XTASK - eXTreme fine-grAined concurrent taSK invocation runtime

POORNIMA NOOKALA, DR. PETER DINDA, DR. KYLE HALE, DR. IOAN RAICU
pnookala@hawk.iit.edu, pdinda@northwestern.edu, khale@cs.iit.edu, iraicu@cs.iit.edu
Department of Computer Science, Illinois Institute of Technology
Department of Electrical Engineering and Computer Science, Northwestern University

***Abstract*—Exascale computers are expected to be made of millions of nodes and billions of threads of execution. To support high degrees of parallelism for various applications, the threads and task scheduling needs to be fine-grained and should be able to execute in the order of tens to a few hundred CPU cycles. Over-decomposition of applications to fine-grained applications is ideal to achieve maximum speed up and there is a need for a parallel runtime system which can launch tasks for execution and report the results with very low latency at high levels of concurrency. This work aims at enabling the launch of independent tasks on many-core accelerator hardware architectures and mechanisms to support tasks of fine granularity on the order of tens of few hundreds of CPU cycles at a large scale. This work also focuses on analyzing the performance of various queue-based data structures commonly used in parallel programming languages and runtime systems. This analysis is essential for designing an efficient runtime system for scheduling billions of tasks with very low latency and high throughput. Lastly, the runtime would also support data dependencies and task dependencies required for task-based shared memory parallel programming.**

***Index Terms*—Parallel runtime; Fine-grained parallelism; Queues; Many-task computing; Accelerators; Intel Xeon Phi; Coprocessor; Xtask**

I. INTRODUCTION

Moore's law states that number of transistors in an Integrated Circuit will double approximately every two years. But, according to processor manufacturers, we are approaching the limits of scaling with silicon and the future of Moore's law is currently undetermined. Alternatives to improving processor performance is by packing more processors together so they can work in parallel on different parts of the same workload. It is typical nowadays to have two, four or eight cores all running in parallel in most devices and systems. This necessitates the need for a programming approach that can leverage and processor's full capacity.

A. Many-Task Computing

Many-Task Computing (MTC) [2] has been an emerging paradigm and area of research for some years now. An MTC workload consists of task that run uninterrupted from start to completion. The task duration may be highly variable, ranging from tens of cycles to hundreds and thousands of cycles. Their dependency and data-passing characteristics may range from many similar tasks to complex, and possibly dynamically determined, dependency patterns. Many-task computing differs from high throughput computing (HTC) in the context of using large number of computing resources over short periods of time to accomplish many computational tasks. To efficiently handle MTC workloads, the system needs to exploit parallelism as much as possible. As more and more cores are being added to increase the processing speed, the need for parallel runtime systems that can leverage full capabilities of the processors by over-decomposition of tasks into fine-grained tasks is increasing.

B. Task-Based Parallelism

Task-based parallelism is a simple paradigm for shared-memory parallelism in which a computation is broken-down into a set of inter-dependent tasks which can then be executed concurrently on various cores. Task dependencies and data dependencies are used to control the flow of tasks through the runtime system. Tasks can be modeled as Directed Acyclic Graph (DAG) which can be traversed in order. Given the DAG, tasks can be executed using a set of threads where each thread dequeues a task from a queue and executes it. If the queue is empty, thread waits for a task to come in to the queue until the whole DAG is processed. Figure1 shows a DAG with set of tasks with arrows showing the dependencies. Nodes at one level can ideally be executed in parallel. Here tasks G and H can be executed in parallel and they do no have dependencies since they are the leaf tasks. Once the dependencies for D,E and F have been resolved, they can be executed in parallel as well.

Figure 1 shows a set of tasks with arrows representing dependencies. G and H represent leaf tasks which can be executed in parallel. D, E and F are the tasks at same level in the graph, so ideally can be run in parallel.

There are several factors that can limit the degree of parallelism and it also depends on the dependencies that exist between the tasks. Most parallel runtime systems today support execution of coarse-grained tasks with very high efficiency, however when it comes to fine-grained tasks, the efficiency decreases due to the overhead of scheduling and dispatching tasks and collecting results from different cores. Hence, there is a need for a parallel runtime system that can dispatch billions of tasks with very low latency and very high throughput. This is the primary motivation for this work.

II. RELATED WORK

There are many existing parallel runtime systems, some which exhibit implicit parallelism and some where parallelism

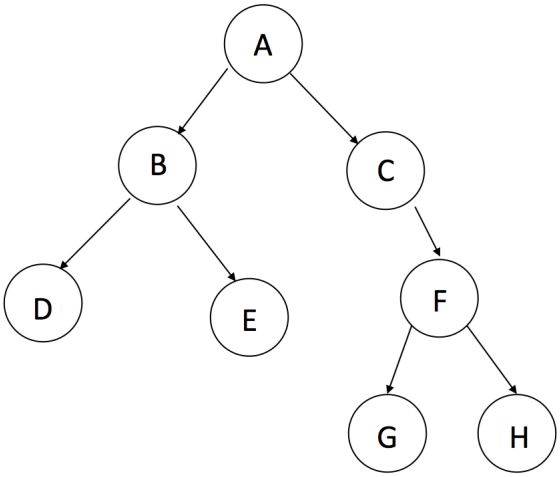


Fig. 1: Directed Acyclic Graph

needs to be explicitly represented. One of the first task-based parallel programming system implementations is Cilk[3]. Cilk is an extension of C language which allows spawning tasks as new tasks. Dependencies can be represented using sync keyword which forces a thread to wait for all tasks that it created to be complete.

StarPU (Augonnet et al., 2011) is a software tool allowing programmers to exploit computing power of CPUs and specialized hardware such as accelerators and coprocessors. QUARK (YarKhan et al., 2011) stands for Queueing and Runtime for Kernels and provides mechanisms to execute tasks with data dependencies in multi-core, multi-socket and shared memory environment. It infers dependencies between tasks from the way data is used and executes tasks in an asynchronous fashion. XKAAPI (Gautier et al., 2007) is a runtime for scheduling irregular fine-grained tasks with dataflow dependencies. In all these parallel runtimes, the programmer specifies what shared data each task will access, and how that data will be accessed. The dependencies between tasks are then generated dynamically by the runtime system, assuming that the data must be accessed and updated in the order in which the tasks are generated. Intel?s Threading Building Blocks (TBB) (Reinders, 2010) provide task-based parallelism using C++ templates in which dependencies are handled either by explicitly waiting for spawned tasks, or by explicitly manipulating task reference counters. Legion uses deferred execution which means all runtime calls are deferred and computation is performed only when it is safe to do so. Finally, the very popular OpenMP standard provides some basic support for spawning tasks, similar to Cilk, as of version 3.0 (Board, 2008). OmpSs (Duran et al., 2011) extends this scheme with automatic dependency generation along with the ability to explicitly wait on certain tasks. Charm++ is a machine independent parallel programming system which uses message-driven execution to hide the latency of communication between tasks and remote data.

In all the above mentioned runtimes, it is the responsibility of the application to decide the level of task granularity and how to decompose algorithms into tasks that can achieve performance gain by exploiting high levels of parallelism. Swift/T, on the other hand, is a data-flow driven implicitly parallel programming model. It dynamically generates a DAG where tasks can generate sub-tasks and uses futures for representing data dependencies. Swift/T has been shown to run well on tens of thousands of nodes with task graphs in the range of millions of tasks[swift]. Swift uses MPI (Message Passing Interface) for inter-node and intra-node communication.

In Xtask system, task dependencies are represented by hierarchical locks on the data objects which ensures tasks are executed only when they are ready and data dependencies have been resolved. Xtask expects a DAG as input to the runtime with dependencies expressed in terms of data locks and holds. It is up to the runtime to schedule tasks based on the dependencies. Queues are basic data structures that are used in most parallel runtime systems. Due to the FIFO nature of queues, it is widely used as message queues for queueing up tasks and processing them in order. Xtask also uses queues to hold a bag of tasks. There is a certain latency associated with the queueing operations including the latency to enqueue, dequeue and wait time in the queue. These latencies can easily become the bottleneck for overall performance of the system under high levels of concurrency. Also, multiple producer multiple consumer queue is a common structure used in parallel runtime system to exploit high degrees of parallelism. Threads are considered a convenient and efficient mechanism to use multicore processors. However, there is cost associated with implementing queues with locks and mutexes vs other mechanisms to make it thread safe. Performance analysis needs to be done on various architectures since the latency and throughput can vary based on the architecture. In this work,we study the performance of various queue implementations using AMD and Haswell processors and Intel Xeon Phi coprocessor. Intel Xeon Phi is a PCI device with roughly 60 cores and over 240 hardware threads. Its design makes it ideal for application that are performance critical and need large levels of parallelism. Moreover, the fact that it implements x86 for its instruction set architecture, makes its integration with existing systems simpler. For these reasons, it is worthwhile to note that the Xeon Phis are considered valuable additions for clusters, grids and supercomputers.

This work is inspired from prior work called GeMTC which stands for GPU-Enabled Many Tasks Computing. The GeMTC framework and its API are limited exclusively to NVIDIA GPUs since it is developed in CUDA. GeMTC also uses queues to hold a bag of tasks to be executed on the GPU. In early work, frameworks similar to GeMTC were designed using OpenMP and SCIF to dispatch tasks on to Intel Xeon Phi Co-processor which also used queues with mutex locks and semaphores as the basic data structure for holding bag of tasks.

III. DESIGN CONSIDERATIONS

The goal of Xtask runtime system is to enable execution of fine-grained tasks on shared memory multi-core architectures with very low latency and high throughput. To achieve this goal, analysis of basic data structures that form the barebones of the runtime is essential so they do not become the bottleneck for performance.

A. Analysis of Synchronization Mechanisms

To run programs in parallel using multithreading, threads need to be synchronized. Various thread synchronization mechanisms exist which ensure that threads do not simultaneously execute a critical section of the program. Common synchronization mechanisms include mutexes (mutual exclusion locks), semaphores, reader/writer locks, condition variables and atomic builtins. While it is essential to synchronize data between threads, it can easily get costly at higher levels on concurrency. Figures 2, 3 and 4 show that all synchronization mechanisms exhibit higher latencies due to contention at higher levels of concurrency. These benchmarks are obtained by running a tight loop of operations for a specific period of time and collecting the aggregate of the results. These benchmarks used RDTSCP as the timing function.

Vanilla shared represents the latency of incrementing a shared variable which is not thread safe. Vanilla unique represents the cost of incrementing a variable local to a thread which is not shared between any other threads. At full cpu utilization, the average cost of atomic operations is around 2500 CPU cycles for Haswell and Xeon Phi co-processor and the cost is around 9000 cycles for AMD processor. The latencies increase as the systems are over-provisioned by increasing the number of threads. There are many factors that impact the cycle counts like cache coherence, communication latency between cores on different sockets, interrupts, cache misses, etc. Hence, it is important to run multiple iterations of these benchmarks and to compute the average number of cpu cycles to estimate the latency of these operations.

B. Analysis of Queue Implementations

A data structure like queue can be made thread safe by using the synchronization mechanisms mentioned above. The most common way is to use locks such as mutexes and semaphores. Queues can also be implemented as lock-free data structures using atomic builtins that are supported by hardware. Squeue is a simple ticket lock based lock-free queue that uses atomic operations. Enqueue operation gets an empty slot in the queue and copies the data in that slot. Dequeue operation gets a slot in the queue and waits in a while loop until an element comes in. If an element is already present in the slot, there is no wait time in the while loop. Basic linux queue is a queue used by Linux kernel and it was made thread safe using mutex locks. Read Copy Update (RCU) and LIBLFDS are open-source libraries for lock-free queues. Another variation of data structures for holding a bag of tasks is to have multiple queues with one producer and one consumer pair for each

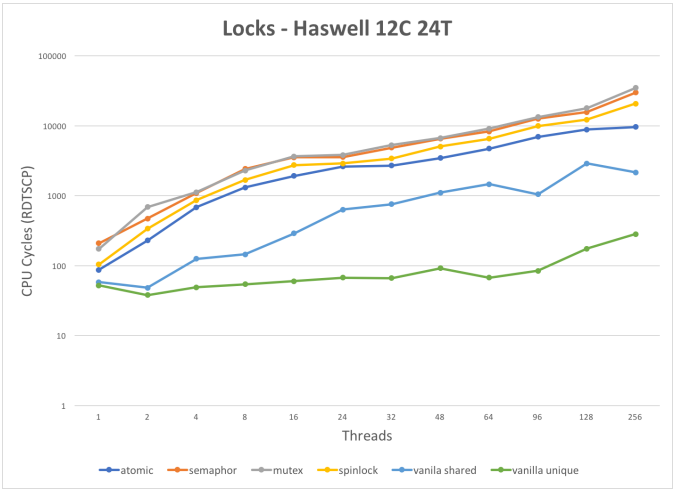


Fig. 2: Latency of synchronization mechanisms on 12C 24T Haswell

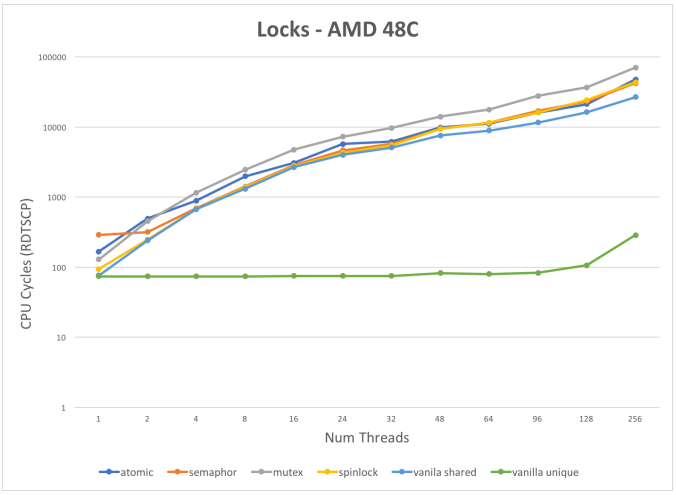


Fig. 3: Latency of synchronization mechanisms on 48C AMD

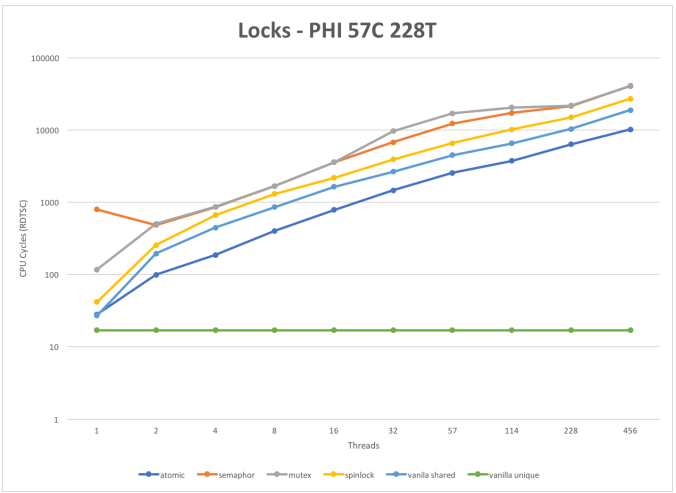


Fig. 4: Latency of synchronization mechanisms on 57C 228T Intel Xeon Phi

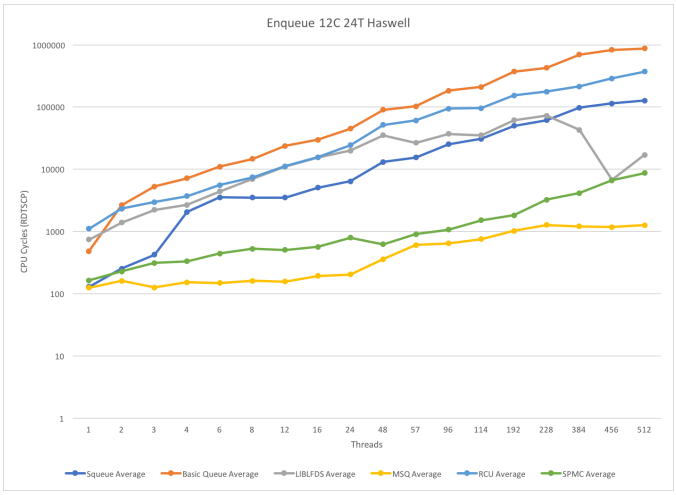


Fig. 5: Latency of enqueue operation on 12C 24T Haswell

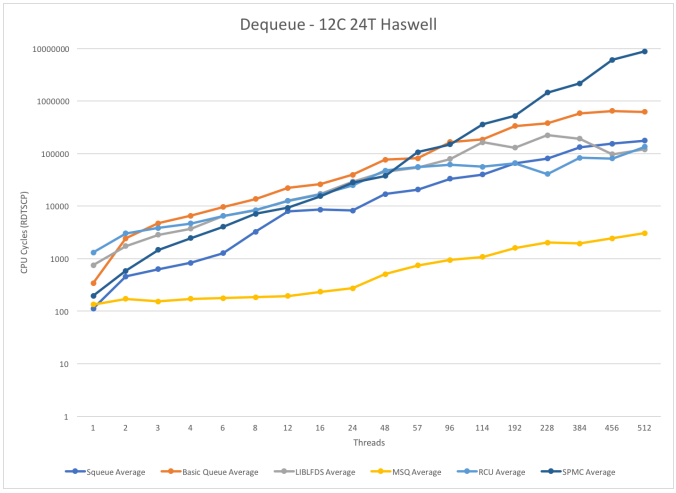


Fig. 6: Latency of dequeue operation on 12C 24T Haswell

queue. Bounded multiple producer multiple consumer is a common model used in many runtime systems and fine-tuning the performance of queues is essential for achieving the best performance.

The data structures mentioned above are benchmarked for latency and throughput. Benchmarking is done using a tight loop of continuous enqueues and dequeues. In some experiments, enqueues follow the dequeues and in others, enqueues and dequeues happen concurrently. Latency is measured in CPU cycles using RDTSCP instruction on Haswell and AMD and RDTSC on Xeon Phi. The test bed for experiments included Haswell with 12 cores and 24 hardware threads, AMD with 48 cores, Xeon Phi Knights Corner with 57 cores and 228 hardware threads and AMD Ryzen with 8 cores and 16 hardware threads. AMD Ryzen is a new architecture with highest clock rate of 4.0 GHz frequency. AMD Ryzen exhibits lowest latencies compared to other x86 architectures given its high clock rate. Results in figures 5 to 12 represent the average latency of few millions of operations.

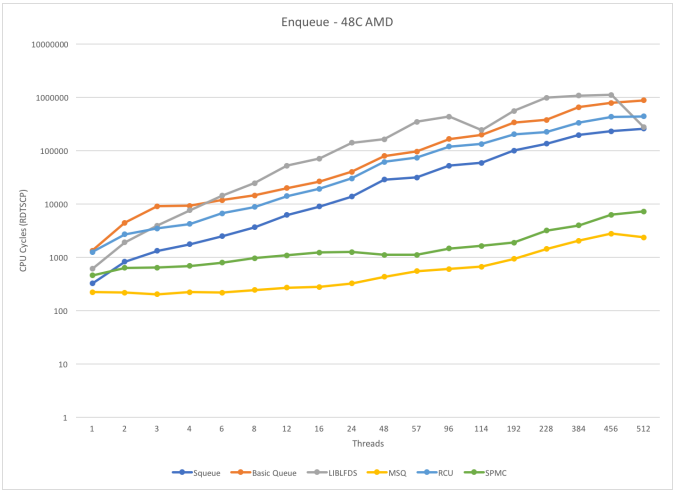


Fig. 7: Latency of enqueue operation on 48C AMD

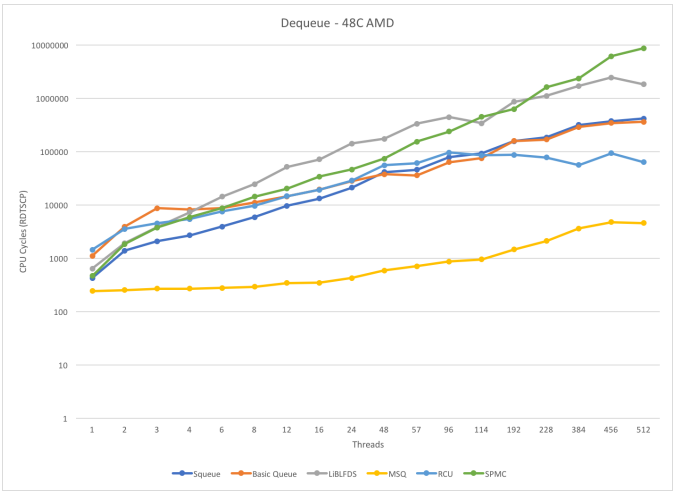


Fig. 8: Latency of dequeue operation on 48C AMD

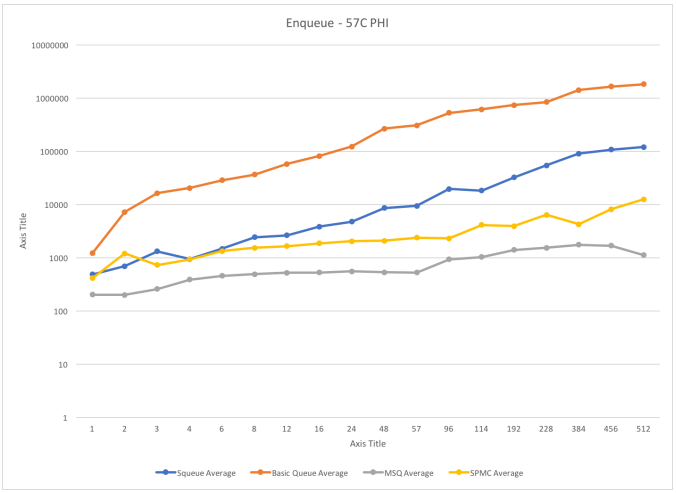


Fig. 9: Latency of enqueue operation on 57C Phi

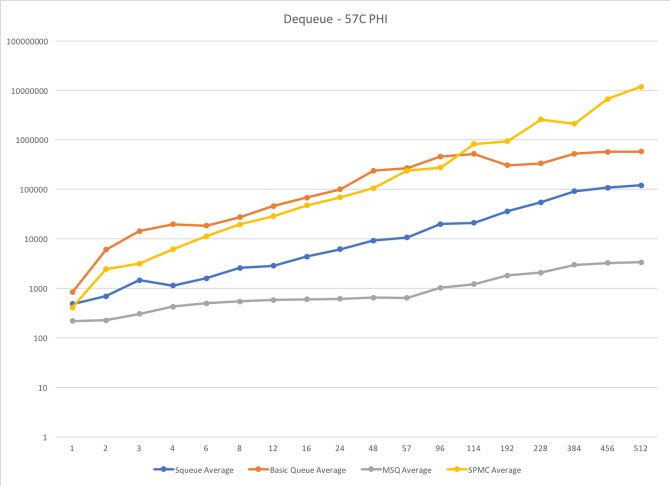


Fig. 10: Latency of dequeue operation on 57C Phi

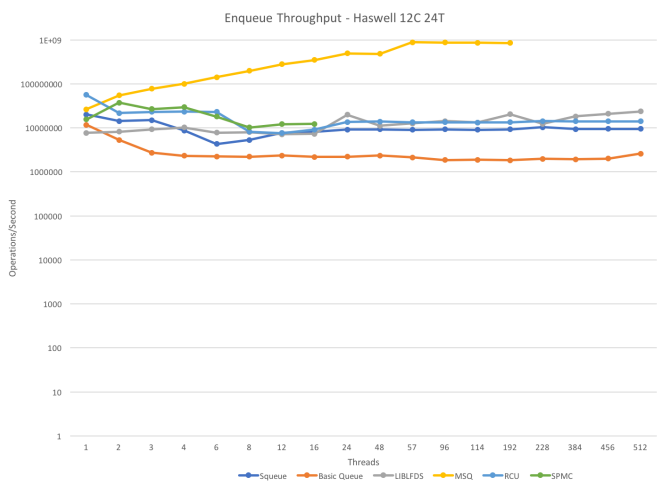


Fig. 13: Throughput of enqueue operation on 12C 24T Haswell

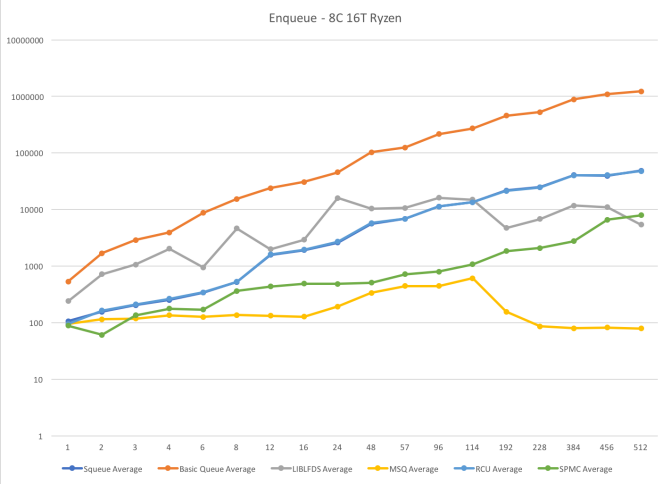


Fig. 11: Latency of enqueue operation on 8C 16T AMD Ryzen

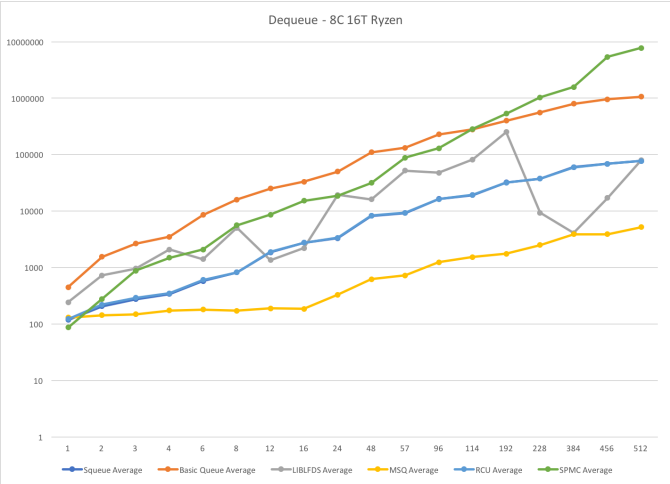


Fig. 12: Latency of dequeue operation on 8C 16T AMD Ryzen

Dequeue operation has higher latency compared to enqueue operation due to waiting in the while loop for an element to come in. This implementation can be improved to reduce the overall latency. However, with current implementation, the benchmarks ensure that queue is full most of the time. Hence the results presented here can be considered as the best average case values.

Throughput is an important metric when it comes to measuring the performance of a runtime system. Throughput of various queue operations can provide a baseline of how many tasks per second can be pushed in and out of the queue in a second. Throughput is measured by running a tight loop for few seconds and measuring the number of operations processed per second. As seen in latency measurements in figures 5 to 12, since latency per operation increases as concurrency increases, throughput remains constant for most implementations except multiple queues approach. With multiple queues, since contention remains the same as threads are scaled up, latency stays constant and higher throughput is obtained with threads equal to number of cores. However as we scale up the threads, due to overhead of context switches, latency and throughput suffer.

While most of the lock-free queue implementations provide 10 million operations per second in throughput, multiple queues approach has an order of magnitude better throughput. Although the caveat here is to distribute the load across multiple queues so no workers are idle waiting for work. This is currently done using naive round robin distribution, however work stealing can be implemented to further improve the performance. Intel Xeon Phi also provides throughput from 6 million to 8 million when all the cores are utilized. However, an important point to note here is that basic queue with locks provide an order of magnitude less throughput compared to other lock-free implementations.

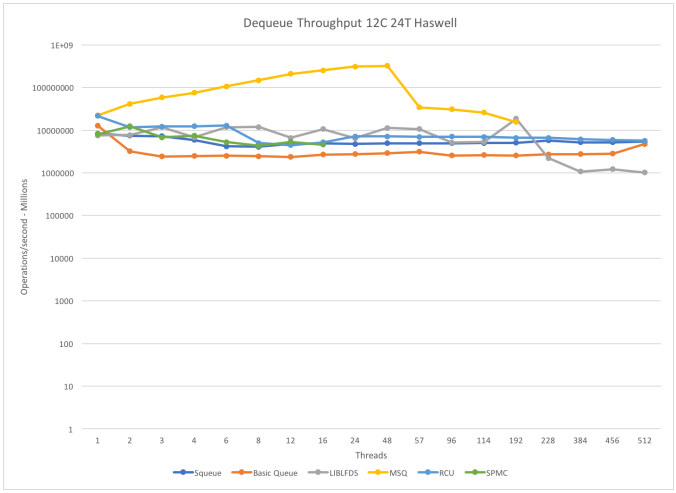


Fig. 14: Throughput of dequeue operation on 12C 24T Haswell

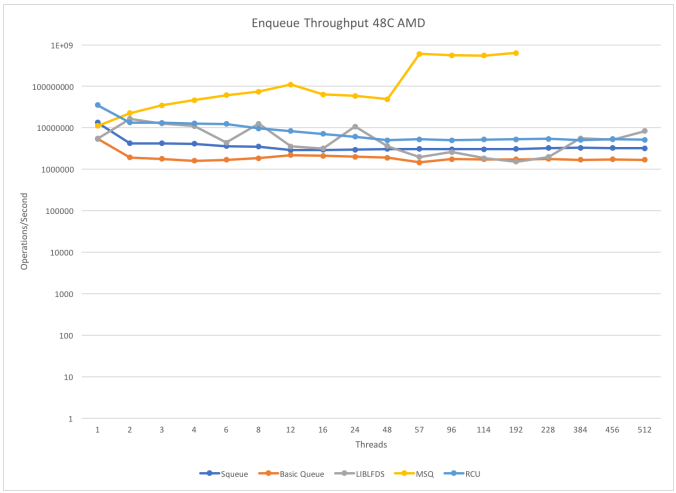


Fig. 15: Throughput of enqueue operation on 48C AMD

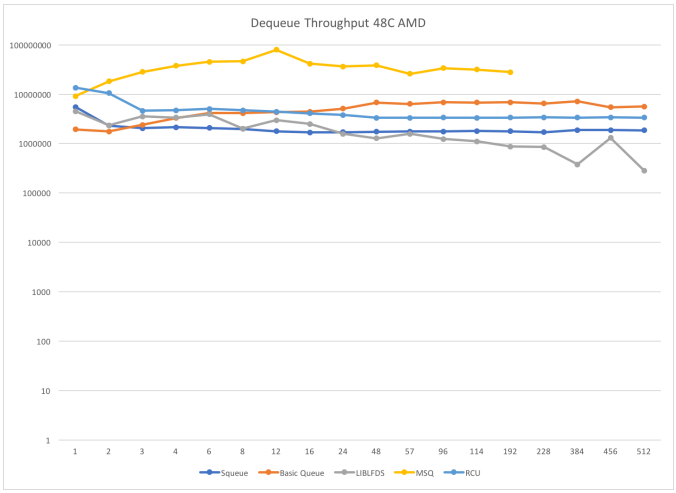


Fig. 16: Throughput of dequeue operation on 48C AMD

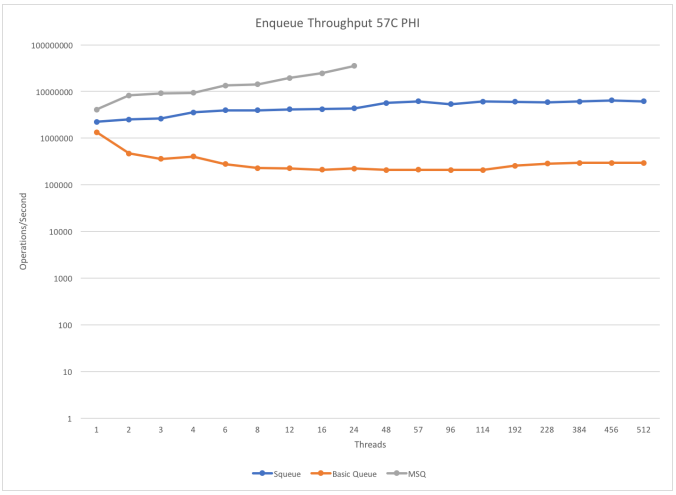


Fig. 17: Throughput of enqueue operation on 57C Phi

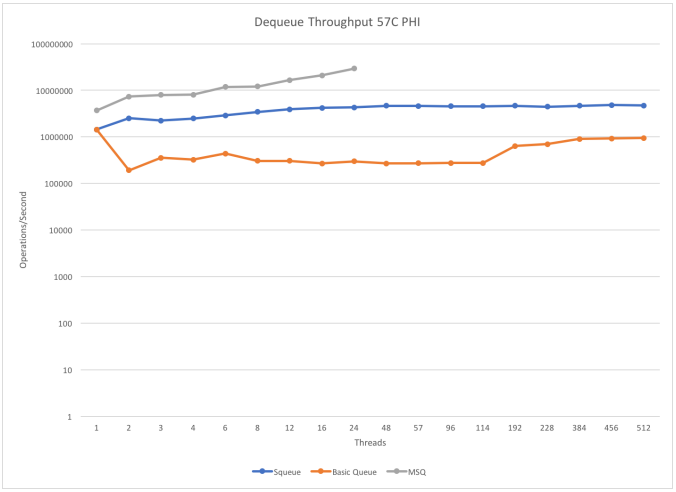


Fig. 18: Throughput of dequeue operation on 57C Phi

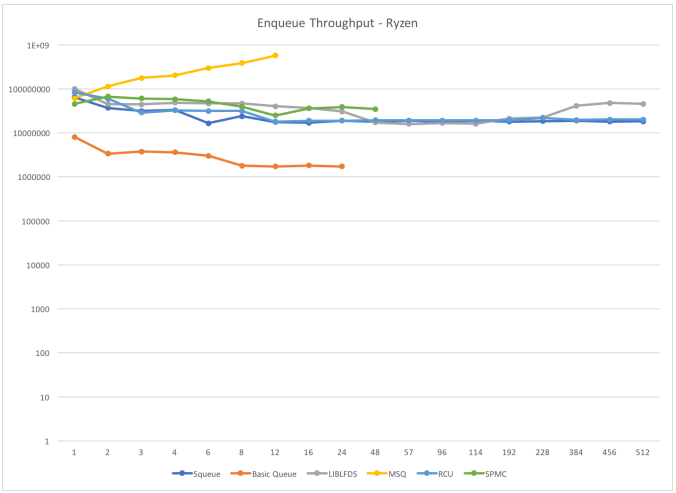


Fig. 19: Throughput of enqueue operation on 8C 16T AMD Ryzen

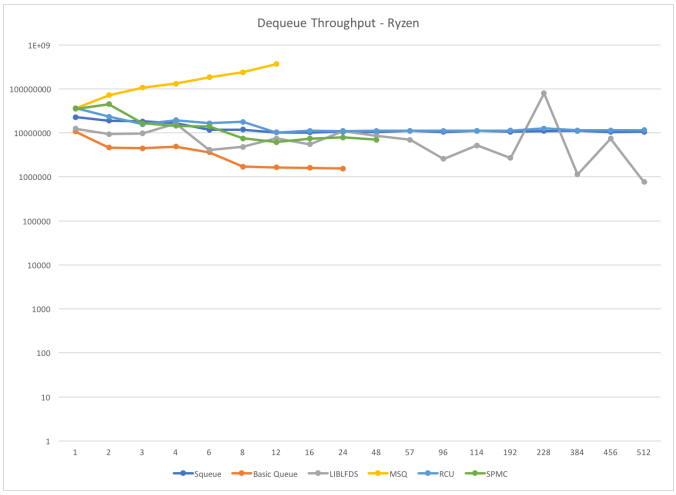


Fig. 20: Throughput of dequeue operation on 8C 16T AMD Ryzen

IV. DESIGN AND IMPLEMENTATION OF XTASK

The Xtask framework employs a producer consumer architecture. Underlying data structure is a bounded lock-free queue which is implemented using ticket lock algorithm. Number of producers and consumers is configurable at runtime along with the queue size. When the framework bootstraps, it allocates memory for the queues using malloc. A pool of worker threads are launched waiting to consume tasks from a queue. Currently the framework uses pthreads as workers. Two versions of Xtask exist, one which processes independent tasks (it will be called XtaskS for the purposes of discussion in this paper) and other which handles dependencies within tasks (it will be called XtaskD for the purposes of discussion in this paper). XtaskS and XtaskD together will be called Xtask in the remainder of this paper. XtaskS treats all the tasks in the queue as independent and schedules them for execution. Results are put in another queue so they can be retrieved later by the program. Since this framework is written in C, it runs on Intel Xeon Phi which also has x86 architecture. In preliminary stages of this work, the runtime also included the ability to offload tasks on to Intel Xeon Phi for processing using OpenMP constructs. Sleep and NOP tasks are used for benchmarking of this framework.

Swift/T is an implicitly parallel data-flow language which uses MPI for inter-node as well as intra-node communication. Pthreads have less communication overhead as compared to an MPI implementation. However, to be able to replace lower layer of Swift/T with a threaded model, it is essential for XtaskS to be able to deal with task and data dependencies. A new model of the framework XtaskD was implemented which could deal with task dependencies. In many existing runtime systems, DAG is generated dynamically as execution progresses. Each task can create sub tasks which also creates dependencies using in, out or inout parameters. In some parallel runtimes, dependency is expressed explicitly by the programmer using pragma or some constructs which require a special compiler. XtaskD uses a different approach than traditional runtime systems. XtaskD requires a DAG to be the input to the runtime system. Leaf tasks that do no depend on any other tasks are readily pushed on the queue for execution. Dependencies can be expressed using a structure which has task dependency information. When the dependencies are resolved, the tasks will be added to the queue for execution. This design was chosen so the runtime system can make informed decisions by knowing the structure of the whole DAG before execution starts. The disadvantage of this approach is that the burden of generating task graph is on the programmer, however this can be addressed as part of future work. XtaskD framework is currently under development and work needs to be done to optimize it for lower latencies and higher throughput. Fibonacci program was implemented on top of XtaskD framework for the purposes of evaluation. Currently, since the framework is in initial stages of implementation, it only supports computing one integer as the output. However, it can be easily extended and will be extended to support multiple outputs as part of future work.

V. OPTIMIZATIONS

There are multiple points to consider when implementing or benchmarking systems on multicore processors for high performance. The way operating systems bind threads and the way they move threads between cores has great impact on multicore performance. Threads needs to be pinned to the cores to achieve optimal performance. Hence Xtask employs a topology aware thread pinning to improvise performance. A tool for hardware locality called HWLOC is used to identify the CPU architecture and the placement of cores. HWLOC provides constructs to get information about the topology, placement of NUMA nodes, IO devices and how the numbering of physical cores.This information is retrieved during bootstrap of the runtime system and threads are pinned to CPUs to achieve maximum performance. Below is a figure generated by HWLOC tool on a Haswell machine.

VI. EVALUATION

This section presents evaluation on XtaskS and XtaskD runtime systems. XtaskS was benchmarked with two implementations of queues, one being the basic linux queue implemented using mutexes and the other being Squeue which was implemented using atomics. Prior evaluation of lock-based queues showed that the latencies are higher at high levels of concurrency and lock-free queues are a better alternative to obtain lower latency and higher throughput. Sleep jobs and NOP tasks are used to evaluate the framework. Micro benchmarks for such fine-grained tasks can be obtained by using RDTSC/RDTSCP instructions. These instructions are available in most architectures and there is subtle difference between the two versions. RDTSCP flushes the CPU pipeline before reading the time stamp counter which is not the case with RDTSC. This is necessary to ensure the operation in flight has completed execution to get an accurate benchmark. However, the caveat to using RDTSCP is that

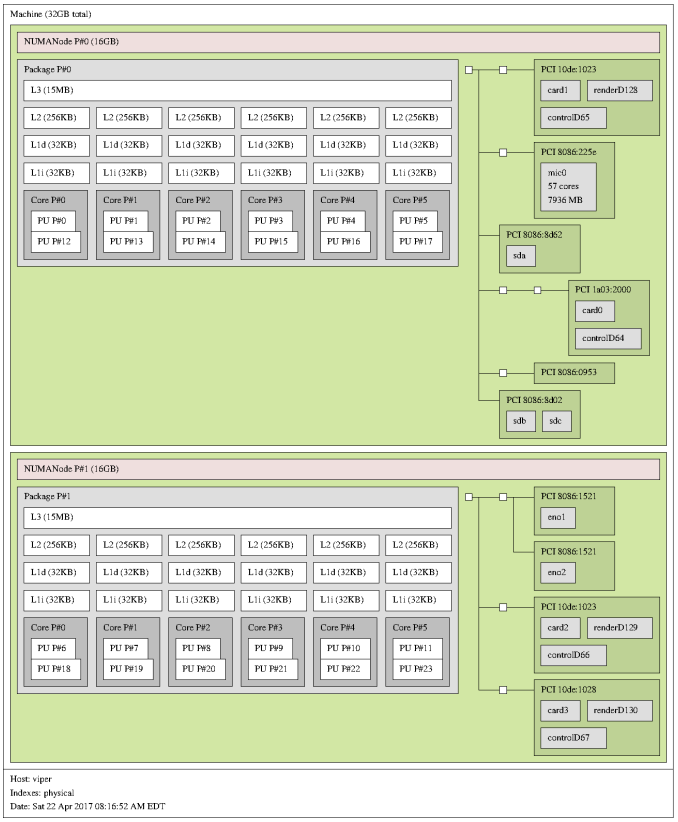


Fig. 21: Topology of a Haswell Machine with 12 Cores and 24 Hardware Threads using HWLOC

subsequent instructions can start execution while time stamp counter is being read which can skew the benchmark results. Hence a combination of CPUID + RDTSC can be used as the start timer which ensures all instructions before have been executed and RDTSCP can be used as the end timer. Also, every core has it's own time stamp counter (TSC). Hence careful measurement is required to synchronize the the counters to get an accurate measurement. Also,when collecting raw data for benchmarking, it is important to make sure data collection does not perturb the execution of the program in a way that it could affect the contention. Extra attention is paid to the way data is being collected for micro-benchmarks since the measurements range in few hundreds to few thousand cycles and any major data collection can perturb the overall execution thereby skewing the results. Figures 21 and 22 show the latency of sleep and nop operations using single producer single consumer (SPSC) lock-based queue as the underlying data structure. Latencies go up as level of concurrency increases as expected and seen the queue behavior. The latencies are higher with a lock-based queue measured to be around 30k cycles at 24 hardware threads on Haswell. Throughput is measured for SPSC version of framework with lock-based queues which averaged around 100 thousand tasks per second on all the architectures.

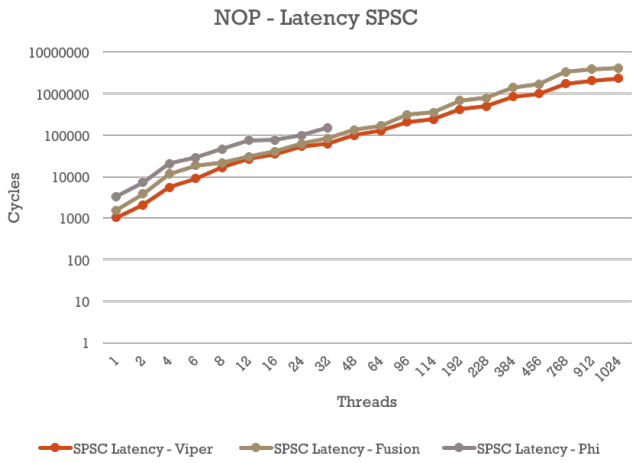


Fig. 22: Latency of Xtask using Basic Linux Queue

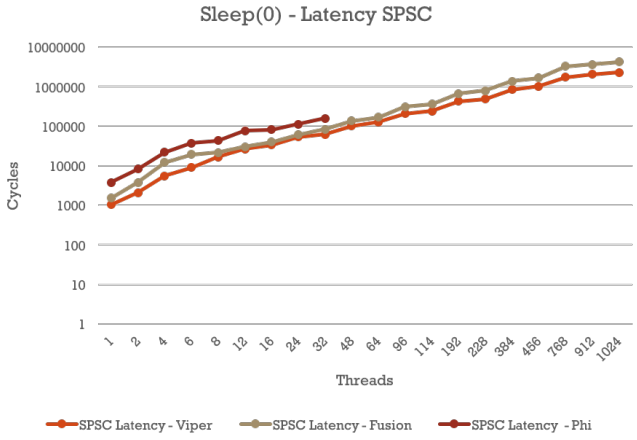


Fig. 23: Latency of Xtask using Basic Linux Queue

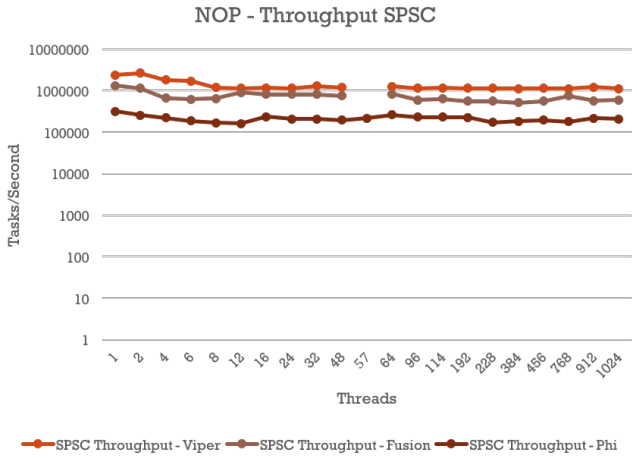


Fig. 24: Latency of Xtask using Basic Linux Queue

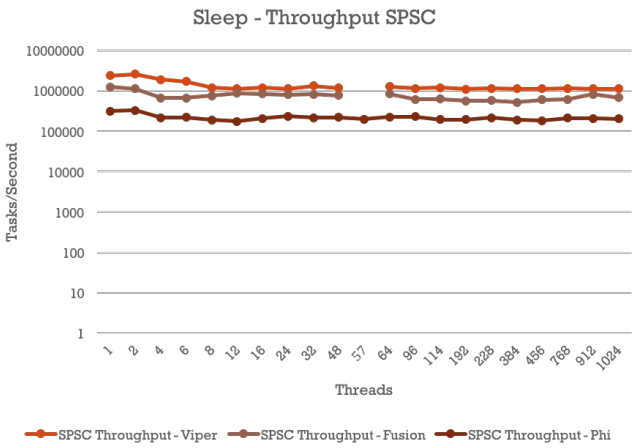
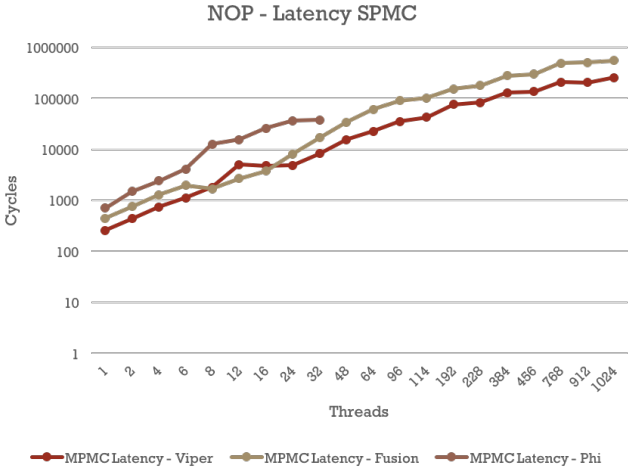
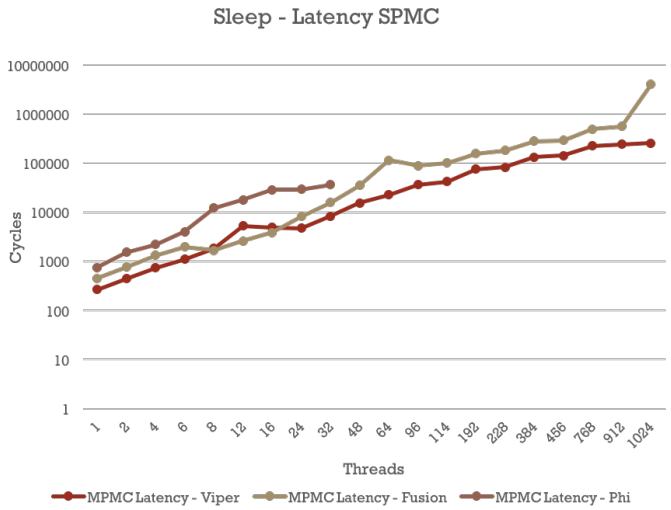


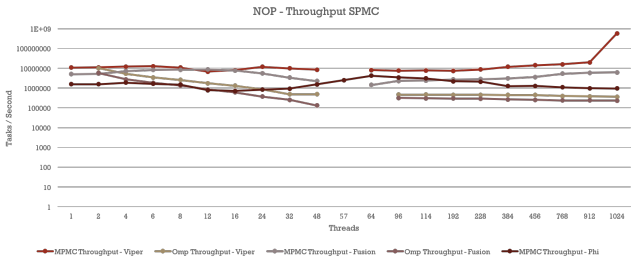
Fig. 25: Latency of Xtask using Basic Linux Queue

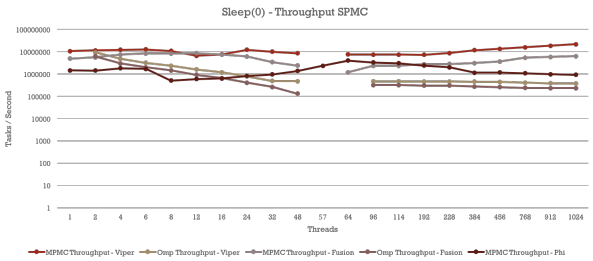
Single producer multiple consumer (SPMC) is a variation that is used in real world applications. There is a single thread that produces tasks and multiple threads processing them as and when they are ready. This was chosen for evaluation where producer would put tasks in the queue in a round robin or random fashion. A challenge with this approach is that the throughput of the queue is limited by the speed of producer being able to push tasks on to the queue. This is quite a possibility with real world application like Fibonacci where a single thread could be producing the tasks while multiple consumers could be consuming the tasks. Below is the evaluation of SPMC variation with lock-free queues. Latencies were measured to be around 4k cycles and throughput around 1 million tasks per second to dequeue, execute and enqueue the result.





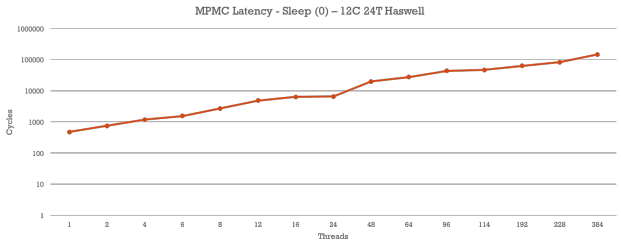
OpenMP is a parallel runtime which can automatically parallelize sections of code within pragmas specified by the user. The task construct of OpenMP [11] resembles bag of tasks pattern and using which we can implement an SPMC version of the program. Since OpenMP is widely used to parallelize code within single node, it was used as comparison in the experiments. OpenMP showed throughput which is comparable to basic queue implementation with locks with values ranging from 60K to 100K. Hence there is certainly room for improvising the OpenMP runtime to get better throughput. OpenMP uses queues based on ticket locks and it can be made better to reduce the overall latency of the operations.





To summarize, Xtask achieved a throughput of 10 million tasks per second with full utilization of all the hardware threads. While OpenMP version started at 10 million tasks per second, it dropped as threads were scaled up to match to the number of hardware threads. This behavior shows that there is room for improvement in OpenMP which is also part of our research focus. Final variation of Xtask version was made to use multiple queues with multiple producers and

multiple consumers. This version performed the best when evaluation was done on queue operations. The motive here is to leverage the advantages of using multiple queues approach and evaluate the approach with naive load balancing to get an idea of the performance. Below are the preliminary results obtained which needs to be investigated to validate the results.



Swift/T uses MPI for sending and receiving tasks within a node and between various nodes. MPI send/receive latency is measured to understand the performance of Swift on a single node and to provide a comparison with a naive pthread implementation. Figure below shows the results obtained for throughput of simple MPI send/receive pairs. With one MPI process, throughput achieved is around 3 to 4 million, however as number of processes goes up, throughput quickly drops to one million. Hence Swift/T can certainly be improved to achieve better performance on single node which is a work in progress currently.

VII. CONCLUSION AND FUTURE WORK

This paper proposed a thread based runtime and execution framework Xtask which can process tasks with low latencies and high throughput. Various queue implementations and synchronization mechanisms are analyzed and benchmarked to understand the cost associated with these operations at high levels of concurrency. Clearly, lock-free implementations are better choice to achieve high throughput although latency suffers with severe contention. Optimal solution would be to use multiple queues with good load balancing technique like work stealing to achieve maximum efficiency. OpenMP achieved less throughput compared to other queueing mechanisms described in this paper. Comparison with OpenMP showed that our simple framework can provide ten times higher throughput compared to OpenMP. These are preliminary results and there is much room for improvement in the framework which is part of future work. Also we presented another version of Xtask called XtaskD which deals with task dependencies. This is an essential feature of a runtime which needs to implemented fully and evaluated. Immediate goals include evaluating the framework for multiple producers and multiple consumers. Also, plan is to finish implementation of XtaskD to be able to efficiently handle dependent tasks.

Swift/T [7] uses MPI for sending tasks to worker processes within a node. Certainly MPI send receive has higher latency compared to enqueue and dequeue operations of queues. Currently work is being done to replace MPI with threads in Swift/T. Also future research plan is to evaluate various kernel level abstractions like interrupts that can be leveraged to notify workers of new tasks which will be an entirely different approach of building a runtime. End goal is to develop a complete parallel runtime system XTASK which can dispatch tasks with very low latency and very high throughput.

VIII. ACKNOWLEDGEMENTS

I would like to express my gratitude to Dr.Ioan Raicu (Illinois Institute of Technology), Dr. Peter Dinda (Northwestern University) and Dr. Kyle Hale (IIT) for suggesting such an interesting idea to explore user space and kernel space capabilities for designing a parallel runtime system to support fine grained parallelism.

REFERENCES

[1] Poornima Nookala, *"Benchmarking Source Code Repository"*, https://github.com/pnookala/MyWork/tree/master/QueueBenchmarks.
[2] Ioan Raicu, Ian T. Foster, Yong Zhao, *Many-task computing for grids and supercomputers*, IEEE, 2008.
[3] ROBERT D. BLUMOFE, CHRISTOPHER F. JOERG, BRADLEY C. KUSZMAUL, CHARLES E. LEISERSON, KEITH H. RANDALL, AND YULI ZHOU, *"Cilk: An Efficient Multithreaded Runtime System"*, October 17, 1996.
[4] YarKhan, A., *"Dynamic Task Execution on Shared and Distributed Memory Architectures,"*, University of Tennessee, December, 2012.
[5] S. Krieder, J. Wozniak, T. Armstrong, M. Wilde, D. Katz, B. Grimmer, I. Foster and I. Raicu, *"Design and Evaluation of the GeMTC Framework for GPU-enabled Many-Task Computing"*, ACM HPDC, 2014.
[6] Justin M. Wozniak, et al., *Swift/T: Scalable Data Flow Programming for Many-Task Applications*, PPoPP 2013.
[7] J. Johnson, S. Krieder, B. Grimmer, J. Wozniak, M. Wilde and I. Raicu, *"Understanding the Costs of Many-Task Computing Workloads on Intel Xeon Phi Coprocessors"*, GCASR, 2013.
[8] David R. Butenhof, *"Programming with POSIX Threads"*, 1997.
[9] Bryant, O'Hallaron, *"Computer Systems: A Programmer's Perspective"*, 2011.
[10] Allen B. Downey, *"The Little Book of Semaphores"*, http://greenteapress.com/semaphores/.
[11] Allen B. Downey, *"Swift/T"*, https://www.mcs.anl.gov/project/swift-fast-parallel-scripting-language.
[12] J. Johnson, S. Krieder, B. Grimmer, J. Wozniak, M. Wilde and I. Raicu, *"Understanding the Costs of Many-Task Computing Workloads on Intel Xeon Phi Coprocessors"*, GCASR, 2013.
[13] David R. Butenhof, *"Programming with POSIX Threads"*, 1997.
[14] Bryant, O'Hallaron, *"Computer Systems: A Programmer's Perspective"*, 2011.
[15] Allen B. Downey, *"The Little Book of Semaphores"*, http://greenteapress.com/semaphores/.
[16] Allen B. Downey, *"Swift/T"*, https://www.mcs.anl.gov/project/swift-fast-parallel-scripting-language.
[17] *"Computing a Fibonacci number"*, http://www.cs.ucsb.edu/projects/jicos/tutorial/fibonacci/.
[18] *"HWLOC"*, https://www.open-mpi.org/projects/hwloc/doc/v1.11.6/.
[19] *"ADLB"*, https://www.cs.mtsu.edu/~rbutler/adlb/.
[20] *"OpenMP Tasks"*, https://www.kth.se/polopoly_fs/1.224493!/Menu/general/column-content/attachment/openmp_tasks.pdf.