SAND2010-1878C

Dynamic Loop Scheduling in Balanced Loops

Omitted for review

Abstract

Qthreads directly supports programming with light-weight threads and a variety of synchronization methods. The qthread 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. Qthreads virtualizes hardware resources, simplifying the application's job of finding and distributing parallel work among the available parallel hardware. Performance is sensitive to the current level of parallelism and often impossible to determine statically. There must be sufficient threads to saturate hardware resources, but the threads must be large enough to amortize start up costs.

Qthreads supports runtime adaptation though a dynamic control mechanism to define the length of parallel regions. Threads are associated and scheduled within a thread mobility domain or "shepherd". Threads may be explicitly migrated. Qthreads also supports a parallel loop construct with runtime blocking of the iterations.

This paper presents Qthreads and examines one early interaction between Qthreads and applications, dynamic blocking of OpenMP benchmark loops to improve load balance and limit overhead. For two OpenMP benchmarks without load balance problems, using an automatic translation process, Qthreads performance matched or exceeded GNU OpenMP for multiple hardware configurations with improved performance and stability. Overall application load balance was improved by over-decomposing parallel loops and intelligent scheduling of the resulting threads. The overhead of loop decomposition and thread scheduling approximated the gain from improved application load balance.

1 Introduction

Many-core chips, nodes and systems need to be programmed in parallel today and parallel programming is difficult. Currently, to maximize performance applications must not only find parallel threads, but must also match the number of threads to available hardware and guarantee that each thread is ‘big’ enough. However, matching threads to hardware resources is best carried out by the runtime rather than the application.

Many-core microprocessors and multi-socket microprocessor systems exist today with Intel, AMD and IBM announcing and demoing systems with 48 to 64 hardware threads within a single address space[2, 4, 10]. These new systems promise greater computational performance than ever before. The number of hardware threads available is only expected to rise in the coming years.

The current trend has memory bandwidth increasing at a slower rate than the thread count, and most of the bandwidth improvement is due to buffering and faster transfer speeds, not faster memory cells. The effective memory bandwidth available per hardware thread is holding even or dropping as the thread count increases[11]. Bandwidth per thread is unlikely to increase until users demand faster, rather than additional, memory.

Turning the massive amount of available computational horsepower (and shrinking memory bandwidth) into application performance is the challenge facing system and application programmers. Parallelism adds to the challenge by introducing synchronization, communication and data race considerations—problems that are not found in old-fashioned serial programming.

The current popular parallel programming models present a basically static view of parallel hardware that the application is expected to map itself on to. Not only does the programmer/system have to find regions of independent work for each hardware thread, for efficiency on most systems the number of parallel regions (software threads) needs to match the available hardware threads and each software thread must be large enough to amortize the cost of creating it. For MPI programs, the overhead of parallel threads is typically on the order of program startup, while OpenMP programs reduce the overhead to an OS call to start a new pthread. The cost and inflexible nature of software thread startup, has caused parallel programming models to downplay the dynamic nature of parallelism in actual applications.

The Qthreads runtime is designed to support dynamic programming and performance features not typically seen in either OpenMP or MPI systems. Parallel regions of any size and location are specified and the Qthread runtime maps the software threads onto available hardware resources. By moving software threads to idle hardware resources, it is possible for the runtime to automatically load balance many applications that would otherwise either have idle hardware threads or require application level load balancing techniques.

It is easy with this flexibility to generate more software threads than needed for regions of an application. The overhead of creating and starting these parallel activities can be substantial. By communicating between the runtime and the application, Qthreads has a dynamic control mechanism to modify the both the creation and the length of application software threads.

Just loading the required execution environment for a new software thread takes time, even if the thread creation mechanism is free. Static models can roll this assignment cost up, so that it is paid only once per hardware thread. To approximate the performance of a static model, the dynamic model needs to watch the percentage of execution time spent in thread creation. One way to accomplish this is to have the runtime dynamically block the execution of threads so that multiple identified software threads are instantiated every scheduling operation. Increasing the number of iterations in a software thread increases application performance, by reducing the scheduling overhead. However, increased thread size can also decrease performance though load imbalances at the end of each loop.

Dynamic interaction between the Qthreads and the application has long-term advantages for performance and reliability. As systems grow in size and complexity, performance will not be limited by the computational resources, but by other critical shared resources (memory bandwidth, network bandwidth or latency, IO, etc.). An interface between the runtime and the application to modify thread creation and scheduling potentially has significant performance impacts. For example, threads sharing a single data location or that use adjacent data locations can be generated independently but forced to schedule together resulting in cache reuse that may other wise be difficult or impossible to program. Threads with IO can be placed near the physical disk subsystem for better performance. A map of an application's network utilization can be computed to create better virtual to physical node layouts.

One problem that any new programming model must address is the lack of programs written to use it. Qthreads initially addressed large graph problems with dynamic synchronization patterns that are hard to program on other systems. Unfortunately, these problems and the languages and libraries they are in, are not familiar to most application programmers. In an effort to increase the number of test programs for Qthreads, OpenMP programs were automatically translated to the Qthreads API. Each iteration of a parallel for loop is considered an independent software thread. The OpenMP to Qthreads translator greatly increases the number of programs that can be run on Qthreads. OpenMP programs are generally designed to have balanced loops which can be efficiently scheduled by its static scheduling mechanism. Qthreads is designed for loops which require a more dynamic mechanism. The efficiency of Qthreads scheduling can be examined by comparing its performance to OpenMP's on the balanced loop OpenMP was designed to execute.

By comparing Qthreads dynamic blocking both against a variety of static techniques and against OpenMP, an accurate picture of the benefits can be obtained. Dynamic loop blocking in Qthreads on some systems can outperform OpenMP by as much as 13%. On other systems, OpenMP's load balance is better and the additional Qthread scheduling overhead results typically in a performance loss of less than 5%. Further work is planned to better understand the source of the overheads.

Qthreads is described in more detail in Section 2. It is followed by a discussion of the OpenMP to Qthread translator in Section 3. The results of our experiments with dynamic loop blocking in Qthreads and OpenMP are in Section 4.

2 Qthread

Qthreads [19] is a cross-platform general purpose parallel runtime designed to support lightweight threading and synchronization within a flexible integrated locality framework. Qthreads directly supports programming with light-weight threads and a variety of synchronization methods, including both non-blocking atomic operations and potentially blocking full/empty bit (FEB) operations. The qthread 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.

Only two functions are required for creating threads: `qthread_initialize()`, which initializes the library, and `qthread_fork(func,arg,ret)`, which creates a thread to perform the equivalent of `*ret = func(arg)`. Threads are always associated with—and scheduled in the bailiwick of—a thread mobility domain, or "shepherd," which is assigned when the thread is created. These shepherds are grouping constructs that define immovable regions within the system. Shepherds establish locality within the system, and may correspond to nodes, memory regions, or protection domains. The number of shepherds is defined when the library is initialized. Threads can be spawned directly to a specific shepherd via the `qthread_fork_to(func,arg,ret,shep)` function.

By calling `qthread_migrate_to(shep)`, threads can explicitly migrate between shepherds if necessary. This movement is explicit in order to provide the scheduling guarantee that each thread will not move unexpectedly. This guarantee is important in order to exercise control over the placement of threads and communication between machine locations. The distance between locations can be determined by using the `qthread_distance(src,dest)` function.

Qthread lightweight threads, once created, cannot be directly controlled by other threads, but their status can be monitored via an FEB-protected return value. When a thread is created, its return value location is marked as empty, and when the thread completes, the location is filled. Thus, threads may efficiently wait for other threads to complete.

The scheduler in the Qthread runtime uses a cooperative-multitasking approach. When threads block, such as when performing an FEB operation, a context switch is triggered. Because this context switch is done in user space via function calls and therefore does not require signals or 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 swapping tasks. Logically, this only hides communication latencies that take longer than a context switch.

The Qthread runtime uses a hierarchical threading architecture with pthread-based shepherds to allow multiple threads to run in parallel. The lightweight threads are created in user-space with a small context and small fixed-size stack, and are then executed by the pthread shepherds. For performance, memory is pooled in shepherd-specific structures, to avoid unnecessary communication between shepherds. Much of the qthread library is implemented with lock-free algorithms, including thread queuing, memory pools, and threaded loop tracking. Mutual exclusion regions are used in places and can be sources of contention.

Because computers are finite and have limited resources, threads must deal with those limits. For example, in a 32-bit environment, millions of concurrent threads are not practical: if each of a million threads receives some small amount of thread-specific state, then each thread can have at most four kilobytes of thread-specific state, which would consume all available address space. In recognition of this basic fact, a qthread stack is explicitly limited, and the remaining space can be queried at runtime.

Each lightweight thread is also inherently localized: it exists in a specific place in the system, where "places" are defined by the shepherds. This is quite unlike the fundamental assumptions of most other threading runtimes, which make no guarantees about thread location. It is instead assumed that the operating system or runtime will do something "intelligent". Some system-specific tools, such as Linux's libnuma[7] and Solaris's liblgrp

```
size_t indices[BIG_NUM];
for (size_t i = 0; i < BIG_NUM; i++) {
    indices[i] = i;
}
```

(a) C Loop

```
void func(qthread_t *me, const size_t startat,
          const size_t stopat, void *arg)
{
    size_t *indices = arg;
    for (size_t i = startat; i < stopat; i++) {
        indices[i] = i;
    }
}

size_t indices[BIG_NUM];
qt_loop_balance(0, BIG_NUM, func, indices);
```

(b) Qthread API Equivalent

Figure 1: `qt_loop_balance()` Example

[16], allow thread locations to be specified after the thread is created, but that is an inefficient model, at best. Thread-specific data is allocated when the thread is created; specifying the location after creation forces thread-specific data to be copied to the new location, which is unnecessary when a thread's intended location is specified as a part of the thread creation process.

The Qthread API includes several threaded loop interfaces, built on top of the core threading components. There are four basic parallel loop behaviors: `qt_loop()` spawns a separate thread for each iteration of the specified loop; `qt_loop_balance()` divides the iteration space among all shepherds evenly and spawns a single thread for each shepherd, `qt_loop_queue()` uses a queue-like structure to distribute sub-ranges of the loop's iteration space among all shepherds, and `qt_parallel_for()` distributes sub-ranges of the loop's iteration space to among cooperating shepherds.

The `qt_loop_balance()` function provides a convenient example of the interaction between the library and the application to efficiently distribute work. The application provides the library with the range of data to iterate over as well as a function that can process a range of data. The library then divides the iteration range into partitions based on the

```
#pragma omp parallel
{
     int sum = 0;
#pragma omp for
     for (i = 0; i < ARCHnodes; i++)
sum += Array[i];
     GlobalSum += sum;
}
```

(a) OMP C Loop

```
void func2(qthread_t *me, const size_t startat,
            const size_t stopat, void *arg) {
      int i;
      int *sum = arg[0];
      int *Array = arg[1];
      int *ret = arg[2];
      for (i = startat; i < stopat; i++) *sum += Array[i];
      *ret = sum;
}
void func1(qthread_t *me, const size_t startat,
            const size_t stopat, void *arg) {
     int sum = 0;
     int * Array = arg[];
     int ret = 0;
     void args[3] = {&sum, Array, &ret};
     qt_parallel_for(func1, startat, stopat, args);
     GlobalSum += ret;
}
void arg[2] = {&Array};
qt_parallel_for(func1, 0, qthread_num_shepherds() - 1, arg);
```

(b) Qthread API Equivalent

Figure 2: `qt_parallel_for()` Example

available parallel hardware, and schedules threads for each partition using the function provided by the application. Figure 1 demonstrates the interface. In this case, it is being used to initialize an array in parallel.

One common parallel programming idiom is existing parallel threads sharing the iterations of a parallel loop. OpenMP, for example, has pragmas to create a number (normally equal to available threads) of parallel workers. Each worker does some private initialization and then they join forces to execute one or more parallel loops, also specified by a pragma. To support this idiom, Qthreads requires parallel work be described as a parallel loop equal with iterations equal to the number of shepherds. Each parallel worker uses `qt_parallel_for()` to rendezvous and distribute the work. Figure 2 gives an example of the interface in use.

`qt_parallel_for()` allows worker threads to have a rendezvous point where they can acquire iterations. Work commences as soon as the first worker arrives and completes when all of the iterations are complete. If a worker arrives late it joins the work in progress. The first arrival is responsible for allocating the loop control structure, and releasing it after all of the iterations are completed.

One of the key tuning issues for such a queue-based loop structure is the size of the ranges that are pulled out of the queue every time. Pulling out a single iteration whenever a thread needs work is the most balanced, but adds the most overhead to the loop. Pulling out larger iteration ranges reduces the overhead—because ranges will be fetched less often and cache entries may be shared between adjacent iterations—but increases the potential for load imbalance. Providing Qthreads with dynamic control of thread size and scheduling allows it to balance the benefits and drawbacks according to the situation.

Several loop scheduling algorithms could be used, chunk self-scheduling [8], guided self-scheduling [13] or factoring self-scheduling [6] are all possible. Currently, Qthreads uses a form of guided self-scheduling. To decide the number of iterations to block together, Qthreads assumes that all shepherds are working on this loop. If fewer shepherds arrive, this only increases the number of partitions slightly. The disadvantage of guided self-scheduling is when the threads are short computation of the last threads can be overwhelmed by the scheduling overhead to start them. Qthreads times each iteration and when the previous thread block was too short repeats the previous block size rather than computing a new (yet smaller) block size. For most loops, this can limit the overhead to around 10% in the worst case. All of the iterations must be complete before the participating shepherds are allowed to return from `qt_parallel_for()` to continue execution.

Not only does this mechanism handout loops with low overhead it also allows workers to join a computation after the computation begins. The mechanism should also be effective when parallel loops are inherently imbalanced in unpredictable ways. Iterations are divided into many scheduling units. If a shepherd picks a slow unit, other shepherds will acquire more iterations and the loop should complete without any shepherds waiting undue amounts of time.

3 OpenMP to Qthread Translation

Qthreads is a general purpose parallel runtime able to efficiently support multiple user level parallel programming models. Qthreads directly supports its own programming model with light-weight threads and full/empty bit synchronization. To test the generality of the intermediate layers of the runtime, it was decided to support a second programming model. By choosing a relatively common second model to support, a large number of test programs become available and the overheads of supporting lots of small software threads can be compared against a reasonably tuned implementation.

We chose to support OpenMP[12]. It is a common parallel programming language that uses a relatively small set of extensions to the base language (C, C++ or FORTRAN) to implement it's parallel features. OpenMP has been implemented and tuned for many parallel systems and is generally available though multiple sources including the GNU compiler. The OpenMP programming model has medium-to-heavy weight threads, that can contain multiple parallel loops, serial regions and synchronization between the threads.

Several compiler and translation systems already support OpenMP syntax allowing for faster integration with Qthreads. We chose to support OpenMP using a source-to-source translator. Each OpenMP pragma and function call is detected and converted to the equivalent Qthreads calls. The resulting source file can then be compiled using a standard compiler linking against the Qthreads library.

The Rose compilation system[15, 14, 9] is designed to support parallelization and highly optimizing compiler transformations using a source-to-source mechanism. This allows them to focus on the transformations without spending effort on code generation for each supported architecture. Rose supports several base languages including C, C++ and FOR-

TRAN. The focus on parallel systems results in excellent coverage of the OpenMP extensions for those languages. Rose's parallel transformations resemble the transformations that are required for Qthreads conversion, greatly simplifying the task. The initial OpenMP-to-Qthreads Rose translator effort only supports C. Adding the other base languages should be straightforward once C is completed.

The translation process takes an OpenMP program, identifies every parallel region, and creates an out-of-line call to that region, passing pointers to any data used within the region. We take advantage of the fact that there is a the shared address space between all of the threads, but that is not required for later ports of the Qthreads runtime. Any data declared to be private for a region is replicated within the region. OpenMP function calls are identified and translated to the Qthread equivalent. A Qthread include file is added allowing the translated functions to call Qthread functions and use Qthread data types.

The translation process is not complete, but significant portions are done and simple OpenMP programs can be translated to Qthreads with no additional modifications. OpenMP parallel pragma are supported including the definition of variables private to the region. Each parallel region results in one Qthread thread per shepherd being created. The threads wait for all to complete before leaving the parallel region. The OpenMP parallel `for` pragma by creates a software thread per iteration. Private variables are supported and if found outside a parallel region one is automatically inserted. The `for` loop is pulled out of line and a Qthread `qt_paralell_for` call is inserted. All shepherds active in processing for loop iterations wait for the loop to complete before continuing. OpenMP barrier pragmas are replaced a call to a Qthread barrier. This is either a full/empty barrier or a tree based rendezvous barrier depending on the context and current default setting. OpenMP function calls like `omp_num_threads()` are replaced with the Qthread equivalent `qthread_num_shepherds()`.

Although not yet translating the entire OpenMP standard, the translator is complete enough to allow small OpenMP programs such as the SPEC OMP benchmark `quake` and the OpenMP version of the NAS benchmark `IS` to be translated, compiled, linked against

Qthreads and run correctly.

4 Loop Scheduling

Modern and future systems with tens or hundreds of hardware threads, allow runtimes like Qthreads to have more control of application performance than ever before. For a fixed and relatively low overhead, a portion of the computation can be dedicated to the runtime to examine execution state and adjust the application accordingly. The initial case of watching program behavior and adjusting thread creation in Qthreads is dynamic loop blocking of parallel for loops.

One problem for all parallel runtimes is variable thread length. If the threads are too long, the wait for the last thread to complete is high and load balance suffers. If the threads are too short the overhead of scheduling them is high and overall performance suffers. The translation of OpenMP programs to the Qthreads API generated parallel loops with only one or two instructions in each iteration. The overhead for the Qthreads implementation, when treating each iteration as a separate thread, was very high. A loop iteration chunking mechanism is needed.

The Qthread loop scheduling mechanism allows workers to arrive late, or not all. The flexibility to start loops as soon as one worker is ready, reduces overall execution time and facilitates the handling of nested parallel regions. For ease of programmability in large applications to expose parallelism within libraries, correct efficient nested parallel regions will be a requirement.

Dynamic control of thread creation can, in the future, enable additional optimizations for performance. As computational bottlenecks are replaced by memory, network or file IO bottlenecks the number of threads required to saturate the bottleneck will change. A runtime such as Qthreads could reduce overall scheduling overhead by only using as many threads as necessary to obtain peak performance. By determining threads that share data and co-scheduling, better utilization of shared levels of the memory hierarchy could be achieved.

4.1 Implementation

The conversion from OpenMP to Qthreads is handled by a Rose source-to-source translator. An OpenMP program is input to Rose and several passes over the source are made, first separating the parallel regions from the original source and then converting them to the form required by Qthreads.

The first pass of Rose adds a Qthread include file (with types, function and variable definitions) and uses the OpenMP pragmas to identify any parallel regions or parallel loops. Each parallel region is marked with an internal pragma for later identification.

The second pass uses that internal pragma to identify parallel regions and moves them into separate function calls (using the naming template `OUT_XX_YYYY__` where XX is the parallel function number and YYYY is the global source file number). The source is turned into a function and moved to the end of the file. In its place, a function call and its declaration is inserted. During this pass all OpenMP function calls are replaced by Qthread equivalents (calls or code fragments).

The last pass goes into the newly created functions and rewrites them for Qthreads. A Qthread parallel loop function has a fixed number of parameters, so the parameter list is compressed before the call and expanded within the function. The range parameters of the `for` loop are changed to be parameters of the function call (`__startat` and `__stopat`) allowing the Qthreads scheduler to modify the number of iterations dynamically.

Figure 3 shows a simple example from the SPEC OMP benchmark `quake`. Note that all parameters are passed in as pointers. A private variable is allocate within the new function when required. As an performance optimization, depending on the compiler that eventually compiles the modified source, the pointers may be captured into local variables to allow better optimization within the function. Parallel regions are similar, but use `qt_parallel` instead of `qt_parallel_for`.

```
#pragma omp parallel for private(i)
    for (i = 0; i < nodes; i++) {
      w2[j][i] = 0;
    }
```

(a) OpenMP Loop

```
void *__qthreads_argv8[5] = {0, &w2, &i, &j};
qt_parallel_for(OUT__3__2057__,(nodes - 0) / 1,__qthreads_argv8);


void OUT__3__2057__(void *__qthread_stream,int __startat,
                  int __stopat,void *__qthreads_arg_list)
{
  void **__qthreads_arg;
  __qthreads_arg = ((void **)__qthreads_arg_list);
  int ***w2;
  w2 = ((int ***)(__qthreads_arg[1]));
  int i__priv__;
  int *i;
  i = &i__priv__;
  int *j;
  j = ((int *)(__qthreads_arg[4]));
  for ( *i = __startat;  *i < __stopat; ( *i)++) {
    (( *w2)[ *j])[ *i] = 0;
  }
}
```

(b) Rose Produced Qthread Equivalent

Figure 3: OpenMP-to-Qthread Example

4.2 Results

Various blocking strategies for distributing the thread iterations were used with a Qthreads translated version of `quake`. To reduce scheduling overhead to a small fixed percentage of thread execution, initial fixed sized blocks were produced. Because the amount of work varies in the parallel for loops, there is no one perfect block size for all loops. Figure 4 shows the best execution time of five runs of the SPEC OMP benchmark `quake` on a quad-socket quad-core Dell M905 blade with AMD 2.2GHZ 8354 processors (QB1) for block sizes ranging from 5 to 10,000.

Execution times drop dramatically for all threads counts until the block size is about 100. This implies that the cost of scheduling plus the cost of the `for` loop is equivalent to a small number of "average" loop bodies. Load balancing issues are more severe for small

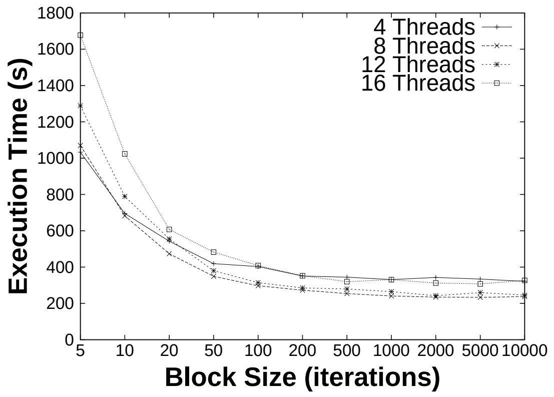


Figure 4: SPEC OMP quake - Static Loop Chunk Size

thread counts (4 threads 54 - 78% slower for large block sizes than 8, 12, or 16 threads). Overhead from scheduling is worse for large thread counts (16 threads take 23% more time than 4 threads when the block size is only 5 iterations). Large block sizes are a good idea for `quake`, in general, however, the scheduler needs to also handle loops with iterations than run for millions of instructions. Block sizes in the thousands will perform poorly for those loops.

The variability of work means that the blocking technique needs to be responsive and dynamic. Guided self-scheduling [13] was chosen to allow bigger blocks to be started when load balancing is likely to be good early in the loop and smaller blocks later when a slow iteration would delay loop termination. The overhead from very small block counts on very small loops, caused the overall performance to suffer. The thread scheduler was modified to hand out the same size block as the previous pass, if the measured execution time was less than approximately 10 times the cost of scheduling an empty loop. By adding a very lightweight timer call built into Qthreads, the overhead can be controlled on a loop-by-loop basis.

Figure 5 compares the performance of this version of guided self-scheduling to the best fixed block size for several threads counts. In general, dynamic blocking performed 2-6%, better than any static block size. Static blocking has very low scheduling overheads. The overhead of dynamic blocking is higher, but in pratice the improved improved load balance

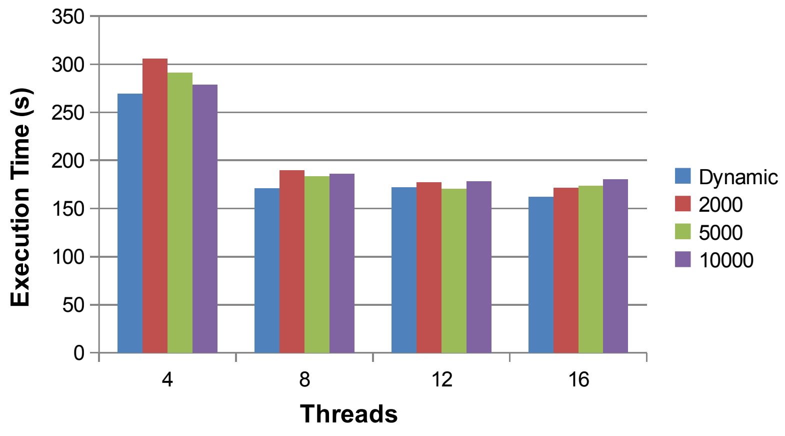


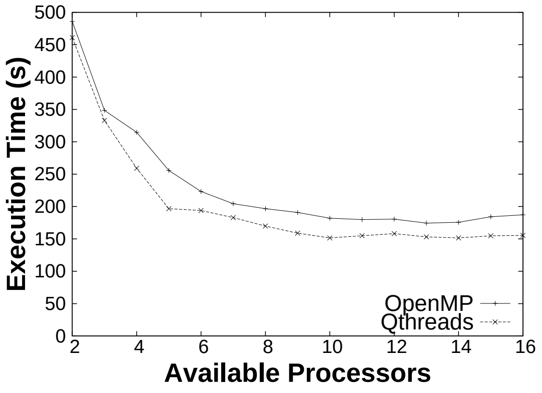
Figure 5: Dynamic vs. Static Blocking in Quake on QB1

lowers overall execution time.

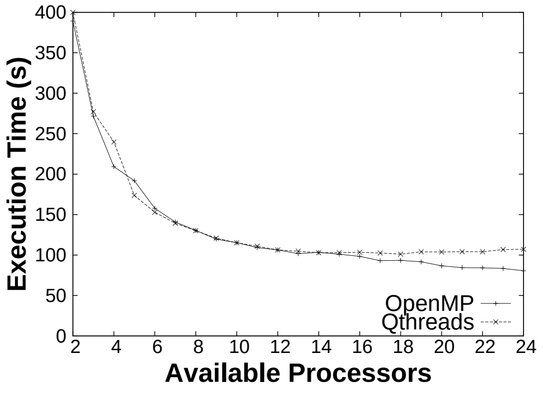
4.2.1 SPEC OMP quake

Comparing to various internal implementations is instructive, but the real test is comparing to a production version of OpenMP running on the same hardware with the same compiler. `quake` was compiled using `gcc -O2` and run on all available thread counts of QB1 in Figure 6(a). Thread counts from 2 to 16 were tested for OMP and Qthreads and in every case Qthreads was faster (normally 11-14%). Even though the loops are well balance and each iteration executes the same number of instructions, Qthreads finds opportunities to improve load balance in excess of it's increased scheduling costs.

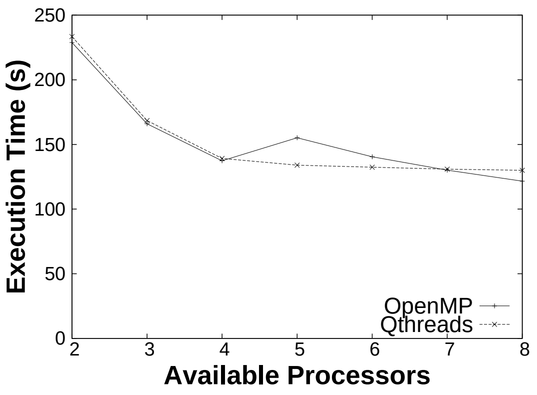
The room for overall performance improvement though load balancing is very system dependent. Results from running the same test on two other systems QB3 - a quad-socket hex-core Dell M905 with 2.1 GHZ AMD 8425 HE processors, and IN1 - a dual-socket quad-core hand built box with 2.93Ghz Intel Core i7 940 processors are shown in figures 6(b) and 6(c). For both systems running with maximum threads OMP outperforms Qthreads. Qthread performance reaches peak performance with less than a full complement of active threads and maintains it while OpenMP continues to improve. Mutual exclusion code in the Qthread thread scheduler may be the cause of Qthreads performance. For low thread counts Qthreads



(a) QB1



(b) QB3



(c) IN1

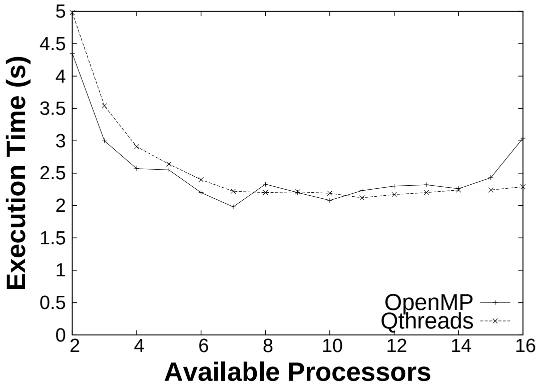
Figure 6: SPEC OMP Quake

performance matches or exceeds OMP.

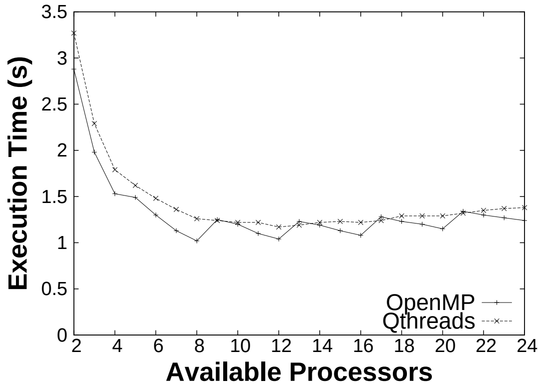
4.2.2 NAS IS

Qthread performance was tested against OMP on a second common OMP C benchmark, NAS NPB IS. IS in the size tested (CLASS B) is an integer sort of 2^{25} elements. For the systems tested, the parallel regions are small and the work is balanced. Figures 7(a), 7(b) and 7(c) compare the performance of Qthreads and OMP for all thread counts.

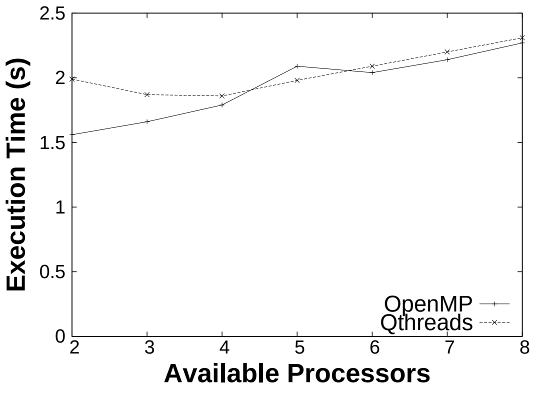
Overall for IS, Qthreads performance relative to OMP is slightly worse than it was for quake. The small parallel region (and short execution times) make it hard for any system to maximize performance, and Qthreads scheduling overhead adds a little overhead. For QB1, Qthreads outperformed OMP with more than 10 threads active. For QB3, depending on



(a) QB1



(b) QB3



(c) IN1

Figure 7: NAS IS

the thread count performance ranged from 12% worse to 3% better. Performance for both runtime systems on IN1 was unusual. Performance degraded as more threads were active. The cost of scheduling and the barriers to guarantee completion increase execution time for both. Qthreads scheduling overhead is noticeable with only 2 or 3 threads, but is hidden as the thread count increases

The IS graphs, particularly QB3, point out one possibly significant advantage of Qthreads loop scheduling over OMP. As workers are added, or subtracted performance is relatively constant. Qthreads just moves the work to available shepherds. OMP's fixed distribution does not work as well when the threads are unevenly distributed across sockets. Contention for shared hardware resources actually slows down processors with more active cores. With OMP, this appears as the jagged performance line on QB3. Qthreads dynamic distribution will also allow it to provide smoother and better performance in the presence of OS jitter. Moving the work to a core/socket that the OS is not stealing, allows better dynamic load balancing.

Qthreads loop scheduling provides more stable performance without adding significant overhead for balanced parallel loops. This allows it to perform as well as or better than OMP for several standard benchmarks, while supporting multiple programming models.

5 Implications and Further Related Work

Having a runtime, like Qthreads, that supports multiple parallel programming models efficiently, is a step toward real model-based comparisons on a common runtime. When programs use completely different implementations, it is hard to separate the performance efficiency of the program from that of the system's implementation of the model. By supporting multiple programming models with a single runtime, Qthreads and the OpenMP translator should allow better comparisons of programs written in different parallel programming models than is typically available.

Very light-weight ubiquitous threads predate Qthreads. A line of systems leading to the Cray XMT[3], all support a version of this threading model. Being an integrated hardware/software solution, the systems take advantage of specialized hardware features not available on commodity x86 hardware (like single instruction thread creation, and FEBs on every word of memory). Qthreads attempts to provide XMT-like programming features on commodity hardware, while supporting multiple parallel programming models.

Handling loops efficiently is a requirement for every parallel runtime. Loop scheduling mechanisms were extensively researched in the 1990's. Chunk self-scheduling[8], guided self-scheduling[13] and factoring self-scheduling[6] are the standard examples. Qthreads uses current light-weight hardware timing mechanisms to implement slightly modified guided self-scheduling. Cilk[1] is a programming language (and runtime) that addresses many of the issues with scheduling over-partitioned loops on parallel hardware. Qthreads runtime is designed to support multple existing parallel programming models in addition to its own.

Compiler and library autotuning research has generated substantial speedup for applications and libraries (ATLAS[18],FFTW[5]) by doing sweeps over compiler and program options to locate the best options. The CHiLL[17] transformation framework generates alternate implementations and quickly finds the best performing version. These projects are, by design, static and tailored to a specific system, a specific input, and a specific load level. By delaying the blocking decision to runtime, Qthreads can react to differences in the input data and load on the system.

6 Conclusions

As computer systems and applications become larger and more complicated even at the single node level, getting good performance out of the system will be increasingly difficult. For many of the applications and systems, any static layout will produce load imbalances during portions of execution. To support dynamic programs on highly multi-threads systems of tomorrow, flexible runtimes will be needed to dynamically map the changing application to the available hardware.

Qthreads allows the application to use an abstract machine model to simplify programming. The performance cost of this more abstract model at runtime is minimal and often covered by the improved load balancing between hardware threads, allowing it to outperform current parallel runtimes on specific programs and systems. Qthreads loop performance is generated by over-partitioning hardware resources and allowing hardware resources to work when ready. Dynamic loop scheduling for load balance and interactive loop chunking to control overhead is critical to the performance.

Once the runtime has a mechanism to control an application to adjust it to changing conditions, it can be applied to several dynamic optimization problems. 1) Threads which share data can be co-scheduled to take advantage of share levels of the memory hierarchy to improve individual thread performance. 2) Threads can be limited to prevent over subscription of shared resources such as the network or file IO. 3) Threads and memory can be moved to take advantage of heterogeneous accelerators such as GPGPUs.

Qthreads has shown that a light-weight threading environment can match the performance of conventional parallel programming models for balanced parallel loops. For unbalanced loops, guided self scheduling should be substantially better than any blocked or round-robin technique. Qthreads is a attractive vehicle for parallel runtime and language research.

References

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[2] Intel Corporation. Intel spotlights new extreme edition processor, software developer resources at game conference. Press Release, March 2010.

[3] Cray XMT platform. http://www.cray.com/products/xmt/index.html, October 2007.

[4] Advanced Micro Devices. What would you do with 48 cores? Press Release, March 2010.

[5] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[6] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.

[7] Andi Kleen. An NUMA API for Linux. http://halobates.de/numaapi3.pdf, August 2004.

[8] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik. Locality and loop scheduling on numa multiprocessors. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 140–147, Washington, DC, USA, 1993. IEEE Computer Society.

[9] Chunhua Liao, Daniel J. Quinlan, Richard Vuduc, and Thomas Panas. Effective source-to-source outlining to support whole program empirical optimization. In *LCPC'09: 22th Annual Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer Verlag, 2010.

[10] International Business Machines. IBM unveils new POWER7 systems to manage increasingly data-intensive services. Press Release, February 2010.

[11] Anirban Mandal, Rob Fowler, and Allan Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *ISPASS'10: International Symposium on Performance Analysis of Systems and Software*, White Plains, NY, March 2010.

[12] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 2.5 edition, May 2008.

[13] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):14425–1439, 1987.

[14] Daniel J. Quinlan, Markus Schordan, Qing Yi, and Bronis de Supinski. A C++ infrastructure for automatic introduction and translation of OpenMP directives. In *WOMPAT'03: Workshop on OpenMP Applications and Tools*, volume 2716 of *Lecture Notes in Computer Science*, pages 13–25. Springer Verlag, June 2003.

[15] Markus Schordan and Daniel J. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, August 2003.

[16] Sun Microsystems, Inc., Santa Clara, CA. *Memory and Thread Placement Optimization Developer's Guide*, June 2007.

[17] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary W. Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS*, pages 1–12. IEEE, 2009.

[18] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001.

[19] Kyle Wheeler, Richard Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *MTAAP'08: Workshop on Multi-Threaded Architectures and Applications*, Miami, Florida, USA, April 2008.