OpenMP Task Scheduling Strategies for Multicore NUMA Systems

Stephen L. Olivier
University of North Carolina at Chapel Hill
Campus Box 3175
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
Albuquerque, NM 87185, USA
kbwheel@sandia.gov

Michael Spiegel
Renaissance Computing Institute (RENCI)
100 Europa Drive, Suite 540
Chapel Hill, NC 27517, USA
mspiegel@renci.org

Jan F. Prins
University of North Carolina at Chapel Hill
Campus Box 3175
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. 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 non-uniform memory access (NUMA) characteristics. In this paper, we propose a hierarchical scheduling strategy that leverages different scheduling 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 as between a parent task and its newly created child tasks. In order to evaluate scheduling strategies, we extended the open-source Qthreads threading library to implement our schedulers, accepting OpenMP programs through the ROSE compiler.

We present a comprehensive performance study of diverse OpenMP task parallel benchmarks, comparing seven different task parallel run time scheduler implementations on an Intel Nehalem multi-socket multicore system: our hierarchical work stealing scheduler, a fully-distributed work stealing scheduler, a centralized scheduler, and LIFO and FIFO versions of the Qthreads fully-distributed scheduler. In addition, we compare our results against the Intel and GNU OpenMP implementations. Hierarchical scheduling in Qthreads is competitive on all benchmarks tested. On five of the seven benchmarks, hierarchical scheduling in Qthreads demonstrates speedup and absolute performance superior to both the Intel and GNU OpenMP run time systems. Our run time also demonstrates similar performance benefits on AMD Magny Cours and SGI Altix systems, enabling several benchmarks to successfully scale to 192 CPUs of an SGI Altix.

Keywords

Task parallelism, Run time systems, Work stealing, Scheduling, Multicore, OpenMP

1. INTRODUCTION

Task parallel programming models offer a simple way for application programmers to specify parallel tasks in a problem-centric 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.

An efficient task scheduler must meet challenging and sometimes conflicting goals: exploit cache and memory locality, maintain load balance, and minimize overhead costs. When there is an inequitable distribution of work among processors, load imbalance arises. Without redistribution of work, some processors become idle. Load balancing operations, when successful, redistribute the work more equitably across processors. However, load balancing operations can also contribute to overhead costs. 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 either fully-distributed or centralized 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 Intel system.** Seven run time implementations supporting task parallel OpenMP programs are compared: five schedulers that we added to the open-source Qthreads library, the GNU 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 fully-distributed work stealing.

3. **Additional performance evaluations on a two-socket multicore AMD system and a 192-processor SGI Altix.** These

evaluations demonstrate performance portability and scalability of our run time implementations.

This paper extends work originally presented in [26]. 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.

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++ adds explicit task parallelism to complement its existing data parallel constructs [27, 3]. 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 [24], we have created an equivalent method to compile and run OpenMP programs with the Qthreads [32] 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 invokes the GNU C++ compiler to compile 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. The ROSE common OpenMP run time library (XOMP) maps the run time calls to functions in the Qthreads library.

2.1 Qthreads

Qthreads [32] is a cross-platform general-purpose parallel run time library 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 and its implementation are intended to match future hardware environments by providing efficient software support for massive multithreading.

In the Qthreads execution model, lightweight threads (qthreads) are created in user-space with a small context and small fixed-size stack. Unlike heavyweight threads such as pthreads, qthreads do not support expensive features like per-thread identifiers, per-thread signal vectors, or preemptive multitasking. Qthreads are scheduled onto a small set of worker pthreads. Logically, a qthread is the smallest schedulable unit of work, such as a set of loop iterations or an OpenMP task, and a program execution generates many more qthreads than it has worker pthreads. Each worker pthread is pinned to a processor core and assigned to a locality domain, termed a **shepherd**. Whereas Qthreads previously allowed only one worker pthread per shepherd, we added support for multiple worker pthreads per shepherd. This support enables us to map shepherds to different architectural components, e.g., one shepherd per core, one shepherd per shared L3 cache, or one shepherd per processor socket.

The default scheduler in the Qthreads run time uses a cooperative-multitasking approach. When qthreads 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 qthreads to execute uninterrupted until data is needed that is not yet available, and allows the scheduler to attempt to hide communication latency by switching to other qthreads. Logically, this only hides communication latencies that take longer than a context switch.

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 qthread 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 used the Qthreads queueing implementation as a starting point for our scheduling work.

We added support for the ROSE XOMP calls to Qthreads allowing it to be used as the run time for OpenMP programs. Although Qthreads XOMP/OpenMP support is not fully complete, it accepts every OpenMP program accepted by ROSE. We implement OpenMP threads as worker pthreads. Unlike many OpenMP implementations, default loop scheduling is self-guided rather than static, though the latter can be explicitly requested. For task parallelism, we implement each OpenMP task as a qthread. (We use the term **task** rather than qthread throughout the remainder of the paper, both for simplicity and and because the scheduling concepts we explore are applicable to other task parallel languages and libraries.) We used the Qthreads FEB synchronization mechanism as a base layer upon which to implement `taskwait` and `barrier` sychronization.

3. TASK SCHEDULER DESIGN

The stock Qthreads scheduler, called *Q* in Section 4, was engineered for parallel loop computation. Each processor executes chunks of loop iterations packaged as qthreads. 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 interdependent 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 efficiently schedule tasks as lightweight threads in 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 [7], and they implemented it in their Cilk run time scheduler [6]. 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.

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.

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, observing 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 locking, 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. [12] 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 [5]. We implemented a centralized shared LIFO queue, *CQ*, for Qthreads, but it 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, typically L3, and all are proximal to a local memory attached to that socket. Within each shepherd, we map one pthread 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. Centralized task queueing for workers within each shepherd reduces the need for remote stealing by providing local load balance. By allowing only one representative worker to steal at time, in bulk for all workers in the shepherd, communication overheads are reduced. While a shared queue can be a performance bottleneck, the number of cores per chip is bounded, and intra-chip locking operations are fast.

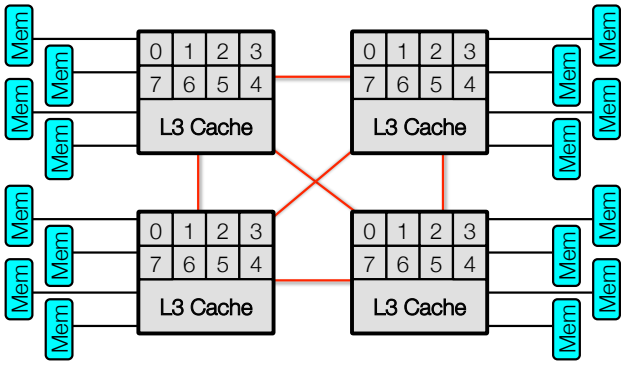

Figure 2: Topology of the 4-socket Intel Nehalem.

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 [15]. The suite comprises a set of task parallel applications from various domains with varying computational characteristics [16]. 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

```
#pragma omp single
   for (si = 0; si < nseqs; si++)
      for (sj = i+1; sj < nseqs; sj++)
         #pragma omp task firstprivate(si, sj)
            compare(seq[si], seq[sj]);
```

```
#pragma omp for schedule(dynamic)
   for (si = 0; si < nseqs; si++)
      for (sj = si+1; sj < nseqs; sj++)
         #pragma omp task firstprivate(si, sj)
            compare(seq[si], seq[sj]);
```

Figure 1: Simplified code for the two versions of *Alignment*: `single` (left) and `for` (right).

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 execution times 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.

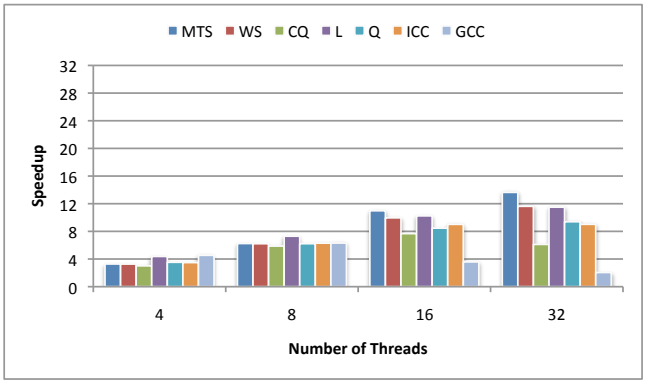

Figure 3: Health on 4-socket Intel Nehalem

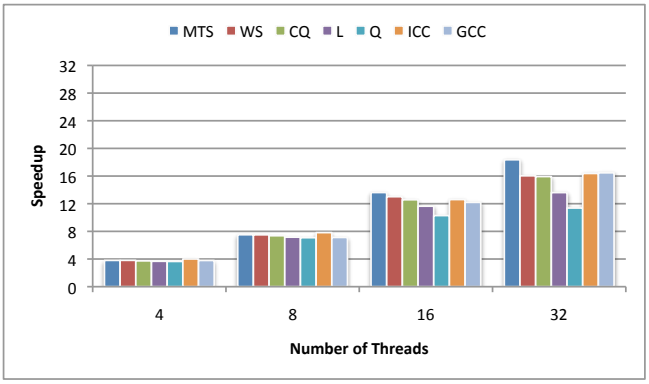

Figure 4: Sort on 4-socket Intel Nehalem

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. Other BOTS benchmarks are not presented here: UTS and FFT use very fine-grained tasks without cutoffs, yielding poor performance on all run times, and floorplan raises compilation issues in ROSE.

For both the *Alignment* and *SparseLU* benchmarks, BOTS provides two different source files. Simplified code given in Figure 1 illustrates the distinction between the two versions of *Alignment*. In the first (*Alignment*-`single`) the loop nest that generates the tasks is executed sequentially by a single thread. This version creates only task parallelism. In the second (*Alignment*-`for`) the outer loop is executed in parallel, creating both loop-level parallelism and task parallelism. Likewise, the two versions of *SparseLU* are one in which in which tasks are generated within single-threaded loop executions and another in which tasks are generated within parallel loop executions.

We ran the battery of tests on seven scheduler implementations: five versions of Qthreads[1], the GNU GCC OpenMP implementation [17], and the Intel ICC OpenMP implementation, as summarized in Table 1. The Qthreads implementations are as follows.

- *Q* is the original version of Qthreads and defines each core to be a separate locality domain or shepherd. It uses a non-blocking FIFO queue to schedule tasks within each shepherd (individual core). Each shepherd only obtains tasks from its local queue, although tasks are distributed across shepherds on a round robin basis for load balance.

- *L* incorporates a simple double ended locking LIFO queue to replace the original non-blocking FIFO queue. Concurrent access at both ends is required for work stealing, though *L* retains round robin task distribution for load balance rather than work stealing.

- *CQ* uses a single shepherd and centralized shared queue to distribute tasks among all of the cores in the system. This should provide adequate load balance, but contention for the queue limits scalability as task size shrinks.

- *WS* provides a shepherd (and individual queue) for each core, 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 large enough (tunable) to keep all of the cores busy.

[1] all compiled with GCC 4.4.4 -O2

4.1 Overall Performance on Intel Nehalem

The first hardware 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. The processors are fully connected using Intel QuickPath Interconnect (QPI) links, as shown in Figure 2. 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.

All executables using the Qthreads and GCC run times were compiled with GCC 4.4.4 with -g and -O2 optimization, for consistency. Executables using the Intel run time were compiled with ICC 11.1 and -O2 -xHost -ipo optimization. Reported results are from the best of ten runs.

Overall the GCC compiler and ICC compiler produce executables 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. If the -ipo and -xHost flags are not used with ICC on *SparseLU*, the GCC serial executable runs 3x faster than the ICC executable compiled with -O2 alone. The significance of this difference will be clearer in the presentation of parallel performance on *SparseLU* in Section 4.2. 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]). Thus even as a research prototype, ROSE/Qthreads provides competitive OpenMP task execution.

4.2 Individual Performance on Intel Nehalem

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 achieved speedup reveals the strengths and weaknesses of each scheduling approach.

The *Health* benchmark, Figure 3, 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* 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 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

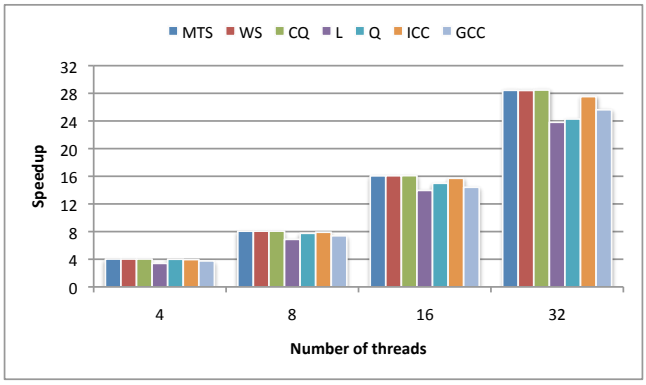

Figure 5: NQueens on 4-socket Intel Nehalem

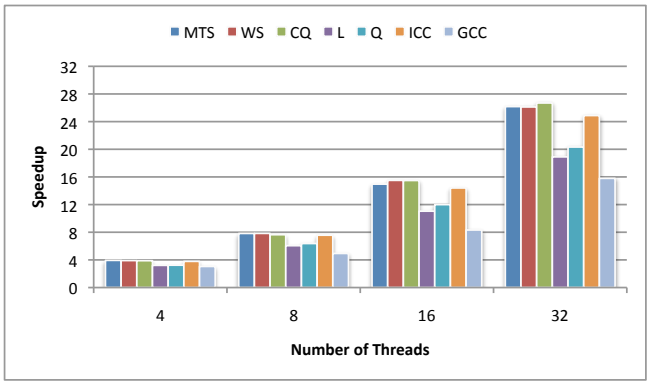

Figure 6: Fib on 4-socket Intel Nehalem

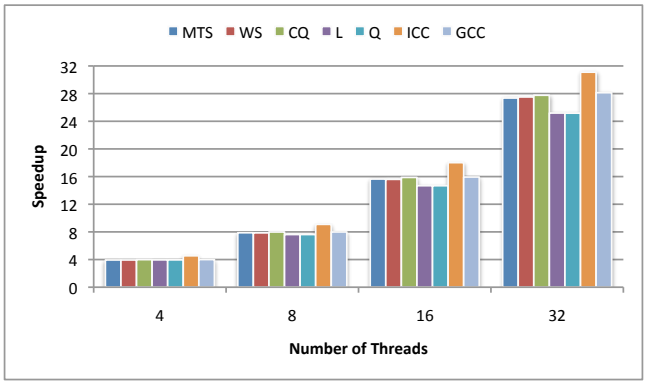

Figure 7: Alignment-single on 4-socket Intel Nehalem

example of how both work stealing and queue sharing within a system can independently and together improve performance, though the failure of any run time to reach 50% efficiency on 32 cores shows that there is room for improvement.

The benefits of hierarchical scheduling can also be seen in Figure 4. *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

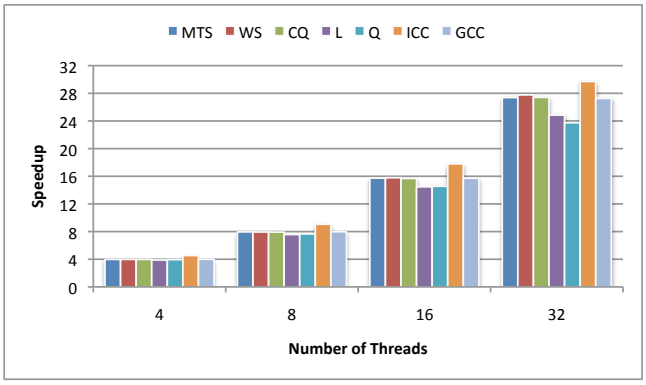

Figure 8: Alignment-for on 4-socket Intel Nehalem

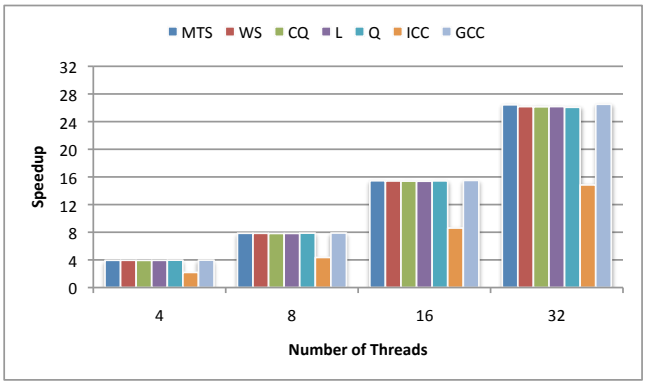

Figure 9: SparseLU-single on 4-socket Intel Nehalem

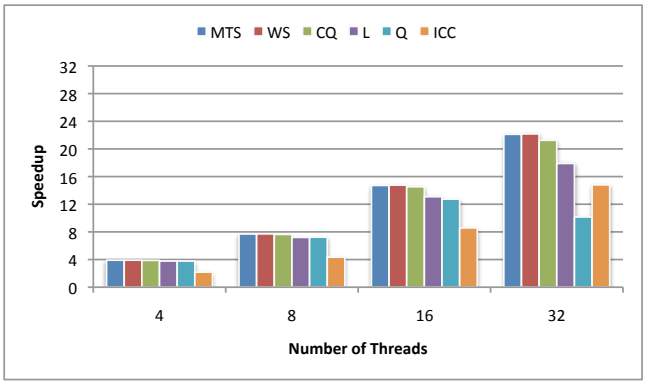

Figure 10: SparseLU-for on 4-socket Intel Nehalem

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 5, 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

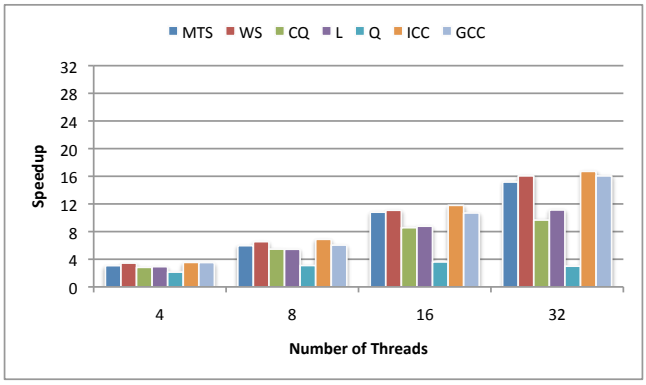

Figure 11: Strassen on 4-socket Intel Nehalem

three versions results in a execution time 12.6% faster than ICC and 10.9% faster than GCC.

Fib, Figure 6, 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 at 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 7 and 8, 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 9, the performance of GCC and the various Qthreads versions is effectively equivalent, with speedup reaching 26.2x. Due 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 10, the GCC OpenMP runs were stopped after exceeding the sequential time; 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 11, 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.

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 on 4-socket Intel Nehalem 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 5: 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 6: Memory performance data for *Health* using MTS and WS. Average of ten runs on 4-socket Intel Nehalem.

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 variation in execution time for 3 of the 9 programs. For 8 of the 9 benchmarks, both *WS* and *MTS* show less variability than ICC.

In three cases (*Alignment*-`single`, *Health*, *SparseLU*-`single`), Qthreads *WS* variability 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 variability 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 variability 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.

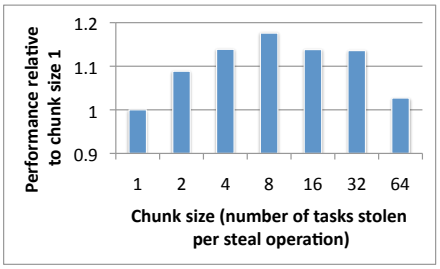

Figure 12: Performance on *Health* using MTS based on choice of the chunk size for stealing. Average of ten runs on 4-socket Intel Nehalem.

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 MTS and WS. Average of ten runs on 4-socket Intel Nehalem.

4.4 Performance Analysis 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 5. 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 12 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, and a target of eight tasks stolen per steal request. This coincides with the peak performance: When the target number of tasks stolen corresponds to the number of workers

	Alignment (single)	Alignment (for)	Fib	Health	NQueens	Sort	SparseLU (single)	SparseLU (for)	Strassen
Tasks Stolen	5900	450	2181	159386	423	5214	93117	38198	1355
Tasks Per Steal	5.8	4.1	3.4	5.5	4.1	4.6	5.1	2.8	6.0

Table 4: Tasks stolen and tasks per steal using the MTS scheduler. Average of ten runs.

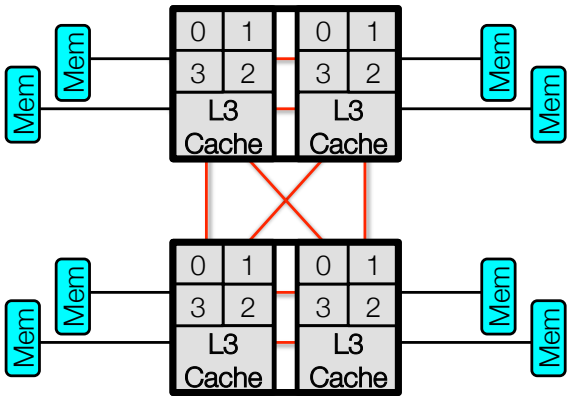

Figure 13: Topology of the 2-socket/4-chip AMD Magny Cours.

in the shepherd, all workers in the shepherd are able to draw work immediately from the queue as a result of the steal.

Frequently the number of tasks available to steal is less than the target number to be stolen. Table 4 shows the total number of tasks stolen and the average number of tasks stolen per steal operation. Across all benchmarks, the range of tasks stolen per steal is 2.8 to 6.0. The numbers skew downward due to a scarcity of work during start-up and near termination, when only one or few tasks are available at a time. Note the lower number both of total steals and tasks per steal for the `for` versions of *Alignment* and *SparseLU* compared to the `single` versions. Loop parallel initialization provides good initial load balance so that fewer steals are needed, and those that do occur sporadically are near termination and synchronization phases.

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 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*. QPI traffic occurs when data is requested and transferred from either remote memory or remote L3 cache, i.e., attached to a different socket of the machine. Not only are remote accesses higher latency, but they also result in remote cache invalidations of shared cache lines and subsequent coherence misses. Increased QPI traffic in *WS* reflects more remote steals and more accesses to data in remote L3 caches and remote memory. In summary, *MTS* gains advantage by exploiting locality among tasks executed by threads on cores of the same chip, making good use of the shared L3 cache to access memory less frequently and avoid high latency remote accesses and coherence misses.

4.5 Performance on AMD Magny Cours

We also evaluate the Qthreads schedulers against ICC and GCC on a 2-socket AMD Magny Cours system, one node of a cluster at Sandia National Laboratories. Each socket hosts an Opteron 6136 multi-chip module: two quad-core chips that share a package connected via two internal HyperTransport (HT) links. The remaining two HT links per chip are connected to the chips in the other socket, as shown in Figure 13. Each chip contains a memory controller with 8GB attached DDR3 memory, a 5MB shared L3 cache, and four 2.4 MHz cores with 64kb L1 and 512kb L2 caches. Thus, there are a total of 16 cores and 32GB memory, evenly divided among four HyperTransport-connected NUMA nodes (one per chip, two per socket). The system runs Cray compute-node Linux kernel 2.6.27, and we used the GCC 4.6.0 with -O3 optimization and ICC 12.0 with -O3 -ipo -msse3 -simd optimization.

We ran the same benchmarks with the same parameters as the Intel Nehalem evaluation. Sequential execution times are reported in Table 8. Again, interprocedural optimization (-ipo) in ICC was essential to match the GCC performance; execution time was more than 500 seconds without it. The greatest remaining difference between the sequential times is on *Alignment*, where GCC is 20% slower than ICC.

Speedup results using 16 threads are given in Figure 14, for Qthreads configurations with one shepherd per chip, *MTS (4Q)*; one shepherd per socket, *MTS (2Q)*; one shepherd per core (flat work stealing), *WS*; ICC; and GCC. At least one of the Qthreads variants matches or beats ICC and GCC on all but one of the benchmarks. Moreover, the Qthreads schedulers achieve near-linear to slightly super-linear speedup on 6 of the 9 benchmarks: the two versions of *Alignment*, *Fib*, *NQueens*, and the two versions of *SparseLU*. Of those, speedup using ICC is 22% and 23% lower than Qthreads on the two versions of *Alignment*, 10% and 18% lower on the two versions of *SparseLU*, 9% lower on *NQueens* and 7% lower on *Fib*. GCC is 42% lower than Qthreads on Fib, 9% and 27% lower on the two versions of *SparseLU*, and close on *NQueens* and both versions of *Alignment*.

On three of the benchmarks, no run time achieves ideal speedup. *Strassen* is the only benchmark on which ICC and GCC outperform Qthreads, and even ICC falls short of 10X. On *Sort*, the best performance is with Qthreads *WS*, *MTS(4Q)*, and GCC all at roughly 8x. Speedup is lower with Qthreads *MTS (2Q)* and still lower with *CQ*, indicating that centralized queueing beyond the chip level is counterproductive. ICC speedup lags behind the other schedulers on this benchmark. Speedup on *Health* peaks at 3.3x on this system using the Qthreads schedulers, with even worse speedup using ICC and GCC.

The variability in execution times is shown in Table 9. The standard deviations for all of the benchmarks on the *MTS* and *WS* Qthreads implementations are below 2% of the best case execution time. On all but two of the benchmarks, the *MTS* standard deviation is less than 1%.

The Magny Cours results demonstrate that the competitive, and in some cases superior, performance of our Qthreads schedulers against ICC and GCC is not confined to the Intel architecture. At first glance, differences in performance using the various Qthreads configurations seem less pronounced than they were on the four socket Intel machine. However, those differences were strongest on the Intel machine at 32 threads, and the AMD system only has 16 threads. Some architectural differences go beyond the difference in

Configuration	Alignment	Fib	Health	NQueens	Sort	SparseLU	Strassen
ICC	23.93	107.9	10.18	60.56	18.51	156.0	214.9
GCC	29.77	105.0	10.67	58.16	17.72	153.4	211.1

Table 8: Sequential execution times using ICC and GCC on the AMD Magny Cours.

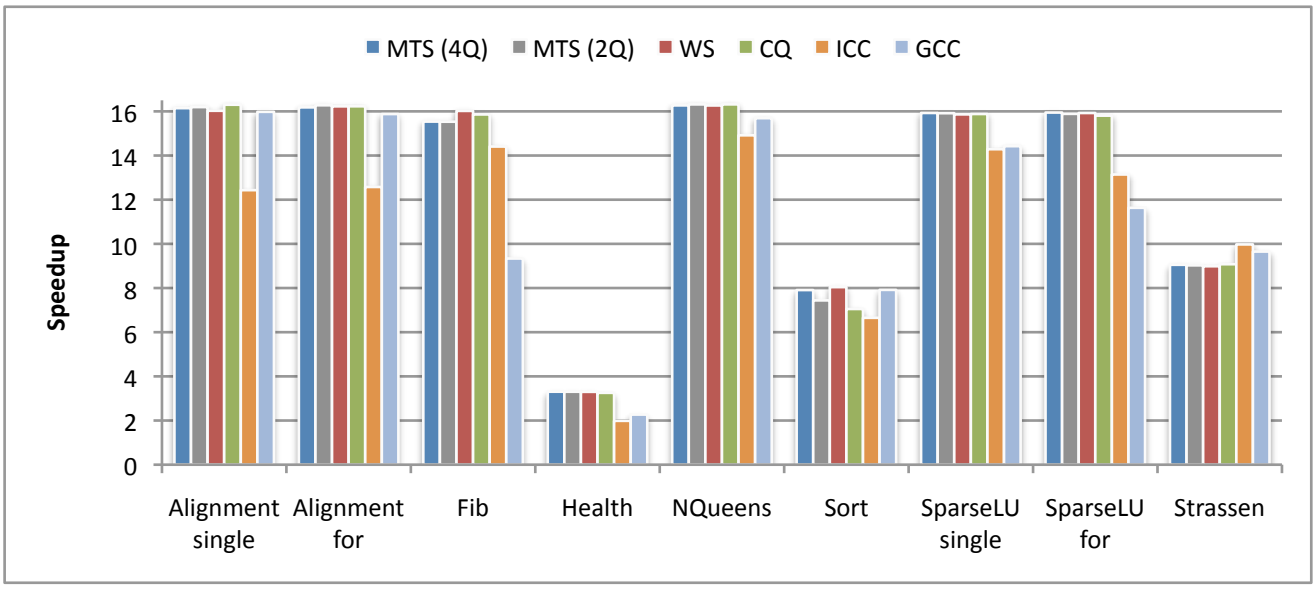

Figure 14: BOTS benchmarks on 2-socket AMD Magny Cours using 16 threads

core count. *MTS* is designed to leverage locality in shared L3 cache, but the Magny Cours has much less L3 cache per core than the Intel system (1.25MB/core versus 2.25MB/core). Less available cache also accounts for worse performance on the data-intensive *Sort* and *Health* benchmarks.

4.6 Performance on SGI Altix

We evaluate scalability beyond 32 threads on an SGI Altix 3700. Each of the 96 nodes contains two 1.6MHz Intel Itanium2 processors and 4GB of memory, for a total of 192 processors and 384GB of memory. The nodes are connected by the proprietary SGI NUMALink4 network and run a single system image of SuSE Linux kernel 2.6.16. We used the GCC 4.5.2 compiler as the native compiler for our ROSE-transformed code and the GCC OpenMP run time for comparison against Qthreads. The version of ICC on the system is not recent enough to include support for OpenMP tasks. Sequential execution times, given in Table 10, are slower than those of the other machines, because the Itanium2 is an older processor, runs at a lower clock rate, and uses a different instruction set (ia64).

The best observed performance on any of the benchmarks was on *NQueens*, shown in Figure 15. *WS* achieves 115x on 128 threads (90% parallel efficiency) and reaches 148x on 192 threads. *MTS* reaches 134x speedup. (On this machine, the *MTS* configuration has two threads per shepherd to match the two processors per NUMA node.) *CQ* tops out at 77x speedup on 96 threads, beyond which overheads from queue contention become overwhelming. GCC gets up to only 40x speedup. Although no run time achieves linear speedup on the full machine, they all reach 30x to 32x speedup with 32 threads; this underlines the importance of testing at higher processor counts to evaluate scalability. On the *Fib* benchmark, shown in Figure 16, *MTS* almost doubles the performance of *CQ* and GCC on 192 threads, with a maximum speedup of 97x. *CQ* peaks at 68x

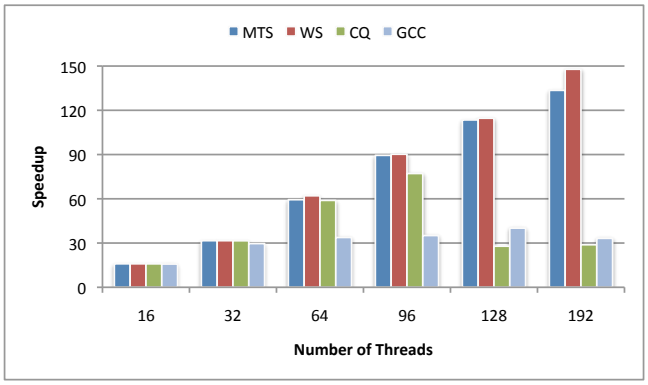

Figure 15: NQueens on SGI Altix

speedup on 128 threads and *WS* exhibits its worst performance relative to *MTS*, maxing out at 77x speedup on 96 threads.

We see better peak performance on *Alignment*-`for` (Figure 18) than *Alignment*-`single` (Figure 17). *WS* reaches 116x speedup on 192 threads and *MTS* reaches 107x, with *CQ* and GCC performing significantly worse. On the other hand, *SparseLU*-`single` (Figure 19) scales better than *SparseLU*-`for` (Figure 20). Peak speedup on *SparseLU*-`single` is 89x with *MTS* and 86x with *WS*, while *SparseLU*-`for` achieves a peak speedup of 60x. As was the case on the 4-socket Intel machine, GCC is unable to complete after a timeout equal to the sequential execution time.

For three of the benchmarks, no improvement in speedup was observed beyond 32 threads: *Health*, *Sort*, and *Strassen*. As shown in Figure 21, none exceed 10x speedup on the Altix. These were also observed to be the most challenging on the four-socket In-

Configuration	Alignment (single)	Alignment (for)	Fib	Health	NQueens	Sort	SparseLU (single)	SparseLU (for)	Strassen
ICC	2.2	0.80	1.3	14	1.1	8.2	0.62	0.31	2.5
GCC	0.035	0.27	5.4	0.38	0.96	3.5	0.016	0.025	1.1
Qthreads MTS (4Q)	0.25	0.63	1.5	0.17	0.13	1.1	0.012	0.16	0.98
Qthreads MTS (2Q)	0.46	0.68	1.4	0.069	0.24	0.30	0.015	0.081	0.87
Qthreads WS	0.21	1.3	1.5	0.15	0.13	1.8	0.036	0.094	1.4

Table 9: Variability in performance on AMD Magny Cours using 16 threads (standard deviation as a percent of the fastest time).

Configuration	Alignment	Fib	Health	NQueens	Sort	SparseLU	Strassen
GCC	53.96	139.2	45.60	63.62	33.59	632.7	551.3

Table 10: Sequential execution times on the SGI Altix.

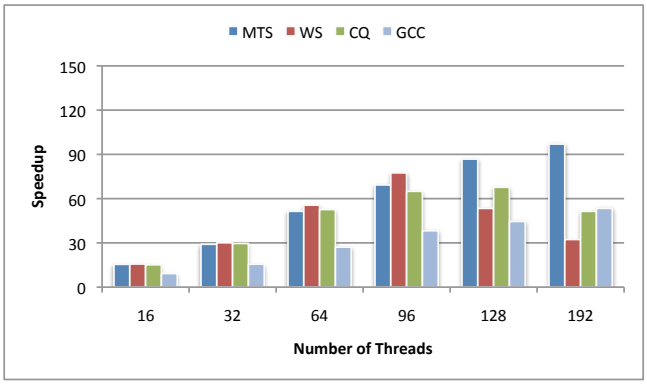

Figure 16: Fib on SGI Altix

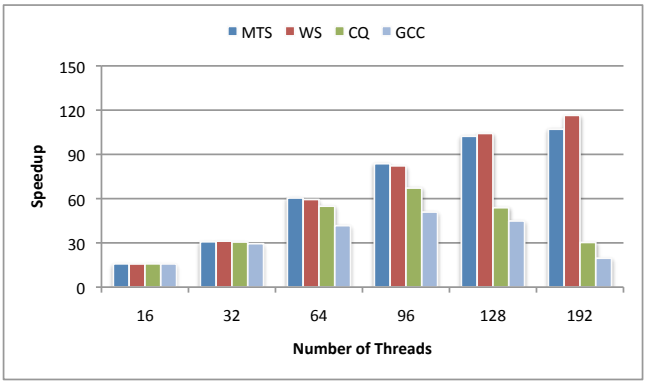

Figure 18: Alignment-for on SGI Altix

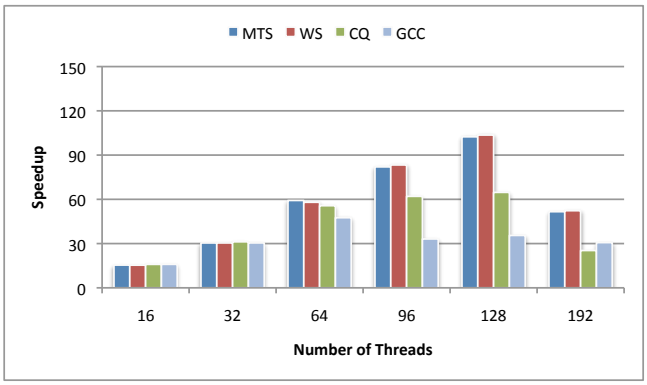

Figure 17: Alignment-single on SGI Altix

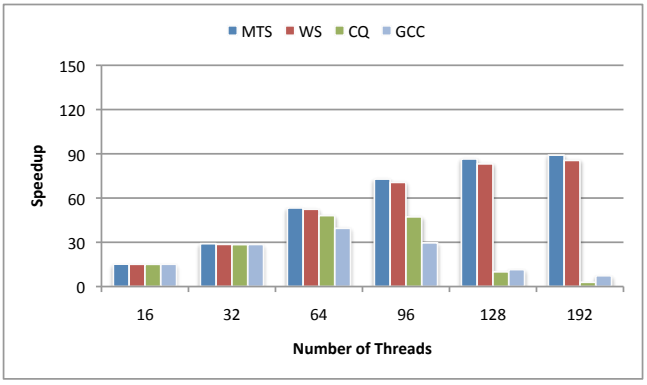

Figure 19: SparseLU-single on SGI Altix

tel and two-socket AMD systems. *Health* and *Sort* are the most data-intensive and require new strategies to achieve performance improvement, an important area of research going forward.

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 [6, 18]. 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 [7]. Our use of shared queues is inspired by Parallel Depth-First Scheduling (PDFS) [5], which attempts to maintain a schedule close serial execution order, and its constructive cache sharing benefits [12].

The first prototype compiler and run time for OpenMP 3.0 tasks was an extension of Nanos Mercurium [30]. An evaluation of scheduling strategies for tasks using Nanos compared centralized breadth-first and fully-distributed depth-first work stealing schedulers [14]. Later extensions to Nanos included internal dynamic cut-off methods to limit overhead costs by inlining tasks [13].

In addition to OpenMP 3.0, there are currently several other task parallel languages and libraries available to developers: Mi-

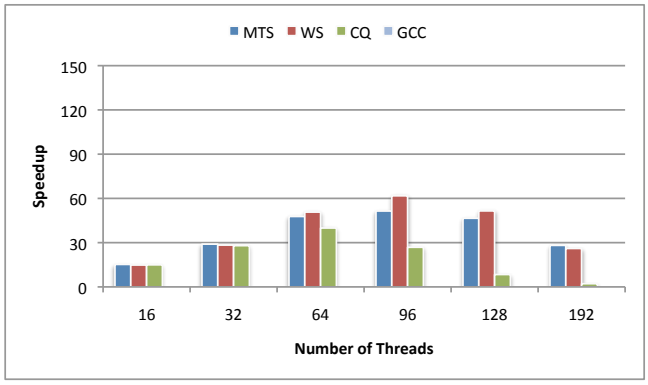

Figure 20: SparseLU-for on SGI Altix

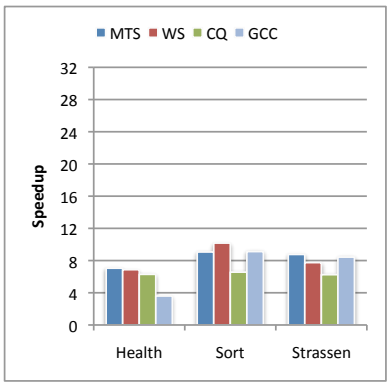

Figure 21: Health, Sort, and Strassen on SGI Altix: 32 threads

crosoft Task Parallel Library [23] for Windows, Intel Thread Bulding Blocks (TBB) [22], and Intel Cilk Plus [21] (formerly Cilk++). The task parallel model and its run time support are also key components of the X10 [10] and Chapel [9] languages.

Hierarchical work stealing, i.e., stealing at all levels of a hierarchical scheduler, has been implemented for clusters and grids in Satin [31], ATLAS [4], and more recently in Kaapi [28, 19]. 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 [8] 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. CONCLUSIONS AND FUTURE WORK

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.

The scalability results on the SGI Altix are important because previous BOTS evaluations [16, 26] only presented results on up to 32 cores. It is encouraging that several benchmarks reach speedup of 90X-150X on 196 cores. The challenge on those benchmarks is to close the performance gap between observed speedup and ideal speedup through incremental reductions in overhead costs and idle times and better exploitation of locality. Other benchmarks fail to scale well even at 32 threads or less. On the data-intensive sort and health benchmarks we have observed a sharp increase in computation time due to increased load latencies compared to sequential execution. To ameliorate that issue, we are investigating programmer annotations to specify task scheduling constraints that identify and maintain data locality.

One challenge posed by our hierarchical scheduling strategy is the need for an efficient queue supporting concurrent access on both ends, since workers within a shepherd share a queue. Most existing lock-free queues for work stealing, such as the Arora, Blumofe, and Plaxton (ABP) queue [2] and resizable variants [20, 11], allow only one thread to execute `push()` and `pop()` operations. Lock-free doubly-ended queues (deques) generalize the ABP queue to allow for concurrent insertion and removal on both ends of the queue. Lock-free deques have been implemented with compare-and-swap atomic primitives [25, 29], but speed is limited by their use of linked lists. We are currently working to implement an array-based lock-free deque, though even with a lock-based queue we have achieved results competitive with and in many cases better than ICC and GCC.

7. ACKNOWLEDGMENTS

This work is supported in part by a grant from the United States Department of Defense. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

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] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proc. 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129. ACM, 1998.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proc. 17th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28. ACM, 2005.

[12] 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. ACM, 2007.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] D. Hendler, Y. Lev, M. Moir, and N. Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18:189–207, 2006.

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

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

[23] 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.

[24] 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.

[25] M. M. Michael. CAS-based lock-free algorithm for shared deques. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003: Proc. 9th Euro-Par Conference on Parallel Processing*, volume 2790 of *LNCS*, pages 651–660. Springer, 2003.

[26] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins. Scheduling task parallelism on multi-socket multicore systems. In *ROSS '11: Proc. Intl. Workshop on Runtime and Operating Systems for Supercomputers (in conjunction with 2011 ACM Intl. Conference on Supercomputing)*, pages 49–56. ACM, 2011.

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

[28] 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.

[29] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In T. Higashino, editor, *OPODIS 2004: 8th Intl. Conference on Principles of Distributed Systems*, volume 3544 of *LNCS*, pages 240–255. Springer, 2005.

[30] 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.

[31] 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.

[32] 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.