Scheduling Task Parallelism on Multi-Socket Multicore Systems

Stephen L. Olivier
University of North Carolina
at Chapel Hill, CB3175
Chapel Hill, NC 27599, USA
olivier@cs.unc.edu

Allan K. Porterfield
Renaissance Computing
Institute (RENCI)
100 Europa Drive, Suite 540
Chapel Hill, NC 27517, USA
akp@renci.org

Kyle B. Wheeler
Department 1423:
Scalable System Software
Sandia National Laboratories
Albuqurque, NM 87185, USA
kbwheel@sandia.gov

Jan F. Prins
University of North Carolina
at Chapel Hill, CB3175
Chapel Hill, NC 27599, USA
prins@cs.unc.edu

ABSTRACT

The recent addition of task parallelism to the OpenMP shared memory API allows programmers to express concurrency at a high level of abstraction and places the burden of scheduling parallel execution on the OpenMP run time system. This is a welcome development for scientific computing as supercomputer nodes grow "fatter" with multicore and manycore processors. But efficient scheduling of tasks on modern multi-socket multicore shared memory systems requires careful consideration of an increasingly complex memory hierarchy, including shared caches and NUMA characteristics. In this paper, we propose a hierarchical scheduling strategy that leverages different methods at different levels of the hierarchy. By allowing one thread to steal work on behalf of all of the threads within a single chip that share a cache, our scheduler limits the number of costly remote steals. For cores on the same chip, a shared LIFO queue allows exploitation of cache locality between sibling tasks as well between a parent task and its newly created child tasks. We extended the open-source Qthreads threading library to implement our scheduler, accepting OpenMP programs through the ROSE compiler.

We also present a comprehensive performance study of diverse OpenMP task parallel benchmarks, comparing seven different task parallel run time scheduler implementations on current generation multi-socket multicore systems: our hierarchical work stealing scheduler, a fully-distributed work stealing scheduler, a centralized scheduler, and LIFO and FIFO versions of the original Qthreads fully-distributed scheduler. In addition, we compare our results against OpenMP implementations from Intel and GCC. Hierarchical scheduling in Qthreads is competitive on all benchmarks. On several benchmarks, hierarchical scheduling in Qthreads demonstrates speedup and absolute performance superior to both the Intel and GCC OpenMP run time systems.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Concurrency, Scheduling, Threads*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Design, Measurement, Performance

1. INTRODUCTION

Task parallel programming models offer a simple way for application programmers to specify parallel tasks in a form that easily scales with problem size, leaving the scheduling of these tasks onto processors to be performed at run-time. Task parallelism is well suited to the expression of nested parallelism in recursive divide-and-conquer algorithms and of unstructured parallelism in irregular computations.

Recent interest in task parallel languages can be traced to the emergence of multicore processors and the realization that future performance improvements will increasingly require the use of additional cores rather than increasing performance of individual cores. A problem-centric approach to the specification of parallelism is attractive compared to a processor-centric SPMD specification of parallelism because it simplifies programming, can reasonably express a larger class of problems, and may offer more transparent scaling as the core count in a shared memory system increases.

An efficient task scheduler must meet challenging and sometimes conflicting goals: exploit cache and memory locality, maintain load balance, and minimize overhead costs. Load imbalance arises when there is an inequitable distribution of work among processors at a particular time. Without redistribution of work, idleness results. Load balancing operations, when successful, redistribute the work more equitably across processors. However, load balancing operations contribute to overhead costs. Furthermore, load balancing operations between sockets increase memory access time due to more cold cache misses and more high-latency remote memory accesses. This paper proposes an approach to mitigate these issues and advances understanding of their impact through the following contributions:

1. **A hierarchical scheduling strategy targeting modern multi-socket multicore shared memory systems** whose NUMA

architecture is not well supported by flat schedulers. Our approach combines work stealing and shared queues for low overhead load balancing and exploitation of shared caches.

2. **A detailed performance study on a current generation multi-socket multicore system** comparing seven run time implementations supporting task parallel OpenMP programs: five implementing different schedulers in our extensions to the open-source Qthreads library, GNU's GCC OpenMP run time, and the Intel OpenMP run time. In addition to speedup results demonstrating superior performance by our run time on many of the diverse benchmarks tested, we examine several secondary metrics that illustrate the benefits of hierarchical scheduling over flat work stealing.

The remainder of the paper is organized as follows: Section 2 provides relevant background information, Section 3 describes existing task scheduler designs and our hierarchical approach, Section 4 presents the results of our experimental evaluation, and Section 5 discusses related work. We conclude in Section 6 with some final observations and proposed future work.

2. BACKGROUND

Broadly supported by both commercial and open-source compilers, OpenMP allows incremental parallelization of serial programs for execution on shared memory parallel computers. Version 3.0 of the OpenMP specification for FORTRAN and C/C++ officially adds explicit task parallelism to complement its existing data parallel constructs [22, 2]. The OpenMP `task` construct generates a task from a statement or structured block. Task synchronization is provided by the `taskwait` construct, and the semantics of the OpenMP `barrier` construct have also been overloaded to require completion of all outstanding tasks.

Execution of OpenMP programs combines the efforts of the compiler and an OpenMP run time library. Intel and GCC both have integrated OpenMP compiler and run time implementations. Using the ROSE compiler [21], we have created an equivalent method to compile and run OpenMP programs with the Qthreads [26] library. The ROSE compiler is a source-to-source translator that supports OpenMP 3.0 with a simple compiler flag. In one compile step, it produces an intermediate C++ file and compiles that file with additional libraries to produce an executable. ROSE performs syntactic and semantic analysis on OpenMP directives, transforming them into run time library calls in the intermediate program. A common run time library (XOMP) serves as an intermediate representation to support OpenMP functionality. We have slightly modified ROSE (mostly bug fixes) to produce a file that is compiled by GCC and uses the Qthreads library to produce an executable.

2.1 Qthreads

Qthreads [26] is a cross-platform general-purpose parallel runtime designed to support lightweight threading and synchronization in a flexible integrated locality framework. Qthreads directly supports programming with lightweight threads and a variety of synchronization methods, including 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, e.g., performing an

FEB operation, a context switch is triggered. Because this context switch is done in user space via function calls and requires neither signals nor saving a full set of registers, it is less expensive than an operating system or interrupt-based context switch. This technique allows threads to process 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. Lightweight threads are created in user-space with a small context and small fixed-size stack and are then executed by worker pthreads. Each worker pthread is mapped to a locality domain, termed a **shepherd**, which is enforced with CPU pinning. Whereas Qthreads previously allowed only one worker pthread per shepherd, we added support for multiple worker pthreads per shepherd.

The Qthreads API includes several threaded loop interfaces, built on top of the core threading components. The API provides three basic parallel loop behaviors: 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-like structure to distribute sub-ranges of the iteration space to enable self-scheduled loops.

We added support for the ROSE produced XOMP calls to Qthreads allowing Qthreads to be used as the run time for OpenMP programs. Although Qthreads XOMP/OpenMP support is incomplete, it has accepted every OpenMP program accepted by ROSE. Note that our version of ROSE/Qthreads differs from the OpenMP standard in two ways: Default loop scheduling is self-guided, rather than static (though it can be explicitly requested), and one common OpenMP 3.0 idiom requires the addition of an extra `taskwait` statement.

3. TASK SCHEDULER DESIGN

The stock Qthreads scheduler, called Q in Section 4, was engineered for parallel loop computation: each processor executes a set of loop iterations packaged as lightweight threads. Round robin distribution of the iterations among the shepherds and self-scheduling are used in combination to maintain load balance. A simple lock-free per-shepherd FIFO queue stores iterations as they wait to be executed.

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 Qthreads, the run time must support more general dynamic load balancing while exploiting available locality among tasks. We implemented a modified Qthreads scheduler, L, to use LIFO rather than FIFO queues at each shepherd to improve the use of locality. However, the round robin distribution of tasks between shepherds does not provide fully dynamic load balancing.

3.1 Work Stealing & Centralized Schedulers

To better meet the dual goals of locality and load balance, we implemented work stealing. Blumofe et. al proved that work stealing is optimal for multithreaded scheduling of DAGs with minimal overhead costs [6], and they implemented it in their Cilk run time scheduler [5]. Our initial implementation of work stealing in Qthreads, *WS*, mimics Cilk's scheduling discipline: Each shepherd schedules tasks depth-first locally through LIFO queue operations. An idle shepherd obtains more work by stealing the oldest tasks from the task queue of a busy shepherd. We implemented two different probing schemes to find a victim shepherd, observ-

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 (non-blocking)	none
L	LIFO	one per core	round robin	LIFO (blocking)	none
CQ	Centralized Queue	one	N/A	LIFO (blocking)	N/A
WS	Work Stealing	one per core	local	LIFO (blocking)	FIFO stealing
MTS	Multi-Threaded Shepherds	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 1: Scheduler implementations evaluated: five Qthreads implementations, ICC, and GCC.

ing equivalent performance: choosing randomly and commencing search at the nearest shepherd ID to the thief. In the work stealing scheduler, 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, e.g., 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. [10] 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 acheived using a shared LIFO queue [4]. We implemented a centralized shared LIFO queue, *CQ*, for Qthreads, but it too is a poor match for multi-socket multicore systems since not all cores, but only cores on the same chip, share the same cache. Moreover, the centralized queue implementation is not scalable, as contention drives up the overhead costs.

3.2 Hierarchical Scheduling

To overcome the limitations of both work stealing and shared queues, we developed a hierarchical approach: multithreaded shepherds, *MTS*. We create one shepherd for 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, we map one worker 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. Thus, 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 queueing 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.

4. EVALUATION

To evaluate the performance of our hierarchical scheduler and the other Qthreads schedulers, we present results from the Barcelona OpenMP Tasks Suite (BOTS), version 1.1, available online [13]. The suite comprises a set of task parallel applications from various domains with varying computational characteristics [14]. Our experiments used the following benchmark components and inputs:

- *Alignment*: Aligns sequences of proteins using dynamic programming (100 sequences)

- *Fib*: Computes the *n*th Fibonacci number using brute-force recursion ($n = 50$)

- *Health*: Simulates a national health care system over a series of timesteps (144 cities)

- *NQueens*: Finds solutions of the *n*-queens problem using backtrack search ($n = 14$)

- *Sort*: Sorts a vector using parallel mergesort with sequential quicksort and insertion sort (128M integers)

- *SparseLU*: Computes the LU factorization of a sparse matrix (10000×10000 matrix, 100×100 submatrix blocks)

- *Strassen*: Computes a dense matrix multiply using Strassen's method (8192 x 8192 matrix)

For the *Fib*, *Health*, and *NQueens* benchmarks, the default manual cut-off configurations provided in BOTS are enabled to prune the generation of tasks below a prescribed point in the task hierarchy. For *Sort*, cutoffs are set to transition at 32K integers from parallel mergesort to sequential quicksort and from parallel merge tasks to sequential merge calls. For *Strassen*, the cut-off giving the best performance for each implementation is used. For both the *Alignment* and *SparseLU* benchmarks, BOTS provides two different source files: one in which computation starts with a single initial task and another in which tasks are generated in a loop.[1] 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

---

[1] 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.

Configuration	Alignment	Fib	Health	NQueens	Sort	SparseLU	Strassen
ICC -O2 -xHost -ipo Serial	28.33	100.4	15.07	49.35	20.14	117.3	169.3
GCC -O2 Serial	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
Qthreads MTS 32 workers	1.024	3.189	1.122	1.591	1.080	4.530	10.72

Table 2: Sequential and parallel performance using ICC, GCC, and the Qthreads MTS scheduler (time in sec.). For *Alignment* and *SparseLU*, the best time between the two parallel variations (`single` and `for`) is shown.
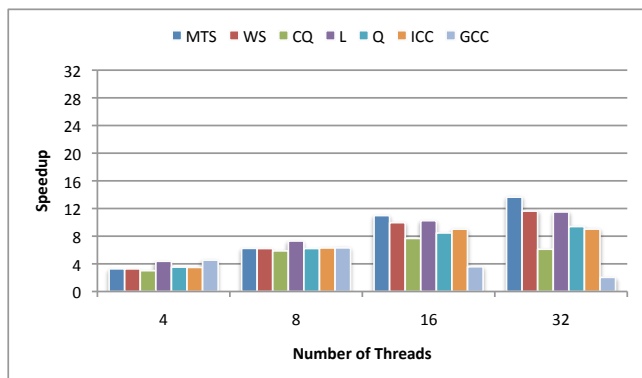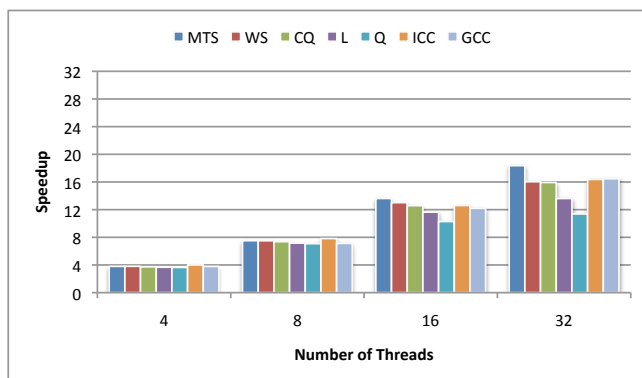

Figure 1: Health

Figure 2: Sort

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 systems, including five versions of Qthreads[2] and the widely available implementations from Intel and the Free Software Foundation (GNU) [15], as described in Table 1. 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 tasks are distributed across shepherd in a round robin basis when generated for load balance. 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

_____
[2]all compiled with GCC 4.4.4 -O2

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. *MTS* assigns one shepherd to 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.

4.1 Overall Performance

Overall the GCC compiler and ICC compiler produce excutables with similar serial performance, as shown in Table 2. 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* equalled GCC's performance [see discussion on Strassen in sec. 4.2]). Even as a research prototype, ROSE/Qthreads provides a competitive OpenMP task parallelism execution platform.

4.2 Individual Performance

Individual benchmark performance on multiple implementations of the OpenMP run time demonstrates 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.

The *Health* benchmark, Figure 1, shows significant diversity in performance and speedup. 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* ex-
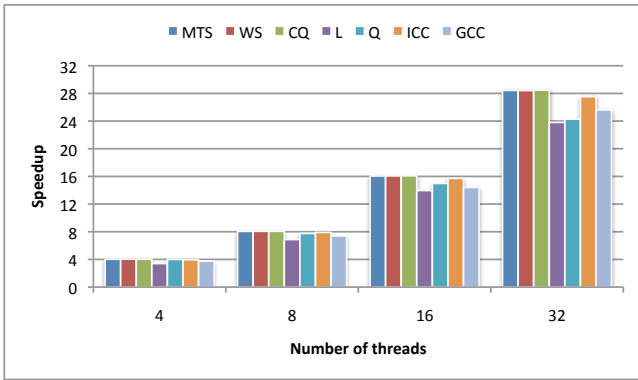
Figure 3: NQueens

Figure 4: Fib

Figure 5: Alignment-single

Figure 6: Alignment-for

periences 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 Qthread *MTS* 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.

The benefits of hierarchical scheduling can also be seen in Figure 2. *Sort*, for which we used a manual cutoff of 32K integers to switch between parallel and serial sorts, achieved speed up of about 16x for 32 cores on ICC and GCC, but just 11.4x for the base version of Qthreads, *Q*. The switch to a LIFO queue, *L*, improved speedup to 13.6x by facilitating data sharing between a parent and child. Independent changes to add work stealing, *WS*, and improve load balance, *CQ*, both improved speedup to 16x. By combining the best features of both work stealing and multiple threads sharing a queue, *MTS* increased speedup to 18.4x and achieved an 13.8% and 11.4% reduction in overall execution time compared to ICC and GCC OpenMP versions.

Locality effects allow *NQueens* to achieve slightly super-linear speedup for 4 and 8 cores using Qthreads. As seen in Figure 3, speedup is near-linear for 16 threads and only somewhat sub-linear for 32 threads on all OpenMP implementations. By adding load balancing mechanisms to Qthreads, its speedup improved significantly (24.3x to 28.4x). *CQ* and *WS* both improved load balance beyond what the LIFO queue (*L*) provides and little is gained by combining them together in *MTS*. The additional scaling of these three versions results in a execution time 12.6% faster than ICC and 10.9% faster than GCC.

Fib, Figure 4, uses a cut-off to stop the creation of very small tasks, and thus has enough work in each task to amortize the costs of queue access. *CQ* yields performance 2-3% faster than *MTS* and the other versions of Qthreads, since load balance is good and no time is spent looking for work. The load balancing versions of Qthreads (26.1x - 26.7x) scale better than Intel 24.9x. Both systems beat GCC substantially at only 15.8x. Overall, the scheduling improvements resulted in *MTS* running 26.5% faster than ICC and 28.8% faster than GCC but 2.0% slower than *CQ*.

The next two applications *Alignment* and *SparseLU*, each have two versions. For *Alignment*, Figures 5 and 6, speedup was near-linear for all versions and execution times between GCC and Qthreads were close (GCC +2.7% single initial task version; Qthreads +0.5%
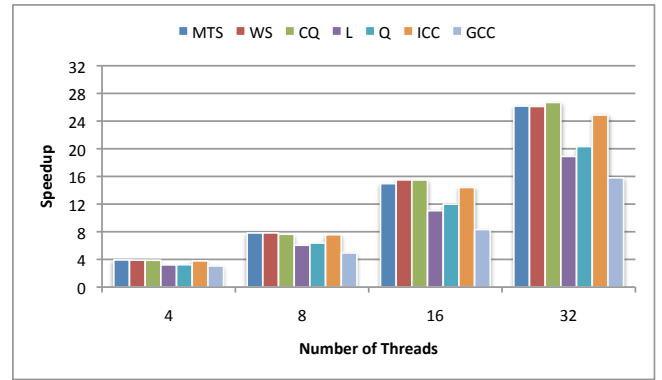
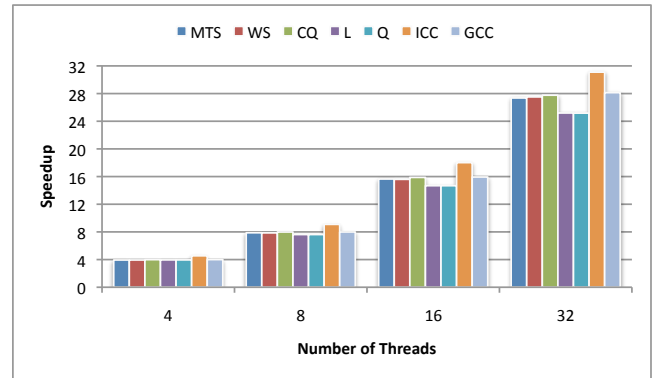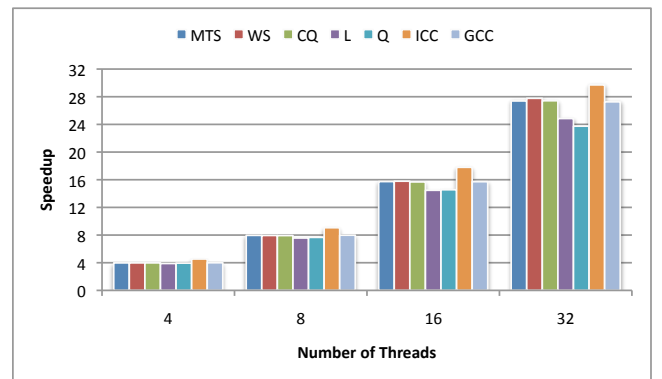parallel loop version). ICC scales better than GCC or Qthreads *MTS, WS, CQ*, with 12.4% lower execution time. Since *Alignment* has no `taskwait` synchronizations, we speculate that ICC scales better on this benchmark because it maintains fewer bookkeeping data structures in the absence of synchronization.

On both *SparseLU* versions, ICC serial performance improved nearly 3x using the -ipo and -xHost flags rather than using -O2 alone. The flags also improved parallel performance, but by only 60%, so the improvement does not scale linearly. On *SparseLU-single*, Figure 7, the performance of GCC and the various Qthreads versions is effectively equivalent, with speedup reaching 26.2x. Due
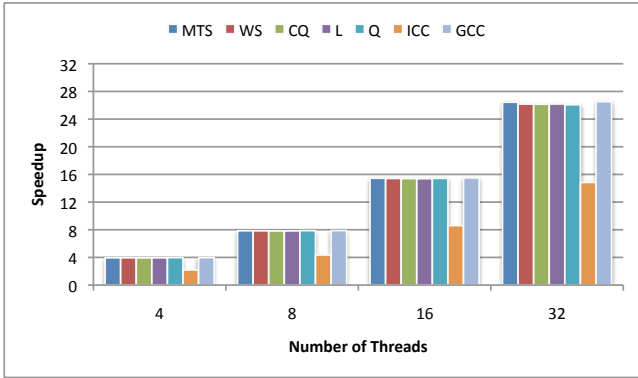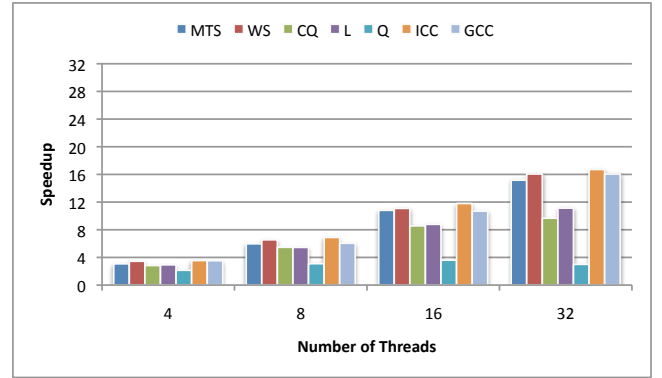
Figure 7: SparseLU-single
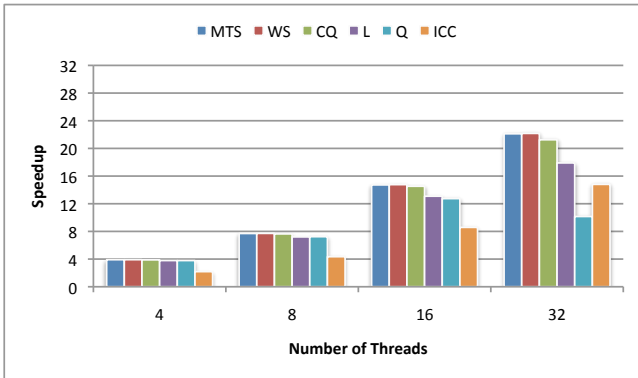

Figure 9: Strassen

Figure 8: SparseLU-for

to the aforementioned scaling issues, ICC speedup reaches only 14.8x. The execution times differ by 0.3% between GCC and *MTS* with both about 74.4% faster than ICC. On *SparseLU*-for, Figure 8, the GCC OpenMP runs were stopped after 30 minutes; thus data is not reported. ICC again scales poorly (14.8x), and Qthreads speedup improves due to the LIFO work queue and work stealing, reaching 22.2x. *MTS* execution time is 46.3% faster than ICC.

Strassen, Figure 9, performs recursive matrix multiplication using Strassen's method and is challenging for implementations with multiple workers accessing a queue. We used the cutoff setting that gave the best performance for each implementation: coarser (128) for *CQ* and *MTS* and the default setting (64) for the others. The execution times of GCC, and *WS* are within 1% of each other on 32 cores, and Intel scales slightly better (16.7x vs 16.1x). For *MTS*, in which only 8 threads share a queue (rather than 32 as in *CQ*) the speedup reaches 15.2x. For *CQ*, however, the performance hit due to queue contention is substantial, as speedup peaks at 9.7x. *Q* performance suffers from the FIFO ordering: not enough parallel work is expressed at any one time, and speedup never exceeds 4x.

4.3 Variability

One interesting feature of a work stealing run time is an idle thread's ability to search for work and the effect this has on performance in regions of limited parallelism or load imbalance. Table 3 gives the standard deviation of 10 runs as a percent of the fastest time for each configuration tested with 32 threads. Both Qthreads 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, as explained in the previous section.

4.4 Performance Benefits of MTS

Limiting the number of inter-chip load balancing operations is central to the design of our hierarchical scheduler (*MTS*). Consider the number of remote (off-chip) steal operations performed by *MTS* and by the flat work stealing scheduler *WS*, shown in Table 4. These counts exclude the number of on-chip steals performed by *WS*, and recall that *MTS* uses work stealing only between chips. We observe that *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 *MTS* scheduler aggregates inter-chip load balancing by permitting only one worker at a time to initiate bulk stealing from remote shepherds. Figure 10 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.

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

Table 3: Variability in performance using ICC, GCC, MTS, and WS schedulers (standard deviation as a percent of the fastest time).

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 4: Number of remote steal operations during execution of *Health* and *Sort* by Qthreads 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 5: Memory performance data for *Health* using MTS and WS. Average of ten runs.

Another benefit of the *MTS* 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 5 and 6 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.

5. RELATED WORK

Many theoretical and practical issues of task parallel languages and their run time implementations were explored during the development of earlier task parallel programming models, both hardware supported, e.g., Tera MTA [1], and software supported, e.g., Cilk [5, 16]. Much of our practical reasoning was influenced by experience with the Tera MTA run time, designed for massive multithreading and low-overhead thread synchronization. Cilk scheduling uses a *work-first* scheduling strategy coupled with a randomized work stealing load balancing strategy shown to be optimal [6]. Our use of shared queues is inspired by Parallel Depth-First Scheduling (PDFS) [4], which attempts to maintain a schedule close serial execution order, and its constructive cache sharing benefits [10].
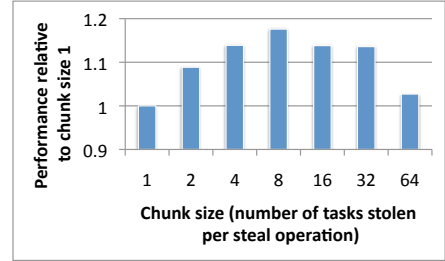

Figure 10: Performance on *Health* using MTS based on choice of the chunk size for stealing. Average of ten runs.

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 6: Memory performance data for *Sort* using MTS and WS. Average of ten runs.

The first prototype compiler and run time for OpenMP 3.0 tasks was an extension of Nanos Mercurium [24]. An evaluation of scheduling strategies for tasks using Nanos compared centralized breadth-first and fully-distributed depth-first work stealing schedulers [12]. Later extensions to Nanos included internal dynamic cut-off methods to limit overhead costs by inlining tasks [11].

In addition to OpenMP 3.0, there are currently several other task parallel languages and libraries available to developers: Microsoft Task Parallel Library [20] for Windows, Intel Thread Bulding Blocks (TBB) [19], and Intel Cilk Plus [18] (formerly Cilk++). The task parallel model and its run time support are also key components of the X10 [9] and Chapel [8] languages.

Hierarchical work stealing, i.e., stealing at all levels of a hierarchical scheduler, has been implemented for clusters and grids in Satin [25], ATLAS [3], and more recently in Kaapi [23, 17]. Those libraries are not optimized for shared caches in multi-core, which is the basis for the shared LIFO queue at the lower level of our hierarchical scheduler. The ForestGOMP run time system [7] also uses work stealing at both levels of its hierarchical scheduler, but like our system targets NUMA shared memory systems. It schedules OpenMP nested data parallelism by clustering related threads (not tasks) into "bubbles," scheduling them by work stealing among cores on the same chip, and selecting for work stealing between chips those threads with the lowest amount of associated memory. Data is migrated between sockets along with the stolen threads.

6. FUTURE WORK AND CONCLUSIONS

Our work on scheduling in Qthreads is ongoing. Inside the run time implementation, our goal is to decrease the minimum effective task size. The queue should be re-implemented, or perhaps re-

placed with a lock-free queue or a partially ordered heap, to reduce contention and further improve performance on programs with fine-grained tasks. We are also investigating ways to reduce the number of failed steals through refinements to the stealing protocol.

As multicore systems proliferate, the future of software development for supercomputing relies increasingly on high level programming models such as OpenMP for on-node parallelism. The recently added OpenMP constructs for task parallelism raise the level of abstraction to improve programmer productivity. However, if the run time can not execute applications efficiently on the available multicore systems, the benefits will be lost.

The complexity of multicore architectures grows with each hardware generation. Today, even off-the-shelf server chips have 6-12 cores and a chip-wide shared cache. Tomorrow may bring 30+ cores and multiple caches that service subsets of cores. Existing scheduling approaches were developed based on a flat system model. Our performance study revealed their strengths and limitations on a current generation multi-socket multicore architecture and demonstrated that mirroring the hierarchical nature of the hardware in the run time scheduler can indeed improve performance. Qthreads (by way of ROSE) accepts a large number of OpenMP 3.0 programs, and, using our *MTS* scheduler, has performance as high or higher than the commonly available OpenMP 3.0 implementations. Its combination of shared LIFO queues and work stealing maintains good load balance while supporting effective cache performance and limiting overhead costs. On the other hand, pure work stealing has been shown to provide the least variability in performance, an important consideration for distributed applications in which barriers cause the application to run at the speed of the slowest worker, e.g., in a Bulk Synchronous Processing (BSP) application with task parallelism used in the computation phase.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] G. A. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. J. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *ICS '92: Proc. 6th ACM Intl. Conference on Supercomputing*, pages 188–197. ACM, 1992.

[2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418, March 2009.

[3] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. Atlas: an infrastructure for global computing. In *EW 7: Proc. 7th ACM SIGOPS European Workshop*, pages 165–172, NY, NY, 1996. ACM.

[4] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *JACM*, 46(2):281–321, 1999.

[5] 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.

[6] 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.

[7] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst. Structuring the execution of OpenMP applications for multicore architectures. In *IPDPS 2010: Proc. 25th IEEE Intl. Parallel and Distributed Processing Symposium*. IEEE, April 2010.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *IJHPCA*, 21(3):291–312, 2007.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proc. 20th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 519–538, NY, NY, 2005. ACM.

[10] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA '07: Proc. 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115, NY, NY, 2007. ACM.

[11] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *SC08: ACM/IEEE Supercomputing 2008*, pages 1–11, Piscataway, NJ, 2008. IEEE Press.

[12] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In R. Eigenmann and B. R. de Supinski, editors, *IWOMP '08: Proc. Intl. Workshop on OpenMP*, volume 5004 of *LNCS*, pages 100–110. Springer, 2008.

[13] A. Duran and X. Teruel. Barcelona OpenMP Tasks Suite. http://nanos.ac.upc.edu/projects/bots, 2010.

[14] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *ICPP '09: Proc. 38th Intl. Conference on Parallel Processing*, pages 124–131. IEEE, Sept. 2009.

[15] Free Software Foundation Inc. GNU Compiler Collection. http://www.gnu.org/software/gcc/, 2010.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223. ACM, 1998.

[17] T. Gautier, X. Besseron, and L. Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07: Proc. 2007 Intl. Workshop on Parallel Symbolic Computation*, pages 15–23. ACM, 2007.

[18] Intel Corp. Intel Cilk Plus. http://software.intel.com/en-us/articles/intel-cilk-plus/, 2010.

[19] A. Kukanov and M. Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4), Nov. 2007.

[20] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. *SIGPLAN Notices: OOPSLA '09: 24th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, 44(10):227–242, 2009.

[21] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski. A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, editors, *IWOMP 2010: Proc. 6th Intl. Workshop on OpenMP*, volume 6132 of *LNCS*, pages 15–28. Springer, 2010.

[22] OpenMP Architecture Review Board. OpenMP API, Version 3.0, May 2008.

[23] J.-N. Quintin and F. Wagner. Hierarchical work-stealing. In *EuroPar '10: Proc. 16th Intl. Euro-Par Conference on Parallel Processing: Part I*, pages 217–229, Berlin, Heidelberg, 2010. Springer.

[24] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé. Support for OpenMP tasks in Nanos v4. In K. A. Lyons and C. Couturier, editors, *CASCON '07: Proc. 2007 Conference of the Center for Advanced Studies on Collaborative Research*, pages 256–259. IBM, 2007.

[25] R. van Nieuwpoort, T. Kielmann, and H. E. Bal. Satin: Efficient parallel divide-and-conquer in Java. In *Euro-Par '00: Proc. 6th Intl. Euro-Par Conference on Parallel Processing*, pages 690–699, London, UK, 2000. Springer.

[26] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS 2008: Proc. 22nd IEEE Intl. Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.