Scheduling Chapel Tasks with Qthreads on Manycore: A Tale of Two Schedulers

Noah Evans
Center for Computing Research
Sandia National Laboratories
Albuquerque, New Mexico
nevans@sandia.gov

Stephen L. Olivier
Center for Computing Research
Sandia National Laboratories
Albuquerque, New Mexico
slolivi@sandia.gov

Richard Barrett
Sandia National Laboratories
Albuquerque, New Mexico
rfbarre@sandia.gov

George Stelle
Programming Models Team
Los Alamos National Laboratory
Los Alamos, New Mexico
stelleg@lanl.gov

ABSTRACT

This paper describes improvements in task scheduling for the Chapel parallel programming language provided in its default on-node tasking runtime, the Qthreads library. We describe a new scheduler distrib which builds on the approaches of two previous Qthreads schedulers, Sherwood and Nemesis, and combines the best aspects of both –work stealing and load balancing from Sherwood and a lock free queue access from Nemesis– to make task queuing better suited for the use of Chapel in the manycore era. We demonstrate the efficacy of this new scheduler by showing improvements in various individual benchmarks of the Chapel test suite on the Intel Knights Landing architecture.

CCS CONCEPTS

• **Software and its engineering → Massively parallel systems**; **Parallel programming languages**; **Runtime environments**;

GENERAL TERMS

Parallel Programming Languages, Runtime Systems, Multithreading, Task Parallelism

KEYWORDS

Chapel, Qthreads, Task Queues, Work Stealing

ACM Reference format:
Noah Evans, Stephen L. Olivier, Richard Barrett, and George Stelle. 2017. Scheduling Chapel Tasks with Qthreads on Manycore: A Tale of Two Schedulers. In *Proceedings of ROSS '17, Washingon, DC, USA, June 27, 2017,* 9 pages.
DOI: 10.1145/3095770.3095774

1 INTRODUCTION

As systems grow in size and computational power much of the scaling gains come not from faster processors or instruction-level parallelism but from greater numbers of cores and hardware threads per processor. This sea change in computing threatens programmer productivity because parallel programming is a challenging activity. It is much more difficult to reason about than sequential programming and there are fewer developers with either the capability or the inclination to create large parallel programs that are both correct and scalable.

Therefore the ability to fully exploit parallelism requires tools and programming models that abstract away or simplify the parallelism available to the user. By pushing the complexity of parallel programs into the runtime developing large parallel programs becomes tractable. The success of the message passing model and supporting MPI implementations like MPICH and OpenMPI demonstrate the advantages of this approach.

In task parallel programming models, the programmer specifies the smallest sequential units of work to be performed (tasks) and data or control dependences between them. Efficient scheduling of the tasks, concurrently where allowed by the dependences, is the responsibility of the run time system. Task parallelism is a key feature of Chapel [11], a programming language developed at Cray Inc. as part of DARPA's High Productivity Computing System (HPCS) program that attempted to make parallel programming available to a wider spectrum of programmers.

A core idea of Chapel is the idea of *multiresolution* development. The programming language allows the user to develop coarse grained programs relying on traditional sequential data structures and control flow and gradually introduce parallel and high performance capabilities, like sparse arrays and parallel tasking, during the natural course of program development. This allows Chapel applications to utilize disparate runtime systems and different hardware with minimal changes to applications themselves.

The software architecture of Chapel's run time system uses a modular approach and provides a common tasking layer interface which has binding to a variety of tasking layer implementations including MassiveThreads [22], Qthreads [29], POSIX threads [9] and (on Cray systems) Cray-proprietary lightweight threading.

The current default tasking layer for Chapel is Qthreads, a user-level threading library developed by Sandia National Laboratories. The key idea of Qthreads is to provide abstractions for lightweight threading and synchronization that directly model parallel hardware even though they are implemented in software. Locality is also a first class concept: locality domains are specified by work queue controllers called *shepherds* that correspond to hardware locality domains.

The structure of Qthreads locality and synchronization primitives correspond directly to Chapel's parallel primitives which make it possible to make modifications to Qthreads which interact with the *resolution* of an application by specifying new synchronization and scheduling primitives which Chapel can then use transparently.

This paper describes a new scheduler **distrib** which takes lessons learned from previous Qthreads schedulers to implement a new scheduler that combines the best aspects of each, specifically work stealing and lock free queuing, in order to make a new scheduler with better performance under Chapel than its predecessors.

The remainder of the paper is organized as follows. Section 2 gives and overview of task parallelism, the Chapel language, and the Qthreads multithreading library. Section 3 describes the existing Qthreads schedulers and the new distrib scheduler. Section 4 presents an evaluation of the different schedulers. Section 5 discusses related work. The paper closes with conclusions and future work in Section 6.

2 BACKGROUND

2.1 Task parallelism

The task parallel programming model for parallel programming breaks down computations into work units and then schedules and executes these units on available hardware resources, often dynamically at run time. Early work on the task parallel programming model and efficient run time system support for its use is exemplified by the Cilk [15] extention to C. In recent years, the number of task parallel languages, libraries, and run time systems has increased manyfold, as shown in the related work (Section 5).

The task parallel programming model differs in approach from other models of parallel computation like Single Program Multiple Data (SPMD) and Bulk Synchronous Processing (BSP), which typically depend on the programmer's specification of a static, regular, balanced work distribution. In the task parallel model, different tasks can perform different amounts of work, and they can begin and end at different times. While providing greater opportunities for parallelism and the interleaving of computation and communication, this asynchrony among tasks can lead to contention and load balancing problems as work can be unevenly distributed and multiple tasks can contend for resources.

To ensure that work is evenly distributed and computations are making progress, correct and efficient scheduling by the run time system are paramount. Much of the research into task parallel runtimes involves ensuring that load balance is maintained, often by work stealing [7], to move tasks from busy computational units to units that are underutilized.

A task parallel computation is often represented as a Directed Acyclic Graph (DAG) or tree of tasks. During execution, the runtime partitions the graph into subgraphs such that each hardware resource executes some set of subgraphs over the course of the program. One especially important aspect of this partitioning of the task graph is maintaining proper spatial and temporal locality of data in the computations. Since computational bandwidth often outstrips data bandwidth the runtime should schedule data to be closer to the computation involved.

2.2 Chapel Tasking Layer

The Chapel language [11] incorporates the task parallel programming model through the parallel constructs **begin**, **cobegin**, and **coforall**. Task management in the Chapel runtime is implemented as a C API that provides support for these constructs, as well as synchronization variables for managing concurrent access to data. Chapel code compiles to C code that delegates its tasking and synchronization behavior to the Chapel tasking API. The API consists of the following functionality:

- The Startup/Teardown layer initializes and finalizes the task runtime as well as creating singleton tasks for **begin** statements.
- The Creation and Execution of Task Lists implements Chapel's cobegin and coforall statements.
- The Synchronization functions implement the full/empty semantics of Chapel's synchronization variables.
- Task Control yields the processor or sleeps.
- Query functions allow Chapel to query the number of tasks, threads or states.

In accordance with Chapel's multiresolution approach this API make no assumptions about the behavior of the underlying runtime. This API is modular, so it is possible to choose different tasking implementations at runtime via an environment variable.

2.3 Qthreads

The default Chapel tasking layer is the Qthreads [29] a cross platform, general purpose, parallel runtime from Sandia National Laboratories. Qthreads is composed of two fundamental abstractions, lightweight threads scheduled onto locality domains called *shepherds* and Full/Empty Bit (FEB) synchronization primitives. The goal of these abstractions is to match hardware threading architectures that implement massive lieghtwieght multithreading and synchronization, such as such as the Tera MTA / Cray XMT [1]. Individual threads of computation can be anonymous, have explicit resource allocations, and exploit explicit locality.

This approach means that Qthreads are fundamentally different from traditional threading models. Qthreads do not have individual thread identifiers, signal vectors or preemption. They share more in common with coroutines and scheduler activations [2] than OS-level threads.

Scheduling in Qthreads is cooperative. When one Qthread can no longer make progress, either via a synchronization primitive or an explicit yield, control is then passed to another waiting Qthread. This context switch occurs entirely within user space, typically much faster than a system call and does not require the saving of signal handlers or the full set of system registers. These user level context switches allow the Qthreads runtime to interleave computation with data access. A Qthread can –for example– launch a new Qthread to produce some data and write a FEB, then yield to be rescheduled when the FEB is available for reading. In the interim, another Qthread can be scheduled so that hardware resources are not idle.

3 SCHEDULER DESIGN IN QTHREADS

Like the Chapel runtime, the Qthreads library also uses a modular design, and among the configurable options is the choice of cooperative scheduler. Various schedulers are implemented in terms of thread queues with defined interactions within and between *shepherds* (locality domains) and workers.

Table 1: Qthreads schedulers

Scheduler	Queue	Workstealing
Sherwood	One per NUMA domain OR one per worker thread	Yes
Nemesis	Only one per worker thread	No
Distrib	Only one per worker thread	Yes

To implement a thread queue a developer satisfies an API that provides the following functionality:

- initialization and teardown;
- enquequeing and dequeuing, as well as filtering mechanisms to remove certain classes of threads from the queue;
- stealing control and statistics, which are optional;
- policy support which dictates whether a shepherd can support multiple workers or only one.

The original Qthreads scheduler was a simple lock free queue that distributed tasks in FIFO order with only one queue and one worker per shepherd. More sophisticated schedulers followed, and several of these are described below and summarized in Table 1.

3.1 Sherwood

The Sherwood scheduler was the first work stealing scheduler for Qthreads, developed originally to support OpenMP tasking over Qthreads [23]. It takes an alternate approach to the organization of shepherds and queues. Rather than maintaining a one-to-one mapping of shepherd and work queue to hardware thread, Sherwood generalizes the idea of a shepherd to correspond to a particular resource with locality constraints (e.g., a NUMA domain) with multiple workers per shepherd that share a queue mediated by mutex locks.

Sherwood uses a two-level load balancing scheme combining the methods of *work stealing* [7] and LIFO shared queuing among topologically nearby threads, known as *parallel depth-first (PDF)* scheduling [6]. All workers within a shepherd share a single queue. This arrangement enables them to benefit from cooperative caching since they share cache and memory resources, an effect of PDF schedulers [13]. Tasks are scheduled in LIFO order, so newly created or recently yielded tasks are executed first in order to exploit cache locality. When a work queue is empty Sherwood attempts to work steal from other shepherds on the system, examining other work queues in a round robin fashion. When work is found the scheduler attempts to take *n* qthreads from the victim queue, where *n* is a tunable parameter that defaults to the number of workers per shepherd. This approach maximizes throughput by doing slow cross-domain transfers at once rather than attempt to steal one qthread at a time.

Sherwood excelled on NUMA architectures with small numbers of cores per NUMA domain and large shared caches, and became the default Qthreads scheduler. On recent many-core systems it has struggled to scale with larger numbers of cores and threads, often experiencing significant queue contention. While one possible configuration of Sherwood for decreased contention is to use only one worker per shepherd and queue, such a configuration was not the original design point for this scheduler.

3.2 Nemesis

The Qthreads Nemesis scheduler is a thread queue based on the MPICH2 Nemesis queueing subsystem. It is currently the default thread queue used in Chapel. In contrast to Sherwood, Nemesis uses a simple FIFO queueing scheme for jobs designed to be a highly optimized for SMP systems.

The original Nemesis queuing scheme acted as a progress engine for MPICH2 [8]. As part of the MPI progress engine it provided a queue implementation for multiple producers to write MPI messages to a queue which had one receiver process.

The Nemesis Qthreads thread queue adopts this intranode message queueing scheme from MPICH2, but rather than receiving MPI messages it receives tasks and dequeues them for scheduling. Unlike Sherwood locality is set to a per hardware thread basis rather than per NUMA domain. This minimizes contention and maximizes scalability at the cost of NUMA-based locality, e.g., shared caching.

To minimize memory bus traffic Nemesis aligns all thread data structures to cache lines. The queue then uses a compare and swap instruction to swap the head and tail of the queue. If the swapped tail is not null there is contention writing the queue and the queue should wait for the operation to complete before attempting to write. This ensures progress and throughput in aggregate: Although some writers can starve, the system itself is guaranteed to make progress.

Nemesis is the Qthreads scheduler currently used by the Chapel runtime, and is better suited to manycore architectures than the Sherwood scheduler due to lower queue contention. However, it lacks the load balancing benefits of a workstealing approach.

3.3 Distrib

The recently added Distrib scheduler is a new thread queue scheduler designed from scratch to combine the NUMA efficiency and load balancing of Sherwood and the queueing performance of Nemesis. The primary insight is that Nemesis mitigates memory contention in the runtime by spreading its work across many queues, and therefore many cache lines. In contrast, Sherwood runs into contention issues due to every queue operation per NUMA domain touching a single cache line. While this was less of an issue for Sherwood in the past, when there were fewer cores per NUMA domain and per node, modern many-core architectures have shown that this approach scales poorly. By spreading work across many cache lines and adding mechanisms to limit work stealing, Distrib enables reduced-contention load balancing.

At its core Distrib is a reimplementation of Nemesis's queue with work stealing functionality added to its behavior. It maintains the same lock free behavior as Nemesis, swapping head and tail pointers with compare and swap before adding elements to the queue. All queue data structures are also cache aligned. However Distrib differs from Nemesis in its queueing order, items are scheduled in LIFO order to preserve cache locality.

Distrib departs significantly from Sherwood in its work stealing implementation. Work stealing is heavily simplified, stolen elements are still stolen from the head of individual work queues in round robin order, but they are stolen a job at a time according to a user defined environment variable `STEAL_RATIO` defines ratio of attempts to run an enqueued job before stealing. If there is no work to steal and no jobs on the queue another user defined variable `COND_BACKOFF` specifies a number of cycles to wait before sleeping on a condition variable and setting a counter of sleeping worker threads. If a task attempts to enqueue work it signals the condition variable to wake up the sleeping workers and decrements the number of waiting workers.

Distrib is currently being tuned and tested as a candidate to be the new default Qthreads scheduler for Chapel.

4 EVALUATION

To measure the differences between the different schedulers we chose several benchmarks from the Chapel nightly performance benchmarks and ran them using the baseline Qthreads FIFO threadqueue, Sherwood, Nemesis and Distrib in order to understand the differences between each of the schedulers and their behavior. Specifically we wanted to see how the performance of each queue affected the running time of Chapel benchmarks and to see the effects of work stealing and lockfree queuing in isolation.

4.1 Experimental Setup

The evaluation was conducted on a node of the Bowman Advanced Architecture Testbed cluster at Sandia National Laboratory. This node hosts an Intel Xeon Phi Knights Landing Processor, model number 7250, with 68 cores and 272 hardware threads operating at 1.6 GHz. The processor includes 16GB of high bandwidth memory (MC-DRAM) on the package, which we operate in cache mode. In addition

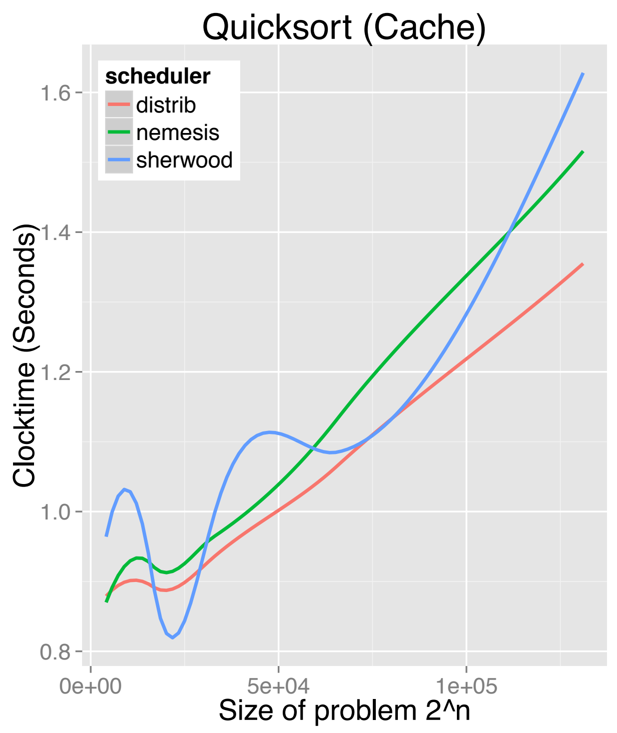

Figure 1: Chapel Quicksort Benchmark
Lower is better

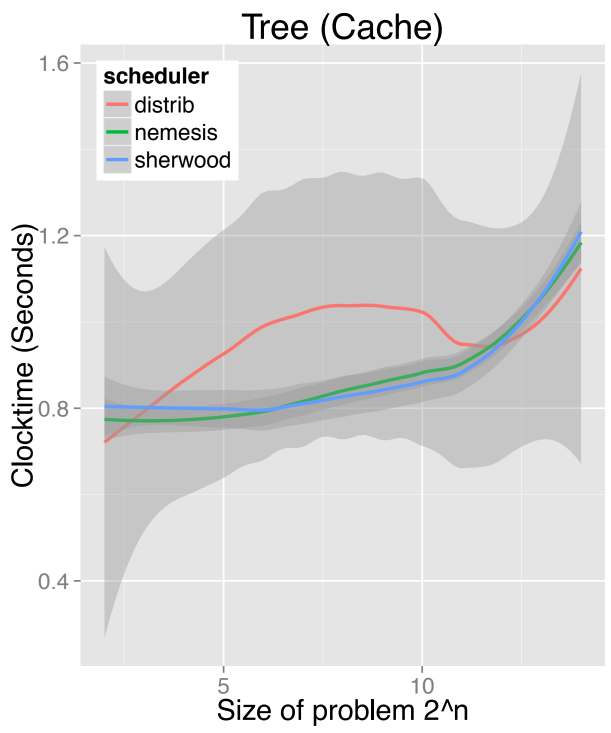

Figure 2: Chapel Tree Benchmark
Lower is better

there is 96GB of DDR4 main memory. We compiled with Chapel version 1.14.0.5f9253e and GCC version 4.8.3 using the -O3 and -march=native flags. Performance comparisons were performed by using Linux's perf tools to do full system profiling and the flamegraph tools [16] to do time comparisons. The Chapel benchmarks used are all available in the main Chapel distribution.

4.2 Quicksort

The Chapel quicksort benchmark is an implementation of a parallel quicksort executing the partition of each pivot in parallel. We use quicksort to act as a stress test for task spawning and communication. We also set the threshold sufficiently high that it never serializes using the same techniques as [28].

From Figure 1 we can see that –while both Nemesis and distrib are roughly comparable at small scales– as the problem size grows distrib is almost 10% faster. This change in behavior is almost entirely due to distrib's backoff strategy freeing up the kernel to do work. More than 59% of the cycles spent in quicksort are spent doing page fault handling from memory allocation. By quickly relinquishing it's timeslice when no work is available distrib is able to better accommodate the needs of the kernel.

4.3 Tree Exploration

The chapel tree benchmark computes the sum of uniquely identified randomly generated tree nodes. This is done in parallel using Chapel's cobegin. It measures how well work is scheduled across tasks using an easily parallelizable workload.

Figure 2 illustrates the difference performance characteristics of the different schedulers. Initially, from problem sizes of 2^3 to 2^{10} nodes distrib is slower than Nemesis and Sherwood which show roughly equivalent performance characteristics. However from a problem size of 2^{11} nodes onward distrib shows better scaling.

Like quicksort this behavior can be traced almost entirely to memory allocation however, in this case the kernel only spends around 36% of its time in the clearing memory in the kernel in distrib (slightly more in Nemesis). However, the extra overhead in this case comes from Nemesis spawning fewer tasks and spending more time sleeping in the kernel. Nemesis shows a 4% improvement in spin wait time compared to distrib but that gain is offset by an addition 7% of time spent in the kernel.

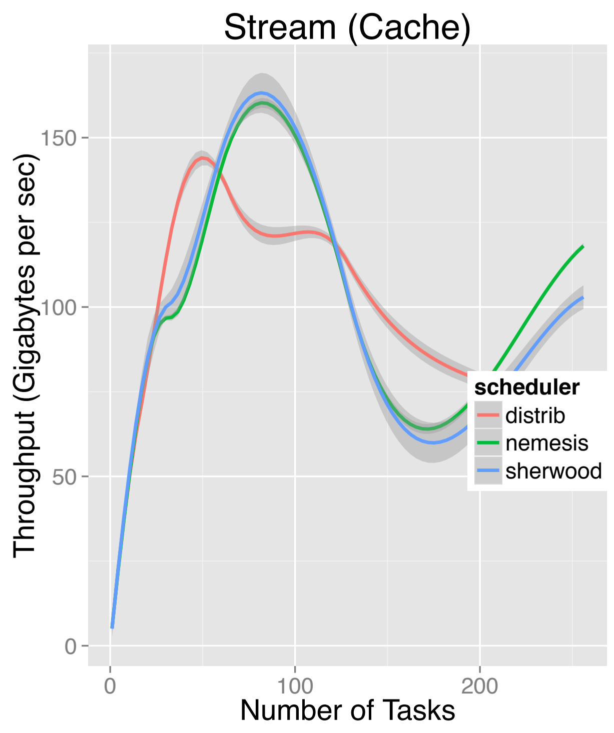

Figure 3: Chapel Stream Benchmark
Higher is better

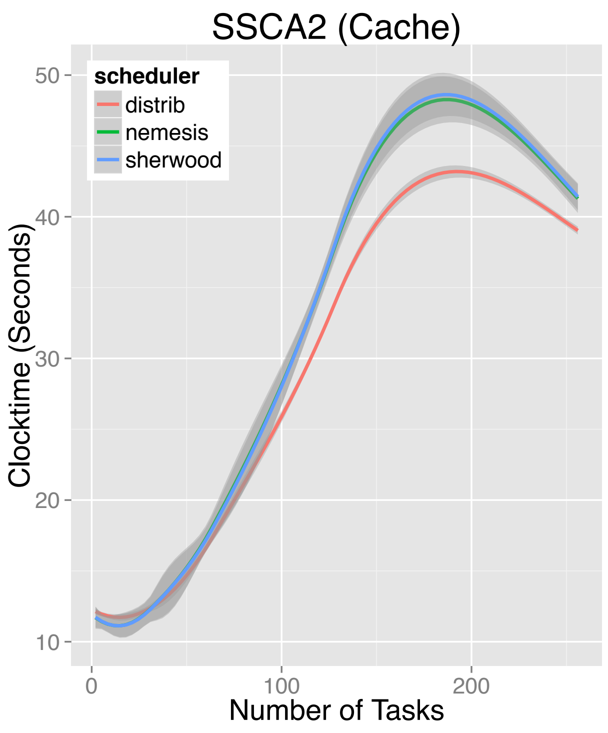

Figure 4: Chapel SSCA2 Benchmark
Lower is better

4.4 STREAM

The stream benchmark measures the amount of sustainable memory throughput of a system using a simple vector kernel which operates on arrays larger than machine cache and the data is structured to ensure that data is not reused.

Since the workload on stream is regular and stresses the system, stream is a good benchmark for measuring the overhead of a task based system, any decline in throughput across schedulers indicates overhead costs for a particular scheduler. To test this overhead we use a constant problem size of 16 Megabytes of data (the smallest problem size that achieves maximum memory throughput for the benchmark) but vary the number of cooperating tasks to measure the impact of the threading runtime on the benchmark performance.

Figure 3 describes our stream result.

We see that –using Nemesis– stream achieves a maximum throughput 130 GB/s, close to the maximum for the memory bus, at 64 tasks, corresponding to one task per physical core on Knights Landing. We see that as the number of tasks equals the number of hyperthreads performance begins to scale upwards again.

We see that distrib is slightly better for smaller numbers of tasks where distrib demonstrates 6% better performance, while at 64 tasks Nemesis is slightly more than 2% faster than distrib.

This performance variation can be attributed to aggressiveness of the default backoff behavior in each queue. Distrib's more aggressive backoff behavior leads it to give up control of the processor more aggressively than Nemesis, leading to more time spent in Linux's irq path with larger numbers of tasks, while at smaller numbers of tasks distrib is capable of providing slightly more work to tasks with a full work queue.

4.5 SSCA2

The SSCA benchmark uses different analysis kernels operating on the same data structure with irregular access patterns to simulate HPC graph workloads. Since each kernel potentially has different access patterns optimizing different access patterns.

In Figure 4 we see roughly equivalent performance between Sherwood and Nemesis (less than 1%) while distrib is roughly 10% faster than both. Similar to the Tree and Quicksort benchmarks this performance improvement is due to much more aggressive backoff when there is lack of work or contention as the problem scales up. Nemesis and Sherwood both spend roughly equivalent times in computation

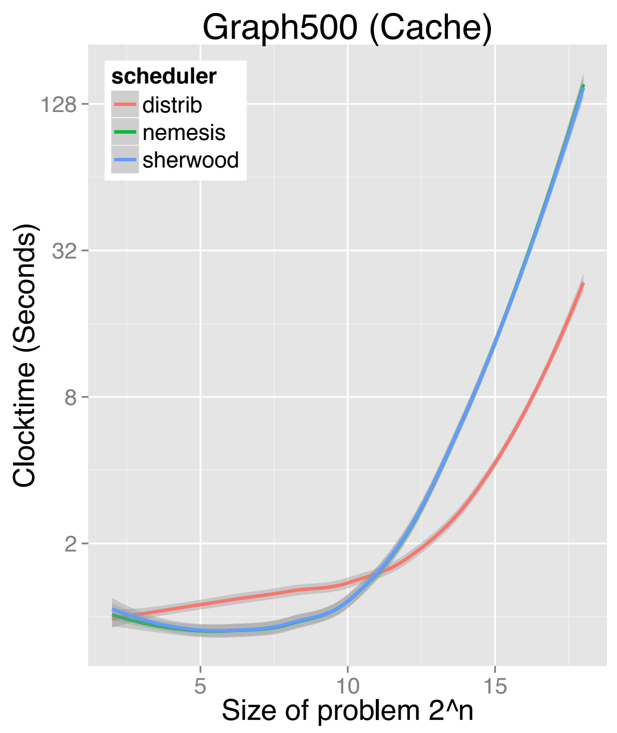

Figure 5: Chapel Graph500v2 Benchmark Lower is better

and spin locking, while distrib spends 4% time in user space and 600% as much time in the kernel, translating to a 5% improvement overall at scale.

4.6 Graph500

The Graph500 "Search" benchmark models data-intensive super computer applications. Like SSCA2, the Graph500 benchmark uses multiple kernels which operate on a weighted undirected graph. The first kernel constructs the graph and a second kernel performs a breadth first search on the graph.

In the Figure 5 we see similar scaling profiles between Nemesis and distrib, however distrib is 600% faster than Nemesis and Sherwood at scale, thanks to work stealing reducing the amount of time spent looking for work (20% less time in userspace overall) and better use of backoff to avoid monopolizing the processor leading to better throughput overall.

5 RELATED WORK

The space of task parallel programming models and run time systems is wide and varied. High-level on-node languages and libraries include Cilk [15], Intel Cilk Plus [26], OpenMP tasking [4] (available in OpenMP versions 3.0 [24] and above), Intel Threading Building Blocks (TBB) [25], and Microsoft Task Parallel Library (TPL) [19]. Libraries other than Qthreads that also provide on-node low-level lightweight threading include MassiveThreads [22] and Argobots [27]. OCR [20] represents an effort to unify run time system development. X10 [12] is a Java-based language that, like Chapel, was developed as part of the HPCS program and incorporates task parallel features as well. Habanero [10] builds on X10 and contributes to OCR. Other task parallel frameworks for distributed memory execution include Charm++ [18], HPX [17], and Uintah [21]. Legion [5], StarPU [3], and OmpSs [14] are designed to support task parallelism on heterogeneous systems.

6 CONCLUSIONS AND FUTURE WORK

This paper has described the two schedulers for Qthreads, Nemesis and distrib, that are most performant for manycore execution, contrasting them with each other and the default multicore Sherwood scheduler. Together these schedulers support a variety of HPC workloads and performance profiles. Choosing among the Qthreads schedulers and their configurations provides a way to tune the Chapel runtime to particular workloads in a way that is transparent to the users of higher level programming constructs like Chapel's cobegin and coforall statements, adhering to Chapel's goal to be a multiresolution programming language.

The most important result of this paper is that for the different HPC workloads that Chapel supports there is no one "correct" Qthreads scheduler with the best performance. Depending on the pattern and needs of the workload either of the Nemesis or distrib schedulers may be the optimal choice, specifically the Nemesis scheduler for workloads with regular data access patterns and execution flow and the distrib scheduler for workloads with irregular data access patterns and execution flow.

There is still a significant amount of work to be done to make Qthreads better fit Chapel's vision of a multiresolution programming language. Currently schedulers are statically compiled into the Qthreads library, but a better approach would be to allow Qthreads to dynamically choose schedulers at runtime. There is also much room for improvement in optimization of schedulers for manycore HPC architectures such as Intel's Knights Landing processors, including work on backoff, workstealing and other tunable parameters.

ACKNOWLEDGMENT

Sandia is a multi-mission laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. We wish to express our appreciation for the use of the Advanced Architecture Test Bed, Bowman, at Sandia National Laboratories.

The test beds are provided by National Nuclear Security Administration's Advanced Simulation and Computing (ASC) program for research and development of advanced architectures for exascale computing.

REFERENCES

[1] Gail A. Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton J. Smith. 1992. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *ICS '92: Proc. 6th ACM Intl. Conference on Supercomputing*. ACM, 188–197.

[2] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. 1992. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 53–79.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009: 15th International Euro-Par Conference on Parallel Processing*. Springer, 863–874.

[4] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishn an, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems* 20 (March 2009), 404–418. Issue 3.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *SC 12: Proc. Intl. Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 66.

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

[7] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[8] D. Buntinas, G. Mercier, and W. Gropp. 2006. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Sixth IEEE International Symposium on Cluster Computing and the Grid*, Vol. 1. IEEE, 521–530.

[9] David R. Butenhof. 1997. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[10] Vincent Cave, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *PPPJ 2011: 9th Intl. Conference on the Principles and Practice of Programming in Java*. ACM, 51–61.

[11] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[12] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proc. 20th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*. ACM, 519–538.

[13] Shimin Chen, Phillip B Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C Mowry, and others. 2007. Scheduling threads for constructive cache sharing on CMPs. In *Proc. 19th ACM Symposium on Parallel Algorithms and Architectures*. ACM, 105–115.

[14] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21, 02 (2011), 173–193.

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

[16] Brendan Gregg. 2016. The flame graph. *Commun. ACM* 59, 6 (2016), 48–57.

[17] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *8th Intl. Conf. on Partitioned Global Address Space Programming Models (PGAS '14)*. ACM, Article 6, 11 pages.

[18] Laxmikant V. Kale and Sanjeev Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proc. 8th Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '93)*. ACM, New York, NY, USA, 91–108.

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

[20] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar. 2015. The Open Community Runtime Interface. (2015).

[21] Qingyu Meng, Justin Luitjens, and Martin Berzins. 2010. Dynamic task scheduling for the uintah framework. In *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*. IEEE, 1–10.

[22] Jun Nakashima and Kenjiro Taura. 2014. MassiveThreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*. LNCS, Vol. 8665. Springer, 222–238.

[23] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. 2012. OpenMP task scheduling strategies for multicore NUMA systems. *The International Journal of High Performance Computing Applications* 26, 2 (2012), 110–124.

[24] OpenMP Architecture Review Board. 2008. OpenMP API, Version 3.0. (May 2008).

[25] James Reinders. 2007. *Intel Threading Building Blocks - Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, Sebastopol, CA.

[26] A. D. Robison. 2013. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science Engineering* 15, 2 (March 2013), 66–71.

[27] S. Seo, A. Amer, P. Balaji, P. Beckman, C. Bordage, G. Bosilca, A. Brooks, A. Castellas, D. Genet, T. Herault, P. Jindal amd L. V. Kale, S. Krishnamoorthy, J. Lifflander, H, Lu, E. Meneses, M. Snir, and Y. Sun. 2015. Argobots: A lightweight low-level threading/tasking framework. http://collab.mcs.anl.gov/display/ARGOBOTS/. (2015).

[28] Kyle B. Wheeler, Richard C. Murphy, Dylan Stark, and Bradford L. Chamberlain. 2011. The Chapel tasking layer over qthreads. In *2011 Cray Users' Group Conference (CUG 2011)*. Cray User Group, Inc.

[29] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *IPDPSW 2008: Proc. 22nd IEEE Intl. Symposium on Parallel and Distributed Processing Workshops*. IEEE, 1–8.