SANDIA REPORT

SAND2012-10594
Unlimited Release
Printed December 2012

A Comparative Critical Analysis of Modern Task-Parallel Runtimes

Kyle Wheeler, Dylan Stark, Richard Murphy

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2012-10594
Unlimited Release
Printed December 2012

A Comparative Critical Analysis of Modern Task-Parallel Runtimes

Kyle Wheeler
Dylan Stark
Scalable System Software Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
{kbwheel, dstark}@sandia.gov

Richard Murphy
Micron Technology, Inc.
Mailstop 1-407
8000 S. Federal Way
P.O. Box 6
Boise, ID 83707-0006

Abstract

The rise in node-level parallelism has increased interest in task-based parallel runtimes for a wide array of application areas. Applications have a wide variety of task spawning patterns which frequently change during the course of application execution, based on the algorithm or solver kernel in use. Task scheduling and load balance regimes, however, are often highly optimized for specific patterns. This paper uses four basic task spawning patterns to quantify the impact of specific scheduling policy decisions on execution time. We compare the behavior of six publicly available tasking runtimes: Intel Cilk, Intel Threading Building Blocks (TBB), Intel OpenMP, GCC OpenMP, Qthreads, and High Performance ParalleX (HPX). With the exception of Qthreads, the runtimes prove to have schedulers that are highly sensitive to application structure. No runtime is able to provide the best performance in all cases, and those that do provide the best performance in some cases, unfortunately, provide extremely poor performance when application structure does not match the scheduler's assumptions.

Acknowledgments

Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Company, for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Contents

Chapter 1

Introduction

The traditional presumption for how to deal with supercomputers with a high degree of node-level parallelism is "MPI + X", where X is some node-level thread-based execution environment. For example, "MPI + OpenMP" refers to a message-passing (and typically BSP-oriented) execution model enhanced with OpenMP-style parallel loops and tasks. Recent interest in the use of novel execution models for high performance computing [6, 10] has suggested an increased level of importance for node-level parallelism, and a merging of the node- and system-level parallel models. Such efforts have led to the creation of runtime system software meant to embody those models by serving as an emulation facility on today's platforms. Both the traditional and novel approaches agree that the success of building a full-scale distributed memory system parallel runtime will be determined by its ability to leverage some form of a node-level, shared memory, threaded runtime. Performance on today's hardware must be the basis for an effective evaluation of the potential of future parallel models. Analysis of runtime performance on existing platforms enables the development of hardware features designed to alleviate the bottlenecks of these runtimes. Thus both the absolute performance of the model and the scalability of its core constructs are critical points of evaluation.

In this work, we examine six publicly available parallel tasking runtimes: Intel Cilk [2], Intel Threading Building Blocks (TBB) [12], Intel and GCC implementations of OpenMP [16], Qthreads [20], and High Performance ParalleX (HPX) [19]. These runtimes are typical node-level, shared memory, threaded runtimes exhibiting the basic characteristics of task-based parallelism. Cilk, TBB, and OpenMP are examples of industry-supported runtimes that are optimized around existing platforms. The Qthreads and HPX runtimes are characteristic of experimental systems intended as foundations for exploring future potential platforms with enhanced hardware capabilities. Both categories of runtime systems are represented here because each has an important role in determining the state of the art as well as paths forward.

This paper examines the common requirements of a task-parallel system in the context of four microbenchmarks that each expose a different application-level task spawning pattern:

Serial Spawn the most basic spawning pattern in which a single task spawns many children one at a time – typical loop control construct (e.g., *for* or *while*) is used in combination with a simple task spawn operation.

Parallel Spawn a more advanced spawning operation in which the runtime system (as a whole) spawns many children in parallel. This pattern is often used by more advanced loop-parallel programming environments to reduce load balancing overhead and improve spawn times.

Fibonacci a structured series of spawns with inter-spawn dependencies and synchronization required (requiring the underlying tasking system to manage many "waiting" requests). It provides a basic well-balanced spawn-tree pattern, common among many parallel-recursive algorithms, such as divide-and-conquer algorithms, and spawn/sync parallel programming environments, such as Cilk.

Unbalanced Tree Search (UTS) a highly unstructured and unbalanced set of spawn requests with a large amount of synchronization. This is a spawn tree that is highly irregular and requires a great deal of load-time load rebalancing. This pattern is similar to input-dependent parallel-recursive algorithms such as graph traversal algorithms.

These four benchmarks were chosen for several reasons. First, they are simple, which makes comparison between such different tasking environments much more straightforward. Second, they represent a reasonable selection of task

spawning patterns used in task-parallel code, specifically loop-level parallelism and recursive tree-based parallelism. These benchmarks have little to no computation in the tasks that are scheduled, which serves several purposes. Runtime overhead can take several forms: "hard" overhead, which is the time it takes to perform specific actions, and "soft" overhead, which is the effect of runtime behavior on actual work, often referred to as "work time inflation." This paper focuses on "hard" overhead, and the lack of computation highlights the differences between runtimes in specific behaviors, which more complex computation, resource consumption, and memory references would serve to mask. This "hard" overhead is what drives decisions about task granularity, and as such understanding this overhead is crucial to designing applications to leverage such runtimes.

By examining these benchmarks, this paper examines the state of the art in threaded runtime task creation and management and provides a basis for evaluating the utility of these platforms in the exploration and development of future runtime systems. Multi-node performance is not examined, since most runtimes lack support for multi-node execution. However, the results of this paper provide a baseline of comparison for future multi-node tests.

The contribution of this paper is the comparison of multiple task-parallel runtimes across a range of task spawning and synchronization behaviors. One would expect to find that most runtimes are good at some problems and relatively bad at others. What is surprising, however, is the magnitude of the worst-case scenarios. In almost all cases, if a tasking runtime is the fastest at one benchmark, it is the worst by a significant margin at another benchmark. This has important implications not only for algorithm selection and design, but for the ability of parallel algorithms to be composed. Future studies will attempt to examine multiple runtimes for more specific applications, but this initial characterization provides a basis for differentiating the runtimes based upon determining the task behavior they are designed to handle efficiently. More generally, the contribution of this paper is the demonstration that few task-parallel runtimes are truly general-purpose; most cannot deal well with behaviors that do not match the assumptions of the runtime. This comparison enables software programmers to choose the runtime most appropriate for their application, or to choose algorithms most appropriate for their runtime, and to guide future research and development in runtime systems.

The remainder of this paper is organized as follows: Section 2 discusses related work; Section 3 describes the experimental setup; Section 4 presents our experimental results; Section 5 analyzes those results; and Section 6 draws together the major conclusions from this work.

Chapter 2

Related Work

Task-parallel programs generate a dynamically unfolding set of interdependent tasks. Efficiently scheduling tasks with a fully arbitrary set of inter-task dependencies is an NP-hard problem, requiring the use of heuristics in practice. As a result, typically either restrictions are placed on task dependencies or tasks are split into two tasks at communication and other synchronization boundaries such that the tasks can be modeled as a directed acyclic graph (DAG) [3, 5, 11, 8].

Scheduling policies for task DAGs have been well studied. Blumofe et al. proved that work stealing is an optimal multithreaded scheduling policy for DAGs that has minimal overheads [4], and Cilk [2] became the first runtime to use that mechanism. Some of the limitations of a pure work stealing approach in modern machine topologies, such as the general lack of topology awareness and cache-affinity, have been explored as well [15]. Intel Threading Building Blocks presents a limited form of work stealing known as depth-restricted work stealing [17, 18].

The effectiveness of a variety of DAG schedulers has also been examined at a theoretical level. Lutz et al. [13] looked at the tradeoff between communication and computation in DAG scheduling, proving that while the lower bounds of scheduling DAGs are tight, there are alternative valid schedules that are ineffective at limiting communication and as such would have significant performance problems. Alternatively, Blelloch et al. [1] examined the tradeoff between space and computation in DAG scheduling, again demonstrating that some schedules are highly effective and some are not. If, as is often the case, memory allocation is slow or memory space is limited, scheduling without awareness of the memory requirements can result in significant performance penalties.

What has not been studied is the effectiveness of the implementations of task-DAG schedulers. While static scheduling of fixed DAGs can prove highly advantageous, the DAG for a given program typically cannot be determined before execution; e.g., for emerging important graph application areas such as informatics. Further, execution time is highly dependent upon implementation details, such as the overheads of synchronization and load-balancing queue implementation. Non-deterministic issues such as queue contention are often ignored in mathematical models of scheduling policies, yet have a significant impact upon overall performance.

Chapter 3

Experimental Setup

The evaluation was conducted on a four-socket 32-core Intel Nehalem X7550 machine. Each socket has eight cores, each core has 32K of L1 cache, 256K of L2 cache and shared 18MB of L3 cache. Each socket has 128GB of dedicated memory for a total of 512GB DRAM. Hyper-threading was disabled. The machine ran CentOS Linux, kernel version 2.6.32-220.7.1.el6.centos.plus.x86_64. This testbed was chosen for its compatibility with the Intel runtimes, its relatively high core count, and its non-uniform memory topology, as these characteristics are likely to be representative of future commodity systems.

3.1 Runtimes

We tested six task-parallel runtimes: Intel Cilk Plus 12.1.0 [2], Intel Threading Building Blocks (TBB) 4.0 [12], Intel OpenMP 12.1.0, GCC OpenMP 4.7.0 [16], HPX 0.9 [19], and Qthreads 1.8 [20]. All runtimes were compiled with ICC 12.1.0, with the exception of GCC OpenMP and HPX, both of which require the GNU compiler and were compiled with GCC 4.7.0.

For each fixed-size problem, the number of utilized cores was scaled from 1 to 32. The default processor pinning mechanism was used for Cilk, GCC OpenMP, HPX, Intel OpenMP, and TBB. The Qthreads runs were manually pinned such that one work queue was associated with each utilized core, so that the CPU affinity resembled that of the other runtimes, to eliminate affinity differences as a cause of performance differences.

The runtimes can be divided into two categories: three compiler-based runtimes and three library-based runtimes.

OpenMP was chosen because it has become a cross-vendor way of expressing task-based parallelism, as of the release of the OpenMP 3.0 spec. The results for Intel and GCC OpenMP demonstrate that the implementation can be far more important for performance than the language. Cilk, a spawn/sync extension to C, was chosen as the historic standard-bearer for low overhead work-stealing runtimes. Each compiler-based runtime technically has an advantage in that it can recognize null-tasks or create special-case tasks to be used under specific conditions, though it still often relies on a runtime library of some kind under the covers.

Library-based runtimes can provide compiler-independence, but typically force programmers to reorganize their parallelism in a way that fits the library's API. Intel TBB was chosen as a production-quality library-based parallel tasking runtime. The other two library-based runtimes are research platforms. High Performance ParalleX (HPX) is a runtime developed at Louisiana State University under the DARPA UHPC program. It is intended to be a high-performance implementation of the ParalleX execution model [10]. The ParalleX model is intended to be a revolutionary execution model, based on parallel tasks and abstract synchronization in the form of local control objects (LCOs), all of which are globally named objects. Qthreads [20] is a lightweight tasking runtime developed by Sandia National Laboratories to serve as a compiler target for task-based languages, such as OpenMP and Chapel. In this paper, the benchmarks were written directly to the Qthreads API, rather than using OpenMP or Chapel as a frontend.

Due to the flexibility of the Qthreads scheduler system, we were able to investigate variations on scheduling policy to see their impact on performance. For simplicity, a single-worker-per-queue regime was used in all tests. Two

variations on load-balancing policy within Qthreads are examined: Nemesis, Sherwood. The Nemesis scheduling policy uses exclusively spawn-time load balancing: spawns from each worker thread are spread across the other worker threads in a round-robin fashion. The queue implementation is the NEMESIS queue from MPICH2 [7]. The Sherwood scheduling policy [15] uses work-stealing but steals half of the victim's queue in a single steal operation. This policy is intended to mitigate queue contention issues common to programs with large numbers of small tasks.

3.2 Benchmarks

The benchmarks chosen to test these runtimes exhibit low levels of real work, so that their scheduling impacts are demonstrated plainly. While task scheduling has a real impact on performance in codes with coarse-grain tasks – for instance in the amount of cache affinity it can maintain between tasks as well as the amount of load balance it can maintain among participants in barrier operations – these benchmarks are primarily tests of the ability of these runtimes to balance task creation and synchronization load. As such, the benchmarks were chosen to demonstrate a variety of different kinds of task spawn and synchronization load imbalance. For instance, the OpenMP versions use only untied tasks and no task coarsening options (such as mergeable or final) in order to compare the runtimes' ability to handle large numbers of moveable tasks.

3.2.1 Serial Spawn

The Serial Spawn benchmark simply spawns 2^{20} null tasks and runs all of them to completion. The tasks are spawned by a single task using a standard serial loop, a pattern common to many loop-parallel programming environments. Depending on the memory allocation scheme used, task spawning can be the primary determinant of performance in this benchmark. This benchmark provides a 1-to-N load imbalance, and requires N-to-1 synchronization. Depending on the implementation, it may also provide pressure on the serial memory allocator performance, particularly in contention with parallel memory deallocation.

3.2.2 Parallel Spawn

The Parallel Spawn benchmark is a progression from the Serial Spawn benchmark. Rather than use a single source to spawn all 2^{20} null tasks, the tasks are spawned in parallel, from a parallel loop construct. The parallel loop construct itself may use a Serial Spawn mechanism, and often does in many loop-parallel programming environments, but the details of such a construct are not restricted by this test. In the OpenMP implementation of this benchmark, a `#pragma omp parallel for` loop is used to spawn the tasks. Cilk uses a `_Cilk_for` loop to spawn the tasks. TBB uses a `parallel_for` function call, and Qthreads uses the `qt_loop` construct. HPX, unfortunately, does not have a parallel loop construct, and so cannot compete in this benchmark.

The use of a parallel loop construct allows the runtime system to better manage the 1-to-N load imbalance, but still requires the N-to-1 synchronization. Thus, it can be viewed as a concurrency test: the tasks can be spawned in a balanced manner, negating the liability of aggressive load balancing during parallel loop start. Further, the memory footprint of this benchmark depends heavily upon whether the tasks execute concurrently or if the null tasks must wait for a large number of them to be spawned first.

3.2.3 Fibonacci

The Fibonacci benchmark naively computes the 30th number in the Fibonacci series; i.e., the task computing the ith Fibonacci number, F_i, spawns task to compute F_{i-1} and F_{i-2} and returns the sum of the results of the two tasks. Specifically, while tasks can be spawned in parallel, the tasks must now behave in a level-synchronized manner. This

well-balanced level-synchronized spawn tree pattern is common among many parallel-recursive algorithms and spawn/sync parallel programming environments.

Because each task spawns two tasks and waits for them, this places high demands on the performance of the tasking runtime's synchronization mechanisms. Further, because an entire depth-first tree must be instantiated at some point during computation, the benchmark requires a much larger memory footprint than the previous two benchmarks. Due to the synchronization-enforced dependency tree, performance in this benchmark is highly sensitive to task execution order: the right order means that tasks will almost never need to perform an expensive synchronization operation, while the wrong order may require almost every task to wait. While there is a great deal of parallelism in this benchmark, load balance is a particular challenge due to the performance-sensitive nature of the execution order. However, due to the tree structure of the dependency graph, load-balancing operations are generally performed only at the beginning and end of the benchmark's execution: the two points where the number of available parallel tasks is relatively low.

3.2.4 Unbalanced Tree Search

The Unbalanced Tree Search (UTS) benchmark [14] is similar in many ways to the Fibonacci benchmark: it also uses a recursive tree algorithm to spawn tasks. These random task trees characterize highly unpredictable and difficult to load/balance medium- to fine-grain tasking workloads. The load per task varies, as each parent task is required to run a random number generator a random number times, and each parent task can spawn a random number of children, while leaf nodes are essentially null tasks. The exact structure of the task tree is governed by a set of parameters discussed in [14]; we used the *T3* data set[1] which builds a task tree with approximately 4 million nodes.

Much like Fibonacci, synchronization performance is key to performance in the UTS benchmark. Similarly, load balance – in terms of actual work, not just the number of tasks executed – is also critical to performance, as the benchmark is designed to generate a great deal of imbalance. As a result, ironically, this imbalance tends help by relieving contention-based pressure on the memory allocation algorithms. Unlike Fibonacci, each task represents an unpredictable amount of work, which presents a real challenge for runtimes that assume that tasks spawned earlier in the execution necessarily represent more work.

[1] The specific parameters for the *T3* data set are defined in the UTS code repository available at http://sourceforge.net/p/uts-benchmark. They are: a binomial tree with a root branch-factor of 2000, a probability of spawning a non-leaf node of 0.124875, a non-leaf branch-factor of 8, and a random seed of 42.

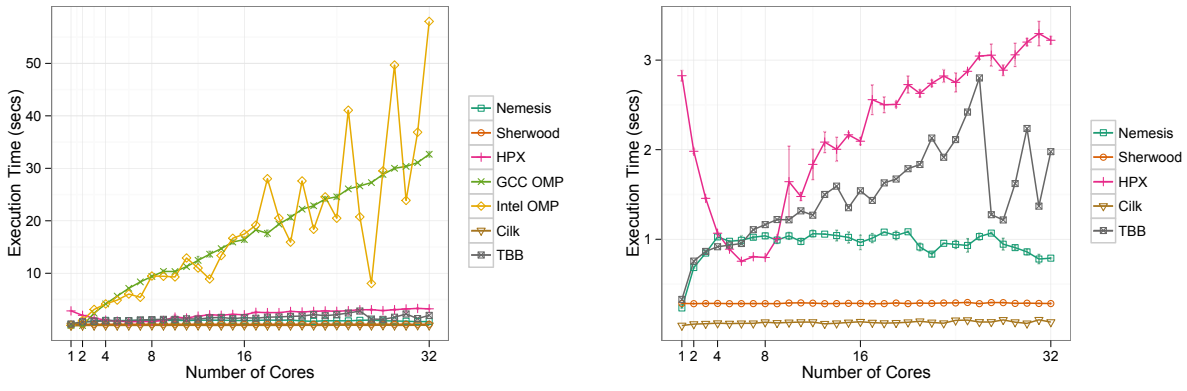
Chapter 4

Results

Sections 4.1 through 4.4 present raw performance and scaling trends based on execution time for the different runtimes, per benchmark. Each data point is the arithmetic mean of three non-sequential samples.

4.1 Serial Spawn

Figure 4.1 presents strong scaling results for the Serial Spawn benchmark discussed in Section 3.2.1. The GCC OpenMP and Intel OpenMP results show an increase in execution time roughly proportional to the increase in the number of workers. The Intel OpenMP results have significant performance swings which approximately correspond to socket boundaries, but with the same overall trend as the GCC OpenMP results, suggesting similar algorithms with more cross-socket communication in Intel OpenMP. With those two runtimes removed (Figure 4.1) the HPX and TBB runtimes also show a strong correlation between worker count and increased execution time. HPX is able to reduce execution time for workers up to approximately the size of a socket (8 workers), suggesting that cross-socket communication is a significant problem at scale. Nemesis Qthreads shows an inverse relationship with respect to worker count and execution time: performance degrades until the socket boundary is reached. This suggests that Nemesis Qthreads is particularly hard on cache contents, and that once enough cache can be obtained, by using some from neighboring sockets, performance stabilizes. Execution time for Cilk and Sherwood Qthreads is largely invariant with respect to the number of cores, which suggests that the majority of the overhead is in the spawn operation, since it is the portion of the benchmark that does not vary with the number of cores. One obvious conclusion to draw from these results is that these two OpenMP runtimes are not good at serial task spawning.



(a) *All Runtimes*

(b) *Excluding GCC and Intel OpenMP.*

Figure 4.1. Execution time for the Serial Spawn benchmark with $t = 2^{20}$.

4.2 Parallel Spawn

Figure 4.2 presents strong scaling results for the Parallel Spawn benchmark discussed in Section 3.2.2. HPX numbers for this benchmark are not available, because the runtime does not provide a built-in parallel loop construct. The two Qthreads scheduling options have poor performance for low numbers of workers, but Sherwood Qthreads provides highly competitive performance once the socket boundary is crossed. This suggests that the overheads involved, though clearly not minimal, are ones that are either amenable to parallel speedup or result from excessive cache use. Nemesis Qthreads and TBB are the worst performing for this benchmark relative to the other runtimes, with TBB having better execution time for low worker counts and worse time for higher counts, and Nemesis Qthreads showing the opposite behavior. Cilk, GCC OpenMP, and Intel OpenMP are the three best performers, showing the best execution time consistently across all core counts; though at scale, Cilk, GCC OpenMP, and Sherwood Qthreads are the three best performers.

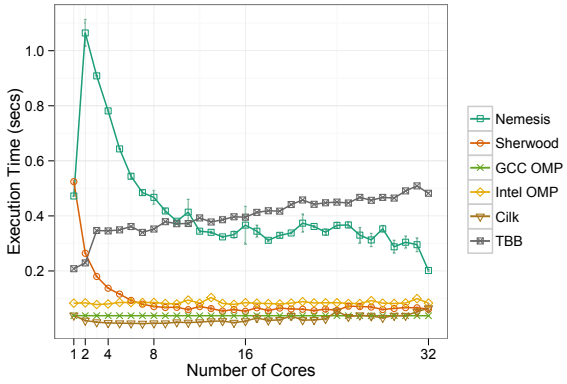
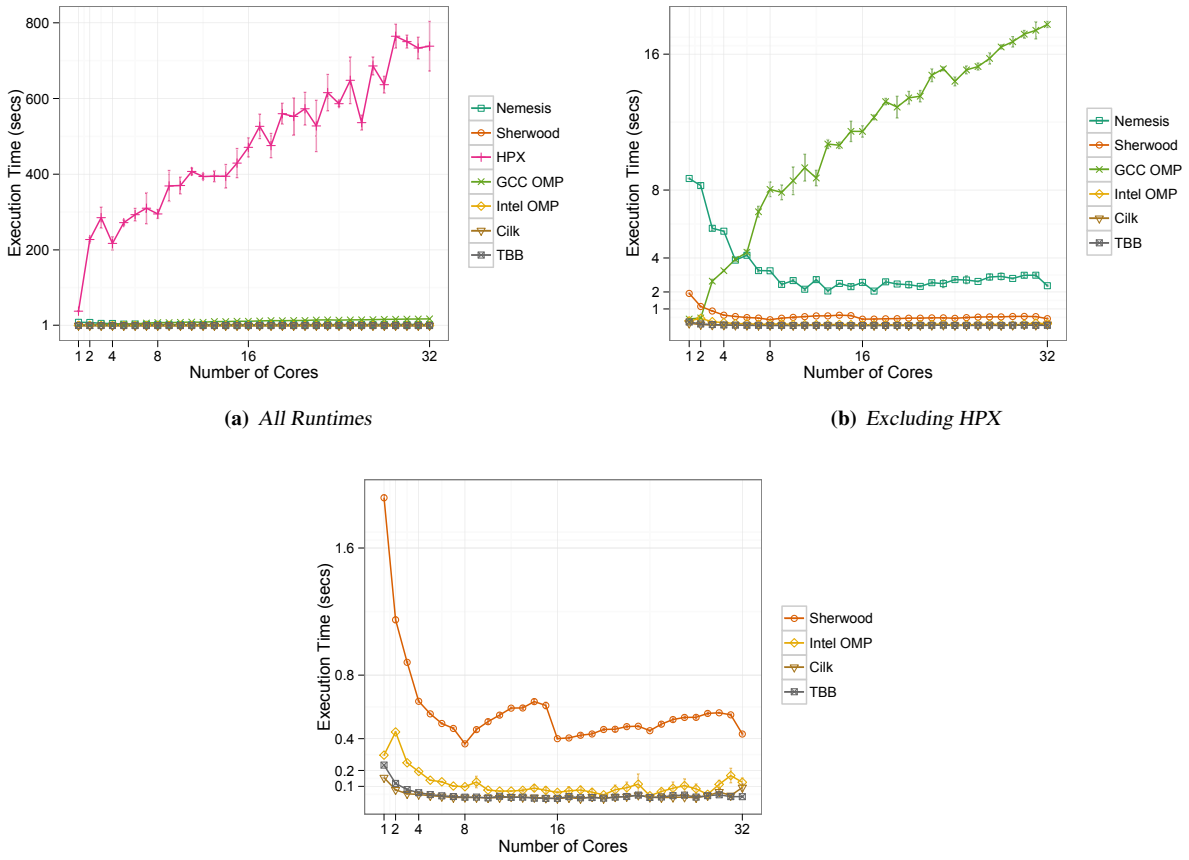


Figure 4.2. Execution time for the Parallel Spawn benchmark with $t = 2^{20}$ for all runtimes.

4.3 Fibonacci

Figure 4.3 presents strong scaling results for the Fibonacci benchmark discussed in Section 3.2.3. In Figure 4.3(a), the HPX runtime shows a strong correlation between increased number of workers and increased execution time. Its single-threaded performance does not compare favorably, at roughly 3× the execution time of the next-slowest runtime, and the problem gets worse as more resources are used, at roughly 295× the execution of the next-slowest runtime at full scale. The HPX implementation of Fibonacci uses a futures construct in order to asynchronously spawn tasks and later wait on their results. This is considered one of HPX's fundamental constructs, so it is unfortunate that it does not perform very well. Removing the HPX results (Figure 4.3(b)) shows that GCC OpenMP follows a similar trend, though not as severe in terms of execution time. Nemesis Qthreads is also not in the same league as the rest of the schedulers, though it does at least provide modest scaling. Figure 4.3(c) presents the results with Nemesis Qthreads, GCC OpenMP, and HPX results removed. Sherwood Qthreads again shows relatively poor performance for low core counts, but strong performance for medium to high core counts, with a distinct performance improvement as additional cache is added by crossing the second socket boundary. Cilk, Intel OpenMP, and TBB are the three best performers, showing the best execution time across increasing numbers of workers. It is likely that TBB's depth-limited scheduling is particularly beneficial on this benchmark.



(a) *All Runtimes*

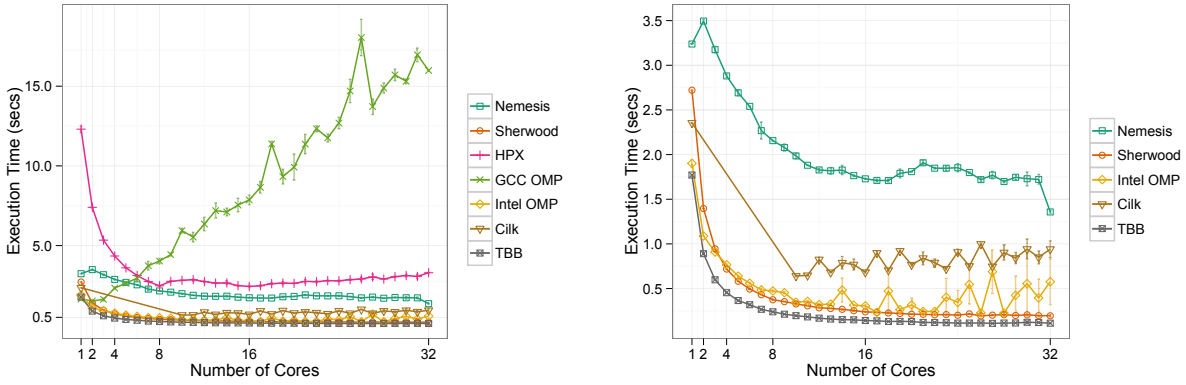
(b) *Excluding HPX*

(c) *Excluding HPX, GCC OpenMP, and Nemesis Qthreads.*

Figure 4.3. Execution time for the Fibonacci benchmark with $n = 30$.

4.4 Unbalanced Tree Search

Figure 4.4 presents strong scaling results for the UTS benchmark discussed in Section 3.2.4. In Figure 4.4(a), GCC OpenMP shows a correlation between an increased number of workers and increased execution time. HPX shows initial speedup up to 8 cores, after which performance degrades as additional workers are added. Note that the implementation of UTS in HPX does not use their futures construct, and instead uses a global atomic counter for termination detection. The reason for this is that, despite the contention overhead inherent in a global atomic counter, the futures implementation was several orders of magnitude slower. Removing GCC OpenMP and HPX results (Figure 4.4(b)) provides a clearer view of the faster runtimes. Nemesis Qthreads shows relatively low performance across core counts, but does demonstrate modest scaling. Cilk shows reasonable performance for medium to high core counts, though performance does degrade with increased core counts. We were unable to collect Cilk results for 2–8 cores due to their stack-based task allocation, which ran out of space and crashed. This is not a fundamental limitation of the Cilk model, but is instead a limitation of the implementation of their runtime. Intel OpenMP, Sherwood Qthreads, and TBB are the best performing runtimes for this benchmark, though TBB has a noticeable performance advantage across all core counts. Like the Fibonacci benchmark, TBB's performance advantage is likely a result of its depth-limited load balancing.



(a) *All Runtimes*

(b) *Excluding GCC OpenMP and HPX*

Figure 4.4. Execution time for the UTS benchmark with the T3 tree for all runtimes.

Chapter 5

Analysis

5.1 High-Level

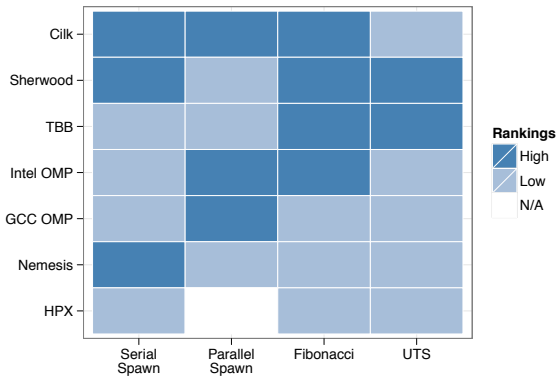


Figure 5.1. Relative ranked performance for each runtime across each benchmark. The data is row-ordered by the number of "High" blocks.

The first observation to make is that each runtime has different strengths and weaknesses. To demonstrate that, we can classify the runtimes as "high performance" and "low performance" for each microbenchmark. We use the Serial Spawn benchmark to separate the runtimes by designating those that have a roughly constant execution time independent of parallelism "High" – Cilk, Nemesis Qthreads, Sherwood Qthreads – and those whose execution time increases with additional processors "Low" – TBB, HPX, Intel OpenMP, and GCC OpenMP. The Parallel Spawn benchmark is used to separate the runtimes by designating those that keep their execution time below 0.2 seconds at all scales "High" – Cilk, GCC OpenMP, Intel OpenMP – and those that don't "Low" – TBB, HPX, and all forms of Qthreads. The Fibonacci benchmark differentiates the runtimes by designating those that keep their execution time below 2 seconds "High" – Sherwood Qthreads, Cilk, TBB, and Intel OpenMP – and those that take more time "Low" – Nemesis Qthreads, GCC OpenMP, and HPX. The UTS benchmark is used to separate runtimes by designating those that have a full-scale execution time of less than a half second "High" – TBB, Sherwood Qthreads – and those that take more time "Low" – Intel OpenMP, Cilk, GCC OpenMP, HPX, and Nemesis Qthreads. Figure 5.1 summarizes these binary categorizations, and demonstrates pictorially that no runtimes are in the "High" category for all benchmarks.

The choice of cutoff between the two classes classes for each benchmark is necessarily subjective, but the classifications are nevertheless instructive. Usually there are fairly significant divisions, but some do miss the cutoff by a relatively small amount. Intel OpenMP is almost a high performer on UTS, but seems to be struggling with some load balancing issues at high core counts. Similarly, Sherwood Qthreads is almost a high performer on the Parallel Spawn benchmark because it is *usually* below the cutoff, but has some overhead issues with very low core counts and some load balancing interference with very high core counts. Yet, in almost all cases, when a runtime is in the "Low"

category, it has missed the cutoff significantly. For these runtimes (with the potential exception of Sherwood Qthreads), using the wrong design pattern comes with a significant performance penalty.

For instance, TBB is the best performer on the heavily unbalanced UTS spawn tree, and is extremely good at the Fibonacci spawn tree. However, when the spawn pattern is not tree-based, such as the Serial and Parallel Spawn benchmarks, TBB seems to suffer from parallelism rather than benefit from it: it has the worst at-scale performance of any runtime on the Parallel Spawn benchmark. Cilk is one of the best performers at both the Serial and Parallel Spawn benchmarks, both in terms of absolute performance and in terms of scalability, as well as the Fibonacci benchmark. But it cannot even always run the UTS benchmark – it runs out of stack space – and appears to struggle with the additional computational resources, rather than benefit from them. With Cilk, the best number of cores to use is the fewest number (greater than one) that will actually run the benchmark to completion.

The most reliably performant runtime is Sherwood Qthreads (the default Qthread scheduler option), which provides dependably strong performance at scale in all of the benchmarks, and only misses the cutoff for the Parallel Spawn benchmark due to its poor performance with six cores or less.

5.2 Inferring Details

When answering the question of *why* these benchmarks perform the way they do, there is a limit to inspection, based on the proprietary nature of some of the runtimes and the perturbation of timing imposed by profiling. However, we can make a few observations based on the benchmarks and what is known about the design of the runtimes. The first two benchmarks, Serial Spawn and Parallel Spawn, do not use the tasks to perform any kind of work, and spawn them in a predictable fashion. This makes them highly susceptible to compiler-based static analysis, which can collapse task spawns when the tasks can be identified as null tasks or even merely side-effect free. The benchmark results for Parallel Spawn bear this out as the compiler-based runtimes perform especially well. What is interesting is that, in the Serial Spawn benchmark, of the compiler-based runtimes only Cilk appears to be able to fully analyze the tasks being spawned; the OpenMP implementations are unable to automatically collapse the null tasks.

5.2.1 Static Analysis

Task-aware compilers have the opportunity for performing task-based static analysis to optimize performance, and Cilk's compiler has a history of deep static analysis for parallel execution. A good example of this analysis is the fast-clone feature of Cilk [9] wherein the compiler creates a synchronization-free version of each task that is used whenever it is known (by virtue of the fact that the task's children have not been stolen) that it is safe to do so. This behavior helps greatly, especially when the structure of the application is such that work-stealing is rare, as with the Fibonacci benchmark. The Fibonacci benchmark's structure is such that there is a lot of load balancing overhead at the beginning of execution and at the end, but in the middle there is very little. This structure benefits greatly from the fast-clone feature of the Cilk compiler. The UTS benchmark, on the other hand, is designed to make it impossible to statically analyze and to force a great deal of runtime load re-balancing. Cilk's relatively poor performance at the UTS benchmark suggests that runtime load re-balancing is not something that Cilk is especially good at. The OpenMP runtimes also have the opportunity to leverage static analysis, and their performance on the Parallel Spawn benchmark may be an indication of that. However, the performance differences on the other benchmarks suggest that the OpenMP compilers do not yet have the level of task-based static analysis sophistication that Cilk has.

5.2.2 Scheduling Policy and Synchronization Avoidance

An interesting comparison to examine is the Cilk and Sherwood Qthreads results. Cilk's work-stealing algorithm only steals a single task per steal operation under the assumption that a single old task is likely to result in a great deal more work than a single task that has been spawned very recently, and is less likely to require data that is in cache.

Further, Cilk's work-stealing algorithm is intended to reduce the number of tasks that are stolen as much as possible, to further assist the fast-clone optimization. Qthreads cannot leverage a compiler-based fast-clone, regardless of the scheduling policy. As such, Sherwood Qthreads suffers from a large number of load-balancing operations and consequently significant queue contention. Cilk has similar problems, but the benefit of the fast-clone usually compensates, in those cases where it can be used.

5.2.3 Context Swapping

Scheduling policy details explain why Cilk and Sherwood Qthreads have such different performance on the first three benchmarks, but does not explain the performance difference on the UTS benchmark. Because the UTS benchmark's design negates most of Cilk's compiler-based and structural advantages, the difference in performance must be based on other runtime overheads, such as the cost to perform synchronization and execute tasks (i.e. context swap). Cilk's context swap implementation is entirely compiler-based, saving off the active-set of variables at every synchronization point. The Qthreads context-swap is based on the function-call ABI, and as such must only save the caller-saved registers and the stack pointer. Which context-swap method is faster is dependent on both the application – how many active variables it has – and the function-call ABI – how many registers must be saved by the callee. If, as is true for the UTS benchmark on an x86_64 architecture, the size of the set of active variables is larger than the size of the set of caller-saved registers, Cilk's compiler-generated scope-based context swap method is necessarily slower than an ABI-based context-swap. The x86_64 ABI requires storing 11 64-bit registers, which makes it reasonably probable that in complex code, the ABI-based context-swap has the advantage. However, on platforms like the POWER7 architecture – where the function-call ABI requires saving 19 64-bit general-purpose registers, 18 64-bit floating-point registers, 11 256-bit vector registers, and 8 more 64-bit bookkeeping registers – it is much more likely that a scope-based context swap, or even a compiler-assisted context swap, will outperform an ABI-based swap.

5.2.4 Synchronization Semantics

It is also possible that the relative efficiencies of the Qthreads and Cilk synchronization primitives are responsible for the performance difference in the UTS benchmark, though that is hard to determine definitively given Cilk's proprietary nature. Cilk uses, exclusively, the `sync` keyword, which is equivalent to OpenMP's `taskwait` and TBB's `spawn_root_and_wait`: it can only wait for all spawned tasks to complete. Qthreads uses a more flexible, direct synchronization mechanism: full/empty bits.

Synchronization primitives designed around the "wait for all my child tasks" semantic work well with a load-balancing technique known as depth-restricted work stealing [18], which is essentially designed for recursive spawn trees. Depth-restricted work stealing requires that when a task blocks, if its worker thread must steal a task to get more work, the only tasks eligible to be stolen are those with a greater recursive depth than the blocking task. The underlying idea is that by working on tasks that are deeper into the recursive tree, the task is more likely to be assisting in the forward progress of the blocked tasks. This assumption is true in deep recursive spawn trees, such as those in Fibonacci and UTS. Only one runtime, TBB, uses this load balancing technique, which explains why it does so well on the Fibonacci and UTS benchmarks despite not being able to leverage compiler assistance. However, depth-restricted work stealing requires additional work to decide which tasks can be stolen, and as such imposes additional, unhelpful overheads on task patterns that are not deeply recursive spawn trees, which is likely why TBB performs so poorly on the Serial and Parallel Spawn benchmarks.

It is tempting to think that perhaps TBB and Cilk ought to be combined: leverage the compiler benefits of Cilk with the scheduler technology of TBB. However, these two technologies assume inherently opposed technical strategies. Depth-restricted work stealing encourages stealing child tasks, rather than parent tasks, which would significantly reduce the utility of the fast-clone feature of Cilk, since it relies on child tasks not being stolen in order to reduce synchronization.

5.3 Profiling

Profiling a performance-optimized code is a challenging endeavor. Function inlining and function multi-purposing restrict the ability for a given runtime to know precisely what is taking the most time and why. For instance, a single scheduling function may participate in context swapping, synchronization, and load balancing depending on the situation, which is often impossible to differentiate without requiring a line-by-line analysis of the runtime. Additionally, load-balancing runtimes are reactive to the behavior of a system at runtime, and so naturally are highly sensitive to profiling overhead. Further, some of these runtimes are intentionally resistant to profiling. The Intel TBB implementation in particular hides its internal functions from the profiler, identifying them merely as "`[TBB Scheduler Internals]`".

With those caveats, however, we can still obtain rough estimates of what the runtimes are spending their time on by using a statistical profiler and binning functions into categories based on educated guesswork. There is obviously some inaccuracy to this, but it is a reasonable first-order analysis. We ran the benchmarks at full scale and profiled them using the HPCToolkit software to identify how much time each function was taking. We then categorized the functions by what they were most likely doing. The categories were as follows:

Task the body of the tasks themselves.

Spawning any function associated with creating new tasks.

Scheduler Misc any function associated with the scheduler but not clearly in another category.

Yielding any call that explicitly relinquishes control of the processor.

Memory Allocation function calls clearly associated with memory allocation and deallocation.

Load Balance functions associated with load balance.

Context Swap functions associated with saving task state and restoring or initializing task state.

TLS functions associated with thread-specific data access.

Synchronization atomic operations, mutexes, and all other synchronization-related functions.

Figure 5.2 presents stacked bar graphs of the percentage of execution time used by the runtimes in executing the Serial Spawn benchmark. GCC OpenMP spent most of its time in a function called `do_spin`, which is an internal spin-lock involved in `taskwait`. One aspect that is particularly interesting is the heavy use of yielding functions, specifically `__sched_yield()`, in both Cilk and Intel OpenMP. This is most likely a result of their load balancing mechanism, wherein when no tasks are available, the CPU is relinquished. This behavior provides a fall-back to handle cases where the computer is over-subscribed. The large portion of HPX's runtime that falls in the miscellaneous category is related to CPU affinity and topology, specifically, spent creating and destroying topology objects. Excluding topology manipulation, most of HPX's runtime is spent doing synchronization operations. The most-used function in Qthreads, is `qt_scheduler_get_task`, which is the primary load balancing function. TBB, unfortunately, spends most of its time in the ambiguous `[TBB Scheduler Internals]`.

Figure 5.3 presents stacked bar graphs of the percentage of execution time used by the runtimes in executing the Parallel Spawn benchmark. The purpose of spawning tasks from a parallel loop was to reduce the need for runtime load-balancing, and it appears to have been largely successful. GCC OpenMP is still stuck with `do_spin`, but because of the natural load-balance of the benchmark, manages to be performant. Intel OpenMP spends most of its time in a function called `__kmp_wait_sleep`, though it is unclear what it is waiting for. Qthreads appears to be doing a surprising amount of load balancing, though this may be an artifact of the dual-purposes of the `qt_scheduler_get_task` function.

Figure 5.4 presents stacked bar graphs of the percentage of execution time used by the runtimes in executing the Fibonacci benchmark. Despite the large volume of synchronization operations in this code, most runtimes spend

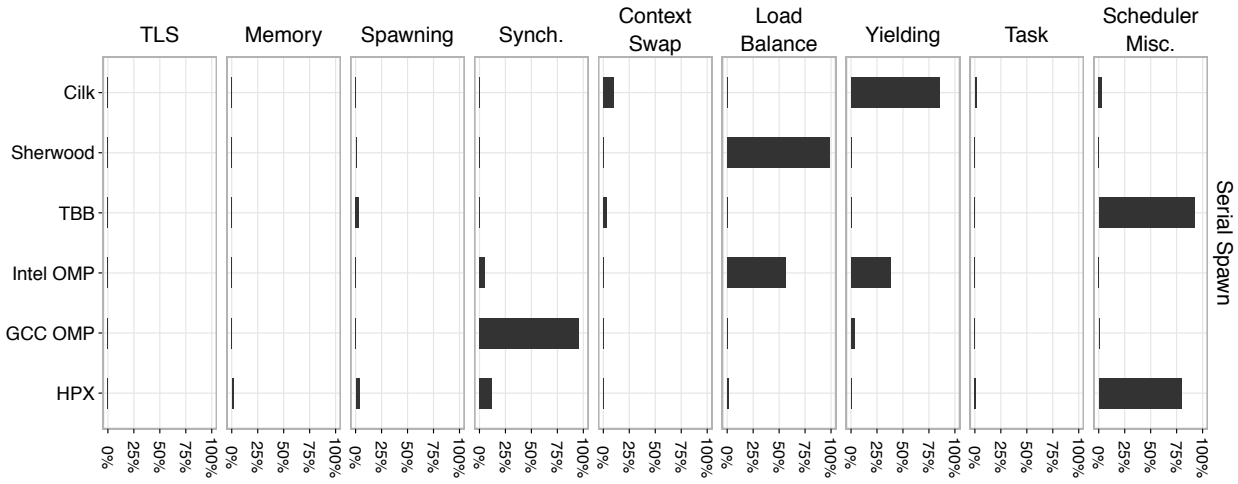


Figure 5.2. Execution time profile of Serial Spawn based on function binning.

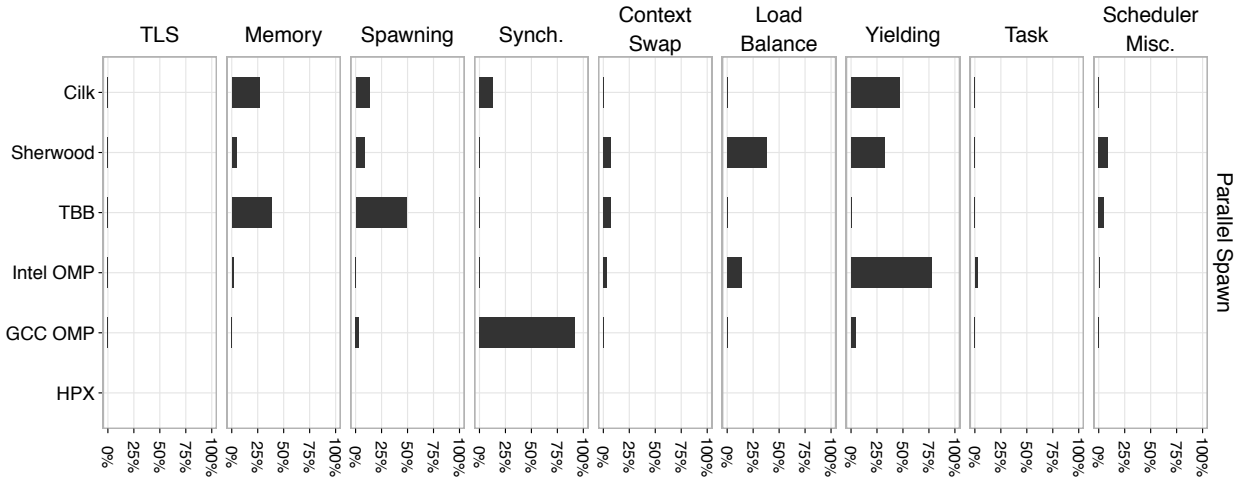


Figure 5.3. Execution time profile of Parallel Spawn based on function binning.

relatively little time performing that synchronization. This may be, as in Cilk, because the synchronization is avoided altogether by scheduling policy, or it may be another dual-purpose function artifact. For instance, Intel OpenMP spends most of its time in `__kmp_wait_sleep`, which may be a part of synchronization. Qthreads, on the other hand, spends most of its time in `qt_hash_lock` and its caller `qthread_readFF`, which is the synchronization used for its full/empty-bit support. Once again, HPX spends most of its time manipulating topology data structures, but excluding topology, most of the runtime is spent in synchronization operations. For instance, other than topology-related functions, the most frequently called function in HPX is an atomic `from_integral` function.

Figure 5.5 presents stacked bar graphs of the percentage of execution time used by the runtimes in executing the Unbalanced Tree Search benchmark. Cilk, which performed unusually poorly on this benchmark, spent a large amount of time allocating memory, presumably for saving off state. Perhaps predictably, the fastest runtimes on this benchmark are those that actually spend some time load balancing (assuming that the time TBB spends in "miscellaneous" is time spent load balancing).

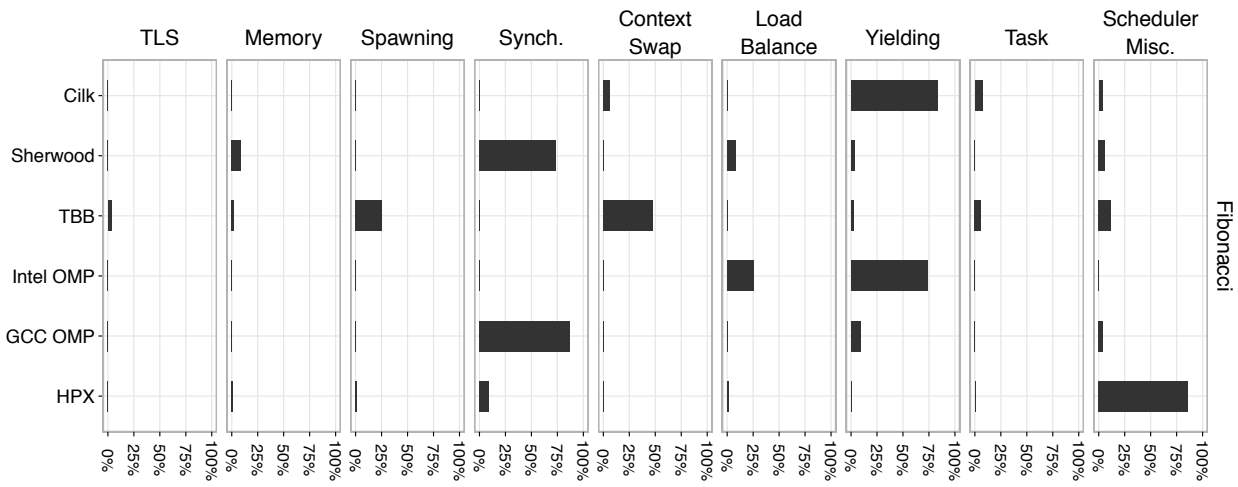


Figure 5.4. Execution time profile of Fibonacci based on function binning.

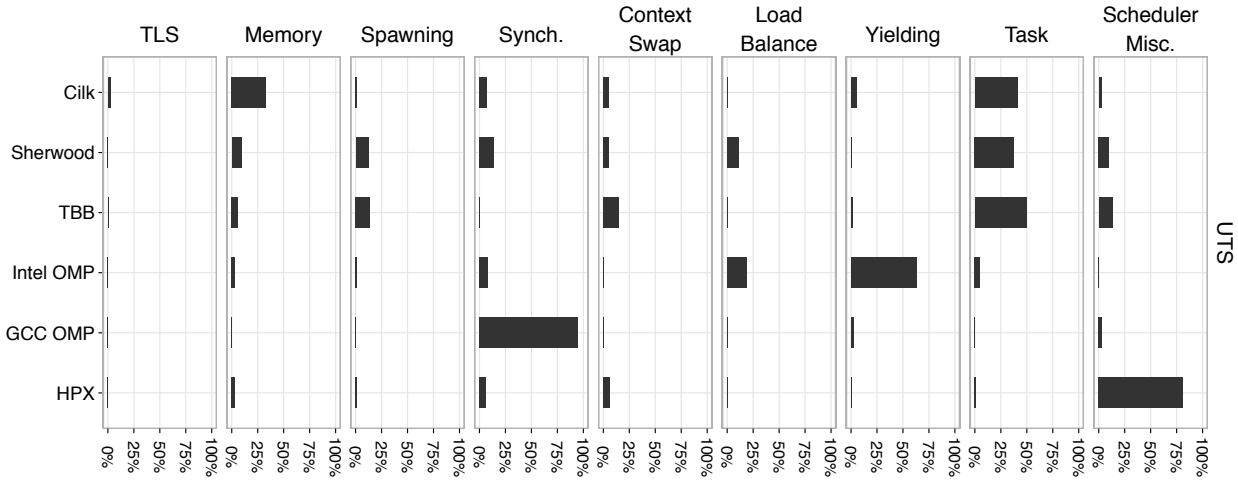


Figure 5.5. Execution time profile of Unbalanced Tree Search based on function binning.

In all of these benchmarks, except in cases where the total runtime is exceedingly small anyway, memory allocation accounts for a relatively small percentage of the execution time. As a result, alternative memory allocation schemes are unlikely to significantly impact the overall time unless the application itself places a great deal of stress on memory allocation. A few runtimes exhibit consistent behaviors across benchmarks. For instance, GCC OpenMP appears to have made a poor choice in using a spin-lock to do task-wait synchronization, which is only beneficial when the code is naturally balanced (i.e. Parallel Spawn). Both Cilk and Intel OpenMP spends a lot of time in yielding functions, though that may suggest that they are either mis-categorized or dual-purpose; they may also be used for context swapping or synchronization. It is also possible that they're forced to yield as a precaution against over-subscription.

Based on these results, while there is no single tasking operation that would be beneficial to hardware-accelerate for all runtimes, there are a few that usually account for most of the execution time: synchronization and context swapping.

Chapter 6

Conclusion

This paper presented a comparison of four task spawning benchmarks (Serial Spawn, Parallel Spawn, Fibonacci, and UTS) across six publicly available tasking runtimes (Intel Cilk Plus, Intel Threading Building Blocks (TBB), Intel OpenMP, GCC OpenMP, Qthreads, and HPX). The results show that no single runtime provides the highest performance for each of the given benchmarks, and, in fact, each runtime exhibits a set of underlying design assumptions for which it is optimized. Blindly choosing one of these systems as part of the overall model of computation for a future platform implies accepting those assumptions for that platform's application set, which has significant ramifications on application structure and performance. Specifically, it places an implicit requirement on future application structure in order to obtain performance.

Where further analysis of those underlying assumptions was possible, we examined the results of our study and demonstrated that:

1. No single runtime is optimal in all cases.
2. Using the wrong task spawning pattern for the runtime has a significant performance penalty.
3. Runtimes with the best performance for one benchmark usually have the worst performance for another benchmark.

In order to properly support an increase in parallelism of multiple orders of magnitude while still presenting a useful execution model, the concepts must at least scale to commodity levels of parallelism. Similarly, if a runtime implementation is expected to scale to multiple orders of magnitude larger than commodity systems, it must scale to at least commodity levels of parallelism. Further, to be considered general-purpose, both the execution model and the supporting runtime system must not have easy-to-expose pathological cases. Work remains to be done to improve the generality of the task scheduling systems embodied in today's task-parallel runtime systems, specifically with respect to their suitability for real world applications that will compose a wide variety of spawn and synchronization patterns.

Bibliography

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

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, USA, 1995. PPOPP '95, ACM Press.

[3] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 362–371, New York, NY, USA, 1993. ACM.

[4] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[5] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[6] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, August 2010.

[7] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 521–530, Washington, DC, USA, 2006. IEEE Computer Society.

[8] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, March 1989.

[9] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Notices*, 33(5):212–223, May 1998.

[10] Guang R. Gao, Thomas Sterling, Rick Stevens, Mark Hereld, and Weirong Zhu. ParalleX: a study of a new parallel computation model. In *Proceedings of the 2007 International Parallel and Distributed Processing Symposium*, IPDPS, pages 1–6. IEEE International, March 2007.

[11] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Tech. Journal*, 45:1563–1581, 1966.

[12] Intel Corporation. *Intel® Threading Building Blocks*, 1.10 edition, 2008.

[13] Lutz and Jayasimha. What is an effective schedule? In *Proceedings of the 1991 Third IEEE Symposium on Parallel and Distributed Processing*, SPDP '91, pages 158–161, Washington, DC, USA, 1991. IEEE Computer Society.

[14] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An unbalanced tree search benchmark. In George Almási, Calin Cascaval, and Peng Wu, editors, *Languages and Compilers for Parallel Computing*, volume 4382 of *Lecture Notes in Computer Science*, pages 235–250. Springer Berlin / Heidelberg, 2007.

[15] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, Michael Spiegel, and Jan F. Prins. OpenMP task scheduling strategies for multicore NUMA systems. *The International Journal of High Performance Computing Applications*, February 2012.

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

[17] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 1 edition, July 2007.

[18] Jim Sukha. Brief announcement: a lower bound for depth-restricted work stealing. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 124–126, New York, NY, USA, 2009. ACM.

[19] Alexandre Tabbal, Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, and Thomas Sterling. Preliminary design examination of the parallex system from a software and hardware perspective. *SIGMETRICS Perform. Eval. Rev.*, 38(4):81–87, March 2011.

[20] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22nd International Symposium on Parallel and Distributed Processing*, IPDPS/MTAAP, pages 1–8. IEEE Computer Society Press, April 2008.

DISTRIBUTION:

1 Richard Murphy
Micron Technology, Inc.
8000 S. Federal Way
P.O. Box 6
Boise, ID 83707-0006

1 MS 0899 Technical Library, 9536 (electronic copy)

Sandia National Laboratories