the partitioning of computation resources. Shared memory is powerful,

but still very restrictive in its usage compared to memory usage within an application. Since process memory is typically protected from access by other processes, it is not possible to put function pointers in the shared memory to allow a process to call a function into another. Even further, because virtual addresses are different for each process, it is not possible to reliably use pointers across processes for data within the shared memory. Due to these limitations, the partitioning implementation uses a combination of shared memory and signals.

The shared memory portion of the implementation stores the dynamic resource-allocation data structure that has a 64-bit unsigned integer to count active processes and an array of 64-bit signed integers to be used to provide data about the process, hereafter referred to as the "process array." The initialized size of the array is 511, which, combined with the 64-bit active process count variable, keeps the data structure inside a single 4-kilobyte memory page.

Entries in the process array use an encoding to provide information about the process. The encoding uses the most significant bit (MSB) to indicate if the process is in need of additional resources. This allows a simple sign check to determine if it needs resources. After the MSB, the next 31 bits are available for extending the implementation or adding functionality. The final 32 bits are the process ID, which provides a unique identification, as well as the means to send signals to other processes. If a system uses 64-bit process IDs, then the sign and extension space could be eliminated or moved to a secondary 32 or 64-bit integer. The code in Listing 2.1 shows the definitions of the shared memory section and bit masks.

Listing 2.1: Shared memory definitions

```
1 #define MAX_FIM_PROC 511
2 // procs encoding:
3 // 63 status
```

```
4  // 32:62 mailbox
5  // 0:31 pid
6
7  // bit encodings
8  #define FIM_MSB_0 0x7FFFFFFFFFFFFFFF
9  #define FIM_MSB_1 0x8000000000000000
10 #define FIM_LOW   0x00000000FFFFFFFF
11 #define FIM_HIGH  0xFFFFFFFF00000000
12
13 typedef struct fim_t {
14     uint64_t num_procs;
15     int64_t procs[MAX_FIM_PROC];
16 }fim_t;
```

Shared memory provides the necessary data sharing, but processes are not aware of changes to the shared data unless they are monitoring the shared data state. To notify all the processes of changes to shared memory, the user defined signals, `SIGUSR1` and `SIGUSR2`, are used to check if a process is still active/alive (see Section 2.6) or to repartition (see Section 2.5), respectively.

2.4 Data Protection

Using shared memory means that all participating processes have access to the shared data structure. Consequently, there can be a race condition when updating the shared data. Any code that updates the shared data structure is considered a "critical section" and needs protection for multiple simultaneous accesses.

The common method for protecting data is a mutex or semaphore, both part of the POSIX specification [13][12]. Mutexes and semaphores allow a programmer to lock a section, blocking access to other threads and processes, until the thread(s) or process(es) complete their execution of the critical section. These are common in multi-threaded programming in C. However, to provide mutual exclusion across processes, the mutex or semaphore must also reside in the shared memory section.

While a mutex or semaphore may be necessary in cases where the critical section is multiple instructions, they can hurt performance and can introduce the possibility of deadlocks if not programmed carefully.

An alternative to locking the data structure is to modify it only using atomic operations. This approach is viable as long as every possible state the data structure may be in is legal and coherent. Such operations can often be done with compiler built-in functions depending on the target processors [2]. Atomic operations provide common simple manipulations such as add or subtract, which are performed on the data atomically with the option to return either the new value or the old value depending on which atomic function is used. Beyond these simple operations, some additional atomic operations are provided, specifically atomic compare and swap (CAS). CAS operations take a pointer to the data being worked on, as well as an old value and a new value. If the data in memory is equal to the old value, then it is set to the new value, otherwise the data is not modified. Since the data is not modified if it is not equivalent to the old value, this provides safety in the case where another thread modified the data first.

Using a lock-free design for the partition data structure allows the implementation to completely avoid using blocking mechanisms while still preserving data integrity. When a process starts, it atomically increments the active process counter then iterates through the shared array until it finds an empty spot (zero value) and attempts a CAS operation. If the CAS is successful, the rank for that process is set; otherwise, it continues iterating and tries again on the next empty spot. If there is not an empty slot, then it will keep iterating over the shared array until the next empty slot becomes available. Alternatively, it could abort using the partitioning and revert to the standard Qthreads behavior of keeping all workers active at the

cost of the additional oversubscription on all processing units. With the current number of slots, how such an occurrence is managed is unlikely to have a significant impact because at that level of oversubscription the system will have already severely degraded performance.

2.5 Resource Partitioning

Partitioning the resources calculates the number of processing units each process will have based on the total number of both processing units and processes. Each process determines its rank, or relative position in the process array, compared to the other processes and calculates its portion of processing units using its rank. Partitioning reduces the overall amount of parallelism available by reducing the number of active workers, but in exchange provides each process with a set of workers with far fewer processes competing for the associated computational resources.

In cases where there are more parallel processes than processing units, the processes each get two workers: processing unit zero and one calculated based on the process rank. The process may be sharing its calculated processing unit with another process whose rank results in the same processing unit being calculated, and zero is shared by all the parallel processes. This is still a large reduction in contention for resources, however it is also a large reduction in available parallelism.

Once a process has added itself to the shared memory data structure, it broadcasts `SIGUSR2` to all other participating processes, which initiates a new partitioning calculation for all signaled processes. Since this is an expensive operation, it only occurs when a process starts and when a process terminates.

The repartition scheme checks if the calculated number of resources is less than

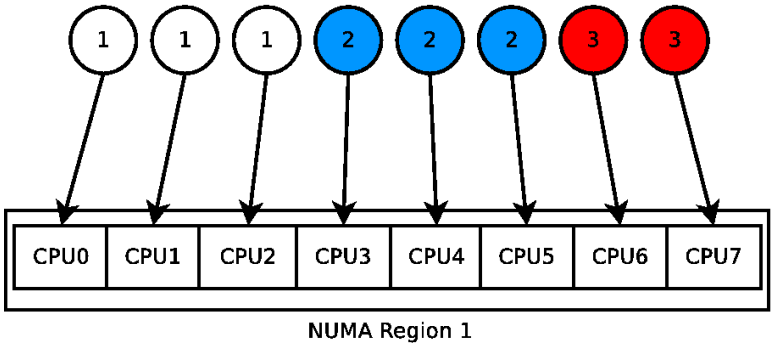


Figure 2.1: This is an example processing unit assignment for a single NUMA region configuration with three processes on an eight processing unit machine. Each circle is a process and the number is the process id, squares are processing units.

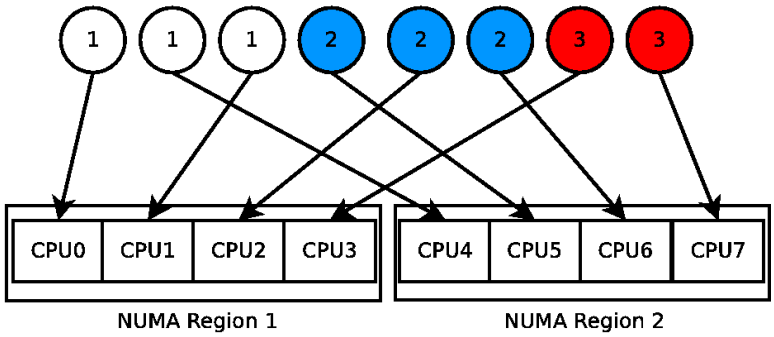


Figure 2.2: This is an example processing unit assignment for two NUMA regions with three processes on an eight processing unit machine. Each circle is a process and the number is the process id, squares are processing units.

one and ensures that every process retains at least one worker, plus worker zero. The system can handle more processes than processing units because the shepherd and worker indices are calculated such that after the worker-process relationship is one-to-one, processes will be oversubscribed incrementally and in order of the worker ID, as opposed to oversubscribing all processing units equally.

Repartitioning is disruptive to running processes because it can cause each process's set of workers to change, resulting in a cold cache, additional signals in the

system, and CPU cycles spent on repartitioning instead of the application. When a worker is disabled, it does not immediately go inactive because it may be currently processing work. It is likely that for workers that stay active for a process through a repartition that the deactivation and reactivation flags will happen in close enough succession that the thread will never actually go inactive and will be unaffected by repartition thrashing.

The partitioning attempts to keep the workers for a process grouped together, giving them a higher probability of belonging to the same NUMA region, and thus able to take advantage of cache locality. The grouping is achieved by using the process rank and the number of shepherds and workers. First, the number of workers for each process is calculated by taking the total number of workers and evenly dividing it by the number of active processes. If there is a remainder from the division, `x`, the first `y` processes, where `y`'s rank is less than `x`, receive an additional processing unit. See Figure 2.1 for a single-shepherd resource allocation and notice that processes 1 and 2 have three processing units while process 3 has two. The number of workers is then used with the process rank to calculate the shepherd and worker indices and then activates that worker and does the same calculation for each of the process's workers.

Qthreads calculates the shepherd and worker for a worker's ID number in a round robin disbursement to the NUMA regions. Figure 2.2 shows how a two-NUMA-region system would be allocated. Because of the round robin disbursement of processing units, processing units for an individual process are going to be split across NUMA regions. This means the benefits of caching among processing units may be reduced. This also increases the probability that a process's threads may be moved across NUMA regions during a repartition, which could result in additional cache-miss penalties. Updating the worker distribution relating to NUMA regions to align better

with the partitioning scheme is future work (see Section 6.2.2).

2.5.1 Worker Zero Constraint

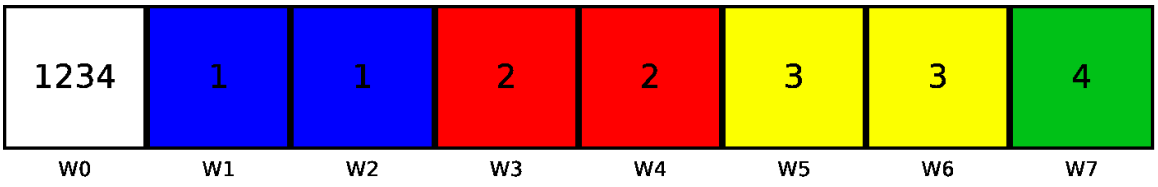


Figure 2.3: An example partitioning with four processes, eight processing units (labeled W0-W7), and the resulting oversubscription on worker zero.

Complicating the partitioning worker assignments, every process must keep worker zero active. This is due to the design of Qthreads, which uses the parent thread as one of the workers. Since the parent thread is a worker, the execution of any code from the parent application becomes a task. However, in order for proper shutdown, the context of the parent thread must return to the originating worker. The parent task cannot stay on worker zero if worker zero is disabled, since the parent task will never run, resulting in a potential deadlock. This means that worker zero is always oversubscribed by the number of processes using Qthreads (beyond any additional oversubscription happening on the system). Figure 2.3 shows an example partitioning with four processes, eight processing units, and the resulting oversubscription on worker zero. This is a significant bottleneck on this implementation. Attempts to allow disabling worker zero resulted in the stack of the parent thread getting corrupted after the task was returned to worker zero (see Section 6.2.3).

2.6 Error Handling

Error handling has many challenges when dealing the multiple processes and shared memory. Without a centralized process, detecting if a process is still running is difficult and if a process crashes without triggering a repartition then the resources allocated for that process will not be reclaimed until a different process initiates a repartition or another process attempts to send the crashed process a signal.

One option is to create a daemon that checks each process periodically to ensure no one crashed and adjust the shared data structure accordingly. A daemon is a relatively robust option, but would require additional resources since it would be a standalone process. A daemon would need access to the shared memory section, or could replace it as a centralized mechanism for communication, but messages would still be signal based. Additionally, a daemon adds an additional layer to the communication: if process C starts, it would need to communicate to the daemon that a repartition is necessary and the daemon would then broadcast the message to all the other processes. Contrast this with process C starting and directly broadcasting its arrival. A daemon would also need a schedule to check the processes.

A second option is to have each process check on the other processes on a time interval. This would likely be in the form of an alarm, which introduces additional signal handling and processing into the system. This is very similar to the daemon approach but removes that daemon layer of message passing.

A third option, and the one implemented, is a passive check. Whenever a process touches the shared memory, it will also send a signal to every process with a non-zero entry in the process array. If the signal fails, then the process is assumed dead and its entry is removed from the process array. Similarly, any time a process is sent a

signal, it doubles as checking that the process is still alive. This approach minimizes the overhead but could potentially allow for resources to go unused until the next process starts or finishes.

CHAPTER 3

TEST MACHINES

3.1 Variance of Environments

The problem and solution detailed in Chapters 1 and 2 were run on several different machines and software environments. The systems were chosen to represent common execution environments and to ensure the viability of the solution across operating systems, compilers, and hardware configurations. Four different operating systems were used in testing: Fedora, CentOS, and Ubuntu Linux distributions and OS X. Two servers, a desktop, and a laptop were used for testing.

The implementation was tested on multiple different machines, operating systems, etc., to ensure portability, show the problem is widespread, and determine how effective the solution is on multiple platforms. The specific versions of gcc, clang, automake, etc., should not have a significant impact, providing that they are consistent across all tests on that machine, and must minimally be able to support running Qthreads and the benchmark tests. The machines and environments provide newer and older hardware and software with various versions of compilers. For each of the machines, the latest `hwloc` version, 1.11, was compiled from source [14].

3.2 Desktop

The desktop machine used for testing uses a Intel Core i7-960 quad core CPU with hyperthreading, which provides eight processing units operating at 3.2 Ghz and 12 GB of DDR3 1600 RAM. The operating system is Ubuntu 14.04, with kernel 3.13.0, and `gcc` version 4.8.4, fully updated as of April 17, 2016. All software used was installed from Ubuntu's APT repository.

3.3 Laptop

The laptop used for testing is a 2015 MacBook Pro (MBP) 15" base model. This provides a 2.2 Ghz quad core Intel Core i7-4770HQ CPU with turbo boost up to 3.4 Ghz. This i7 also provides hyperthreading and eight processing units. The MBP has 16 GB of DDR3L 1600 RAM. The tests were run with a fully updated version of OS X 10.11, and used tools provided by the latest XCode package, including Apple LLVM version 7.3.0, as of April 17, 2016.

3.4 Shared Server

The shared server is a Linux server at Boise State University that is available to all Computer Science students. It represents a common usage case of a system shared by multiple users, where exclusive usage of the computational resources is not likely, and thus can impact the execution time of processes owned by other users. The shared server runs Fedora 22, with kernel 4.2.3, and has gcc version 5.1.1.

The shared server is a two-socket server with dual Intel Xeon E5-2630 CPUs, each with six cores and hyperthreading, giving the total system 24 processing units, at 2.4

Ghz. The shared server has 32 GB of DDR3 1600 RAM.

3.5 Dedicated Server

The dedicated server is another Linux server hosted at Boise State University that is not shared among a large group of people, so additional activity on the system is greatly reduced, making it a less dynamic testing environment. It has 64 GB of DDR3 2133 RAM, and processors similar to the shared server: dual Intel Xeon E5-2620 v3 processors, with six cores and hyperthreading, providing the system with 24 processing units at 2.4 Ghz. The dedicated server runs CentOS 6.7, with kernel 2.6.32, and gcc version 4.4.7.

CHAPTER 4

BENCHMARKS

4.1 Benchmark Selection

The benchmarks were selected as representations of common parallel application patterns. Unbalanced Tree Search creates a tree and visits each node, with each node visit being a new work task. This results in a large number of small tasks with the task computation being independent from other tasks. The High Performance Conjugate Gradient benchmark is a serial application that parallelizes loops with a smaller, set number of tasks, which is very common in scientific computing.

4.2 Unbalanced Tree Search

The Unbalanced Tree Search (UTS) benchmark is aimed at testing parallel processing load balancing. UTS generates a tree with each node having a probability of being a leaf and a branching factor. Each node performs a SHA1 hash as its work, which is used to determine whether it is a leaf node or not. The benchmark can quickly grow many nodes, with each node having an equal probability of producing more work. This makes it difficult to optimize scheduling. The benchmark uses a seed for random number generation allowing for deterministic tree generation, a requisite feature for repeatable results.

For all the tested implementations of UTS, every node is spawned as a task. When a node is visited, a counter is incremented that keeps track of the total number of nodes in the tree. The trees can grow very quickly with little variation in the branching factor and leaf probability, which results in massive time and memory requirements. Due to the rapid tree growth, a depth limit is additionally imposed on the trees to limit the number of nodes in the tree. When the depth limit is reached, it forces the current node to be a leaf node and the leaf probability routine is skipped.

The UTS benchmark was selected for testing because it produces a lot of work to execute and generates a large number of tasks (one per node in the tree), which keeps the processing units and the tasking scheduler busy. UTS itself does not introduce significant synchronization between tasks or threads, so it does not introduce additional jitter delays due to synchronization.

4.2.1 Test Trees

Three tree sizes were selected to be used in testing with the single tree generation time/size selected to keep execution time reasonable, even when there are many parallel processes generating the trees. The sizes are distinguished as small, medium, and large. The size is adjusted by changing the branching factor and tree depths. Table 4.1 shows the values and sizes of the trees.

Table 4.1: UTS Tree Variables

Test	Branching Factor	Tree Depth	Number of Nodes
small	5	55	871691
medium	5	70	24938666
large	6	35	53380645

4.3 High Performance Conjugate Gradient

High Performance Conjugate Gradient (HPCG) is a benchmark intended to behave like complex physics simulationsi [3]. HPCG generates a 3D matrix then iteratively performs a set of operations on the matrix. These operations include sparse matrix-vector multiplication, vector updates, global dot-products, and more. The Qthreads implementation of HPCG parallelizes only the loops, making the benchmark a sequence of parallel and serial sections.

Per the HPCG README included with the benchmark code, it is recommended that the size of tests be based on the memory footprint of the simulation, specifically recommending 25-75% of memory. However, in testing oversubscription, using even just 25% of memory would reach 100% of memory with four processes. Going over the physical memory limits, and spilling into swap space, would be too large of a performance hit to provide usable results within a rational time frame. Instead of basing the size on memory, a cube of size 100x100x100 elements was selected for testing, based on the execution time for a single process, which uses approximately 400 MB of memory, ensuring all tests stay well within the memory bounds of the test machines.

CHAPTER 5

RESULTS

The test results for UTS show that execution times of partitioning generally approximate the Qthreads yielding option, usually executing faster than the yielding method. On the laptop, the execution time disparity is extreme, with partitioning being much faster. On the servers, partitioning is clearly faster on the dedicated server, but on the shared server partitioning did not provide an advantage over yielding until twelve or more concurrent processes were run.

5.1 Testing

The laptop and desktop were rebooted before testing to provide a fresh environment. Testing on all machines was done "out of the box," meaning there was no special tuning done. Similarly, to keep testing representative of the average case, thread/process priorities were not changed for the test applications. For Qthreads, `hwloc` was used for the topology.

5.2 Qthreads Default Configuration

The Qthreads default configuration, compiled with no options passed to the configure script, was tested on the desktop and it quickly became apparent that it is not intended to handle multi-process execution, but instead is optimized for single-process

Table 5.1: Qthreads Default vs Yielding enabled

Test	Default	Variance	Yielding	Variance
small	0.189s	0.063s	0.326s	0.032s
medium	4.321s	0.140s	8.229s	1.003s
3 small	394.939s	445.499s	0.668s	0.054s
3 medium	3.71 hours	3.79 hours	17.509s	0.151s

environments, common to high performance computing. The Qthreads default does well with single applications, but even three concurrent processes shows severe performance degradation. Single small tests had an average runtime of 0.189s, medium tests had an average runtime of 4.300s, and large tests had an average runtime of 9.242s. However, running three small-test processes simultaneously ballooned the average time to 394.939s, and varied with a minimum time of 138.130s and a maximum time of 583.629s. Running three medium-size processes proved even more dire, with an average of 3.71 hours, with a minimum of 1.05 hours, and a maximum of 4.85 hours. The difference between the minimum and maximum medium tests, 3.79 hours, and small tests, 445.499s are larger than the averages themselves. The variability shows that jitter is severely and nondeterministically impacting performance, and that the architecture of Qthreads may be good for single-process performance, but it clearly does not handle multi-process jobs well. Table 5.1 shows the compared execution times of small and medium sized, individual, and three-process UTS tests, averaged over eight runs on the desktop computer. The three process times are the time it took all three processes to execute. Large tree-generating tests are not included due to the amount of time necessary to execute three concurrent processes.

The individual runs show that multi-process execution roughly doubles the execution time due to causing excessive context switches via yielding. However, there is an even more drastic difference when the machine is oversubscribed. The Qthreads

default performance, when oversubscribed, is untenable, growing from 0.189s for an individual, small test to 394.939s with just three processes, and taking almost four hours on average for three medium-size tests. Since Qthreads provides a yielding option, that will be the primary configuration used for comparison with partitioning.

5.3 UTS Results

5.3.1 Single UTS Process

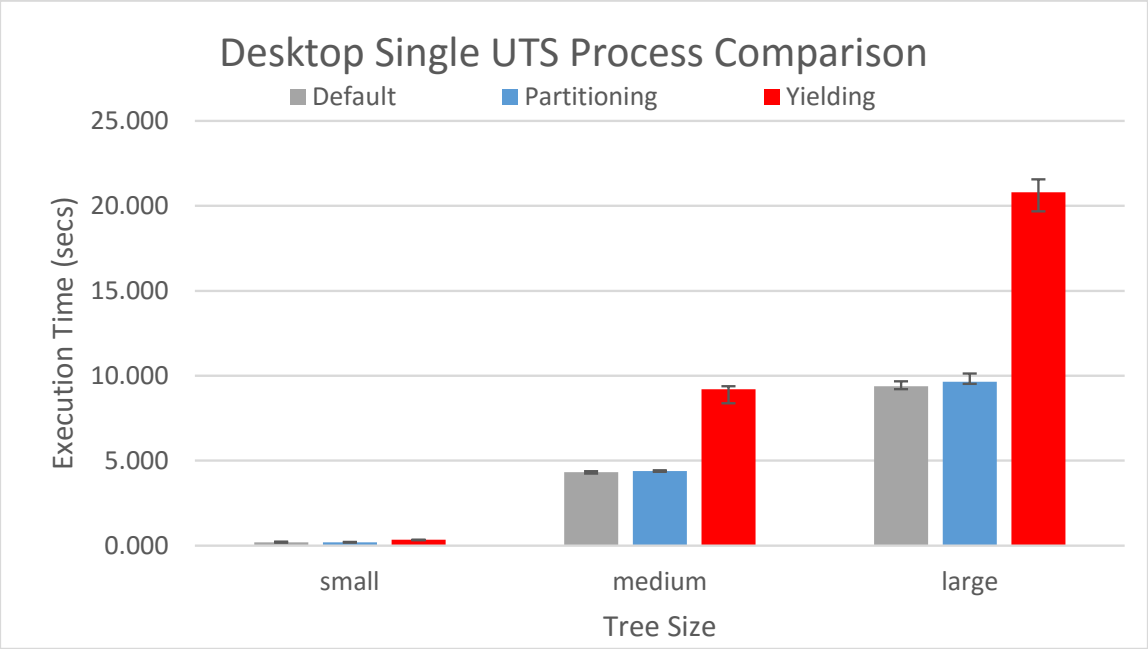


Figure 5.1: Single UTS process execution time using default, partition, and yielding options on the desktop.

As was mentioned in Section 1.2.1, the yielding option adds calls to yield the rest of the thread or processes time slice instead of spinning. While this is beneficial in oversubscribed situations, it inhibits single-process performance.

Table 5.2: Desktop Single Test Variance

Test	Yielding Variance	Partition Variance
small	0.032s	0.038s
medium	1.003s	0.090s
large	1.882s	0.607s

The single-process tests are the average of eight runs of tree generation for each tree size. Figure 5.1 shows the Qthreads default single-process performance compared to both yielding and partitioning, including the maximum and minimum execution times. This demonstrates that the overhead introduced by the standard yielding technique is substantial, while the single-process overhead of the partition scheme is minimal. Table 5.2 shows the variance between minimum and maximum execution times. Partitioning shows much less variance, which could indicate that there is less jitter due to the partitioning. However, the execution time difference makes yielding more susceptible to jitter just by having a larger time window available for system disruption.

The overhead of the startup and shutdown of partitioning was timed to measure the average overhead for a single process. The combined overhead averaged between 91 to 277 microseconds across the four test systems. Table 5.3 shows the combined startup and shutdown times of each system, over eight single-process UTS tests.

Table 5.3: Single Process Partitioning Overhead

System	Partitioning Overhead
desktop	127 µs
laptop	277 µs
shared server	91 µs
dedicated server	120 µs

5.4 Three UTS Processes

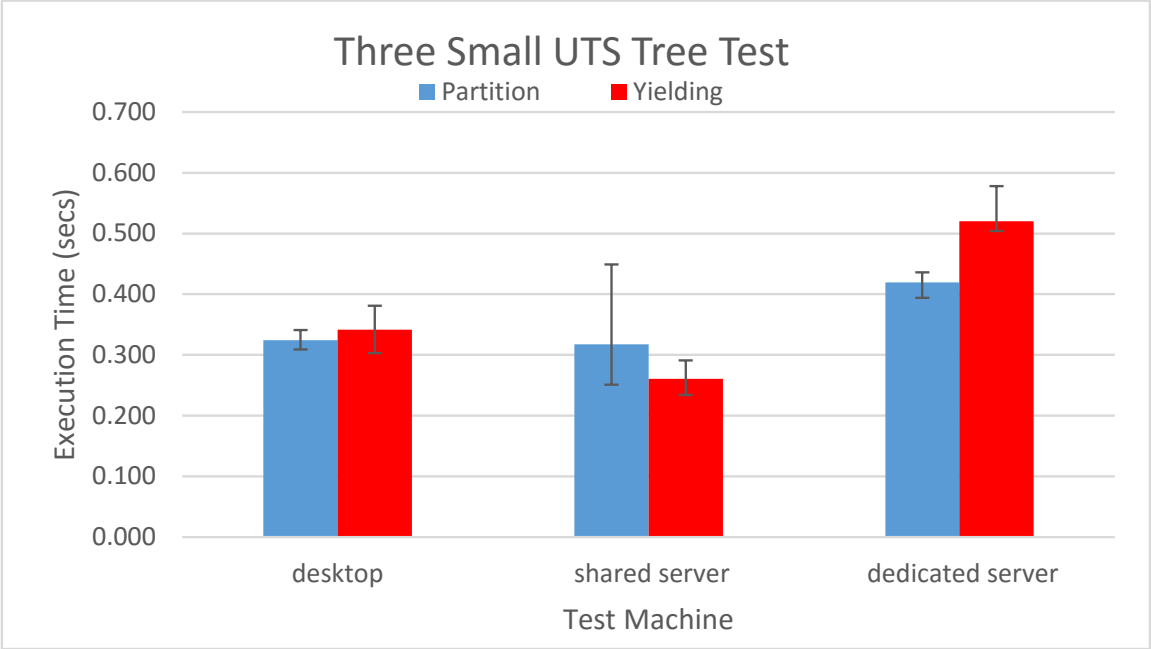


Figure 5.2: Total execution time of three UTS processes, in the small configuration, using partitioning and yielding. The black vertical lines represent the range of result values. Due to the variance of the execution times, the laptop results are shown in Figure 5.4.

For testing three UTS processes, three small-tree tests, three medium-tree tests, and three large-tree tests were run together. These tests were run on all four test systems.

Figure 5.2 shows the results of the three UTS process, small tree tests on the desktop, shared server, and dedicated server. Both the desktop and dedicated server show a performance edge to partitioning. However, the shared server had high variability with partitioning and was slower on average than yielding.

Figure 5.3 shows the results of the large tree, three process UTS experiment.

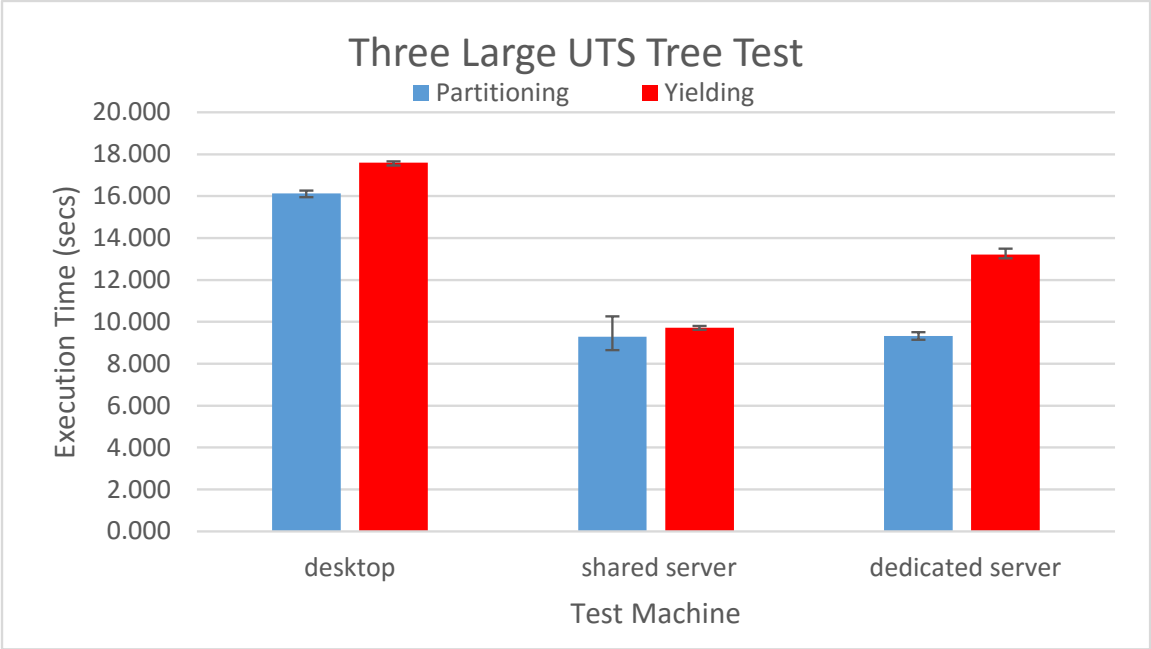


Figure 5.3: Total execution time of three processes, in the large configuration, using partitioning and yielding. The black vertical lines represent the range of result values.

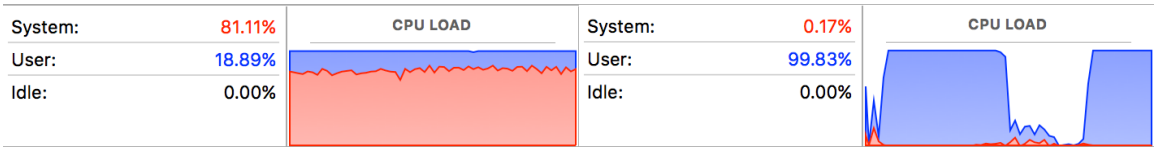
5.4.1 Laptop UTS Oversubscription

The laptop separates itself here in multi-process execution time, and thus is not included in Figure 5.3 and Figure 5.2, to prevent the laptop results from skewing the scale of the rest of the results. Apple OS X's scheduler is based on Mach and BSD, and is derived directly from the scheduler in OSFMK 7.3. Apple states that it will lower the priority of compute-bound processes and threads to avoid locking up the system and prevent starvation of I/O bound threads [5]. This results in erratic and poor performance for the oversubscription tests. Despite a reboot to start with a fresh system, and preventing as many startup processes as possible, the yielding configuration fared poorly. This did not have any apparent affect on the results for the partitioning configuration. Table 5.4 shows the laptop yielding and partition results.

Table 5.4: Laptop Three UTS Process Tests

Tree Size	Yielding	Partition
small	6.008s	0.396s
medium	133.968s	9.130s
large	296.364s	20.043s

The OS X Activity Monitor, Figure 5.4, provides an indication that part of the reason for the slowdown using the yielding option is that the yield calls and context switching are dominating execution of the UTS benchmark code. Figure 5.4a shows only roughly 20% of CPU cycles is going to user code during execution of a test with the yielding option enabled. Compare that to Figure 5.4b, which shows almost 100% of resources being used for user code during execution with partitioning enabled.



(a) Yielding (b) Partitioning

Figure 5.4: The OS X Activity Monitor shows full resource utilization for both tests. However only roughly 20% is for user code when using yielding while partitioning has almost 100% user code utilization.

5.5 UTS Scaling

In addition to the comparison results, the partitioning implementation was tested with varying numbers of processes to test how it scales. The scaling tests were run with the same methodology: each test result being an average comprised of eight runs. The desktop and laptop number of processes selected for testing are two, three, four, eight, and twelve. For the server, three, four, eight, twelve, and thirty-two

processes were run. The dedicated server was also tested with forty-two processes. A forty-two process test was not run on the shared server, due to it being shared with other students, and a forty-two process test would negatively impact anyone else on the system.

5.5.1 Desktop UTS Scaling Results

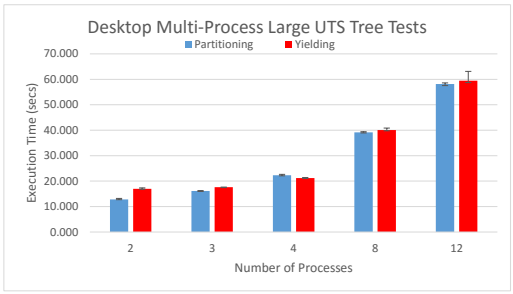
The desktop computer showed that resource partitioning consistently improved execution time over yielding, with the exception of the four-process test. Due to worker zero being used for all processes using partitioning, a four-process test results in three of the processes getting two dedicated workers but, process four only gets one. This is because worker zero is active for all processes, so worker zero is not in the partitioning group of workers. Figure 5.5a shows the large test comparison results. Both the small and large tests follow a similar curve, with partitioning generally being faster than yielding by small margins, except for the four-process tests.

5.5.2 Laptop UTS Scaling Results

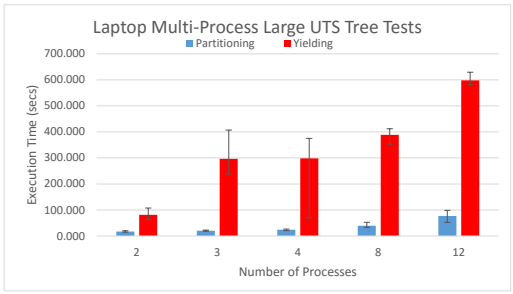
Similar to the three-process tests, the laptop results are heavily skewed due to its poor performance with the yielding option. Figure 5.5b shows the large tree UTS test comparison results. Partitioning is a clear winner, compared to yielding, in not just execution time, but also minimizing the variance in execution time between runs.

5.5.3 Shared Server UTS Scaling Results

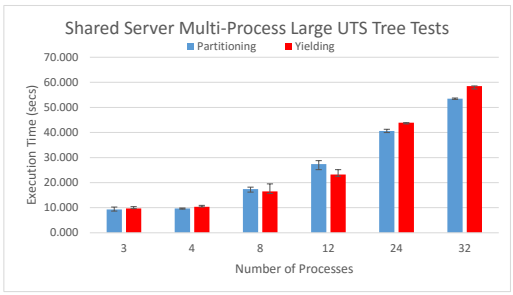
The results of the shared server are given in Figure 5.5c, for the large-tree UTS test comparison results. The shared server had results inconsistent with the other



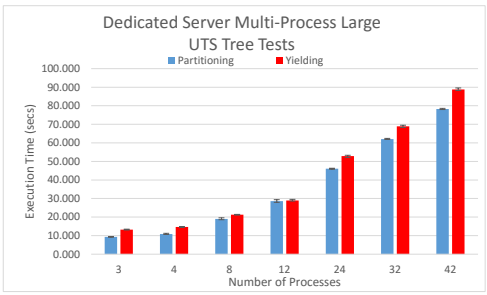
(a) Desktop



(b) Laptop



(c) Shared Server



(d) Dedicated Server

Figure 5.5: Total execution time of large tests comparing partitioning and yielding.

machines in that it had two tests where partitioning was slower than the yielding option.

Partitioning has lower execution time for all tests, except those with eight and twelve processes. For the servers, twelve processes is the same issue as four processes on the desktop described in Section 5.5.1, where the number of processes is half the number of processing units, resulting in all but one process getting two unique workers. However, it remains that the three, four, twenty-four, and thirty-two process tests are clear advantages for partitioning.

5.5.4 Dedicated Server UTS Scaling Results

Despite very similar hardware to the shared server, the dedicated server's results are not very similar. Figure 5.5d shows the large UTS-tree test comparison results. The dedicated server results show a clear performance advantage to partitioning across all tests. Since the dedicated and shared servers are so similar in hardware, the difference between their results must be due to either other users on the shared server or the differences in the software environments.

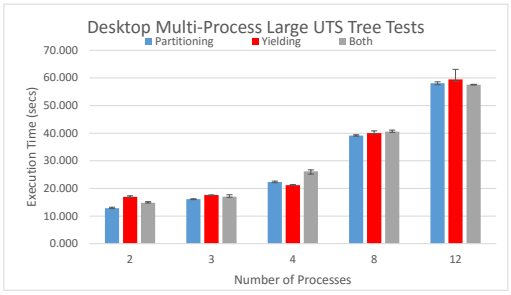
5.5.5 UTS Scaling Conclusion

The scaling tests show that partitioning has a strong performance advantage with fewer processes and, with the exception of OS X, stays pretty close to the yielding results. Since partitioning is effectively reducing parallelism available to the applications, this implies that reducing parallelism is better performing than the costs of oversubscription. This is true even when applications are reduced to two processing units, including worker zero, which is shared by all processes in the partitioning scheme.

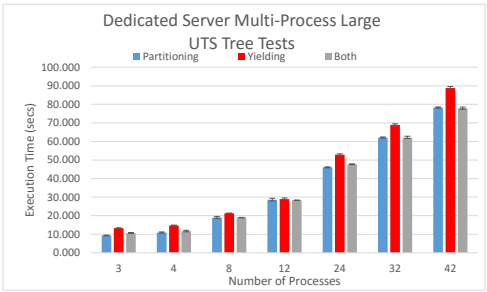
There is an interesting phenomenon in partitioning, when the number of processes is half the number of all processing units. Instead of each process getting two processing units, the result is that one process gets two, and the others get three, including the shared worker zero, due to the limitation that all processes share worker zero. The performance difference is likely a result of that imbalance, though the same would be expected to occur when the number of processes equals the total number of processing units, as well. In the case where the number of processes equals the number of processing units, the amount of oversubscription is likely to the point that

the effect of yielding does not effectively mitigate the contention, allowing partitioning to maintain a lead by keeping contention minimal.

5.6 Partitioning with Yielding



(a) Desktop



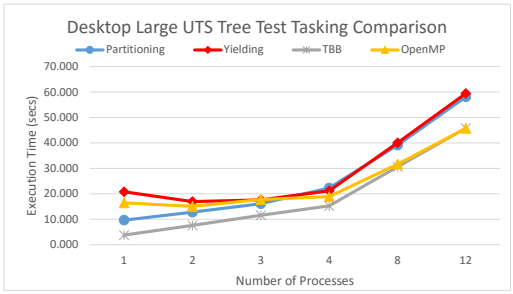
(b) Dedicated Server

Figure 5.6: Total execution time of large tests on the desktop and dedicated server comparing partitioning, yielding, and enabling both.

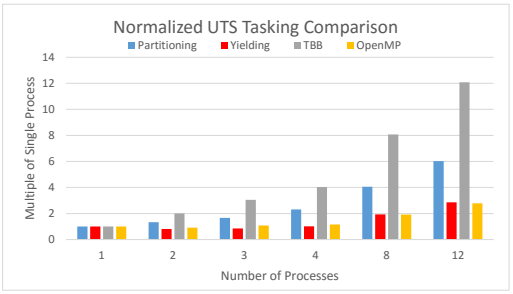
With partitioning, processing unit zero is still always oversubscribed, so it could possibly benefit from using the yielding approach. If both are good separately, then it would make sense that combining them should result in an even larger performance increase. This was not the case. Figure 5.6 shows the combined approach on both the desktop and dedicated server, using the large-tree tests. Mixing partitioning and yielding resulted in the poor behavior of yielding in single-process execution, and roughly equivalent performance to partitioning alone. Even in situations where partitioning results in oversubscription, adding the yields did not appear to make any real difference. Processing unit zero could still likely benefit from yielding, but that would also introduce additional checks (branches), to ensure only processing unit zero is yielding; this was not implemented or tested.

5.7 OpenMP and Intel Threading Building Blocks UTS Comparison

In addition to testing Qthreads, Intel Threading Building Blocks and OpenMP were tested with UTS. Each was tested with single-process executions, for a baseline, then tested with three-process, three tree tests, then run with two, four, eight, and twelve binomial tree processes. These tests were run on the desktop computer.



(a) Execution time



(b) Normalized on single process execution time

Figure 5.7: Large test results comparison with TBB, OpenMP, Qthreads oversubscribed, and Qthreads partitioning showing a comparison of raw execution time and execution time normalized to single-process execution time.

OpenMP was used without setting any environment variables for the tests, which should cause it to default to the number of processing units. However, its processing unit utilization was inconsistent. The OpenMP processes generally restricted themselves to only four processing units at a time for a single process, even when setting the OpenMP environment variable to use more threads. This behavior was observed on multiple machines to confirm it was not isolated to the desktop. Only using roughly four processing units at a time hurt the OpenMP performance considerably for the individual tests, but had the unexpected consequence of helping it in oversubscribed

situations.

Intel's Threading Building Blocks (TBB) offering clearly has a lot of work put into it, and the performance shows. Figure 5.7 shows the curves for OpenMP, TBB, Qthreads with yielding, and Qthreads with partitioning on the desktop computer in both raw execution time and normalized on single process execution time. The normalized graph shows TBB is linear and partitioning is super-linear. However, the normalized graphs for OpenMP and yielding, on the surface, show exceptional performance but that is an artifact of their poor single-process performance. TBB starts off with a sizeable performance advantage over Qthreads, on single-process execution, and maintains that lead through all testing. Monitoring the processing-unit load during OpenMP execution revealed that OpenMP was limited to the number of physical cores. It was not pinning processes to those cores causing execution to migrate across processing units, which is responsible for OpenMP having over seven seconds in variance on its eight large-test runs. The OpenMP curve shows that it struggles in single process performance, due to not utilizing all the processing units, but stays relatively flat as the number of processes increase.

While TBB is still affected by oversubscription, its performance edge on single-process performance carries over to oversubscribed situations as well. So, while Qthreads partitioning was not able to beat TBB or OpenMP oversubscribed performance, the curves of the tests are similar between partitioning and TBB. It is not clear whether OpenMP and TBB could benefit from partitioning, especially considering OpenMP is somewhat partitioning itself. However, compared to yielding, partition tracks much closer with TBB's performance, due to not having the performance hits in single and two process tests.

5.8 Coming and Going

One of the benefits of partitioning is its dynamic nature, making it a better approach than manually tuning processes because processes can start with full system resources, give some up when another process starts, then regain them when the second process ends. To show this, a few tests were constructed that start with a large test then have additional tests added at intervals to see how the performance fared.

The interspersed process tests are:

- LM: Start a large test, wait for three seconds, then start a medium test.
- LSSS: Start a large test, wait three seconds, start a small test, wait two seconds, start another small test, wait two more seconds and start a final small test.
- LMMM: Start a large test, wait three seconds, start a medium test, wait two seconds, start another medium test, wait two more seconds and start a final medium test.
- LLLL: Start a large test, wait three seconds, start a large test, wait two seconds, start another large test, wait two more seconds and start a final large test.

Table 5.5: Interspersed Tests

Test Name	Partition	Yielding
LM	10.843s	18.799s
LSSS	23.682s	20.489s
LMMM	15.218s	18.904s
LLLL	24.233s	27.084s

Table 5.5 shows the results of these tests with partitioning and yielding, run on the desktop computer, averaged over eight runs.

These results are expected, with partitioning generally being faster, which is consistent with other desktop results. However, the test with three small-tests interspersed resulted in very poor performance and reveals a big weakness in the partitioning: signal thrashing. What would happen is one small test would end at roughly the same time another one was starting, causing a lot of jitter due to signals, and also causing successive repartitions, which would harm the large test due to multiple repartitions.

5.9 HPCG Results

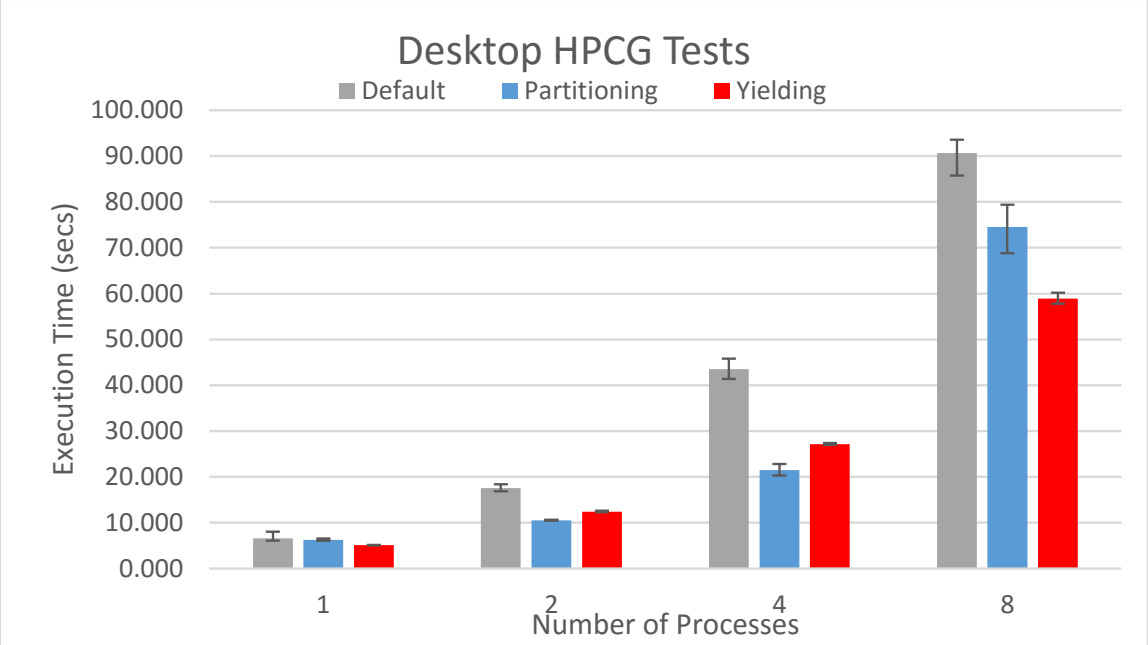


Figure 5.8: Total execution times of HPCG benchmark on the desktop comparing Qthreads default, partitioning, and yielding.

The HPCG results show a different parallel pattern than the UTS benchmark. UTS is a constant spawning of tasks and their execution. HPCG has serial sections

then parallelizes loops. Since HPCG is not alway parallel, the contention for resources is substantially reduced, because the parallel loop sections are unlikely to execute at exactly the same times, and thus won't likely contend for resources.

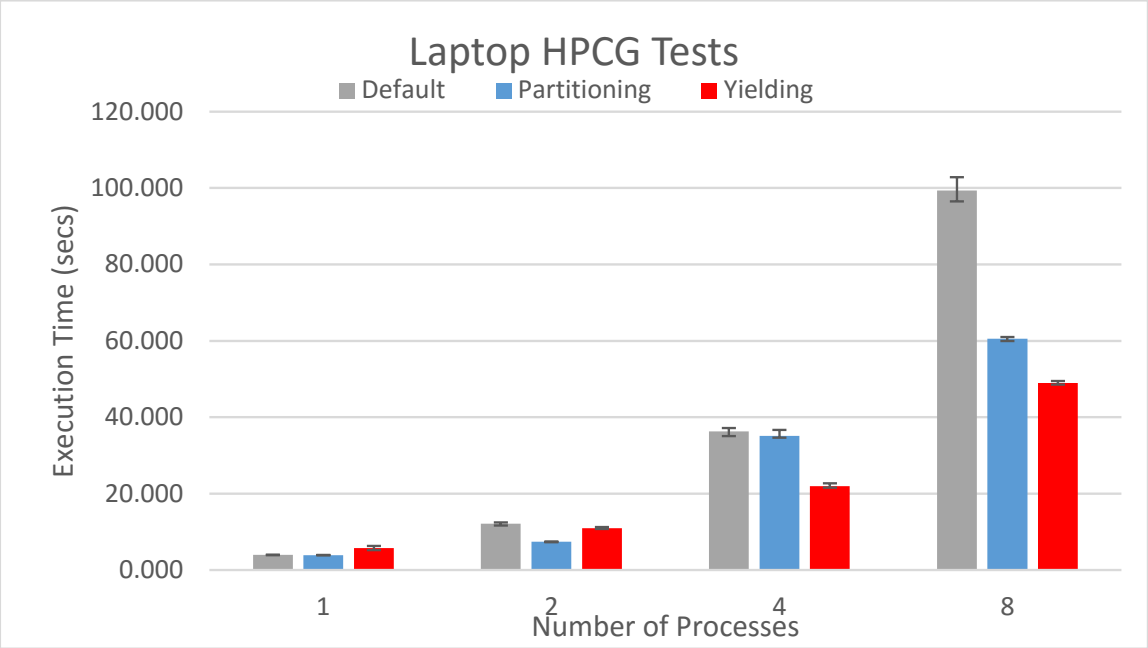


Figure 5.9: Total execution times of HPCG benchmark on the laptop comparing Qthreads default, partitioning, and yielding.

Partitioning reduces parallelism for the application's lifetime, regardless of the actual compute-resource contention. In the case of HPCG, this could work against partitioning in specific cases where the parallel loops of the concurrent processes do not overlap.

All the HPCG tests were run with one, two, four, and eight processes with the two servers additionally running twelve-process tests. Each test was run eight times, and the results are the average execution time of all the processes. In Section 5.2, the Qthreads default configuration was shown to perform very poorly in oversubscribed situations. However, HPCG's different model of parallelism works better with the

default configuration. Thus, the results include Qthreads default, the yielding option, and the partitioning scheme.

Figure 5.8 shows the HPCG results on the desktop. The yielding option is fastest for single process and is a very fast solution throughout. Partitioning is fastest for two and four processes, but is slower than the yielding option for one and eight.

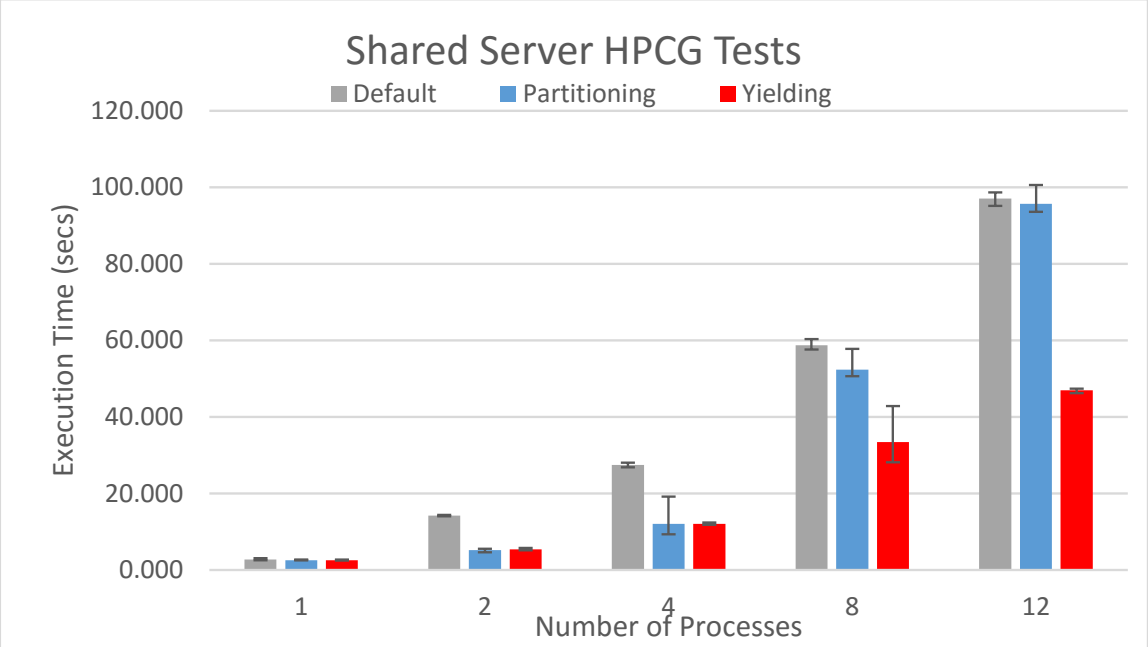


Figure 5.10: Total execution times of HPCG benchmark on the shared server comparing Qthreads default, partitioning, and yielding.

Figure 5.9 shows the results for the laptop. Partitioning is fastest for single and two process execution, but the yielding option has a strong performance advantage, compared to partitioning and the default configuration, at four and eight processes.

The shared server HPCG results can be found in Figure 5.10. On the shared server, all three were equivalent for single process. Partitioning and the yielding option were near identical for two and four processes. Eight process execution is dominated by

the yielding option, with partitioning still being faster than the default, but a distant second.

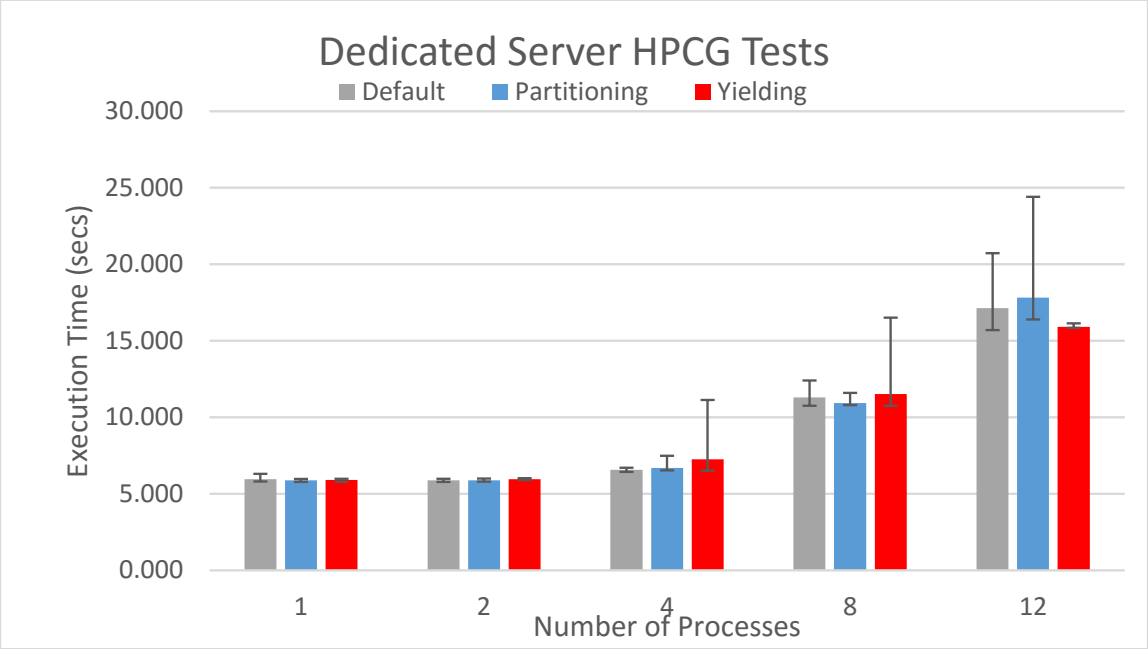


Figure 5.11: Total execution times of HPCG benchmark on the dedicated server comparing Qthreads default, partitioning, and yielding.

The dedicated server blazed through all the tests with single and two process being essentially a three-way tie. Four and eight process saw the default and partitioning be fastest, respectively, but not by significant margins. However, there was a lot of time variance for the yielding option, with four and eight processes, though its average was close to the others.

Partitioning does not show as drastic or consistent an improvement in execution time as it did with UTS. The yielding option was especially dominant in these tests, particularly in the eight-process test. However, partitioning still provides an improvement over the default configuration, and was fastest in some instances. This is likely due to HPCG's model of parallelism, which does not result in as much

contention for resources as a benchmark like UTS. Instead, HPCG benefits from all processes being able to use all cores, since the contention does not increase with the number of processes as rapidly as UTS.

5.10 Multi-User

An additional user account was created on the desktop computer to test if the shared memory section could be used across user accounts. From another computer, two separate accounts were logged into on the desktop, via `ssh`, and each one ran a single UTS large test with the default configuration and then with partitioning. The default configuration execution times were consistent with the previous desktop results.

Running with partitioning enabled had the same result. One issue was that of file permissions. While the shared memory section permission bits were set correctly, the two processes were still not able to both open the shared memory section. Putting both users in the same group fixed this. However, there remained another issue: users cannot send signals to processes owned by other users.

It may be possible to set the `CAP_KILL` capability on the processes, or find some other way around this limitation. However, the concept of being able to send signals to another user's processes is probably a bad idea and has many security implications. By changing the partition model to polling on changes in the shared memory section, sending signals could be avoided completely, but that would harm performance to gain a functionality that would be little, if ever, used.

It is possible to use a `pthread_cond`, instead of signals, for the notifications to other processes for repartitioning. However, this would require an additional thread for the conditional as well as storing a mutex and `pthread_cond` variable in the

partitioning shared memory. Additionally, the use of pthread mutexes for inter-process signaling is not supported on some common POSIX operating systems, such as Mac OS X.

CHAPTER 6

FUTURE WORK

6.1 Code

The implementation is complete and functional, however that does not mean the work is completely finished. The following items are still left to complete regarding the code.

- Logging: Simple logging to a file was implemented solely for the partitioning. Qthreads provides its own logging structure, so the partition implementation should be converted to use the regular Qthreads logging infrastructure.
- Merge with Qthreads Github repository: The code has not been merged into the Sandia National Labs git repository on Github.

6.2 Research

6.2.1 Resource Donation

Partitioning allows processes to determine the number of active workers they should have based on the number of participating processes running. However, the number of active workers remains tied to the number of participating processes. There are scenarios where a process may run out of work, or be executing a serial section, and

thus its workers are sitting idle. Since the process could be aware of its resource utilization, and since the partitioning scheme allows for communication across processes, it is possible a process with underutilized workers could donate a worker to a process that is in need of workers. Since the donating process would be the one with underutilized workers, the overhead of selecting a participating process to donate to would have little impact on the donating process. Additionally, since a donation would not be a repartition, it would have very little impact on the process receiving the donation.

Resource donation can happen when a worker doesn't have work to perform and can then donate a resource to another process, as long as the donating process retains at least one worker after the donation. To perform this, the donating process deactivates the worker to be donated, then iterates over the process array in shared memory until it finds a negative value (indicating that process is in need of resources). Once the receiving process is identified, the donating process copies the receiving processes' shared memory array entry, then sets the MSB to zero, and sets the worker ID field to the worker being donated. Finally, a compare and swap is done with the original value. If the compare and swap is successful, the donating process will then send `SIGUSR1` to the receiving process to notify it that it can activate the worker.

Additionally, when a process finishes execution and calls `finalize` on the Qthreads library, the process will signal for a repartition.

6.2.2 Worker Allocation

The current worker allocation is suboptimal on machines with multiple NUMA nodes/shepherds, as described in Section 2.5. The current calculation from worker ID to shepherd and processing unit indices is relatively simple. Converting the

calculation to change the grouping such that `n` consecutive worker IDs are mapped to consecutive processing units on a single shepherd, only crossing shepherd boundaries as necessary, instead of a round-robin shepherd assignment should not be necessarily difficult, but needs thorough testing for consistency and robustness across different topologies.

6.2.3 Unshackle Worker Zero

All processes being required to keep worker zero active was a strong restriction that undoubtedly hurt performance, due to oversubscription of worker zero. Allowing processes to enable and disable worker zero would reduce contention for all processes. Disabling worker zero may be possible if it is re-enabled before Qthreads is shut down. The first thing the Qthreads finalize function does is identify worker zero and only worker zero completes the cleanup using the parent thread. Thus, worker zero would need to be re-enabled before the Qthreads finalize function is called and the parent thread needs to be migrated back to worker zero.

Additionally, the protections around disabling worker zero also apply to shepherd zero. If the worker allocation from Section 6.2.2 is implemented, this could result in a shepherd being disabled, which should be a better solution, but will certainly have consequences not yet identified.

6.2.4 Benchmarks and Testing

There is always more testing to be done. This work focused on the UTS and HPCG benchmarks for testing, but there are many other benchmarks that could be useful, as well as different configurations of UTS and HPCG, and testing on more machines and topologies. Another benchmark that would be relatively easy to test is the

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [8], which is implemented with Qthreads in the `benchmarks` directory.

6.2.5 Tuning

There are some aspects of the partition implementation that could be fine tuned for better performance. For example, it allows for 511 process entries in the shared data structure, which means that for every partition and check that other processes are still active (error handling) the iteration over the array is all 511 elements. However, it is unlikely that this will be run in an environment with 511 processes on a single machine (in fact, server testing only reached a maximum of 42 processes) so that iterating over the shared array can be done more quickly. 511 was chosen to use a full page of memory, and provide scalability, so further testing could be done to bring this number down.

Similarly, the shared array uses `int64_t` for the array data, which allows for extensibility such as, but not limited to, resource donation, even though process IDs are a `uint32_t`. Changing the type to a `uint32_t` would make the data array much more compact, allowing either further scaling or just reducing the memory footprint. However, this does limit the extensibility, unless the shared data structure is altered.

6.2.6 Error Handling

There are many ways to handle errors in the partition implementation. The method currently employed was chosen for ease of implementation. However, it is only run when a process starts or exits cleanly, meaning processing units could remain allocated to a process that crashed and thus no longer exists. Additionally, the current error handling is based on sending signals, which increases jitter and can hurt performance.

This has no easy solution, and it may be worthwhile to explore user-selectable error-handling policies.

CHAPTER 7

CONCLUSION

This work conclusively shows that partitioning computational resources across concurrent parallel applications using the same tasking library shows an improvement in execution time. For the UTS benchmark, the performance of the default Qthreads configuration in a multi-process environment was severely impacted. However, the yielding feature brings the execution time closer to partitioning, even out-performing it in certain cases. For the HPCG benchmark, partitioning still performs very well, being the fastest in many cases, but cannot keep up with the yielding option in the eight-process tests. The yielding option brings downsides, such as poor single-process performance, and turning yielding on requires recompiling the tasking library. In contrast, partitioning maintains default Qthreads performance for single-process execution and often better performance than yielding, making it an ideal feature that can be turned on and left on with no discernible downside compared to the other options.

Partitioning was thoroughly tested with the UTS and HPCG benchmarks on both servers and personal machines, with four different operating systems and different environments and compilers. It was tested with single-process execution all the way up to 42 concurrent processes, resulting in almost a complete loss of parallel ability and having almost all processing units being oversubscribed, and still returned

performance increases over the yielding feature by reducing resource contention. Partitioning has many uses and allows people to run concurrent parallel applications without trying to manually account for oversubscription and without sacrificing performance when not oversubscribed.

REFERENCES

[1] P. De, R. Kothari, and V. Mann. Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *Cluster Computing, 2007 IEEE International Conference on*, pages 331–340, Sept 2007.

[2] GCC. Atomic builtins - using the gnu compiler collection (gcc). `https://gcc.gnu.org/onlinedocs/gcc-4.5.0/gcc/Atomic-Builtins.html`. Accessed: 2016-03-18.

[3] HPCG. HPCG: High performance conjugate gradient benchmark. `http://www.hpcg-benchmark.org/`. Accessed: 2016-04-04.

[4] C. Iancu, S. Hofmeyr, F. Blagojevi, and Y. Zheng. Oversubscription on multicore processors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11, April 2010.

[5] Apple Inc. `https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html`. Accessed: 2016-04-03.

[6] Intel. Intel threading building blocks. `https://software.intel.com/en-us/tbb-documentation`. Accessed: 2016-02-14.

[7] Intel. Intel threading building blocks - appendix a costs of time slicing. `https://software.intel.com/en-us/node/506127`. Accessed: 2016-04-14.

[8] Lawrence Livermore National Laboratory. Livermore unstructured lagrangian explicit shock hydrodynamics (lulesh). `https://codesign.llnl.gov/lulesh.php`. Accessed: 2016-04-04.

[9] T. J. LeBlanc and E. P. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*, pages 254–263, Dec 1992.

[10] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[11] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[12] POSIX Programmer's Manual. *PTHREAD_MUTEX_LOCK(P)*, 2003. Accessed: 2016-03-28.

[13] POSIX Programmer's Manual. *SEM_OVERVIEW(7)*, May 2012. Accessed: 2016-03-28.

[14] Open MPI. Portable hardware locality (hwloc). `https://www.open-mpi.org/projects/hwloc/`. Accessed: 2016-04-14.

[15] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. Multicore resource management. *IEEE Micro*, 28(3):6–16, May 2008.

[16] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, and Jan F. Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pages 49–56, New York, NY, USA, 2011. ACM.

[17] OpenMP. Openmp application programming interface. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`. Accessed: 2016-02-14.

[18] J. H. Schonherr, J. Richling, and H. U. Heiss. Dynamic teams in openmp. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 231–237, Oct 2010.

[19] K.B. Wheeler, R.C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

APPENDIX A

IMPLEMENTATION CODE

Listing A.1 shows the core implementation code for the partitioning. The partitioning has an initialization called at startup that creates the shared memory section and then signals the other participating processes to partition. Similarly, there is a cleanup function that also tells the other participating processes to partition but unlinks the shared memory section and closes all open files. These functions are called in the Qthreads initialize and finalize functions, respectively. This file also contains the signal handling for the partition scheme, the struct held in the shared memory, and the first pass at two functions for resource donation beyond just the partitioning.

Beyond the partition source file, there were some minor updates to the Qthreads code. Listing A.2 shows the updated wait-loop uses a blocking read to wait for partitioning instead of the spinlock. Similarly, Listing A.3 shows the updated `enable_worker` function that uses a write to a pipe to enable the worker.

This implementation could be reused for other applications and tasking libraries with little modification. However, it was necessary to use some data internal to Qthreads so this is specific to the Qthreads tasking library and is not a generic implementation.

Listing A.1: partition.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
```

```
3  #include <sys/mman.h>
4  #include <fcntl.h>
5  #include <unistd.h>
6  #include <sys/stat.h>
7  #include <inttypes.h>
8  #include <errno.h>
9  #include <signal.h>
10 #include <unistd.h>
11
12 #include ``partition.h''
13 #include ``qthread.h''
14 #include ``qt_shepherd_innards.h''
15 #include ``qthread_innards.h''
16
17 // used 511 as limit to keep shared data structure within a single page.
18 #define MAX_FIM_PROC 511
19 #define VERBOSE        0
20 // procs encoding:
21 // 63 status
22 // 32:62 mailbox
23 // 0:31 pid
24
25 // bit encodings
26 #define FIM_MSB_0 0x7FFFFFFFFFFFFFFF
27 #define FIM_MSB_1 0x8000000000000000
28 #define FIM_LOW   0x00000000FFFFFFFF
29 #define FIM_HIGH  0xFFFFFFFF00000000
30
31 typedef struct fim_t {
32     uint64_t num_procs;
33     int64_t procs[MAX_FIM_PROC];
34 }fim_t;
35
36 // variables
37 static uint32_t my_rank;
38 static int      fd;
39 static int      inited           = 0;
40 static uint32_t num_active_procs = 0;
41 static fim_t   *fim              = NULL;
42 // fim_logging
43 FILE    *fim_log         = NULL;
44
45 // functions
46 static void signal_handler(int);
47 static void donate_all(void);
48 static void get_resources(void);
49 static void pulse(void);
50 static void init_repart(void);
51
52 int init_partition(){
53     // STARTUP
```

```
54      // init pipes
55      for (int x = 0; x < qlib->nshepherds; x++) {
56          for (int y = 0; y < qlib->nworkerspershep; y++) {
57              int fids[2];
58              if (pipe(fids) < 0) {
59                  fprintf(stderr, ''Unable to create partition pipes, reverting to default behavior\n'');
60                  return -1;
61              }
62              qlib->shepherds[x].workers[y].read_pipe_fd = fids[0];
63              qlib->shepherds[x].workers[y].write_pipe_fd = fids[1];
64              fcntl(qlib->shepherds[x].workers[y].write_pipe_fd, F_SETFL, O_NONBLOCK);
65          }
66      }
67      // setup signal handlers
68      if (signal(SIGUSR1, signal_handler) == SIG_ERR) {
69          printf(``unable to receive signals(1)\n'');
70          return -1;
71      }
72      if (signal(SIGUSR2, signal_handler) == SIG_ERR) {
73          printf(``unable to receive signals (2)\n'');
74          return -1;
75      }
76      // create and init shared memory
77      if (VERBOSE) {
78          printf(``opening shared memory\n'');
79      }
80      if((fd = shm_open(``/fim'', O_RDWR | O_CREAT, 0666)) < 0) {
81          printf(``fd is negative\n'');
82          return -1;
83      }
84      struct stat *stats = calloc(1, sizeof(struct stat));
85      fstat(fd, stats);
86      if (0 == stats->st_size) {
87          // printf(``fd is %d, filesize is %lld\n'', fd, stats->st_size);
88          if (-1 == ftruncate(fd, sizeof(fim_t))) {
89              // printf(``ftruncate failed %s\n'', strerror(errno));
90              // return -1;
91          }
92      }
93      free(stats);
94      fim = mmap(NULL, sizeof(fim_t), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
95      if(MAP_FAILED == fim) {
96          printf(``mmap failed\n'');
97          return -1;
98      }
99      pid_t my_pid = getpid();
100     // effectively ignore ret val since that's not guaranteed to be rank
101     my_rank = __sync_fetch_and_add(&(fim->num_procs),1);
102     // There may be gaps in the array from processes
103     // so try to fill those first
104     my_rank = 0;
```

```
105     do {
106         if (__sync_bool_compare_and_swap(
107                 &fim->procs[my_rank], 0, my_pid)) {
108             if (VERBOSE) {
109                 printf(``my rank is %d and my pid is %d\n'', my_rank, my_pid);
110             }
111             break;
112         }
113         my_rank++;
114     } while (1);
115
116     // setup fim_log
117     if (VERBOSE) {
118         char filename[32];
119         sprintf(filename, ``fim%d.log'', my_rank);
120         printf(``logging to %s\n'', filename);
121         fim_log = fopen(filename, ``w+'');
122     }
123     if (1 < fim->num_procs) {
124         init_repart();
125     }
126     inited = 1;
127     if (VERBOSE){
128         fprintf(fim_log,``fim has initialized properly\n'');
129     }
130     return 0;
131 }
132
133 void init_repart() {
134     int num_dead = 0;
135     for (int proc_idx = 0; proc_idx < MAX_FIM_PROC; proc_idx++) {
136         if (proc_idx == my_rank || 0 == fim->procs[proc_idx]) {
137         continue;
138     }
139     // pid section should never change so atomic unnecessary
140     pid_t pid = fim->procs[proc_idx] & FIM_LOW;
141     if (0 != kill(pid, SIGUSR2)) {
142         if (VERBOSE) {
143             fprintf(fim_log, ``signal to %d failed. %s\n'', pid, strerror(errno));
144         }
145         fim->procs[proc_idx] = 0;
146         num_dead++;
147         proc_idx = 0;
148     } else {
149         if (VERBOSE) {
150             fprintf(fim_log, ``signal to %d succeeded.\n'', pid);
151         }
152     }
153     if (VERBOSE) {
154         fflush(fim_log);
155     }
```

```
156      }
157      __sync_sub_and_fetch(&(fim->num_procs),num_dead);
158      get_resources();
159  }
160
161  static uint32_t get_raw_index() {
162      int raw_index = 0;
163      int x =0;
164      while (x != my_rank) {
165          if (0 != fim->procs[x]) {
166              raw_index++;
167          }
168          x++;
169      }
170      return raw_index;
171  }
172
173  void get_resources() {
174      num_active_procs = 0;
175      if (VERBOSE){
176          fprintf(fim_log,``getting resources\n'');
177          fflush(fim_log);
178      }
179      //-1 to account for 0,0 required for all processes
180      int total_pu = (qlib->nshepherds * qlib->nworkerspershep) - 1;
181      if (!total_pu) {
182          return;
183      }
184      int expected_recs = total_pu / fim->num_procs;
185      if (1 > expected_recs) {
186          expected_recs = 1;
187          set_needy();
188          //+1 to shift off 0,0
189          int gworker = (get_raw_index() % total_pu) + 1;
190          for (int x = 0; x < qlib->nshepherds; x++) {
191              for (int y = 0; y < qlib->nworkerspershep; y++) {
192                  if (0 == x && 0 == y) {
193                      continue;
194                  }
195                  int curr_id = qlib->shepherds[x].workers[y]
196                          .packed_worker_id;
197                  if (curr_id == gworker) {
198                      if (VERBOSE) {
199                          fprintf(fim_log, ``enabling %d\n'', curr_id);
200                          fflush(fim_log);
201                      }
202                      qthread_enable_worker(gworker);
203                  } else {
204                      if (VERBOSE) {
205                          fprintf(fim_log, ``disabling %d\n'', curr_id);
206                          fflush(fim_log);
```

```
207                     }
208                     qthread_disable_worker(curr_id);
209                 }
210             }
211         }
212         qthread_enable_worker(gworker);
213         num_active_procs++;
214     } else {
215         int pu_mod = total_pu % fim->num_procs;
216         uint32_t raw_rank = get_raw_index();
217         // add any extra workers to processes in order of availability
218         expected_recs += (raw_rank < pu_mod) ? 1 : 0;
219         int temp_rank = raw_rank * expected_recs;
220         if (raw_rank >= pu_mod) {
221             // account for the mod cpus
222             temp_rank += pu_mod;
223         }
224         //+1 to shift off 0,0
225         int gworker = (temp_rank % total_pu) + 1;
226         int curr_id = 1;
227         for (int x = 0; x < qlib->nshepherds; x++) {
228             for (int y = 0; y < qlib->nworkerspershep; y++) {
229                 if (0 == x && 0 == y) {
230                     continue;
231                 }
232                 if (gworker == curr_id && expected_recs > 0) {
233                     if (VERBOSE) {
234                         fprintf(fim_log, ''enabling %d\n``, curr_id);
235                         fflush(fim_log);
236                     }
237                     qthread_enable_worker(curr_id);
238                     gworker++;
239                     num_active_procs++;
240                     expected_recs--;
241                 } else {
242                     if (VERBOSE) {
243                         fprintf(fim_log, ''disabling %d\n``, curr_id);
244                         fflush(fim_log);
245                     }
246                     qthread_disable_worker(curr_id);
247                 }
248                 curr_id++;
249             }
250         }
251     }
252     if (VERBOSE){
253         fprintf(fim_log,``%u received resources, total procs: %d\n'', my_rank, num_active_procs);
254         fflush(fim_log);
255     }
256 }
257
```

```
258 int donate(int64_t proc_unit_id){
259     if (1 >= num_active_procs || 0 == proc_unit_id) {
260         // fail, can't have zero resources and can't disable 0,0
261         return 1;
262     }
263     fim->procs[my_rank] = fim->procs[my_rank] & FIM_MSB_0;
264     int proc_idx = 0;
265     for (int x = 0; x < fim->num_procs; x++) {
266         while (0 == fim->procs[proc_idx]) {
267             proc_idx++;
268         }
269         if (0 > fim->procs[proc_idx]) {
270             int64_t old_val = fim->procs[proc_idx];
271             pid_t   pid     = fim->procs[proc_idx] & FIM_LOW;
272             int64_t new_val = fim->procs[proc_idx] | (proc_unit_id << 32);
273             if(__sync_bool_compare_and_swap(&fim->procs[proc_idx], old_val, new_val)){
274                 if (0 == kill(pid, SIGUSR1)) {
275                     qthread_disable_worker(proc_unit_id);
276                     num_active_procs--;
277                     if (VERBOSE){
278                         fprintf(fim_log,``sending donation, total procs: %d\n'', num_active_procs);
279                     }
280                     return 0;
281                 } else {
282                     fim->procs[proc_idx] = 0;
283                     __sync_sub_and_fetch(
284                     &(fim->num_procs),1);
285                     init_repart();
286                 }
287             }
288         }
289     }
290     // no needy proc found, no donation occurred
291     return 1;
292 }
293
294 void donate_all() {
295     if (VERBOSE){
296         fprintf(fim_log,``%u shutting down, final total procs: %d\n'', my_rank, num_active_procs);
297     }
298     fim->procs[my_rank] = fim->procs[my_rank] & FIM_MSB_0;
299     int curr_proc = 0;
300     int first     = 1;
301     for (int x = 0; x < qlib->nshepherds; x++) {
302         for (int y = 0; y < qlib->nworkerspershep; y++) {
303             // skip 0,0 because it can't be disabled
304             if (1 == first) { first = 0;
305                 continue;
306             }
307             if (qlib->shepherds[x].workers[y].active) {
308                 if (VERBOSE) {
```

```
309                     fprintf(fim_log,``%u sending resource %d\n'', my_rank,
310                                         qlib->shepherds[x].workers[y].packed_worker_id);
311                 }
312                 while (curr_proc == my_rank || 0 == fim->procs[curr_proc]) {
313                     curr_proc++;
314                 }
315                 if (curr_proc >= fim->num_procs) {
316                     curr_proc = curr_proc % fim->num_procs;
317                 }
318                 int64_t old_val = fim->procs[curr_proc];
319                 pid_t   pid     = fim->procs[curr_proc] & FIM_LOW;
320                 uint64_t proc_unit_id =
321                 qlib->shepherds[x].workers[y].packed_worker_id;
322                 int64_t new_val = fim->procs[curr_proc] | (proc_unit_id << 32);
323                 if(__sync_bool_compare_and_swap(&fim->procs[curr_proc], old_val, new_val)){
324                     if (0 == kill(pid, SIGUSR1)) {
325                         qthread_disable_worker(qlib->shepherds[x].workers[y].packed_worker_id);
326                         num_active_procs--;
327                         curr_proc++;
328                     } else {
329                         fim->procs[curr_proc] = 0;
330                         __sync_sub_and_fetch(
331                                 &(fim->num_procs), 1);
332                         init_repart();
333                     }
334                 }
335             }
336         }
337     }
338 }
339
340 void pulse() {
341     if (VERBOSE) {
342         fprintf(fim_log,``\n\n%u sending pulse\n\n'', my_rank);
343         fflush(fim_log);
344     }
345     int curr_proc = 0;
346     for (int curr_proc = 0; curr_proc < MAX_FIM_PROC; curr_proc++) {
347         if (0 == fim->procs[curr_proc] || my_rank == curr_proc) {
348             continue;
349         }
350         pid_t pid = fim->procs[curr_proc] & FIM_LOW;
351         if (0 > fim->procs[curr_proc]) {
352             fim->procs[curr_proc] = pid;
353             fim->procs[curr_proc] |= FIM_MSB_1;
354         } else {
355             fim->procs[curr_proc] = pid;
356         }
357         if (0 != kill(pid, SIGUSR1)) {
358             fim->procs[curr_proc] = 0;
359             __sync_sub_and_fetch(&(fim->num_procs),1);
```

```
360             init_repart();
361             break;
362         }
363         curr_proc++;
364     }
365 }
366 
367 void set_needy() {
368     int64_t newval = fim->procs[my_rank] | FIM_MSB_1;
369     int64_t oldval = fim->procs[my_rank] & FIM_MSB_0;
370     if (__sync_bool_compare_and_swap(&fim->procs[my_rank], oldval, newval)) {
371         pulse();
372         if (VERBOSE) {
373             fprintf(fim_log, ``set_needy\n'');
374         }
375     }
376 }
377 
378 int cleanup_partition(){
379     if (VERBOSE) {
380         fprintf(fim_log,``\n\n%u shutting down\n\n'', my_rank);
381         fflush(fim_log);
382     }
383     fim->procs[my_rank] = 0;
384     uint64_t active = __sync_sub_and_fetch(&(fim->num_procs),1);
385     if (0 < fim->num_procs) {
386         if (VERBOSE) {
387             fprintf(fim_log,``%d initing repartition\n'', my_rank);
388         }
389         for (int proc_idx = 0; proc_idx < MAX_FIM_PROC; proc_idx++) {
390             if (0 == fim->procs[proc_idx]) {
391                 continue;
392             }
393             // pid section should never change so atomic unnecessary
394             pid_t pid = fim->procs[proc_idx] & FIM_LOW;
395                 if (0 != kill(pid, SIGUSR2)) {
396                     // clean up dead procs
397                     fim->procs[proc_idx] = 0;
398                     __sync_sub_and_fetch(
399                             &(fim->num_procs), 1);
400                 }
401         }
402     }
403     if (VERBOSE){
404         fprintf(fim_log,``num active is now %''PRId64``\n'', active);
405     }
406     close(fd);
407     if (VERBOSE) {
408         fprintf(fim_log,``%d closing worker pipes\n'', my_rank);
409     }
410     for (int x = 0; x < qlib->nshepherds; x++) {
```

```
411         for (int y = 0; y < qlib->nworkerspershep; y++) {
412             close(qlib->shepherds[x].workers[y]
413                     .read_pipe_fd);
414             close(qlib->shepherds[x].workers[y]
415                     .write_pipe_fd);
416         }
417     }
418     if (VERBOSE && fim_log) {
419         fclose(fim_log);
420     }
421     shm_unlink(``/fim'');
422     return 0;
423 }
424
425 void signal_handler(int sig) {
426     if (SIGUSR1 == sig) {
427         if (!inited) {
428             return;
429         }
430         int64_t my_val = fim->procs[my_rank];
431         if (VERBOSE){
432             fprintf(fim_log,``RECEIVED SIGUSR1\n\n'');
433         }
434         my_val &= FIM_MSB_0;
435         // get process unit num
436         uint32_t pu_num = ((my_val & FIM_HIGH) >> 32);
437         if (0 == pu_num) {
438             if (VERBOSE) {
439                 fprintf(fim_log, ``SIGUSR1 was 0\n'');
440                 fflush(fim_log);
441             }
442             return;
443         }
444         // use pu_num resource
445         qthread_enable_worker(pu_num);
446         num_active_procs++;
447         if (VERBOSE){
448             fprintf(fim_log,``receiving donation %d, total procs: %d\n'', pu_num, num_active_procs);
449         }
450     } else if (SIGUSR2 == sig) {
451         if (!inited) {
452             return;
453         }
454         if (VERBOSE){
455             fprintf(fim_log, ``RECEIVED SIGUSR2\n\n'');
456         }
457         get_resources();
458     }
459 }
```

Listing A.2: qthread.c

```
while (!QTHREAD_CASLOCK_READ_UI(me_worker->active)) {
#ifdef HAVE_PARTITION
    qt_spawncache_flush(threadqueue);
    char buf[2];
    if (read(me_worker->read_pipe_fd, buf, sizeof(buf)) < 0) {
        break;
    }
#else
    SPINLOCK_BODY();
#endif
}
```

Listing A.3: worker.c

```
void API_FUNC qthread_enable_worker(const qthread_worker_id_t w)
{
    assert(qthread_library_initialized);

    unsigned int shep   = w % qlib->nshepherds;
    unsigned int worker = w / qlib->nshepherds;

    assert(shep < qlib->nshepherds);

    if (worker == 0) {
        qthread_enable_shepherd(shep);
    }
    qthread_debug(SHEPHERD_CALLS, ``began on shep(%i)\n'', shep);
    if (worker < qlib->nworkerspershep) {
        qthread_internal_incr(&(qlib->nworkers_active), &(qlib->nworkers_active_lock), 1);
#ifdef HAVE_PARTITION
        __sync_bool_compare_and_swap(&qlib->shepherds[shep].workers[worker].active, 0, 1);
        if (write(qlib->shepherds[shep].workers[worker].write_pipe_fd, ``\0'', 1) < 0) {
            qthread_debug(SHEPHERD_CALLS, ``write failed in partition\n'');
        }
#else
        (void)QT_CAS(qlib->shepherds[shep].workers[worker].active, 0, 1);
#endif
    }
}
```