Scheduling OpenMP for Qthreads with MAESTRO

TR-11-02

Allan Porterfield
akp@renci.org
Renaissance Computing Institute
Rob Fowler
rjf@renci.org
Renaissance Computing Institute
Paul Horst
phorst@renci.org
Renaissance Computing Institute
David O'Brien
dobien@renci.org
Renaissance Computing Institute
Stephen Olivier
olivier@cs.unc.edu
Univeristy of North Carolina
Kyle Wheeler
kbwheel@sandia.gov
Sandia National Laboratory
Brad Viviano
viviano@renci.org
Renaissance Computing Institute

September 29, 2011

Scheduling OpenMP for Qthreads with MAESTRO

Allan Porterfield
akp@renci.org
Renaissance Computing Institute

Rob Fowler
rjf@renci.org
Renaissance Computing Institute

Paul Horst
phorst@renci.org
Renaissance Computing Institute

David O'Brien
dobien@renci.org
Renaissance Computing Institute

Stephen Olivier
olivier@cs.unc.edu
Univeristy of North Carolina

Kyle Wheeler
kbwheel@sandia.gov
Sandia National Laboratory

Brad Viviano
viviano@renci.org
Renaissance Computing Institute

September 29, 2011

Abstract

Obtaining good performance from modern Multi- and Many-core processors requires understanding the dynamic performance of the resources shared by multiple cores. Performance of single core systems only requires understanding the way that threads interact with the core on which they are executing. Multi- and Many cores systems have complicated this by adding various shared resources (e.g. L3 cache, I/O, network access, etc.) which are shared by multiple cores. The usage of these resources may be influenced by other threads within an application or by other concurrent programs. Efficient scheduling will require a dynamic scheduler. Fortunately, the increase of cores (and increasing frequency of non-ALU bottlenecks) provides the scheduler an opportunity to acquire the resources necessary to do dynamic monitoring and modeling.

MAESTRO includes a scheduler for the Qthreads runtime to explore any potential benefits from dynamic performance monitoring and modeling on application performance. The idea is to use computational resources that would otherwise be idle (because of memory bottlenecks) to measure and model system performance. MAESTRO implements an experimental scheduler on top of the Qthreads runtime. The scheduler communicates with a dynamic performance model to understand the dynamic state of the system. Scheduling decisions use that knowledge to better determine which threads should be executing and where. Qthreads already has a concept of locality (shepherds), to reason about shared resources. Building on the shepherd concept, MAESTRO supports hierarchical work stealing both intra- and inter- shepherd. Improving dynamic cache hit rates while reducing the number of expensive remote steals operations.

MAESTRO also extended Qthreads with the XOMP interface. XOMP is generated by the ROSE source-to-source translator to handle OpenMP (version 3.0) input files. The ROSE/Qthreads extension allow most C and C++ OpenMP applications to use the Qthreads runtime.

1 Many-Core Systems

High Performance Computing (HPC) is currently undergoing several significant changes. In 2010, AMD introduced nodes with coherent address spaces and 48 cores. In 2011, Intel will introduce nodes with 40 cores that can be HyperTheaded for 80 independent threads. Nodes with this degree of parallelism requires the programmer to consider parallelism within each node as well as between nodes. Multi-level parallelism will be the required programming model to achieve performance.

This is a result of several trends in microprocessor design, that have become increasingly clear in the last couple of years. Core or thread counts are now increasing as transistor count increases. The speed of those cores or threads is holding steady between 2 and 3 GHz. The memory bandwidth is increasing but not at a slower rate, the memory bandwidth per thread is falling as a result. The number of different hardware operations that can be measured by the user increases with each generation, and now includes a number of counters to measure shared resources such as cache activities and mechanisms to characterize the utilization of the various off-chip interfaces (QPI or HyperTransport).

The massive increase in computational power and limited memory bandwidth means that for the first time Runtime and OS overheads may be hidden from application performance by pushing the overheads to cores that would otherwise be effectively idle because of resource contention. If all OS interruptions are moved to a single core, not part of the application, allowing better lock-step execution of loops, we may reduce OS jitter and improve performance between barriers. The Runtime can use computational power to model dynamic performance without impact on the application as long as the model uses no shared resources.

As the systems become more powerful and computational techniques improve, new problems are becoming computationally viable. An important subset of these new problems are irregular and/or use adaptive solution techniques. When combined with the increasing heterogeneity of the hardware, mapping applications to hardware is increasing difficult and dynamic. Competition between the hardware execution threads sharing hardware resources (e.g., L3 cache, I/O, network access, etc.) will result in dynamic contention or bottlenecks.

Historically, overheads in a runtime have limited performance and scalability. Much work has been spent on good implementations reducing the amount of time and space used by the runtime and thereby improving application performance. As the hardware thread count increases and the applications become more dynamic over time, achieving good performance from the current runtime organization will become difficult. Using the a minor fraction of the computational power to model the state of the system will allow better scheduling, improving

overall performance.

The Qthreads runtime and the MAESTRO scheduler are designed to support and explore issues with modern applications on a variety of modern node architectures. Qthreads simplifies programming large problems that access memory in irregular patterns. It provides the programmer with a simple lightweight threading model and straightforward synchronization to facilitate communication between threads. MAESTRO explores techniques to improve dynamic performance though hierarchical work stealing and work throttling.

Qthreads/MAESTRO serves a general purpose high performance runtime for high performance system nodes, each of which encompasses a single address space and many hardware execution units. To prove it's usefulness, Qthreads/MAESTRO needs to support a variety of parallel programming models. Besides the initial Qthreads model for large graph problems, Qthreads developers are working with the Chapel[**?**] language team to support Chapel's distributed work model. OpenMP is a commonly used parallel programming model for shared memory systems. As a ready example/source of test programs, the MAESTRO project also extended Qthreads to export the XOMP interface. The ROSE source-to-source compiler accepts parallel programs written in OpenMP 3.0 and uses the XOMP interface during execution.

OpenMP and Chapel both support multiple programming styles. Traditional OpenMP (through version 2.5) focuses on loop based parallelism. MAESTRO recognizes those loops and runs them in parallel across all available cores. The default scheduling policy can be modified to allow work stealing and work throttling to occur (guided self-scheduling rather than static). In OpenMP 3.0 and 3.1[**?**], task based parallelism has been added and most of those calls are support by ROSE/Qthreads/MAESTRO. Work stealing in a task parallelism environment smooths any potential load imbalances. Work throttling reduces the pressure on shared resources when they are oversubscribed.

One key reason for implementing a new runtime like Qthreads was to allow research into runtime mechanisms and policies. Since the Qthreads source is publicly available and relatively simple, the MAESTRO scheduler can be integrated into a fully functional runtime with little effort. This allows MAESTRO to explore the premise that for many-core processors, most problems are memory bound and that the last core will contribute little to nothing to application performance. MAESTRO examines whether that last worker can improve scheduling without otherwise impacting application performance.

MAESTRO uses the hardware performance counters and the computational power from that last core to track the contention levels of any resource. Resources that are shared between multiple threads are of particular interest. Contention on shared resources means that performance is no longer an effect of only what a particular thread is doing, but also what everyone else sharing that resource is doing. For instance, a thread with a working set of 3 MBytes sustains no conflict cache misses when run on a processor with a 8 MByte L3 cache. A second copy would also run fine, but if four or more tried to share the cache, conflict misses could be a significant performance factor. Similar resource contention issues may exist for DIMMs, networks, or IO accesses.

2 Qthreads

Qthreads [10] is a cross-platform general-purpose parallel runtime designed to support lightweight threading and synchronization within a flexible integrated locality framework. Qthreads directly supports programming with lightweight threads and a variety of synchronization methods, including both non-blocking atomic operations and potentially blocking full/empty bit (FEB) operations. The Qthreads lightweight threading concept is intended to match future hardware threading environments more closely than existing concepts in three crucial aspects: anonymity, introspectable limited resources, and inherent localization. Unlike heavyweight threads, these threads do not support expensive features like per-thread identifiers, per-thread signal vectors, or preemptive multitasking.

The default scheduler in the Qthreads runtime uses a cooperative-multitasking approach. When threads block, such as when performing an FEB operation, a context switch is triggered. Because this context switch is done in user space, via function calls, it is less expensive than an operating system or interrupt-based context switch. This technique allows threads to proceed uninterrupted until data is needed that is not yet available, and allows the scheduler to attempt to hide communication latency by switching tasks. Logically, this only hides communication latencies that take longer than a context switch. The Qthreads runtime uses a hierarchical threading architecture with pthread-based worker threads to allow multiple threads to run in parallel. Lightweight threads are created in user-space with a small context and small fixed-size stack, and are then executed by the worker pthreads. These worker threads are organized into locality domains, termed "shepherds", which are enforced with CPU pinning. As such, each lightweight thread is inherently localized.

The Qthreads API includes several threaded loop interfaces, built on top of the core threading components. There are three basic parallel loop behaviors provided by the API: one to create a separate thread for each iteration, one that divides the iterations space evenly among all shepherds, and one that uses a queue to distribute sub-ranges of the iteration space to enable self-scheduled loops.

As initially implemented, Qthreads used a custom API to express parallelism and synchronization. This required any program using it to undergo extensive porting or be rewritten. The initial thread scheduler was simple and fast, some of the simplicity and speed was obtained by placed the burden of load balancing on the application.

3 MAESTRO for OpenMP

MAESTRO improves Qthreads in two primary ways. It increases the number of programs that can use Qthreads as their runtime library by supporting OpenMP through the the ROSE source-to-source compiler and the XOMP interface. Programs no longer need to be written from scratch or even modified to run with Qthreads. The MAESTRO scheduler is designed to improve application performance by improved utilization of resources across the system. It not

only supports work stealing between the threads to improve load balance, but also dynamic addition (or subtraction) of threads from loops, to improve parallel loop performance. Initially, the dynamic thread count is used to support work throttling when shared resources are over-utilized. Eventually, this could also be used to support preemption and resilience. MAESTRO also improves application performance through better use of the system resources which are shared between multiple computational units.

MAESTRO required several additions to the Qthreads runtime, these include

- ability to compile and link "real" applications against Qthreads
- ability to read system wide hardware performance counters
- ability to dynamically move work to improve load balance
- ability to dynamically change the number of active hardware threads

Compiling and linking "real" applications is supported by use of the ROSE[**?**] source-to-source translator. C or C++ OpenMP (currently version 3.0) source files are compiled by ROSE into C++ files, with the OpenMP pragmas replaced by the XOMP API. The resulting files are then compiled with any C++ compiler (we use GNU's g++) and linked against the Qthreads/MAESTRO library. The Qthreads library supplies implementations for all of the XOMP calls and the required OMP functions. This entire process is completed by a single command line invocation of the ROSECompiler.

MAESTRO supports several versions of scheduler with different mechanisms to support load balance. Qthreads defines each locality with a shepherd. Good load balance requires keeping each shepherd busy. The MAESTRO hierarchical work stealing protocol shares work within a shepherd amongst all workers and only when the local work queue is empty does one worker attempt to acquire work (for workers in that shepherd) from other shepherds. This reduces contention for work queues from stealing and allows work to be co-scheduled for improved cache locality and reuse.

The MAESTRO scheduler reacts to resource contention in the system by changing the number of threads that are executing tasks. This requires that both the task and loop scheduling codes allows workers to both arrive late and leave early. MAESTRO allows this by internally separating the concept of a software task from the hardware executing it. Loops and barriers both require the proper number of software threads to arrive, irregardless of the number of hardware threads present. This also requires barriers to allow hardware threads to exit and search for unfinished tasks. Hardware threads are pinned to specific cores and do not move during execution. The various hardware sharing relationships between the hardware threads is determined during Qthreads initialization and is exposed to the user though the shepherds (don't share) and workers (do share within a single shepherd).

3.1 Compilation

The ROSE source-to-source translator compiles C and C++ (FORTRAN supported by ROSE not Qthreads/MAESTRO) OpenMP programs and produces C++ source code that uses XOMP[**?**] calls. Each OpenMP parallel pragma is translated into an out-of-line procedure call. The body of the parallel region is transformed into a procedure by ROSE and is replaced by a pair of function calls XOMP_parallel_start and XOMP_parallel_end. All data used by the out-of-line call is passed through a created class which is initialized before the call and used within the called function.

ROSE translates each OpenMP call into the correct sequence of XOMP calls. For Example, each OpenMP parallel loop is turned into 3 XOMP calls depending on the loop scheduling protocol requested (possibilities include guided (shown), static, dynamic, and runtime selection all in either ordered or unordered)

- XOMP_guided_init – for initialization
- XOMP_guided_start – set loop bounds; get initial iterations; return TRUE if iterations found
- XOMP_guided_next – get interactions; return FALSE if loop complete

MAESTRO supports all of these calls in the Qthreads environment and dynamically modifies scheduling decisions. ROSE optimizes scheduling decisions when static scheduling is used, so MAESTRO uses a modified version of ROSE which defaults to guided self scheduling rather than the static that most OpenMP implementations use and programs may expect[1].

Qthreads also supports Chapel[**?**], a PGAS language. We expect to extend the MAESTRO scheduling to include Chapel support in the future. Parallelism is likely to be expressed differently in Chapel and the two different languages should test whether Qthreads/MAESTRO can be use for general parallel programming.

3.2 Daemon

The most innovative feature of the MAESTRO scheduler is the resource centric reflection daemon, RCRDaemon. It watches the hardware performance counters and transmits information about dynamic shared resource contention back to the application. RCRDaemon can either be started as part of the Qthreads initialization and run inside the application's address space, or it can be an independent application that spans multiple job launches and communicates though a shared memory region. Both versions also (optionally) create log files that are used to support a performance tuning tool, RCRTool[8].

[1]We expect to add a pragma in the near term which will allow the scheduling change to be user controlled

All recent general purpose microprocessors have a number of hardware performance counters which are accessible at runtime by either the user or the OS. Currently most performance tuning tools use these counters to evaluate various thread specific metrics, like floating point ops, cycles per instruction, or L2 cache misses. With the introduction of multi-core microprocessors, counters were included that measure various resources shared between multiple cores. These include L3 cache, memory controllers and cross-chip interconnect (QuickPath or HyperTransport). RCRDaemon watches the counters on the shared resources over time and uses a simple model to determine if contention currently exists.

During RCRDaemon configuration, in standalone mode or inside a Qthreads application, it configures itself using a trigger file. The trigger file defines the performance bottlenecks that this instantiation of the daemon will detect and what counters will be used. It defines multiple meters, which can be based on either whole chip counters or on individual core counters. In addition to specifying which hardware counters need to be monitored, the trigger file also defines values above (or below) which the resource is considered 'busy' (or 'idle'). The daemon uses these resource contention limits to determine when communication with the MAESTRO scheduler is appropriate. Log entrees are also marked and may contain additional information during periods of contention.

At start up, RCRDaemon also uses `shm_open` to create a Linux shared parallel region that is used as a blackboard to communicate system load to any interested application[2]. Using the information in
`proc`, it automatically builds a system model/map using the available Node, Socket and CPU information. Based on this information, it compiles the system topology to set up the proper RCRTool blackboard data structures for the hardware. Currently, it obtains the following information: number of logical processors, number of cores per socket, number of sockets, and the mapping between cores and sockets. Other information could be added to the graph, including number and size of cache levels, cache line size and the number of nodes.

At each node of the tree, that node's monitoring level is recorded along with any performance meters being watched at that level. At the node level, events like Network traffic can be monitored at the NIC. At the socket level via 'uncore' hardware counters, shared resources can be monitored, such as, memory concurrency can be monitored by watching number of L3 misses, the average memory latency and the clock. For core nodes of the tree, meters like floating point operations per second can be generated by watching counters for floating point operations and the clock. The daemon updates these fields with the latest computed value at a rate dictated by the overheads of computing the meters (normally the limiting case) and the frequency that the underlying hardware counters are updated (typically at a $>$ KHz rate).

When RCRDaemon notices a meter cross one of the thresholds specified in the trigger file, it flags that meter and generates more detailed information. When embedded in MAESTRO, it calls the scheduler through an API to change the number of active hardware threads. This will be described in more detail the next section 3.3. It also supplies the backboard with

[2]If `shm_open` is not available, most functionality can optionally be obtained though `open` command using the *debugfs*.

information about the current location (if supplied) of any executing applications. Several XOMP interface functions within Qthreads currently supply this information. This allows rapid detection and identification of multi-thread performance issues.

3.3 Scheduling

One goal of MAESTRO is to dynamically adjust thread scheduling to account for changes in overall system load, whether internal or external. To successfully accomplish this MAESTRO needs many tools including

- support a common parallel programming model – OpenMP (see sec 3.1)
- understand dynamic system load (see sec 3.2)
- translate compiler output to useful parallelism
- allow hardware threads to arrive late/leave early from a parallel region
- move work to idle hardware resources. "work stealing"
- change the amount of hardware resources used throughout execution, "work throttling"

The first two, OpenMP support and RCRDaemon, have been discussed earlier in this document. Supporting the ROSE/OpenMP interface in such a manner that hardware can show up late and leave early from software threads are both mechanisms that are needed to support the last two, work stealing and work throttling.

XOMP interface For Qthreads, ROSE accepts C or C++ OpenMP 3.0[**?**] and generates a C++ source file with calls to various XOMP functions in place of the OpenMP pragmas (the OpenMP function calls are left alone). Some resulting XOMP calls are simple translations of the OpenMP pragma, like XOMP_barrier() replacing the #pragma omp barrier (other examples include XOMP_single() and XOMP_flush_all()). Several OpenMP pragmas generate a pair of XOMP calls, one to start the parallel action and one to terminate it (examples include XOMP_parallel_start(), XOMP_parallel_stop() – define a parallel region; XOMP_critical_start(), XOMP_critical_stop() – to delineate a critical section and XOMP_atomic_start(), XOMP_atomic_stop() – to delineate an operation that must be atomic). OpenMP for loops are turned into one of a large number of triplet functions, *_init(), *_start() and *_next depending on the type of scheduling requested and the whether the loops are to be ordered.

MAESTRO implements each of these calls to Qthreads, in addition to the various OpenMP function calls needed (like omp_get_num_threads), to allow full OpenMP applications to use Qthreads and the MAESTRO scheduler. This greatly expands the list of potential test programs that can execute on Qthreads allowing much quicker improvements in robustness to be made.

Variable Thread Count Changing the thread count during execution will be one of two scheduling tools which MAESTRO can use to improve application performance. Rather than always running at the maximum possible number of hardware threads all of the time, MAESTRO anticipates being able to detect and react to cases where fewer threads would execute faster. There is anecdotal evidence that for HPC applications some generations of Intel's HyperThreading is often not always a good idea. Another case occurs when each software thread has a cache footprint 1/3 of the shared cache size, and having only 3 rather than 4, 8 or 12 threads executing would improve overall performance by improving cache performance. Support for variable thread counts must be designed into all of the runtime parallel structures. Changing hardware state on commodity hardware can be an expensive operation and should be avoided.

If the number of hardware threads executing an parallel loop changes over time, the implementation barriers (and barrier like structures) must be handled very carefully for correct execution. Waiting for all of the hardware threads that have entered a loop to proceed from a barrier may not succeed. Instead of waiting on the all of the hardware threads to arrive, correctness and performance is best obtained by waiting on all of the software threads to arrive (and allow software threads move to idle hardware threads). Separating hardware execution threads from the notion of software logical threads, allows the runtime to correctly adjust hardware resources to software requirements.

MAESTRO separates the concept of a software thread from the hardware resource executing it. When a software thread enters a barrier, it cannot leave until all the other threads, but the hardware resource is free to search for another software thread and run it. When a parallel loop is reached, MAESTRO creates a software thread for each iteration (possibly bundled to reduce loop overhead, default guided self-scheduling uses bundling in its scheduling). The hardware is then free to execute the threads in any order. When MAESTRO decides to change the hardware thread counts, it has no impact on the software threads other than execution speed.

3.3.1 Work Stealing

Task parallel programs generate a dynamically unfolding sequence of interrelated tasks, often represented by a directed acyclic graph (DAG). A task executing on the same thread as its parent or sibling tasks may benefit from temporal locality if they operate on the same data. In particular, such locality properties are a feature of divide-and-conquer algorithms. To schedule tasks as lightweight threads in the MAESTRO scheduler, the run time supports general dynamic load balancing while exploiting available locality among tasks. To optimize performance on modern systems the scheduler must also attempt to maximize performance of shared resources such as caches.

The initial Qthreads scheduler uses round robin initial task placement and does not provide any automatic mechanism to move tasks after that placement. The degree of load balance is determined by that initial placement. Also since a parent and child often end up on different

queues, cache reuse between tasks is severely limited.

To better meet the dual goals of locality and load balance, MAESTRO implements work stealing. Blumofe et al. proved that work stealing is optimal for multithreaded scheduling of DAGs with minimal overhead costs [3], and they implemented it in their Cilk run time scheduler [2]. Our initial implementation of work stealing in Qthreads mimicked Cilk's scheduling discipline: Each shepherd scheduled tasks depth-first locally through LIFO queue operations. An idle shepherd obtained more work by stealing the oldest tasks from the task queue of a busy shepherd.

MAESTRO implemented two different probing schemes to find a victim shepherd, and observed equivalent performance: choosing randomly and commencing search with the shepherd ID one greater than the thief. Interruptions to busy shepherds are minimized because the burden of load balancing is placed on the idle shepherds. Locality is preserved because newer tasks, whose data is still hot in the processor's cache, are the first to be scheduled locally and the last in line to be stolen. The cost of work stealing operations on multi-socket multicore systems varies significantly based on the relative locations of the thief and victim, i.e. whether they are running on cores on the same chip or on different chips.

Stealing between cores on different chips reduces performance by incurring higher overhead costs, additional cold cache misses, remote memory access costs, and coherence misses due to false sharing. Another limitation of work stealing is that it does not make the best possible use of caches shared among cores. In contrast, Chen et. al. [4] showed that a depth-first schedule close to serial order makes better use of a shared cache than work stealing, assuming serial execution of an application makes good use of the cache. Blelloch et al. had shown that such a schedule can be achieved using a shared LIFO queue [1].

To overcome the limitations of both work stealing and shared queues, MAESTRO uses a hierarchical approach: multithreaded shepherds. Each shepherd manages all the cores on the same chip. These cores share a cache and all are proximal to a local memory attached to that socket. Within each shepherd, a worker is mapped to each core. Among workers in each shepherd, a shared LIFO queue provides depth-first scheduling close to serial order to exploit the shared cache. Load balancing happens naturally among the workers on a chip and concurrent tasks have possible overlapping localities that can be captured in the shared cache.

Between shepherds, work stealing is used to maintain load balance. Each time the shepherd's task queue becomes empty, only the first worker to find the queue empty steals enough tasks (if available) from another shepherd's queue to supply all the workers in its shepherd with work. The other workers in the shepherd spin until the stolen work appears. Aggregate task queuing for workers within each shepherd reduces the need for remote stealing. While a shared queue can be a performance bottleneck, the number of cores per chip is bounded, and intra-chip locking operations are fast within a chip.

3.3.2 Work Throttling

MAESTRO also uses work throttling to improve dynamic application performance. Although vaguely counter-intuitive, it has been know in the OS and compiler communities that running less can actually improve overall performance[**?**]. In the OS running fewer jobs simultaneously, in some cases, prevents swapping and improves the overall performance. For compilers, reducing a thread's working set to something that fits in cache is known to increase the cache hit rate, reducing the number of memory access and the overall average memory latency, thereby improving performance. MAESTRO focuses on these cache issues during runtime. Other resources, such as network/IO bandwidth, can produce the same thrashing behavior and may be modeled by MAESTRO in the future.

As an example of the type of runtime work throttling MAESTRO addresses, consider an application which iterates though several arrays totaling approximately 3 MBytes in size. Running that application on a modern system with an 8 MByte cache would generate only cold misses and run well. If the 4 or more copies of that application were running simultaneously, then the working set would be exceeded and the performance of each application would severely degrade. MAESTRO observes the cache hardware performance counters and limits the number of active threads to two, allowing each to effectively use the cache. When the first two finish, the next two start. When cache thrashing occurs, the total time from the limited runs will be less than from the unlimited runs.

Determining when shared resource contention exists requires a dynamic system model that watches the application execution and can adjust to any changes. The RCRDaemon is used to drive such a model in MAESTRO. RCRDaemon computes the number of concurrent memory references from available hardware performance counters for each microprocessor.[3] Our current simplistic model decides when the number of concurrent accesses is near the limit of what the memory system can sustain. This value is currently determined empirically with a micro-benchmark. A high number of memory references has is the result of a high cache miss rate. These cache misses are either compulsory(first use) misses, capacity(cache not big enough to hold the required data) misses or conflict (somebody else forced you data out of the cache) misses. Streaming applications will use data only once, high compulsory miss rate, and performance cannot be improved though better cache use. Applications where a single thread's cache footprint exceeds the cache size will experience a high number of capacity misses and there is little the runtime can do to dynamically improve performance. However when one thread's cache footprint does fit within the cache, a high number of conflict misses can still be seen as multiple threads compete for the cache. In this case, performance can be improved by limiting the number of active threads until the sum of the cache usage is smaller than the available cache.

This model should eventually take into account other factors (including memory latency

[3]Currently we use AMD's L3_MISSES:ALL to track the total number of misses and the cycle count to track the number of serial misses and divide the two to compute average concurrent misses. On Intel's, we just acquired a drive to read the "uncore" counters. We are currently upgrading the daemon thread to use the new driver and expect to have equivalent Intel numbers within a week or two.

and access history). The RCRDaemon currently recomputes a new memory concurrency value about every 5 milli-seconds. At this rate, MAESTRO can rapidly to dynamic changes in memory usage.

When executing an OpenMP parallel for loop, every time that a software thread attempts to acquire iterations to execute, a call is made to either a XOMP_*_start() or XOMP_*_next(), internally those calls limit the number of outstanding threads. When RCRDaemon increases the allowed active threads, the additional threads are immediately released. When the number of threads is reduced, the system waits until the previous iteration chunk is completed and more requested before removing the thread. The number of active threads is kept for each shepherd, allowing us to evenly reduce the thread count across the system. Eliminating the possibility that when a reduction of 4 global threads occurs that they all occur on one shepherd (causing load imbalances across the shepherds). When the first thread determines that the all loop iterations have been completed, it releases all delayed threads, allowing the loop to complete and work to continue.

To scheduling task-based parallelism, work is placed onto the queues described in sec. 3.1. Each worker thread executes a loop looking for work on the shepherd work queue and processing it. Although not currently implemented, the same delay mechanism will be used to limit the number of workers that are actively executing tasks when the memory concurrency for the system is near peak. Within the loop acquiring additional work, the memory concurrency model will be checked to see if the thread is allowed to continue. If delayed, a thread will be held until either the memory concurrency model decides more work is needed or until all of the tasks complete and the thread is needed to complete task synchronization.

3.4 RCRTool

The RCRDaemon clearly has the ability to communicate with more than the Qthreads runtime and MAESTRO. It's chief purpose is to facilitate a performance debugging tool to support Resource Centric Reflection, or RCRTool. The goal of RCRTool is to allow users to quickly locate portions of an application's execution where performance was limited because of a shared system resource. It not only identifies the programs running concurrently, but when the application uses MAESTRO, gives the current parallel region, task or loop.

When executing as a standalone daemon, a performance logger, RCRLog, can take the information produced by RCRdaemon and build a performance trace. This trace currently grows at a rate of several MBytes per minute (depending on a setable log frequency) and can be used for post execution performance tuning. The RCRTool GUI allows the log to be examined in detail, with each point annotated with the current source location and the current value of the metric of interest. In addition the detail provides information about the current percentage of peak, the number of calls to the current parallel region, task or loop, and the absolute counter values. For more information on RCRTool see [8].

EXAMPLE	GCC	MAESTRO
omp_dynamic	0.003	0.018
omp_mm	0.005	0.009
omp_orphan	0.004	0.015
omp_reduction	0.005	0.008
omp_workshare1	0.005	0.015
omp_workshare2	0.005	0.027
dijkstra_open_mp	0.004	0.009
md_open_mp	14.637	15.112
quad2d_open_mp	1.327	1.332
quad_open_mp	0.017	0.022
satisfy_open_mp	0.030	0.033
schedule_open_mp	3.411	4.353
sgefa_open_mp	1.234	1.337
ziggurat_open_mp	0.938	0.946
fib	22.304	1.987
prime_open_mp	18.189	9.567
heated_plate_open_mp	3.039	16.037
fft_open_mp	34.775	50.595

Table 1:

4 Preliminary Results

4.1 OpenMP

The ROSE/XOMP has been tested on a number of small OpenMP example programs. These test check for correctness and give us a number of simple timing tests, allowing comparisons between the untuned MAESTRO scheduler and the GNU OpenMP implementation. These tests were run on a quad-socket Dell M605 blade populated with six-core AMD (Istanbul) Opteron 8425 HE microprocessors. Each core runs at 2.1GHz and has a 512KB L2 cache. The six cores share a 48-way associative 6MB L3 cache.

They can be divided into four categories based on the times for the test programs. The first group of example programs have no significant work in them. There time to completion is dominated by OpenMP initialization. GNU (and Intel) have substantial benefits at start-up due to the number of structures that Qthreads and MAESTRO create. On average the start-up time for Qthreads/MAESTRO is between 2 and 5 times as long (.008 - .027 vs .004 or .005) as GNU. These programs tend to be small and test/show a particular feature of OpenMP. (For example omp_dynamic is a simple test to show that omp dynamic pragma is correctly recognized and only contains one `printf`.)

The second group of example programs contain a small amount of computation and GNU slightly outperforms MAESTRO. These are the test cases on which MAESTRO tuning efforts

will focus. md_open_mp implements a simple molecular dynamics simulation and runs 4% faster on GNU than MAESTRO. Most of its work is in a well balanced parallel for loop, with each iteration comprising 10,000+ instructions. One advantage MAESTRO has over GNU (and Intel), is better loop balance when threads arrive at the loop at different times. The overhead for this is higher than for static scheduling. If the parallel loop is repeated with each thread performing approximately the same amount of work between the loops, the increased overhead is larger than the loop balancing gains. Since the loop iterations are coarse, the additional overhead for MAESTRO guided self scheduling over the default static is small but visible in these balance loop's execution times.

In the next two programs, fib and prime_open_mp, MAESTRO runs much faster than GNU. Fib is a simple task parallel program with computing Fibonacci numbers no serial cutoff. The performance of task based parallelism will be examined in more detail in section 4.2, but MAESTRO greatly enhances load balance between cores. prime_open_mp computes the number of prime numbers within a region. The iterations do very different amounts of work depending on whether a divisor is found quickly. The default static scheduling results in poor load balance. MAESTRO's improved load balance results in a reduction in execution time of almost 50%. The GNU version can be tuned to improve performance (force scheduling other than static), but before application tuning it performance suffers.

The last two programs, heated_plate_open_mp and fft_open_mp, both run much faster on GNU than for MAESTRO. They both repeatedly call parallel loop with small amount of work in a single iteration. The compiler can compute a static mapping and embed it into the executable, so that starting an iteration only requires a handful of cycles. MAESTRO does not used a fixed iteration set and requires a short function call be executed to obtain the next iteration to execute. The overhead of this function call results in substantially reduced performance.

4.1.1 Discussion and Interpretation of Results

When an application repeatedly creates parallel loops with equal amounts of work for each iteration and little or no intervening work between iteration, GNU (and Intel) beat MAESTRO. The ability of a compiler to pre-compute a static schedule allows very low loop overheads and when it results in good load balance results in very good performance. Guided self scheduling produces better load balance, but at a cost of a dynamic function call and several dozen instructions per iteration. For those applications with great load balance anyway, this overhead is not recovered and performance suffers. When the parallel iteration is small this performance loss may be significant. For applications with either varying amounts of work per iteration or between parallel loop executions, the improved loop balancing can result in improved performance. Picking a static schedule, like GNU and Intel, can for some dynamic or adaptive programs can result in substantial performance reductions before any application tuning.

Even for a relatively small group of random programs, the relative benefits and cost of the two strategies can be easily seen. As program be come larger and and more adaptive,

Qthreads Implementations, compiled ROSE/GCC -O2 -g					
Version Name	Scheduler Implementation	Number of Shepherds	Task Placement	Internal Queue Access	External Queue Access
Q	Stock	one per core	round robin	FIFO	none
L	LIFO	one per core	round robin	LIFO (blocking)	none
CQ	Single Queue	one	N/A	LIFO (blocking)	N/A
WS	Work Stealing	one per core	local	LIFO (blocking)	FIFO stealing
MTS	MAESTRO	one per chip	local	LIFO (blocking)	FIFO stealing
ICC	Intel 11.1 OpenMP, compiled -O2 -xHost -ipo -g				
GCC	GCC 4.4.4 OpenMP, compiled -O2 -g				

Table 2: Scheduler implementations evaluated: five Qthreads implementations, ICC, and GCC.

Configuration	Alignment	Fib	Health	NQueens	Sort	SparseLU	Strassen
ICC 1 thread	28.33	100.4	15.07	49.35	20.14	117.3	169.3
GCC 1 thread	28.06	83.46	15.31	45.24	19.83	119.7	162.7
ICC 32 threads	0.9110	4.036	1.670	1.793	1.230	7.901	10.13
GCC 32 threads	0.9973	5.283	7.460	1.766	1.204	4.517	10.13
MAESTRO 32 workers	1.024	3.189	1.122	1.591	1.080	4.530	10.72

Table 3: Sequential and parallel performance using ICC, GCC, and the Qthreads MAESTRO scheduler (time in sec.). For *Alignment* and *SparseLU*, the best time between the two parallel variations (`single` and `for`) is shown.

the benefits of load balancing schedulers will continue to grow and the overhead liabilities will drop. For modern large scale programs, MAESTRO expects effective load balancing will be one of if not the critical performance metric.

4.2 Work Stealing

To evaluate the performance the MAESTRO hierarchical scheduler against other Qthreads schedulers and two common OpenMP implementations on OpenMP task-based parallelism, we used a subset of the Barcelona OpenMP Tasks Suite (BOTS), version 1.1, available online [5]. The suite comprises a set of task parallel applications from various domains with varying computational characteristics [6]. The subset included: *Alignment, Fib, Health, NQueens, Sort, SparseLU, Strassen.* A more detailed examination of the results can be found in [7].

Task size is often critical to runtime performance, for *Fib*, *Health*, and *NQueens* benchmarks the default manual cut-off to prune tasks is used. For *Sort*, cutoffs are set to transition at 32K integers from parallel to sequential calls. For *Strassen*, the cut-off giving the best performance for each implementation is used. For both the *Alignment* and *SparseLU* benchmarks two versions are supplied and used, one starts computation starts with a single initial task and

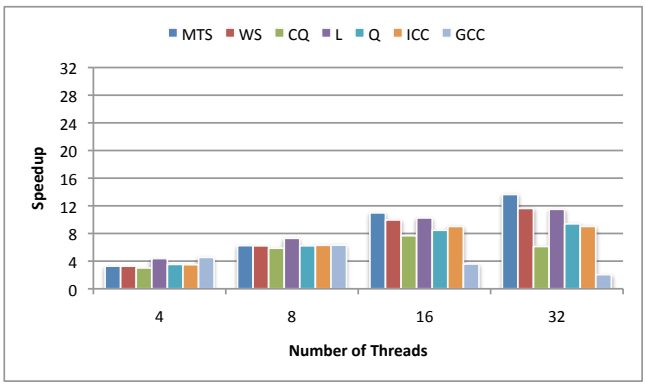

Figure 1: Health

another in which tasks are generated in a loop.[4] Other BOTS benchmarks are not presented here: *UTS* and *FFT* use of very fine-grained tasks without cutoffs, yielding poor performance on all run times, and *floorplan* raises compilation issues in ROSE.

The test system for our experiments is a Dell PowerEdge M910 quad-socket blade with four Intel x7550 2.0GHz 8-core Nehalem-EX processors installed for a total of 32 cores. Each processor has an 18MB shared L3 cache and each core has a private 256KB L2 cache as well as 32KB L1 data and instruction caches. The blade has 64 dual-rank 2GB DDR3 memory sticks (16 per processor chip) for a total of 132GB. It runs CentOS Linux with a 2.6.35 kernel. Although the x7550 processor supports HyperThreading (Intel's simultaneous multithreading technology), we pinned only one thread to each physical core for our experiments.

We ran the battery of tests on a variety of runtime systems, including five versions of Qthreads[5] and the widely available implementations from Intel and the Free Software Foundation (GNU), as described in Table 2. The original version of Qthreads, *Q* defines each core to be a separate locality domain or shepherd. It uses a lock-free FIFO queue to schedule tasks within each shepherd (individual core). Each shepherd only obtain tasks from its local queue, although for load balance tasks are distributed across shepherd in a round robin basis when generated. Work stealing required using a double ended queue, so the lock-free version was replaced with a simple double ended locking LIFO queue for the other versions. *L* incorporates this queue, replacing the original FIFO queue. *CQ* uses a single centralized shared queue to distribute tasks among all of the cores. For large tasks this should produce very balanced load, but as the task size shrinks the contention for the queue limits scalability. Each core is provided its own queue in *WS*, and idle shepherds steal tasks from the shepherds running on the other cores. Initial task placement is not round robin between queues, but onto the local queue of the shepherd where it is generated, exploiting locality among related tasks. The final MAESTRO scheduler, *MTS* (for multi-threaded shepherds), assigns one shepherd to

[4]The single task versions of both required the addition of a `taskwait` statement. The parallel loop versions required minor hand-editing of the ROSE intermediate output because of a compiler bug that has since been fixed.

[5]all compiled with GCC 4.4.4 -O2

every processor memory locality (shared L3 cache on chip and attached DIMMs). Each core on a chip hosts a worker thread that shares its shepherd's queue. Only one core is allowed to actively steal tasks on behalf of the queue at a time and tasks are stolen in chunks big enough (tunable) to keep all of the cores busy. All executables using the Qthreads and GCC run times were compiled with GCC 4.4.4 and -O2 -g, for consistency. Executables using the Intel run time were compiled with ICC 11.1 and -O2 -xHost -ipo. Reported results are from the best of ten runs.

4.2.1 Overall Performance

Overall the GCC compiler and ICC compiler produce executables with similar serial performance, as shown in Table 3. These serial execution times provide a basis for us to compare the relative speedup of the various benchmarks. Note that if the -ipo and -xHost flags are not used with ICC on *SparseLU*, the GCC serial executable runs 3x faster than ICC executable compiled with -O2 alone. Several other benchmarks also run slower with those ICC flags omitted, though not by such a large margin.

Qthreads *MTS* 32 core performance is faster or comparable to the performance of ICC and GCC. In absolute execution time, *MTS* runs faster than ICC for 5 of the 7 benchmarks by up to 74.4%. It is over 6.6x faster for one benchmark than GCC and up to 65.6% faster on 4 of the 6 others. On two benchmarks *MTS* runs slower: for *Alignment*, it is 12.4% slower than ICC and 2.7% slower than GCC and for *Strassen* it is 5.8% slower than both (although *WS* equaled GCC's performance [see discussion on Strassen in sec. 4.2.2]). Even as a research prototype, ROSE/Qthreads provides a competitive OpenMP task parallelism execution platform.

4.2.2 Individual Performance

Individual benchmark performance on multiple implementations of the OpenMP run time demonstrate features of particular applications where Qthreads generates better scheduling and where it needs further development. Examining where the run times differ in performance and speedup on up to 32 cores reveals the strengths and weaknesses of each scheduling approach. Only one benchmark is presented here, for performance of the other BOTS benchmarks see Olivier et al [7].

The *Health* benchmark, Figure 1, best shows how the various load balancing and cache locality mechanisms can work together to improve performance. GNU performance is slightly superlinear for 4 cores (4.5x), but peaks with only 8 cores active (6.3x) and by 32 cores the speedup is only 2x. Intel also has scaling issues and performance flattens to 9x at 16 cores. Stock Qthreads *Q* scales slightly better (9.4x), but just switching to the LIFO queue *L* to improve locality between tasks allows speedup on 32 cores to reach 11.5x. Since the individual tasks are relatively small, *CQ* experiences contention on its task queue that limits speedup to 7.7x on 16 cores, with performance degrading to 6.1x at 32 cores. When work stealing, *WS*, is added to Qthreads the performance improves slightly and speedup reaches 11.6x. *MTS*

further improves locality and load balance on each processor by sharing a queue across the cores on each chip, and speedup increases to 13.6x on 32 cores. This additional scalability allows the MAESTRO scheduler using *MTS* to achieve a 17.3% faster execution time on 32 cores than any other implementation, much faster than ICC (48.7%) and GCC(116.1%). *Health* provides an excellent example of how both work stealing and queue sharing within a system can independently and together improve performance.

Configuration (32 threads)	Alignment (`single`)	Alignment (`for`)	Fib	Health	NQueens	Sort	SparseLU (`single`)	SparseLU (`for`)	Strassen
ICC	4.4	2.0	3.7	2.0	3.2	4.0	1.1	3.9	1.8
GCC	0.11	0.34	2.8	0.35	0.77	1.8	0.49	N/A	1.4
MTS	0.28	1.5	3.3	1.3	0.78	1.9	0.15	0.16	1.9
WS	0.035	1.8	2.0	0.29	0.60	0.90	0.060	0.24	3.0

Table 4: Variability in performance using ICC, GCC, MTS, and WS schedulers (standard deviation as a percent of the fastest time).

4.2.3 Variability

One interesting feature of a work stealing run time is an idle thread's ability to search for work and the stabilizing effect this has on performance in regions of limited parallelism or load imbalance. Table 4 gives the standard deviation of 10 runs as a percent of the fastest time for each configuration tested with 32 threads. Both implementations with work stealing (*WS* and *MTS*) have very small deviations for 3 of the 9 programs. For 8 of the 9 benchmarks, both *WS* and *MTS* show less deviation than ICC.

In three cases (*Alignment*-`single`, *Health*, *SparseLU*-`single`), Qthreads *WS* deviation was much lower than *MTS*. Since *MTS* enables only one worker thread per shepherd at a time to steal a chunk of tasks, it is reasonable to expect this granularity to be reflected in execution time variations. Overall, we see less variation with *WS* than *MTS* in 6 of the 9 benchmarks. We speculate that normally having all the threads looking for work leads to finding the last work quickest and therefore less variation in total execution time. However, for some programs (*Alignment*-`for`, *SparseLU*-`for`, *Strassen*), stealing multiple tasks and moving them to an idle shepherd results in faster execution during periods of limited parallelism. *WS* also shows less deviation than GCC in 6 of the 8 programs for which we have data. There is no data for *SparseLU*-`for` on GCC because of completion problems.

Although in some cases, work stealing with a more queues shows better performance stability, MAESTRO uses the hierarchical scheduler, figuring that the performance impact from better cache performance will generally exceed any benefit in runtime stability.

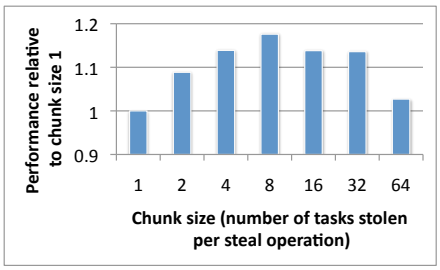

Figure 2: Performance on *Health* based on choice of the chunk size for stealing. Average of ten runs.

4.2.4 Performance Benefits of hierarchical scheduling in MAESTRO

Limiting the number of inter-chip load balancing operations is central to the design of the MAESTRO hierarchical scheduler (*MTS*). Consider the number of remote (off-chip) steal operations performed by it and by the flat work stealing scheduler *WS*, shown in Table 5. Recall that the MAESTRO work stealing operations are between chips (intrachip load balancing done though a shared queue), so these counts exclude the number of on-chip steals performed by *WS*. *WS* steals more than *MTS* in almost all cases, and some cases by an order of magnitude. *Health* and *Sort* are two benchmarks where *MTS* wins clearly in terms of speedup. *WS* steals remotely over twice as many times as *MTS* on *Sort* and nearly twice as many times as *MTS* on *Health*. The number of failed steals is also significantly higher with *WS* than with *MTS*. A failed steal occurs when a thief's lock-free probe of a victim indicates that work is available but upon acquisition of the lock to the victim's queue the thief finds no work to steal because another thread has stolen it or the victim has executed the tasks itself. Thus, both failed and completed steals contribute to overhead costs.

The MAESTRO scheduler aggregates inter-chip load balancing by permitting only one worker at a time to initiate bulk stealing from remote shepherds. Figure 2 shows how this improves performance on *Health*, one of the benchmarks sensitive to load balancing granularity. If only one task is stolen at time, subsequent steals are needed to provide all workers with tasks, adding to overhead costs. There are eight cores per socket on our test machine, thus eight workers per shepherd. This coincides with the peak performance: When the number of tasks stolen corresponds to the number of workers in the shepherd, all workers in the shepherd are able to draw work from the queue as a result of the steal.

Another benefit of the MAESTRO scheduler is better L3 cache performance, since all workers in a shepherd share the on-chip L3 cache. The *WS* scheduler exhibits poorer cache performance, and subsequently, more reads to main memory. Tables 6 and 7 show the relevant metrics for *Health* and *Sort* as measured using hardware performance counters, averaged over ten runs. They also show more traffic on the Quick Path Interconnect (QPI) between chips for *WS* than for *MTS*. The increased QPI traffic reflects more remote steals using *WS* and more snoop probes for data in remote L3 caches.

Benchmark	MTS		WS	
	Steals	Failed	Steals	Failed
Alignment (`single`)	1016	88	3695	255
Alignment (`for`)	109	122	1431	286
Fib	633	331	467	984
Health	28948	10323	295637	47538
NQueens	102	141	1428	389
Sort	1134	404	19330	3283
SparseLU (`single`)	18045	8133	68927	24506
SparseLU (`for`)	13486	11889	68099	32205
Strassen	227	157	14042	823

Table 5: Number of remote steal operations during execution of *Health* and *Sort* by Qthreads MAESTRO(MTS) & WS schedulers. In a failed steal, the thief acquires the lock on the victim's queue after a positive probe for work but ultimately finds no work available for stealing. On-chip steals performed by the WS scheduler are excluded. Average of ten runs.

Metric	MTS	WS	%Diff
L3 Misses	1.16e+06	2.58e+06	38
Bytes from Memory	8.23e+09	9.21e+09	5.6
Bytes on QPI	2.63e+10	2.98e+10	6.2

Table 6: Memory performance data for *Health* using MAESTRO(MTS) and WS. Average of ten runs.

4.3 Work Throttling

The MAESTRO scheduler in conjunction with the RCRDaemon reduces the number of active threads when shared resource contention is detected. The processor model used within the RCRDaemon is currently simple and straightforward. A simple test program was written to test the basic concept and to refine the model parameters for when contention was occurring. Each thread in the test program repeatedly iterated over a variably sized array that the thread had allocated. This is a reasonably common algorithmic occurrence in many dense linear algebra operations. As the array size is increased the cache footprint will first exceed the private L2 caches and then combine to exceed the shared L3 cache capacity.

The work throttling tests were executed using a Dell M605 system with 4 AMD Opteron (Barcelona) processors. Each processor has 4 cores and a shared L3 cache of 2 MBytes. Each core additionally has private L1 (split I/D 64KB/64KB) and L2 (512KB) caches. As the size of the array increases, performance degradations can be expected as each cache size is exceeded.

Table 8 gives the execution times for the test programs on for various array sizes on ICC, GCC, Stock Qthread and MAESTRO Qthreads. Each test used 32 active threads (one per core), except MAESTRO which started with 32 but reduce to 16 active threads when contention

Metric	MTS	WS	%Diff
L3 Misses	1.03e+7	3.42e+07	54
Bytes from Memory	2.27e+10	2.53e+10	5.5
Bytes on QPI	4.35e+10	4.87e+10	5.6

Table 7: Memory performance data for *Sort* using MAESTRO(MTS) and WS. Average of ten runs.

Configuration	ICC	GCC	Qthreads	Work Throttling
25000	0.22	0.04	0.06	0.07
50000	0.22	0.07	0.09	0.10
100000	0.22	0.14	0.16	0.17
200000	0.62	0.45	0.45	0.48
300000	1.62	1.58	1.80	1.42
400000	4.22	4.12	4.27	2.75
500000	5.62	5.44	5.39	5.41

Table 8: Repeated Array Walking for Intel, GCC, Qthreads and Work Throttling Qthreads – all with 32 threads

was detected. The startup time for ICC was higher than the other systems and that penalty was evident though the tested sizes. As the sizes increased from 25K to 100K, the execution time rose slower than the array size as startup remained a significant portion of the time. As the size (and the amount of work) doubled between 100K and 200K, the execution times for all increased by about 3x, or about 50% higher than desired. The sum of working set sizes approximates the cache size and conflict misses are starting to occur. Increasing the working set (and work) another 50% to 300K results in an increasing number of misses and execution times increase up to 400% for the standard OpenMp implementations. Work Throttling shows a significant performance improvement and it reduced the cache pressure allowing better cache hit rates. At 400K, the performance of the standard implementations are all about the same (4.2 seconds) or about 40x the time for 8x the work of 25K. MAESTRO with work throttling completes in about 2/3 the time. As the size continues to increase so that a single threads working set exceeds the cache size, performance of all of the implementations even out.

4.3.1 Discuss Early Throttling Results

In certain sweet spots, work throttling has significant performance potential. As the number of available cores rises, the control of dynamic thread count will also increase. With only four cores per chip, MAESTRO does not have many choices. With microprocessor with dozen or more cores, picking the correct number of threads to be active will become more complex and critical. Currently, if the memory concurrency is too high, the number of threads is cut from 4 to 2 and later when it is too low raised back. Higher thread counts will require more options and feedback between the scheduler and the performance model to determine the proper dynamic number of active threads.

Work throttling is preliminary work and many questions about it have not been answered. Even in applications where throttling is successful, throttling, as currently implemented, only reduces thread count when threads are looking for work. If the work chunks are big (like the beginning of guided self scheduled loops), the system may run for a substantial amount of time with a suboptimal number of hardware threads. The RCRDaemon memory model right now struggles to differentiate between cases where a single thread's footprint is 1/3 cache size and 20x the cache size. In the first case, only 3 threads should run, in the second all threads miss cache and should be run to make sure maximum throughput is maintained from main memory. Concerns also exist for real applications about how often throttling can be successful. Until the memory model is more robust, testing to figure this out will be speculative.

5 Summary

MAESTRO extended Qthreads to support OpenMP by way of the XOMP interface generated by the ROSE compiler. Previously, using the Qthreads runtime required the application to be ported to the Qthreads threading library interface. This implementation (and the independent Chapel compiler work) has greatly increased the number of potential programs using the Qthreads runtime. By supporting multiple programming paradigms the overall strengths and weaknesses of the overall design can be examined. Although not examined here, Qthreads has potential benefits as the target of new parallel languages (or domain specific languages). By using an existing runtime hopefully development time can be reduced.

As a publicly available runtime that supports multiple parallel programming styles Qthreads has become a useful research tool to explore the various programming issues involved in running parallel applications on the complicated nodes of modern systems. Achieving good performance from supercomputers with thousands or tens of thousands of nodes will require getting good to great performance from each level of the hierarchy that makes up those systems. Qthreads by providing a research platform that is competitive with the commonly used implementations of OpenMP allows studies of how changes in the hardware effect performance. As the performance limiting hardware resource moves from CPUs to memory bandwidth, network throughput or I/O performance the pressures on and resources available to the runtime will be changing. MAESTRO/Qthreads is an excellent platform to develop new concepts.

MAESTRO currently uses Qthreads to examine two different ways to improve application performance, hierarchical work stealing and work throttling. By providing different levels of work stealing in the runtime, application performance is improved by taking advantage of locality available in the parent-child task tree (or between siblings). Work is first stolen by threads which share L3 cache, resulting in satisfying references after the work is moved from the cache rather than a remote memory site. For some applications, steal local first significantly improves cache performance and reduces total application execution time. For most applications, it reduces the amount of remote steals and memory traffic, perhaps at a small cost of overall load balance. Work throttling examines ways to detect and improve performance when a shared resource is saturated. MAESTRO's initial resource to examine

the shared cache. When the cache miss rate is high, MAESTRO reduces the number of active threads, to ease cache pressure. Much work is to be done with throttling, but for a simple cache busting test, it produces a performance improvement of 35%.

References

[1] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[2] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP '95: Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216. ACM, 1995.

[3] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *SFCS '94: Proc. 35th Annual Symposium on Foundations of Computer Science*, pages 356–368. IEEE, Nov. 1994.

[4] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA '07: Proc. 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[5] Alejandro Duran and Xavier Teruel. Barcelona OpenMP Tasks Suite. `http://nanos.ac.upc.edu/projects/bots`, 2010.

[6] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *ICPP '09: Proc. 38th International Conference on Parallel Processing*, pages 124–131, Vienna, Austria, September 2009. IEEE Computer Society.

[7] Stephen Olivier, Allan Porterfield, Kyle Wheeler, and Jan Prins. Scheduling task parallelism on multi-socket multicore systems. In *International Workshop on Runtime and Operating Systems for Supercomputers*, Tuson, AZ, USA, June 2011.

[8] Allan Porterfield, Rob Fowler, and Min Yeol Lim. Rcrtool design document; version 0.1. Technical Report RENCI Technical Report TR-10-01, 2010.

[9] Allan Porterfield, Nassib Nassar, and Robert Fowler. Multi-threaded library for many-core systems. In *Workshop on Multi-Threaded Architectures and Applications*, Rome, Italy, May 2009.

[10] Kyle Wheeler, Richard Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium(IPDPS '08, in the MTAAP '08 workshop)*. IEEE Computer Society, 2008.