ABSTRACT
Modern multi-core libraries do an excellent job of abstracting the details of thread programming away from the programmer, delivering a convenient interface for developing portable parallel applications. With a rich collection of synchronization primitives and lock-free data structures, these libraries allow the programmer to easily manage their programs. Their weakness, however, is that at initialization these applications typically launch as many threads as there are processors, so if there were two instances of the library present there would be twice as many threads as CPUs, in addition to any other processes running on the system. The result would be decreased performance for all applications on the system due to overloading of the CPU. Applications using shared resources are especially effected.

Improving lock performance has been the traditional approach to improving performance in systems under high load. While it has proven helpful in the past, such an approach only scales well to a certain point, and still may not provide optimal performance. As we are reaching a point where parallel applications will be commonplace on general purpose machines, care must be taken to make sure that they cooperate. We propose that we can improve the performance and scalability of multi-core libraries by dynamically preventing oversubscription of the CPU at runtime. Such an approach would greatly improve the scalability of multi-core libraries and make them much more attractive for common workloads.

1. INTRODUCTION

1.1 Task Model
A major limiting factor of the development of mainstream multi-core applications has been the lack of simple means to develop such applications and the difficult nature of safe thread programming. Developing multi-core programs from scratch requires an intimate knowledge of the specific processor and of the thread and lock implementations of the operating system. Porting complete applications to other platforms requires drastic refactoring, as well as the knowledge of the platform’s thread and lock implementations.

Until recently this has not been a large issue due to the lack of hardware to support such applications. However, now that clock speeds are approaching their limits and energy use is an increasing concern, multi-core processors are becoming more commonplace [2]. As multi-core processors became available so did multi-core libraries, which provide a convenient level of abstraction, relieving the programmer of the minute details of thread programming. These libraries are not only typically platform independent, allowing instant portability, but also provide additional resources to the programmer, such as optimized locks and lock-free data structures.

Popular among these libraries are Intel Threading Building Blocks (TBB) [7] and Java java.util.concurrent (JSR166) [6], which use a task based model for achieving parallelism. In this model, large workloads are divided into smaller units called tasks and distributed to worker threads which execute them in parallel.

Figure 1: Task based model. In the task based model large workloads are divided into smaller units called tasks and distributed to worker threads which execute them in parallel.
**Figure 2: Lock convoying problem.** Convoying is demonstrated on a two hardware thread machine with software threads sharing a single lock under which all work is done. On the left, threads T1 and T2 are scheduled to run all of the time since there are as many software threads as hardware threads. The lock is passed back and forth and work is being done by some thread at all times. On the right, threads T1, T2, and T3 must share the CPU. At first threads T1 and T2 are scheduled on the CPU, and T3 is not, indicated by the dotted line. At time 4, the CPU is taken away from T1 and given to T3. Since T1 is still holding the lock when the CPU is taken from it, T2 and T3 must wait until T1 is scheduled again for the lock to be released, and they wait, wasting their CPU time. At time 7 T1 is scheduled again and T2 is removed from the CPU. T1 completes its work and T3 attains the lock.

Multiple instances of the library can easily overload the system. For example, if each instance of a multi-core library were to launch 8 threads, and there were 10 instances running at a given time, there would be 80 threads present on the system from the library, plus more threads from other processes. On such a system, when a thread is preempted (exhausts its processor usage time) it would have to wait much longer to be scheduled again than on a system which was not overloaded, due to the amount of threads it is sharing the processor with. Additionally, if a preempted thread is holding a lock, then a significant amount of time is wasted by other threads waiting on that lock, since they need to wait on the thread holding the lock to be rescheduled and the lock to be released.

Our goal has been to investigate the performance of a system under different burdens and to find a way to limit over-subscription (more threads than processors) of the processor in order to prevent performance degradation under large work loads. We have modified TBB to support dynamic runtime adjustment of the number of threads running and examined the impacts.

**Figure 3: Lock convoying performance effect.** Performance of the optimized TBB `queuing_lock` is shown as threads are added to an eight core system. As the number of software threads passes the number of hardware threads the lock begins to suffer from convoying. Section 6 further details this.

2. **BACKGROUND & RELATED WORK**

Most work pertaining to efficiency in multi-core systems has been focused on contention management- minimizing critical sections and efficient lock implementations. Amdahl’s law states that speedup is limited by the time of sequential execution, but as load increases there is more contention for critical sections, reducing speedup further. It is a common paradigm to keep critical sections as short as possible or as rare as possible. However, under enough load, contention still reduces performance due to increased competition for critical sections and longer penalties when a thread within is preempted.

2.1 **Efficient Locking**

A way to avoid the performance cost from the buildup of threads contending to enter a critical section is proposed by Scott *et al.* [8]. They propose a solution where threads waiting to enter a critical section wait on a spin lock, repeatedly polling a value, while they are queued to enter a critical section. While waiting the thread may time out and yield the CPU so another thread may resume execution. The advantage of spinning is that the desired lock may be released soon enough that it is worth the wait instead of incurring the overhead of yielding the processor or blocking until the lock is released. Similar to our work, this is an attempt to negate the effect of multiple threads competing for a single resource by having threads give up CPU time. By yielding the CPU the spinning thread avoids wasting CPU time and allows other thread to do useful work. This approach is highly effective and is utilized in TBB in its `queuing_lock` and `spin_lock`.

Another means of improving parallel performance is to avoid locks in general. Libraries such as TBB deliver mostly lock-free data structures, which are often not available in standard programming languages. As suggested, such data
structures use minimum to no locking and are accessible by multiple threads while still maintaining correctness. Lock-free data structures are extremely difficult to program as well as verify for correctness, so their inclusion in such libraries is very important. These data structures allow the programmer to cleanly implement familiar data structures without having to wrap access to more common non-thread safe implementations with costly locks, which introduce the problems we are attempting to minimize. While lock-free data structures are helpful, they do not address the larger problem of too many threads, and under enough stress we also see performance drop off.

2.2 Lock Convoying

While improved lock performance is important, performance still drops when the number of software threads passes the number of hardware threads, as seen in Figure 3. When a thread holding a lock exhausts its CPU run time it is removed from the CPU, so the lock it is holding will not be released until it is scheduled again and able to complete its work. If another thread is waiting on that lock, it must wait for the holding thread to be scheduled and the lock to be released. When the number of software threads is less than or equal to the number of hardware threads, the holding thread will be rescheduled immediately. However, if there are more software threads than hardware threads then the holding thread will have to wait to be scheduled again, and the waiting threads will suffer.

This phenomenon is called convoying [3], and is illustrated in Figure 2. Each thread waiting on a lock must wait at least as long as the thread holding the lock before it. Threads holding a lock may not be scheduled on the CPU, while other threads waiting on that lock are scheduled and waste their run time waiting. As more threads are added the effects worsen. Convoying is what we aim to eliminate by limiting the number of active threads.

2.3 Contention Management

Johnson et al. [5] compare the performance of several lock implementations and also examine the effectiveness of reducing contention to improve performance. Similar to Scott et al., they developed a proof of concept in which threads spin or block at a lock. In addition to this, they used current system utilization to decide whether or not to sleep a random thread, preventing it from being scheduled on the CPU at all. They found that useful processor utilization was increased and wait time was decreased using this system. They conclude that an effective solution to improve performance in multi-core systems involves a combination of efficient locking and load control. It is their findings that we have built upon and applied to TBB.

The improvement of locks can only do so much, and, as Johnson et al. implied, additional benefits are to be found in the reduction of CPU load. Scott et al. take a step toward relieving workloads by investigating locks that will fairly resolve contention, and in the event of long waits will allow other strands of execution to continue. Johnson et al. improve upon this not only by investigating the performance of more advanced locks, but also by integrating real time information about system utilization to make intelligent blocking decisions. They also use this information to decide when it is appropriate to sleep a thread in order to prevent its scheduling and reduce contention, which is most similar to what we have done.

3. TBB AND TASK BASED PARALLELISM

We have chosen TBB as the base of our experiments due to our familiarity with the library and its convenient locking primitives and data types. TBB is also open source, so we can potentially contribute our findings to the community.

3.1 Task Model

TBB employs a task based model in which large work loads are divided into tasks, either by the programmer directly or helpers provided by the library, and executed in parallel by worker threads. Since the number of tasks is not necessarily dependent on the number of threads, this approach scales extremely well. For example, if there is only a single thread then all tasks will be executed in serial by that thread, one at a time, until completion. If there are two threads than tasks will be split evenly between the threads and each thread will execute its individual tasks in serial to completion. This model continues to scale to an arbitrary number of threads; tasks are divided among workers, which in turn execute their assigned tasks to completion.

There is the case that some threads may finish their work more quickly than others and become idle, while other threads still have a work to finish. Such load imbalance is prevented by the concept of work-stealing, where a thread that has completed all of its tasks may attempt to steal tasks from another thread. TBB implements a work-stealing scheduler similar to Cilk [4], where work is initially evenly distributed evenly across all threads, and as threads complete their work they attempt to steal work from other threads. Our modifications rely heavily on this aspect of the scheduler, as work belonging to threads that we suspend is automatically picked up and completed by other non-suspended threads.

In TBB tasks are represented by the class Task, and their contents are executed by calling the member function execute(), which may return a Task to be executed next, bypassing the scheduler, or null if there is no continuation. This optimization quickly delivers work to threads without having to go through the scheduler, and is a special case we need to consider in our modifications.

3.2 Task Scheduler

The TBB scheduler consists of three levels of nested loops which, from outermost to innermost, steal tasks from other threads, retrieve tasks tasks belonging to the local thread,

```c++
Task *next_task;
for(;;){
    do{
        while(next_task){
            next_task = new_task->execute()
        }
        next_task = get_task()
    }while(t);
    next_task = steal_task()
}
```

Figure 4: TBB Scheduler. The innermost loop executes any continuation specified by the running task, and the outer two retrieve the next task to be run.
and execute tasks and their continuations, respectively. On creation worker threads enter the scheduler and remain there for the duration of program execution. When there is no work left, including work to steal, they sleep until they are notified of more. The main thread enters the scheduler on parallel calls, and remains within the scheduler until all work is complete, when it exits and resumes normal execution.

Nested scheduler instances are allowed, and happen when a parallel call is made from within another parallel call. In this case, the calling thread remains in the innermost instance of the scheduler until the call completes. In certain circumstances a nested instance may exit early to prevent unbounded stack growth from nested calls. Similar to non-nested calls, work is divided into tasks and distributed among the threads. The condition of nested calls requires special consideration in our modifications.

4. APPROACH

We used TBB as a starting point because it is open source and we are most familiar with it. TBB gives us a solid base to work with and provides many features that will be beneficial to our project, most notably work-stealing.

At the highest level our model consists of the scheduler and a monitor thread. Each thread has a separate scheduler instance, so individual schedulers must communicate to decide when one of them should stop doing work and sleep. The monitor thread checks system load at fixed intervals, communicating to the schedulers how many should stop doing work and stop running, and how many should continue execution as normal.

Checkpoints are put within TBB’s scheduler where threads decide whether or not they may proceed or must halt. The first checkpoint is placed right before tasks are executed. Here it is determined whether or not a thread will be allowed to continue to run as normal or if it should give up its work and stop running. The second checkpoint is placed before a new task is retrieved from the local task pool. This checkpoint is only entered if it was decided that the running thread should halt, and contains the code to block the thread. The separate check points ensure that:

- No tasks are executed by threads that have not checked in yet
- Threads exiting a task continuation are blocked the same way as all other threads
- Blocked threads that must exit a nested scheduler instance may do so.

The two check points are separated by very little code and can generally be considered one. Once a thread has passed a checkpoint it retains its status as blocked or allowed to continue and may bypass the checkpoints until the thread count is changed.

Schedulers instances on different threads communicate using shared memory. Most of this memory is accessed during the checkpoints so correctness is imperative. This portion of our modifications has been subject to much testing and verification.

The monitor thread’s job is to use real time data about system load to determine how many threads should be allowed to run at a time. When the monitor detects over-utilization it will throttle down the number of threads, and when it detects under-utilization it will increase the number of threads. When a change in thread count is needed the monitor informs the scheduler instances which adjust accordingly.

5. SYSTEM IMPLEMENTATION

Our work is based on a patched version of TBB 2.2 supporting the SPARCv9 architecture [1]. Our changes were made to the standard version of the TBB scheduler under Unix using all default parameters for compilation. While other options may work they are not tested.

5.1 Scheduler

To allow the dynamic adjustment of threads at runtime we first needed to modify the scheduler to support selectively blocking threads. We built this directly into the scheduler for both speed and uninhibited access to functionality involved with scheduling.

The TBB scheduler loop is located in the function CustomScheduler::local_wait_for_all. This function does some initial set up, and then dispatches tasks in the scheduler loop. As we are modifying this function directly, we have access to all of its local variables, as well as all of the functions of the class CustomScheduler. This is particularly helpful in determining the difference between the two different types of threads that run the scheduler. These are worker threads, which blindly consume work, and master threads, which dispatch parallel work and exit the scheduler on completion.

It is important to treat these two types of threads differently. Worker threads may be blocked indefinitely because they simply pick up work as it becomes available. Blocking a master thread indefinitely would result in deadlock because the master thread would never be able to exit the parallel call it began and resume normal execution. To further
Figure 6: Worker and master threads. An illustration of worker threads and master threads. Thread 0 is the main thread which initializes TBB launching threads 1-3. Thread 0 then makes a parallel call, entering the scheduler, as can be seen by its stack. As part of the work given to it, Thread 3 enters a parallel call, and enters a nested scheduler instance. Thread 3 is now a master thread as well, but threads 1 and 3 are still workers because they are in the outermost scheduler.

To keep track of threads we add a counter of how many threads are running in the local instance. We check this against a variable which stores how many threads are allowed to run in the local instance, which is changed to adjust the thread count. The active thread counter is initialized to 0, and the number of allowed threads starts at the number of threads initially spawned by the library. The first scheduler checkpoint we check the number of threads running against the number of threads allowed to run, and use that to decide the action for the current thread. If the active thread count is below the number of allowed threads the counter is incremented and the thread may continue, otherwise the thread is blocked.

Since multiple threads are accessing the thread count it must be protected to maintain correctness. Originally we updated the counter using atomic instructions, but it proved difficult to maintain correctness this way. Instead, we decided to protect the counter using TBB’s spin_mutex, which is highly efficient and well suited for this application.

To avoid excessive contention for the counter each thread competes to run once. If the thread is allowed to run it will continue to run until it exits or the thread count is changed. If a thread is told to stop it will remain inactive until another thread exits or the thread count is changed. To enforce this global flags are maintained for each thread. These flags are used to decide to bypass checkpoints, or to block at a checkpoint. Whenever the flags are reset the threads will compete for the right to run again.

Flags may be set at any time to cause threads to either compete to run again or to stop. An important use of flags is to cut off task continuations. Because task continuations bypass the rest of the scheduler, flags must be used to stop them when the thread count change requires that particular thread to go to the checkpoint again. When a task continuation is flagged to stop, the new task is put back in the local task pool, to be run later by the owning thread or to be stolen by another thread.

5.2 Monitor

To gauge system load, an extra thread is launched when TBB is initialized. The responsibility of this thread is to sleep and wake up at fixed intervals to calculate system load and adjust the thread count accordingly.

System load is calculated from the information in /proc/loadavg. The monitor reads this file and parses the number of runnable processes, or threads, in the operating system. This number does not include threads that are blocking in a system call, such as pthread_cond_wait or usleep. The result is that threads blocked by our scheduler, regardless of TBB instance, are not counted, as desired. The same goes for worker threads that are sleeping waiting for work.

The number of runnable processes on the system can be used to calculate if the processor is over-utilized or not. If the ratio of runnable processes to hardware threads is greater than one, then there are more runnable software threads than hardware threads, and there is competition for the processor. In this case the processor is over-utilized. If the ratio is less than one then there are idle threads and the processor is under-utilized.

The monitor thread calculates this ratio, subtracting one from the number of runnable processes to omit itself, and uses it to decide how to change the thread count. Because a ratio of 1.0 at all times is unrealistic, the monitor thread has a built in tolerance. When load is within this tolerance the thread count is not changed. When the system load is above the threshold the monitor thread will lower the thread count, and when it is below the threshold it will raise the thread count.

The updated thread count is calculated using multiplicative decrease/additive decrease. When the monitor determines that the system is over-utilized it will halve the number of threads in the instance, and when under-utilized it will add one. Using this system TBB is able to quickly back off when the system is over-utilized, and then reach an optimal value.

Once the new thread count is determined the monitor interacts with the scheduler through shared memory. By resetting flags and awaking threads stuck in pthread_cond_wait it can add or remove threads. If the new number of threads is greater than the old number of threads then flags on just enough threads to make the difference are reset and blocked threads are awakened. If the new number of threads is less than the old number of threads then the flags are set to tell running threads to stop.

6. SYSTEM PERFORMANCE

In this section we explain our methodology and present the performance results and reactivity of our model.
6.1 Methodology

Due to the limited number of benchmarks compatible with TBB we created our own micro-benchmark to demonstrate the performance problem we are trying to solve. We seek to measure the performance of multi-core applications that do most of their work under a limited number of shared locks, a common theme in multi-core programming. We were not able to find a benchmark of this sort which supported TBB.

For our benchmark we developed a histogram with a lock for each bucket. The histogram is updated by supplying an integer whose low order bits are used to choose the bucket. The bucket is first locked, and then the integer is transformed by repeatedly dividing it or multiplying it based on whether or not it is even or odd. Once the transformation is complete the result is added to the bucket and the lock is released. The number of buckets and times to multiply/divide the input value is provided when the structure is initialized. While the inserts are just busy work, we are concerned with how quickly they can happen in parallel, and this is how we judge performance.

The numbers to be inserted are randomly generated at the beginning of our benchmark, for our tests we generated $2^{24}$. Once the numbers are generated they are divided into chunks of about 10,000 and inserted in parallel by TBB. The number of buckets was chosen such that there were significantly more buckets than threads, and so that all locks and counters would fit within the system cache. The number we used for our experiments was $2^{14}$. The number of multiplication/division operations was set at 512.

Of the five lock types used in our benchmark, the following three are native to TBB:

- **mutex** - a wrapper to the native operating system mutex, in our case pthread_mutex
- **queuing_mutex** - a fair mutex, where processes arriving at a lock will be granted access in the order that they arrive. Failed attempts to attain the lock result in the thread yielding to the operating system
- **spin_mutex** - an unfair mutex, where processes arriving at a lock will be granted access as soon as possible. Failed attempts to attain the lock result in the thread yielding to the operating system

Both queuing_mutex and spin_mutex offer much better performance than mutex for short held locks. The main difference between queuing_mutex and spin_mutex is that while queuing_mutex prevents starvation by queuing access to the lock, spin_mutex does not.

The additional two locks we used were modified versions of the TBB spin_mutex and queuing_mutex, such that they would not yield after failure. They are included to show how they would perform without the optimizations added by TBB. We only show the performance of these locks in Figure 7 due to their low performance.

6.2 Performance

The results of these benchmarks prior to any modification can be seen in Figure 7a for an Intel Quad Core2 machine with 8 hardware threads and Figure 8a for a dual-socket Sun Niagara II machine with 128 hardware threads\(^1\). It is

---

\(^1\)These graphs depict performance statistics under the assumption that 128 threads is indeed optimal. While the given machine has 128 hardware threads, every four threads shares an ALU, and every eight threads shares an FPU, so
clear from the graphs that mutex outperforms the other locks under over-subscription. This is because threads waiting on a mutex are managed by the operating system and are not scheduled to run until the given mutex is released. Since waiting threads are not scheduled, any thread waiting on a locked mutex will not run on the CPU and waste its share of run time. Also, because threads waiting on a mutex are not scheduled for CPU time, threads not waiting on a locked mutex will be competing with less processes for CPU time, and will have a shorter wait time.

Performance of the remaining locks degrades as the number of software threads increases beyond the number of hardware threads, as seen in Figures 7a and 8a. queuing_mutex and spin_mutex are implemented in user space, so they perform better than mutex up to a point because they do not suffer the overhead of a system call. Beyond that point they do poorly because a thread waiting on one of these locks remains in the operating system run queue, and when it is scheduled it will waste CPU time if it does not attain the lock. This wasted time extends the wait times for other threads in the run queue. As more threads are introduced to the system performance degrades further. This is an example of convoying, as described in Section 2.2, and its effect is even more apparent for queuing_mutex. If queuing_mutex is released, but the next thread waiting for it has been preempted, every other thread waiting on that lock must also wait for the next thread to be scheduled again.

After adding thread limiting to TBB we see that peak performance of user space locks is maintained for higher thread counts, and they perform better than mutex for most thread counts. This is visualized in Figure 7b for an Intel Quad Core2 machine with 8 hardware threads and Figure 8b for a dual-socket Sun Niagara II machine with 128 hardware threads. The modified, low performance spin_mutex gave a 473% increase in throughput for only 16 threads on the Intel machine. The higher performance TBB spin_mutex showed a 28.2% increase in throughput before performance started to decrease at 128 threads. On the Sun machine it showed a 333% increase in throughput at 1024 threads before performance dropped.

The increase in performance can be attributed to the number of software threads running being limited to the number of hardware threads regardless of thread count. The modified TBB scheduler successfully kept the number of active threads in check and prevented excess threads from running, preventing convoying. The high performance TBB locks did show good performance without thread limiting, but not as good as they did in conjunction with thread limiting. The drop off in performance at high thread counts can be attributed to the overloading of the TBB scheduler. On the Intel machine this was at 16 times more software threads than hardware threads, and on the Sun machine this was at 8 times more software threads than hardware threads. While the modified TBB scheduler is effective, it too suffers under high enough load.

6.3 Responsiveness

The ability of our system to react quickly to changes in system load is imperative for its effectiveness. Figure 9 demonstrates the reaction time of the modified TBB scheduler to changes in system load when dynamic thread adjustment is enabled.

A new instance of TBB is introduced to the system adding 8 software threads. The original instance immediately halves its number of threads, leaving 4 remaining. Eventually the extra instance exits and the original instance adds one thread at a time until the system is fully utilized again.
Figure 9: System responsiveness. Thread count in a single instance of TBB reacts to a change in system load on a machine with 8 hardware threads. Another instance of TBB is introduced to the system adding 8 software threads and the original instance adapts. When the new instance exits, the original returns to full thread count.

We may also extract information about the other instance of TBB from this graph. Since the thread count remains stable at 4 threads for some time, it can be inferred that the other instance of TBB also halved the number of threads it was running. By using system load information the instances were able to effectively communicate and fairly adjust their thread counts.

7. FUTURE WORK

A possible future expansion of this project would be to deepen modify TBB to support thread limiting. While we were successful in what we completed, a lot of things could have potentially been more straightforward and efficient if they were deeper integrated into TBB. The flow of the scheduler did not lend itself well to our modifications and we needed to work around several edge cases. Better integration would include a refactored scheduler and a blocked-thread aware work stealing algorithm that would steal work from blocked threads over running ones.

Another expansion would be more experimentation with how the monitor decides to limit threads. Direct communication between the monitors of different instances and experimentation with different decision heuristics are both potential topics.

Adding operating support for dictating how many threads a process would optimally use would be another topic. Multi-core applications could query the operating system for how many threads they should be allowed to run, instead of checking system load, and then decide to act on that information or not.

8. CONCLUSION

We sought to modify the TBB scheduler to support limiting of threads in real time to eliminate the effects of convoying. As our data demonstrates we were successful in improving the performance of multi-core applications with locking. We added support to our thread limiting scheduler to dynamically change thread count at run time using system load information. This proved to be successful and fast. We conclude that support for thread limiting to avoid convoying multi-core systems is effective.

9. REFERENCES