ILLINOIS TECH
College of Computing

Exploring Extreme Fine-grained Parallelism on Modern Many-Core Architectures

PhD Proposal

at Illinois Institute of Technology
at the Department of Computer Science
Data Intensive Distributed Systems Laboratory

Committee Members
Ioan Raicu
Robert Harrison
Jia Wang
Kyle Hale
Stefan Muller

Primary Advisor:	Dr. Ioan Raicu
Advisor:	Dr. Robert J. Harrison (Stony Brook University)
Author:	Poornima Nookala 10 W 31st Street, Chicago IL 60616 Chicago pnookala@hawk.iit.edu
Submission:	6th May 2022

Abstract

Processors with 100s of threads of execution and GPUs with 1000s of cores are among the state-of-the-art in high-end computing systems. This transition to many-core computing has required the community to develop new algorithms to overcome significant latency bottlenecks through massive concurrency. Implementing efficient parallel runtimes that can scale up to hundreds of threads with extremely fine-grained tasks (less than $\sim$100 μs) remains a challenge. We propose XQueue, a novel lockless concurrent queueing system that can scale up to hundreds of threads. We integrate XQueue into LLVM OpenMP and implement X-OpenMP, a library for lightweight tasking on modern many-core systems with hundreds of cores. We show that it is possible to implement a parallel execution model using lock-less techniques for enabling applications to strongly scale on many-core architectures. While OpenMP is suitable for on-node parallelism, it is crucial to support heterogenous architectures and distributed memory environments. The existing parallel programming environments that support distributed memory environments either discover the DAG entirely on all processes which limits the scalability or introduce explicit communications which increases the complexity of programming. We implement Template Task Graph (TTG), a novel programming model and its C++ implementation by marrying the ideas of control and data flowgraph programming. TTG supports compact specification and efficient distributed execution of dynamic and irregular applications. TTG can address the issues mentioned above without sacrificing scalability or programmability by providing higher-level abstractions than conventionally provided by task-centric programming systems, but without impeding the ability of these runtimes to manage task creation and execution as well as data and resource management efficiently. TTG implementation currently supports distributed memory execution over 2 different task runtimes PaRSEC and MADNESS.

Contents

CHAPTER 1

Introduction

The Department of Energy (DOE) has reported that *"Scientific productivity is one of the top ten exascale research challenges [69]"*. The scientific computing community is facing unprecedented changes in computer architectures that has fueled the emergence of the many-core computing architecture. Processors with 100s and GPUs with 1000s of threads of execution are among the state-of-the-art in high-end computing systems. In a recent report [50], the DOE stated that *"the transition of applications to exploit massive on-node concurrency... create the most challenging environment for developing applications in at least two decades."* Extreme on-node concurrency levels of order 10^4 is required in order to achieve exascale performance levels according to this report. They continued by saying *"much of the performance improvement must come from vectorization and lightweight tasking."* These heterogeneous systems provisioned with many-core accelerators fundamentally make programmability harder as we shift from MIMD (multiple instruction, multiple data) programming to a mixture of MIMD and SIMD (single instruction, multiple data) programming. The era of many-core and exascale computing will bring new fundamental challenges in how we build large-scale systems, how we manage them, and how we program them. The techniques that have been designed decades ago will have to be dramatically changed to support the coming wave of extreme-scale general purpose parallel computing.

Today, the increase in performance for single-threaded processor has come to an end due to the limitation of the current Very Large Scale Integration (VLSI) technology. In response, most hardware companies are designing and developing new parallel architectures [40]. To achieve higher performance, applications need to leverage the parallelism on modern architectures. On the other hand, multicore designs are also encountering scaling problems, notably the "Dark Silicon" phenomenon [37]. Power and cooling concerns suggest the number of dynamically active transistors on a single die may be greatly constrained in the near future. In other words, even if the number of transistors per chip continues to follow Moore's law, we will not be able to use all of them simultaneously. This problem may lead to scenarios in which only a small percentage of the chip's transistors can be "on" at a time [98]. The limitations of current CMOS technology has fueled the emergernce of many-core architectures and many of these massively parallel platforms offer a high ratio performance/cost and an efficient power consumption design [109, 112, 108]. They are also widely used in high performance computing, including systems ranging from a cluster of personal computers, to large scale supercomputers. As per the Top 500 list [102], many of the most powerful supercomputers today are based on platforms that combine multicore and manycore processors with data parallel accelerators.

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. When a task is created by some processor/thread, it is conceptually queued for execution by a future available thread. Task dependencies and/or 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 can dynamically unfold during the execution of the application. Given the DAG, tasks can be executed using a set of processors/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.

Figure 1 shows a DAG with a set of tasks with arrows showing the dependencies. Nodes at one level can ideally be executed in parallel. Here tasks D,E, G and H can be executed in parallel and they do no have dependencies since they are the leaf tasks. Once the dependencies for F have been resolved, task F can execute. The execution models of many parallel languages and libraries [84, 9, 120, 119, 38, 104] rely on such task parallelism.

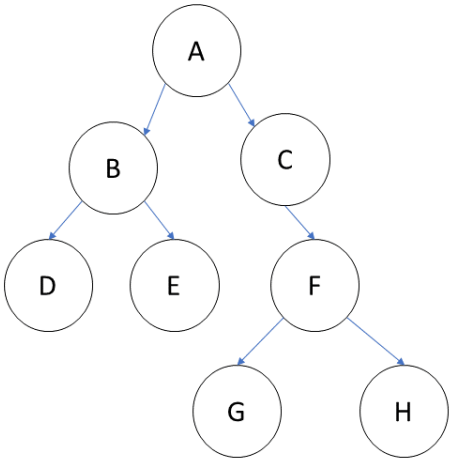


Figure 1 Directed Acyclic Graph (DAG)

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 managing the tasks. Hence, the need for low overhead tasking becomes significant in order to explore extreme parallelism from applications.

Many-Task Computing (MTC) [88] has been an emerging paradigm and area of research for some years now. An MTC workload consists of tasks 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 execution models that can leverage full capabilities of the processors by over-decomposition of tasks into fine-grained tasks is increasing.

1.1 Early Work in Many Task Computing

GPUs have a very restrictive programming model, but provide at least an order of magnitude better throughput for applications painstakingly coded to that model. To program GPUs, typically there is a need to learn another programming language such as CUDA (NVIDIA) or OpenCL (AMD). As a result,

existing vendors must spend extra time and effort to modify or rewrite parts of their codebase to take advantage of the new capabilities provided by General Purpose GPUs (GPGPUs). Besides that, barely rewriting an application just to offload computations to a GPU rarely works well. Because of the architecture of most GPUs out there, applications must be tailored from the ground up to follow the rules of the restrictive programming model of GPUs, otherwise they may suffer from severe performance penalties. Because of that, interested vendors cannot afford to go through the effort involved. Finally, while GPUs are great for massively parallel applications with thread-switching that comes almost at no cost, their performance can take a large hit when executing programs with complex logic (like complicated branching and looping for example). Therefore they may be unsuitable for certain applications of MTC. The Intel Xeon Phi is a family of processors based on the Intel MIC Architecture [56] that incorporates earlier work on the Larrabee architecture [95]. It follows an alternative programming model that, although may not provide the same level of parallelism, provides more flexibility and therefore can be more suitable for certain application of MTC that GPUs are not suited for. The reason is that the Xeon Phi has x86 cores that are more capable (can handle complex branching and looping) than most GPU cores. Another advantage of having x86 cores is that programming the co-processor minimizes the amount of work that needs to be done in order to integrate a Xeon Phi to an existing system. That is because the Phi does not require being programmed in any specific framework and it can natively run applications written in C with Pthreads or OpenMP. This work used the 22nm Knights Corner chip, which was the first commercial product from this family. This product has been discontinued due to the problems with 10nm technology and we briefly discuss our findings from using this chip.

The Knights Corner is a PCIe vector co-processor with integrates up to 61 in-order dual issue x86 cores, which trace some history to the original Pentium core, like the Larrabee predecessor. Among other enhancements, the Corner's cores are augmented with 64-bit support, 4 hardware threads per core (resulting in more than 200 hardware threads available on a single device) and 512-bit SIMD instructions [56]. Each core has a 512KB L2 cache locally but has also access to all other L2 caches in the system through a high-speed bidirectional ring [56]. Unlike previous GPUs, the L2 cache is kept fully coherent by a global-distributed tag directory.

Due to the foundations of Intel architecture, the coprocessor can be programmed in several different ways. We implemented two different frame-

work for handling MTC workloads - OpenMP and SCIF (Symmetric Communications Interface). SCIF is a low-level API implemented by Intel for Xeon Phis for communications across PCIe. For the OpenMP implementation, we used offloading approach for offloading computations from host to the Phi. For the SCIF part, we implemented the framework to run natively on the Phi while accepting jobs from clients running on the host CPU [6]. The major advantage of native execution coupled with SCIF over offloading is that the developer gets more control overall in the configuration and the architecture of their design in order to maximize performance. In addition, different MIC cards can communicate directly with each other basically making certain designs more efficient. The OpenMP version of the framework is developed using a Producer-Consumer architecture which communicates using shared memory for IPC. The Consumer side hosts the framework which runs as multiple worker threads which use the shared memory space as a queue structure, continuously accepting new tasks from producer. Likewise, the producer acts as a client process which submits tasks to the queue. Asynchronous offloading is used to allow the framework to be non-blocking to continue accepting tasks while other tasks are running on the Phi. After submitting the job, the clients can request the result and the server will deliver it to them when the task has finished processing. The whole procedure is non-blocking for the server who can handle multiple requests and submissions at the same time.

To analyze the performance of our implementations, we ran a simple task-based matrix multiplication benchmark on the Midway High Performance Computing Cluster at the University of Chicago. Our testing host is an Intel SandyBridge with 16 cores at 2.6 Ghz and 32 GB of RAM. It has 2 Xeon Phis from the Knights Corner family attached to it. We calculated speedup compared to the sequential version and observed higher performance with fine-granular tasks, but the gain reduces as problem scales up to higher matrix sizes. The overheads of communication via PCIe interface quickly became evident as we scaled up the problem sizes. We also analyzed the performance of sleep(0) tasks to assess the ideal performance of Xeon Phi with very short length tasks. We achieved an efficiency of 90% with tasks lasting as short as 640 microseconds.

1.2 Summary

These preliminary observations motivated us to further explore many-core architectures and fine-grained tasking with the goal to reduce the underlying overheads of the existing parallel runtime systems and to explore extreme fine-grained parallelism. Parallel execution models typically uses concurrent data structures like queues to hold a bag of tasks. We shifted our focus to analysing the performance of concurrent queues to understand the overheads of managing tasks in task-based runtime systems.

CHAPTER 2

How Bad Is Concurrent Queue Performance Across Many Threads?

Concurrent data structures have to deal with data synchronization and communication between threads. Synchronization mechanisms like mutexes, semaphores and spinlocks are known to have significant overhead and can easily become the bottleneck to achieving high performance. Many researchers have proposed better performing lock-free data structures using atomic instructions supported by hardware. Many programming languages like Java and C++ implement lock-free data structures. Many libraries exist for these languages [62] [89] [10], that take advantage of the instructions that hardware can support for implementing high-performance data structures. However, as we move towards many-core architectures, lock-free techniques do not scale well due to mutual exclusion and high contention on memory bus resource.

2.1 Baseline Queue Performance

A single producer single consumer (SPSC) array-based queue provides the lowest latency for enqueue and dequeue operations when both operations do not happen simultaneously since they do not require data synchronization, thread to thread communication and can benefit from data locality. In order to parallelize applications, concurrent queues are necessary for shar-

ing work among various threads and a multiple producer multiple consumer (MPMC) queue is the most commonly used data structure. Thread contention, data synchronization, cache coherence and cache misses are few of the many factors that can highly impact the performance of MPMC queues limiting their scalability.

Table 1 Testbed for evaluation from the Mystic System

Machine	Model	Sockets-Cores/HT@Freq
skylake-192	Intel Xeon Gold 8160	8-192/384@2.1GHz
skylake-48	Intel Xeon Gold 8160	2-48/96@2.1GHz
skylake-32	Intel Xeon Gold 6130	2-32/64@2.1GHz
skylake-16	Intel Xeon Silver 4110	2-16/32@2.1GHz
phi-64	Intel Xeon Phi 7210	1-64/256@1.5GHz
broadwell-16	Intel Xeon E5-2620 v4	2-16/32@2.1GHz
haswell-12	Intel Xeon E5-2620 v3	2-12/24@2.4GHz
epyc-64	AMD Naples 7501	2-64/128@2.0GHz
theadripper-32	AMD Threadripper 2990WX	1-32/64@3.0GHz
ryzen-8	AMD Ryzen 7 1700	1-8/16@3.0GHz
opteron-48	AMD Opteron 6168	4-48/48@1.9GHz
power9-40	POWER9 EP73	2-40/160@3.8GHz
thunderx-96	ThunderX 88XX ARM v8	2-96/96@2.0GHz

In order to show the scalability and performance of MPMC queues compared to SPSC queues, we selected five diverse systems (see Table 1) from the Mystic Testbed [85] that represent different architectures with large core counts. The five systems we choose to evaluate for these initial experiments are: 1) AMD Epyc, 2) ARM ThunderX, 3) IBM Power9, 4) Intel Xeon Phi, and 5) Intel Xeon Scalable Processor. More information about these systems (as well as others used in our work) can be found in Table 1.

We measured the latency and throughput of a simple SPSC array-based circular queue to identify baseline numbers for the lowest latency that can be achieved on latest many-core architectures. Experiments involve running 1 billion enqueue operations followed by a sequence of dequeues. We measured the latency of each operation and calculated the average time per enqueue/dequeue pair. Queue size is set to the number of samples for the purposes of this evaluation. Results in Figure 2 show the average latency of both enqueue and dequeue operations. It can be noted that latency of any operation on queues takes 30 to 70 cycles depending on the architecture and clock frequency. This latency measurement includes a check if queue is full/empty, an increment operation on head/tail, a modulo operation on head/tail to get the position in the circular array and a copy operation to add/remove the item. Figure 3 represents the throughput, which is the rate at which items are being processed by the queue. For throughput experiments,

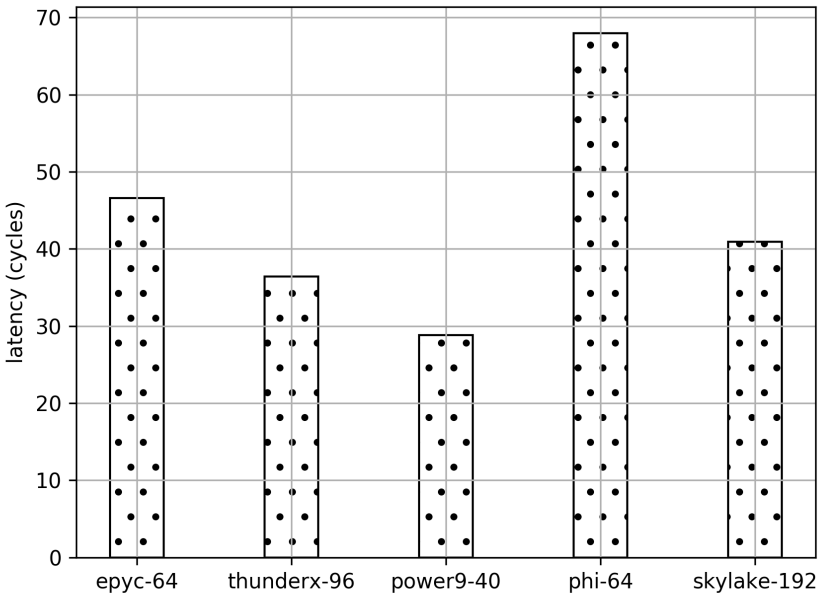


Figure 2 Average latency of enqueue/dequeue operations on SPSC queue

we measured the total time taken for a billion enqueue/dequeue operations and calculated the throughput. Average throughput of enqueue/dequeue operations reaches 270 million operations per second on Intel Skylake 192-core machine. Although these results are significant showing excellent single threaded performance, an SPSC queue is limited because it cannot be used with more than one producer and one consumer.

Figures 4 and 5 depict the results obtained by benchmarking a simple multiple producer multiple consumer queue for latency and throughput. The queue is implemented by using a semaphore which keeps track of free spaces in the queue and `pthread_mutex_lock` to lock the queue during enqueue and dequeue operations. This is the most common and simple way to implement a concurrent queue. We do not expect a single concurrent queue with multiple threads to scale well. This experiment aims at quantifying the poor scalability of MPMC queues using mutex locks. Each experiment enqueues and dequeues one billion items using equal number of producer and consumer threads. For all the experiments, a round robin pinning of threads is employed with producer and consumer thread being on the same core and different hyper threads. Binding threads to processor can result in better cache utilization, thereby reducing costly memory accesses. This thread placement is a result of tests performed by pinning producer to core 0 and

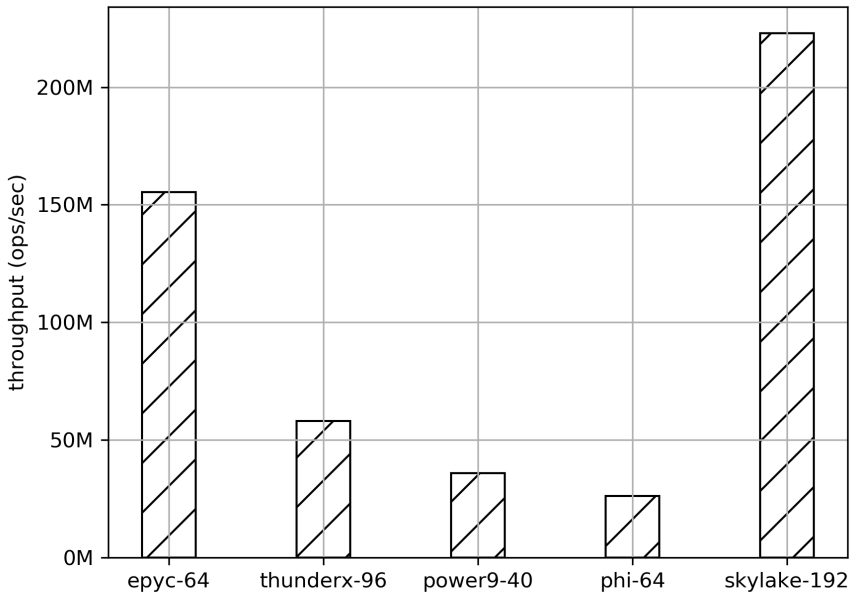


Figure 3 Average throughput of enqueue/dequeue operations in millions(M) on SPSC queue

consumer to each other core available and evaluating the performance obtained for every combination which resulted in separate hyper threads on the same CPU giving the highest performance.

Figures 4 and 5 show the latency and throughput, respectively. Our results indicate that latency can reach up to millions of cycles under high contention, and throughput can drop down to as low as 311,329 operations per second (aggregate over all threads). For the skylake-192 system, which had the best single core performance at 270 million operations/sec, the MPMC approach yielded only 810 operations per second per thread at a 384-thread scale (a 333,333$\times$ loss of performance). The fastest MPMC queue throughput at any scale reached just 5 million operations/sec. These results provide enough motivation to investigate methods to exploit full concurrency on many-core architectures while not compromising on the lowest latency that can be achieved.

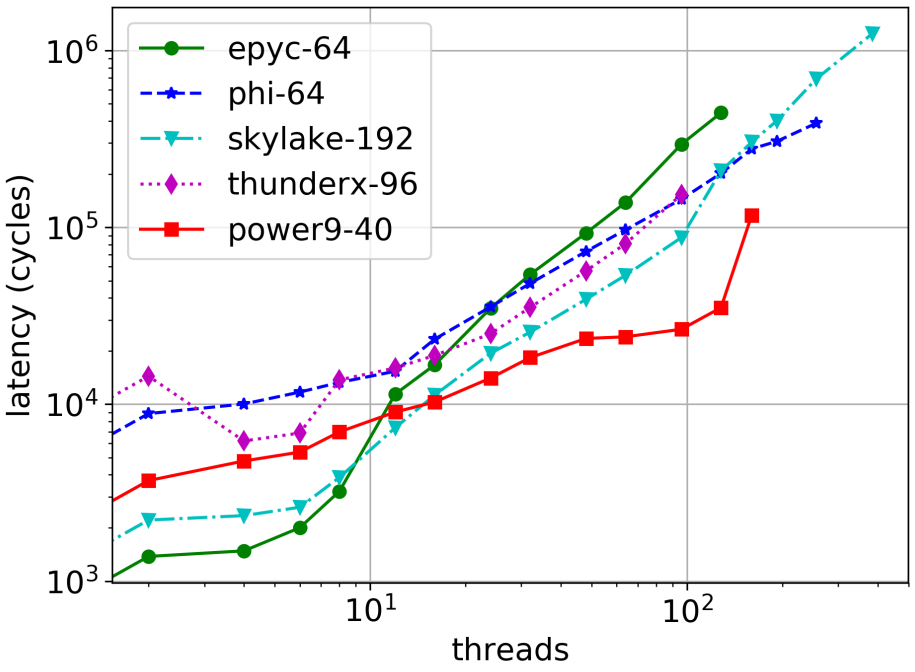


Figure 4 Average latency of enqueue/dequeue operations on a lock-based queue. This graph is shows that simple lock-based queues don't scale beyond 8 threads on any modern processors.

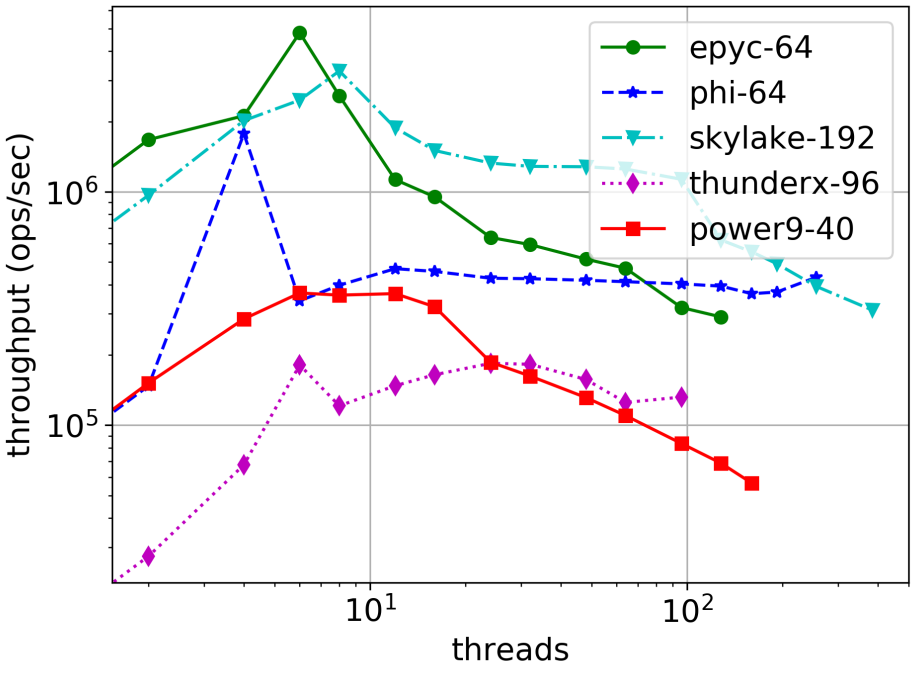


Figure 5 Average throughput of enqueue/dequeue operations on lock-based queue. This graph shows that the throughput of a simple lock-based queue plateaus beyond 8 threads on modern processors

2.2 Analysis of Synchronization Mechanisms

This section conducts a detailed performance study of synchronization mechanisms: 1) mutexes, 2) semaphores, 3) spin locks, and 4) atomic fetch-and-add operations. The evaluation is conducted on a testbed of 13 systems representing today's largest shared-memory systems from Intel, AMD, IBM, and ARM with up to 384 hardware threads.

2.2.1 Tesbed, Software Stack, and Timing Mechanisms

Testbed: Table 1 shows details of the testbed used for experiments in this paper. The testbed covers latest many-core architectures from Intel, AMD, IBM and ARM with processors such as Haswell, Broadwell, Skylake, Phi, Opteron, Ryzen, Threadripper, Epyc, Power9, and ThunderX. The smallest system is an 8-core single socket system from AMD. The largest system is an 8-socket system with 24-core Intel CPUs, for a total of 192-cores and 384 hardware threads. The average system scale is about 50-cores and 100 hardware threads.

Software stack: All experiments in this paper are performed on Ubuntu 18.04 operating system and compiled using GCC version 7.3 with O2 optimization level.

Fine-grained timing: On x86 architectures, latency is measured in CPU cycles using RDTSCP instruction for start time and RDTSC + CPUID instruction for the end time. RDTSCP is a serializing instruction and it prevents instruction reordering around the call. CPUID is also a serializing call and when it follows RDTSC instruction, it prevents any future instructions to be executed before timing information is read. The combination of these two timing functions gives the most accurate results for latency. Timing on ARM and Power9 architectures is quite different from x86 architectures. ARM processor has a PMU cycle counter which is only accessible in privileged mode. The operating system sets up a virtual counter which counts at the same frequency as the physical counter and can be used for fine-grained measurements. The ARM cycle counter ticks at a lower frequency than the frequency that cores are running at and hence calibration is required to get the multiplier that needs to be applied to the cycle count to get a precise value. In Power9, time base register counts cycles at a fixed lower frequency and needs to be calibrated to convert the value to actual cycles at CPU clock frequency. Throughput in all experiments in this paper

is measured using `CLOCK_MONOTONIC` for start and end times. Throughput is calculated for each thread individually and all the results are aggregated to get the final throughput value for the experiment.

2.2.2 Performance of Synchronization Mechanisms

In order to program for shared-memory systems 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. Many languages provide high level abstractions for synchronization to ease parallel programming. Common synchronization mechanisms include mutexes (mutual exclusion locks), semaphores, reader/writer locks and condition variables. Mutex is a mutual exclusion lock which ensures exclusive access to the shared resource. Spinlock is a type of lock which waits in a busy loop if lock cannot be acquired. Atomics operations are instructions supported by hardware and they lock the memory bus to access the shared resource. These operations are inherently atomic and have limited support for data types on various architectures. Semaphores is a type of mutual exclusion where a thread can wait to get access to the critical section or do a post so other threads can get access.

While it is essential to synchronize data between threads, it can easily get very expensive at higher levels of concurrency. This is due to the reason that only one thread can hold exclusive access to the critical section and all other threads are waiting to get the lock using up CPU cycles. Lock-free approaches using atomic operations are believed to be highly efficient, but are hard to implement and maintain. Lock-free algorithms can be implemented by using special hardware primitives such as CAS (compare and swap), FAA (fetch and add) and LL/SC (load-link/store conditional). Most implementations of mutexes are built on top of atomic instructions supported by hardware.

The primary focus here is to analyze the cost of low-level thread synchronization mechanisms and for this purpose, we benchmarked `pthread_mutex_lock`/ `pthread_mutex_unlock`, `sem_wait`/ `sem_post`, `fetch-and-add` and `spin_lock`/ `spin_unlock` to measure latency. Spinlock for this benchmark is implemented using test-and-set algorithm using CAS atomic primitive. Fetch-and-add is supported by x86 architectures using 'lock xadd' instruction. The Power9 variant for FAA instruction is 'lwarx/stwcx' and ARMv8 provides

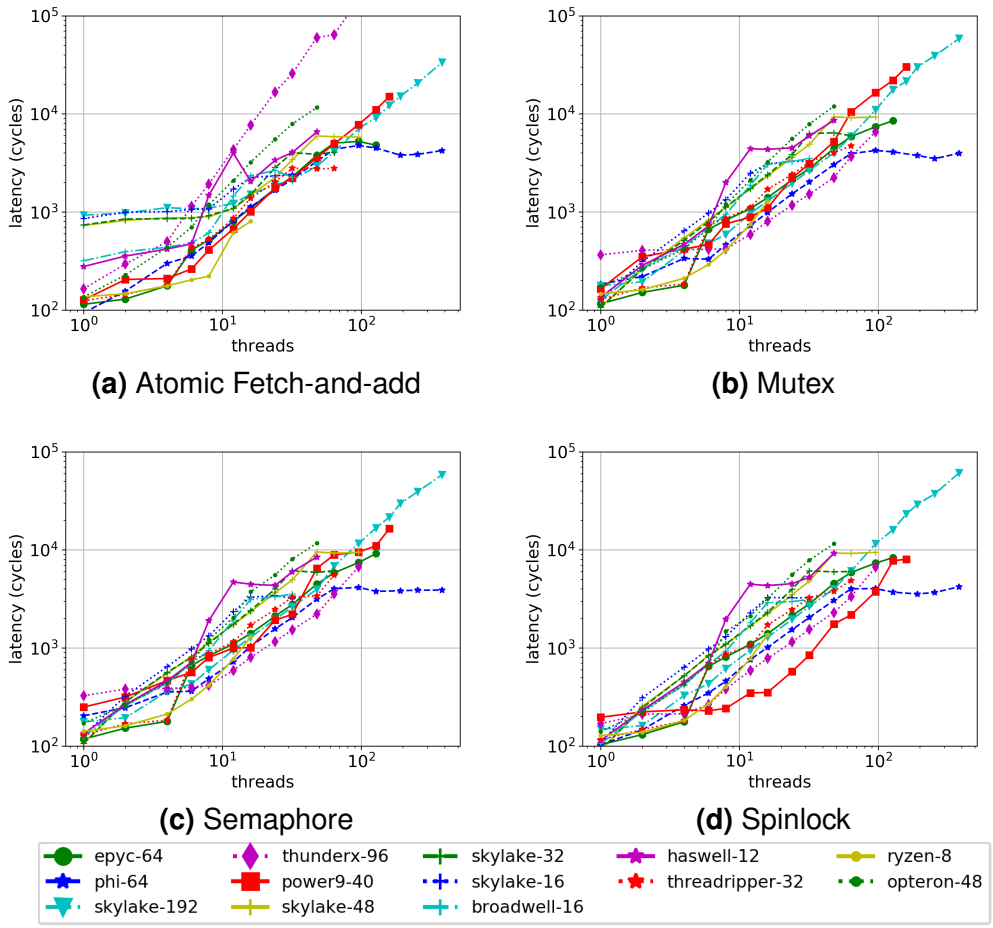


Figure 6 Average latency of incrementing an integer using different synchronization mechanisms. Same trend is observed on all architectures where latency keeps increasing as threads are scaled up except Intel Xeon Phi.

'ldxr/stxr' which are load/store exclusive instructions used for implementing atomic read, modify, write operations. These benchmarks are obtained by running a tight loop of 1 billion operations and collecting the aggregate of the results. Each iteration acquires the lock, increments a shared integer and releases the lock, excluding fetch-and-add which performs an increment operation atomically.

Figure 6 shows that all synchronization mechanisms exhibit higher latencies due to contention at higher levels of concurrency. There are many factors that impact the cycle counts like cache coherence, communication latency between cores on same and 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. Latency of a single atomic increment on a Skylake sys-

tem with 192-cores and 384 hardware threads when running on all threads concurrently is 33592 cycles whereas on Intel Xeon Phi Knights Landing with 64-cores and 256 hardware threads, latency reaches 3868 cycles. Similar behavior is observed on other architectures with latencies reaching up to thousands of CPU cycles solely for acquiring the lock, incrementing a variable and releasing the lock.

Although AMD, Intel, ARM and IBM have distinctly different architectures, it is interesting to note that the latency of synchronization mechanisms steadily increases on all the architectures as concurrency increases. For atomic instructions, most architectures show a slow rise in the latency up to 8 threads and latency linearly increases after 8 threads whereas for mutex, spinlock and semaphor, latency steadily goes up as concurrency level increases. Intel Broadwell, Haswell and Skylake processors exhibit similar performance curve as threads are scaled up where as AMD Ryzen, AMD Threadripper and AMD Epyc processors start with a slow increase in latency up to 8 threads for all four types of locks and then the latency rapidly grows as level as concurrency increases.

Intel Xeon Phi Knights landing with 64-cores shows interesting results. Although latency increases up to 64 threads, the latency remains constant as more threads are added. This behavior can be attributed to the round robin hyper-threading implemented in Intel Xeon Phi (which is different than all the other processor architectures evaluated in this paper). In x86 architectures, hyper-threading allows each physical processor to be perceived as two separate logical processors within the operating system by sharing the resources, which results in both hyper-threads running simultaneously increasing contention on each core. Whereas, in Intel Xeon Phi, every core alternates scheduling hardware threads at each cycle thereby not increasing contention and resulting in a better performance as threads are scaled up to more than the number of cores [106].

2.3 Conclusion

We were not surprised by these findings as it is well known that state-of-the-art synchronization mechanisms do not scale beyond single digit concurrent threads [79]. These limitations are automatically imposed onto concurrent data structures that are implemented using such synchronization mechanisms. Furthermore, use of such concurrent data structures in modern

parallel runtimes have significant overheads for managing extremely fine-grained tasks. Even though at low concurrency these mechanisms only cost hundreds of cycles, these costs quickly grew to tens of thousands and even hundreds of thousands of cycles at hundreds of threads. Our experience with the cost of synchronization mechanisms at high concurrency along with the cost of MPMC queues as a building block for parallel runtimes has motivated our investigation into methods to eliminate synchronization mechanisms in order to unleash the full performance of many-core architectures under high concurrency.

CHAPTER 3

Scalable Concurrent Queues on Modern Many-core Architectures

This work is motivated in large part by the significant latency gap observed with SPSC and MPMC models.

A simple concurrent SPSC queue can enqueue and dequeue items in less than 100 cycles. Independent SPSC queues per core could, in theory, scale linearly with increasing core counts. Thus, we believe that an MPMC lock-less queue can be built using SPSC queues by manipulating the task/data flow carefully.

We introduce XQueue, a novel lock-less MPMC, out-of-order queuing mechanism that can scale up to hundreds of threads. XQueue uses B-queue [116] as a building block. B-queue is a concurrent SPSC lock-free queue designed for efficient core-to-core communication. It is implemented without using any locks, atomic operations, or barriers. The latency of queue operations in B-queue is as low as 20 cycles. B-queue uses batching where both producer and consumer detect a batch of available slots that are safe to use. Batching avoids shared memory access and therefore improves performance. Several fast SPSC queues have been proposed in recent years [75, 71, 5] and we aim to demonstrate that XQueue can be built with any fast and scalable SPSC queue.

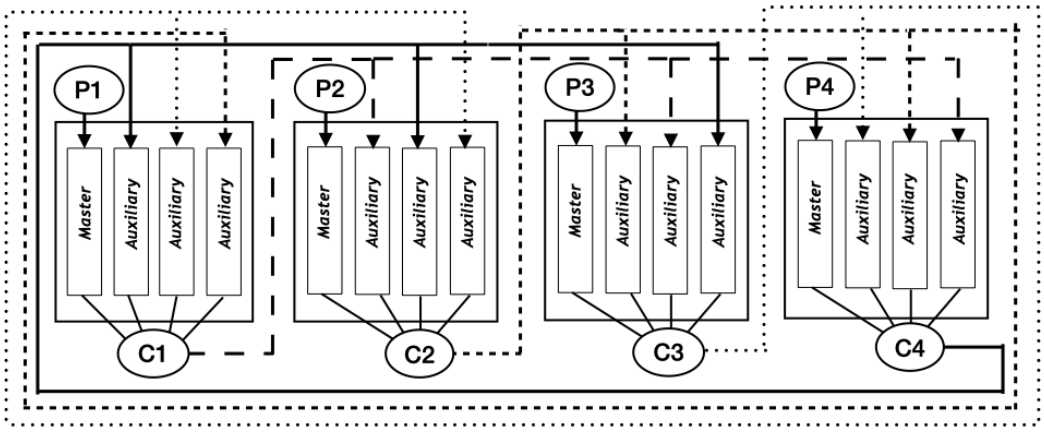


Figure 7 Architecture of XQueue on a 4-core machine with 4 queues per consumer.

Figure 7 shows the architectural of XQueue on a 4-core system. The key idea here is to have N SPSC concurrent queues per worker if there are N workers. There is one master queue and $N-1$ auxiliary queues per worker, with N (equal to number of workers) producers adding items into master queues. Every item is a void pointer that represents a task where a task could be a function pointer or data pointer. One worker exists for dequeueing tasks from the master queue as well as the auxiliary queues. A worker first tries to dequeue a task from the master queue. If a task is dequeued successfully, it is processed immediately. The item when processed can generate one or more items to be enqueued into the auxiliary queues of the other CPU cores. Every worker distributes work to auxiliary queues in a round-robin fashion as shown in Figure 7. A worker then tries to dequeue an item from its auxiliary queues and dequeued items are processed immediately.

A simplified version of pseudocode for worker logic is outlined in Algorithm 1. Since all queues in XQueue are concurrent SPSC queues, producer and consumer threads can act concurrently processing items in the queues. The strategy of distributing work across queues (as shown in Figure 7) ensures that there is a only a single producer and single consumer for every queue at any point in time. Due to this design, locks can be completely avoided thereby reducing the latencies of queue operations and improving overall performance.

Algorithm 1 Worker logic.

$id \leftarrow coreId; next \leftarrow nextCoreId;$

while 1 **do** $ret \leftarrow dequeueFromMaster(id, item);$

if $ret = SUCCESS$ **then** $retItem \leftarrow processItem(item);$

if $retItem \neq NULL$ **then** $enqueueToAuxiliary(next, retItem);$
$ret \leftarrow dequeueFromAuxiliary(id, item);$

if $ret = SUCCESS$ **then** $retItem \leftarrow processItem(item);$

if $retItem \neq NULL$ **then** $enqueueToAuxiliary(next, retItem);$
$next \leftarrow (next + 1)\%numCores;$

if $next == id$ **then** $next$++;

3.1 Load balancing

In most parallel programming systems, it is a common scenario to use multiple queues, one per worker, with work produced and consumed locally by the workers/threads. Load balancing is commonly achieved by using techniques like work stealing [31, 45]. While XQueue also uses multiple queues, it balances load by the virtue of its design with N queues per core and consumer threads inserting items into the auxiliary queues of all the other cores. This architecture enables distribution of task graphs to multiple threads with minimal overhead due to the lock-less design as compared to the state-of-the-art work stealing techniques which primarily use locks or atomics to achieve synchronization.

In a task-parallel program, tasks can be modeled as a Directed Acyclic Graph (DAG) which can be traversed based on inter-dependencies between the tasks. Task graphs have a pool of ready tasks which can be processed by threads and subtasks can be generated. The master and auxiliary queues and the communication between them is modelled after the dynamic execution of a program where a task can generate subtasks. In the case of XQueue with N workers and N queues per worker, as shown in Figure 7, we employ a ring buffer topology for communicating between queues. Essentially, the consumer thread of every set of queues acts as a producer thread of $N-1$ auxiliary queues of all the other threads. This pattern of task distribution ensures optimal load balancing in terms of the number of tasks

processed per worker. However, this may not be the best fit for every scenario for various reasons, such as data locality, task dependencies, and per task execution time. Optimal allocation of work among various threads is known to be NP-hard, but, in the case of XQueue, depending on the nature of work, the topology of connections between queues and task distribution strategy can be changed to achieve best performance.

The load balancing mechanism in XQueue can be considered as a push-based mechanism as opposed to pull-based work stealing approach. This primary difference impacts how initially imbalanced workloads are handled. For example, consider the case of Fibonacci. Execution starts with a single task which recursively unfolds the DAG as execution progresses. In the work stealing approach, idle workers randomly try to steal tasks from other workers. This results in several failed steals and coupled with the cost of locking for every steal, incurs significant overhead. On the other hand, the push-based approach of XQueue handles this efficiently with its round-robin distribution without the use of locks, thus incurring minimal overhead. We discuss the advantages and disadvantages of this approach in Section 3.4.

On modern many-core architectures, it is common to have multiple Non-uniform memory access (NUMA) zones which impact the latency of memory operations from various cores. In XQueue, every worker allocates queues in its respective NUMA zone.This ensures that any memory reads and writes from various threads have the lowest latency possible. However, when tasks propagate through auxiliary queues in the system, the latency of memory read/write is higher across NUMA zones. With XQueue's ring buffer design across N cores with N queues, some latency is unavoidable due to the underlying architecture.

In summary, there is a lot of flexibility for defining the topology for task distribution statically and dynamically during program execution with XQueue. If the nature of the DAG and data access patterns are known, the task distribution can be tuned to achieve best performance as compared to state-of-the-art work stealing approaches.

3.2 Xtask - eXtreme fine-grained TASKing runtime

XQueue is designed to have the lowest latency of operations since it does not use any locks or atomic operations for synchronization. XQueue achieves minimal load balancing automatically as execution progresses and depend-

ing on the nature of the DAG and size of tasks, applications with balanced DAG will benefit from using XQueue as the underlying data structure for managing tasks. To demonstrate this, we developed a prototype parallel runtime system that can process a dynamic task graph with task dependencies. The runtime system employs a producer consumer architecture. Underlying data structure to hold the bag of tasks is XQueue which is implemented using multiple single producer single consumer queues as describe earlier in this section. Number of producers and consumers is configurable at runtime along with the queue sizes. A pool of worker threads are launched waiting to consume tasks from multiple queues. When the framework bootstraps, it allocates memory for the queues on the NUMA node where the thread is created on. The framework uses pthread library for implementing multithreading. We originally developed Xtask wit a long-term goal to accelerate fine-grained parallel applications implemented in OpenMP, OpenCL, Swift/T, etc orders of magnitude beyond the state-of-the-art. However, it quickly became evident that it is cumbersome to implement complex applications using the Xtask API since the applications have to be broken down into tasks manually and rewritten. OpenMP is a popular standard for implementing parallelism by decorating the code with pragmas. We decided to integrate our ideas into OpenMP in order to broaden the scope of our ideas into real applications.

3.3 XQueue Integration with the OpenMP Runtime

In order to extend our research to real systems, we integrated XQueue into OpenMP [14] to enable execution of unmodified OpenMP programs using XQueue. OpenMP's tasking model provides a way to efficiently parallelize dynamic task graphs and recursive algorithms. Several implementations of OpenMP exist: GNU OpenMP (for GCC) [101], LLVM OpenMP [14], and Intel OpenMP. We chose to integrate XQueue into the LLVM OpenMP due to its open source code and its superior performance as compared to GNU OpenMP with fine-grained tasks [86].

Implementation: In the LLVM OpenMP tasking implementation, every thread owns a queue and the enqueue/dequeue operations are protected by locks implemented using Lamport's bakery algorithm. We replaced the task queues in OpenMP with multiple SPSC queues per worker to model XQueue. OpenMP implements a work-stealing scheduler. Every thread first checks it's own queue for tasks. If no tasks are found, a thread is randomly chosen to steal

a single task. We replaced the work stealing scheduler with the scheduler for XQueue as shown in Algorithm 1. In our XQueue-based OpenMP implementation, every thread checks it's own queue for tasks. If no tasks are found, the scheduler checks all auxiliary queues. This process of checking the master queue and auxiliary queues is repeated until a termination condition is satisfied.

Optimizations: We applied few optimizations to the XQueue system during integration with the OpenMP runtime. Since the core design of XQueue is to have multiple queues per worker, at higher thread counts (hundreds), the latency of checking all auxiliary queues can become significant and reduce the overall performance. To solve this issue, we implemented a hinting mechanism where every producer stores the ID of the last queue to which the task was pushed. This hint can possibly be over-written by multiple threads writing to various queues, however this simple mechanism reduces the latency of checking auxiliary queues many times.

3.4 Performance Evaluation

We evaluate the performance of XQueue using synthetic and real workloads. For the purposes of evaluating XQueue independently, we developed a prototype parallel runtime system that can process a dynamic task graph with task dependencies using XQueue. We first evaluate XQueue individually using a series of micro-benchmarks. We deployed XQueue on 13 systems (Table 1); we then picked the system with the highest number of cores, the skylake-192 with 192-cores and 8 NUMA zones to conduct deeper analysis.

3.4.1 Experiment Setup

We implemented three systems for the micro-benchmark evaluation:

(1) **XQueue (SPSC)** uses a single SPSC queue per worker.

(2) **XQueue (MPMC)** uses an MPMC queue with a master queue per worker.

(3) **XQueue (Cilk Deque)** uses a Cilk deque [38] with a separate queue per worker.

Cilk deque is implemented as part of Cilk 5 multi-threaded language [38] and uses a shared-memory, mutual-exclusion protocol called the THE protocol[30] for implementing locks. This mechanism of locking is about 25% faster than hardware locking primitives.

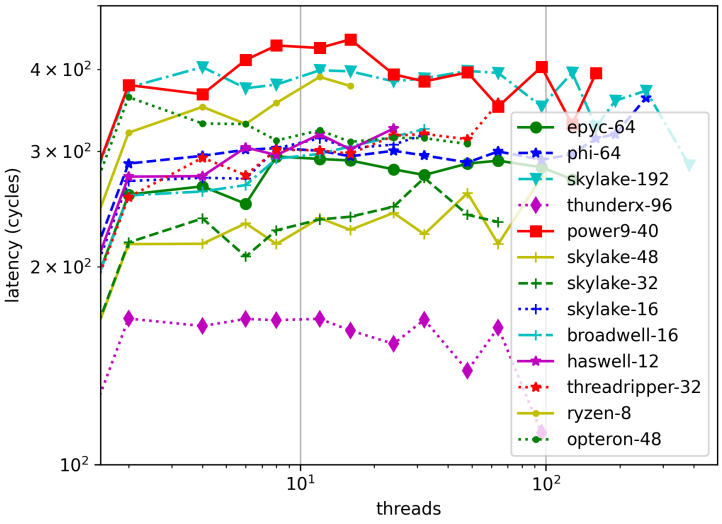
For the macro-benchmarks, we use the XQueue-enabled LLVM OpenMP implementation with N queues per worker and N workers. We compare it with the native LLVM OpenMP and GNU OpenMP libraries.

3.4.2 Micro-benchmark Performance Results

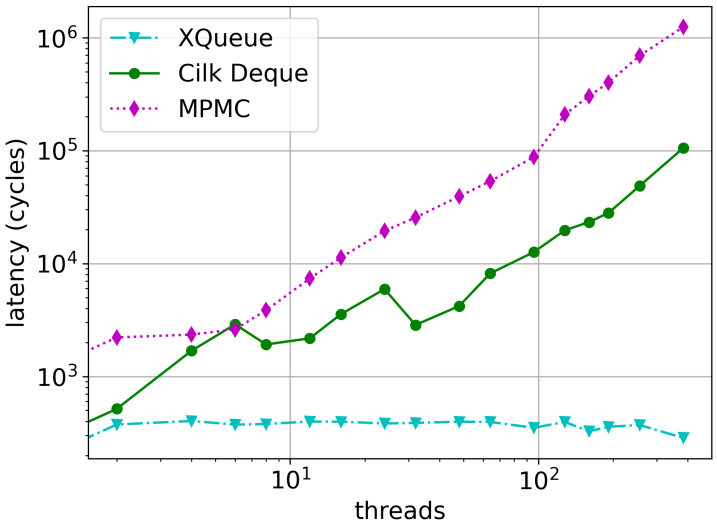
In each experiment we perform 1 billion enqueues/dequeues concurrently by varying the number of threads. We consider a single operation to be the act of dequeing an item from the master queue and executing the function to which that item points to. The function performs a single NOP operation. The X-axis on all the figures represents the number of producers/consumers.

Figure 8a shows the latency of queue operations on XQueue using lock-less queue. Each queue operation takes around 110 to 400 CPU cycles on average on all architectures considered. ARM ThunderX shows the lowest latency and IBM Power9 shows the highest latency in these micro-benchmarks. Intel processors Skylake, Haswell, Broadwell and Xeon Phi show latencies in the range of 180 to 300 CPU cycles on average. The standard deviation is low across all architectures indicating that XQueue with lock-less queue can scale up to hundreds of threads with latencies as low as 110 to 400 cycles.

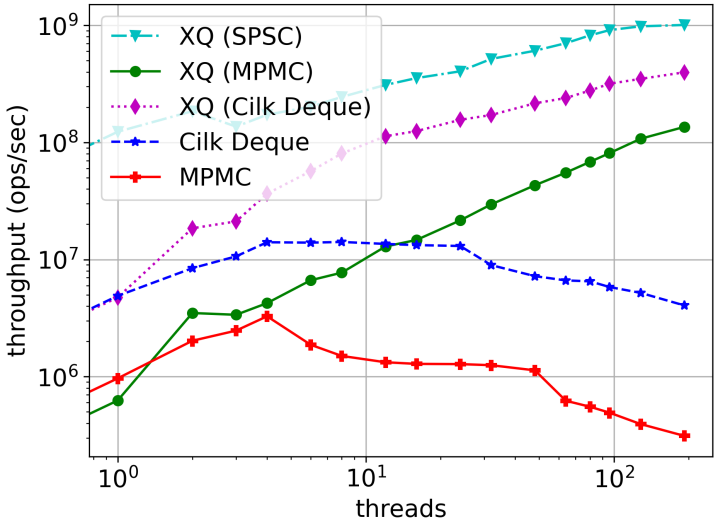
Figure 8b compares the latency of XQueue (SPSC) with Cilk Deque and MPMC queues on skylake-192. Here, Cilk Deque/MPMC is a single queue shared across all the workers. With 192 producers/consumers, latency of MPMC queue is 13$\times$ the latency of Cilk deque. Cilk deque's Dijkstra-like locking mechanism achieves much lower latency than locks implemented using hardware locking primitives. However, the latency is much higher compared to XQueue which does not use any locks. It is noteworthy that XQueue has relatively constant latency as we increase the number of threads by two and half orders of magnitude, while Cilk deque and MPMC show significant latency increases over the same scale.



(a) Average latency of enqueue/dequeue operations using XQueue (SPSC)



(b) Latency Comparison



(c) Throughput Comparison

Figure 8 XQueue Performance on Skylake-192

Figure 8c is a log-log plot showing the throughput of XQueue using lock-based and lock-less queues on the skylake-192 system. The throughput achieved on this system with XQueue with lock-less queue is 1 billion operations per second with all hyper threads being utilized. For XQueue using lock-based queue, the average throughput achieved is 200 million operations per second and 397 million for the Cilk deque. In the case of MPMC queue, each mutex lock is held for short intervals and contention is low, but acquiring the lock has a cost which explains the 5$\times$ gap in performance as compared to XQueue with lock-less queue. Cilk deque also incurs a cost for acquiring and releasing the lock (a 2.5$\times$ gap), although the cost is lower compared to mutex-based locks. Similarly XQueue implementations over both lock-less and lock-based queues highly benefit from data being in the cache avoiding costly memory accesses. A single lock-based queue shared among all threads shows improvement in performance up to a maximum of 8 threads on all architectures and it steadily declines as more threads are added. As noted in Chapter 2 for MPMC queue, with high contention on the mutex lock with more than 8 threads, throughput drops to about 300K operations per second on skylake-192 with 384 threads. In case of Cilk deque, the throughput drops to 4 million operations per second. This clearly shows a 3300X gap in throughput between XQueue with lock-less queue and single lock-based queue with hundreds of threads.

The results obtained from micro-benchmarks using XQueue with lock-less queue and lock-based queue are significant and show that this architecture can scale to at least hundreds of threads with any scalable concurrent SPSC queue implementation. It can be noted that these micro-benchmarks do not take into consideration the cache effects of task distribution to other cores in XQueue since there are no auxiliary queues. Hence, this benchmark shows the lowest latency and highest throughput that can be achieved, providing a baseline.

3.4.3 Macro-benchmark Performance Results

To quantify the improvements in real application workloads, we evaluate the speedup achieved using XQueue-enabled LLVM OpenMP as compared to the native LLVM OpenMP and GNU OpenMP libraries. We evaluate five out of nine applications from the BOTS benchmark suite [36]: Fibonacci, FFT (Fast Fourier Transform), Multisort, NQueens and Health. Results are shown in Figure 9. We also evaluate the breadth first search application

from the GAP benchmark suite [12] with real-world social network graphs such as those from Friendster and Twitter. Results are shown in Figure 11. The application workloads are summarized in Table 2.

Table 2 Application - number of tasks

Application	Inputs(S,M,L,XL)	Highest Task Count
Fibonacci	44, 46, 48, 50	40.7B
FFT	134M, 268M, 536M, 1B	128M
Multisort	134M, 268M, 536M, 1B	14M
Nqueens	14, 15, 16	1.1B
Health	small, medium, large	126M
BFS	friendster	79M
BFS	twitter	40M

Fibonacci (Fib) computes the Nth Fibonacci number using recursive parallelism. While Fib is hardly a critical parallel application, it *does* have extremely fine-grained tasks (e.g., addition of two numbers) with extremely large number of tasks, and thus exposes the limits of a tasking runtime in terms of granularity. Figure 9 shows the results obtained on skylake-192. OpenMP with XQueue achieves 3$\times$ speedup as compared to the native LLVM and GNU versions for Fib(50). The performance gap increases with problem size due to the increase in overhead of locking operations in the native OpenMP versions with more fine-grained tasks. Further analysis using Intel Vtune Profiler showed that about 50% of the execution time is spent in these operations which includes waits and atomics, where as this overhead is negligible in the XQueue version due to the lack of locks or atomics. The overall runtime overhead for managing fine-grained tasks of this application reduced from over 90% to 29% of the CPU time when using XQueue.

Multisort sorts 32-bit randomly generated numbers using a fast parallel sorting variation of mergesort. It uses a recursive algorithm with a base condition of 2048 numbers and they are sorted using serial quicksort and insertion sort is used for arrays with less than 20 elements. The application scales well up to 96 threads for all the runtimes and XQueue is faster for all problem sizes with 1.97$\times$ speedup for the largest problem size. However, the performance drops by 50% at 192 threads. As shown in Figure 9, XQueue achieves similar performance compared to LLVM and GNU versions using 192 threads. LLVM and GNU versions of OpenMP exhibit high CPI (cycles per instruction) rate (0.5 for XQueue vs 24 for both LLVM and GNU for the largest problem size) which is the result of waits, atomics, and locks in the GNU/LLVM versions. However, since this application is heav-

ily memory-bound, the benefits of avoiding locks and lower CPI in XQueue are outweighed by the data movement across cores, thereby resulting in no performance benefit.

Health simulates the Columbian Health Care System [29]. A list of potential patients in a village with one hospital are simulated with several possibilities of getting sick, needing treatment or reallocating to an upper level hospital. Every village being simulated is run as a task. The different probabilities at each step cause indeterminism and load imbalance. On skylake-192, the performance of this application is heavily impacted due to remote memory accesses for moving the village data across NUMA zones. Despite some load imbalance, XQueue achieves 6$\times$ speedup compared to LLVM variant and 4$\times$ speedup compared to GNU variant using the large input data.

Fast Fourier Transform (FFT) computes the 1D FFT of a vector with N complex values using the Cooley-Tukey Algorithm. This algorithm recursively divides the FFT into several smaller Discrete Fourier Transforms (DFTs) creating multiple tasks at each step. Although the XQueue version has the advantage of reduced overhead due to lock-less queues, the task distribution suffers due to the static round-robin placement of tasks resulting in similar overall execution time as compared to other versions of OpenMP. Figure 10 shows the timeline view of the OpenMP parallel region for the largest problem size, where green represents effective work and black represents the spin/wait/overhead time introduced by load imbalance. It is noteworthy that OpenMP with XQueue with worse load balancing can still achieve slightly improved performance (between 0.9$\times$ to 1.2$\times$) due to the smaller overheads incurred by avoiding locks.

NQueens computes all the solutions for placing N queens on an $N \times N$ chess board such that no queens can attack each other. The algorithm prunes certain branches of the tree that cannot reach the solution which creates load imbalance. Figure 9 shows that the XQueue-enabled OpenMP implementation achieves 4X speedup compared to the GNU version. The performance loss in XQueue as compared to standard LLVM is due to the significant load imbalance. On the other hand, GNU OpenMP incurs huge synchronization overheads for managing fine-grained tasks (about 60% on skylake-192) and the performance is significantly lower for GNU OpenMP compared to OpenMP with XQueue.

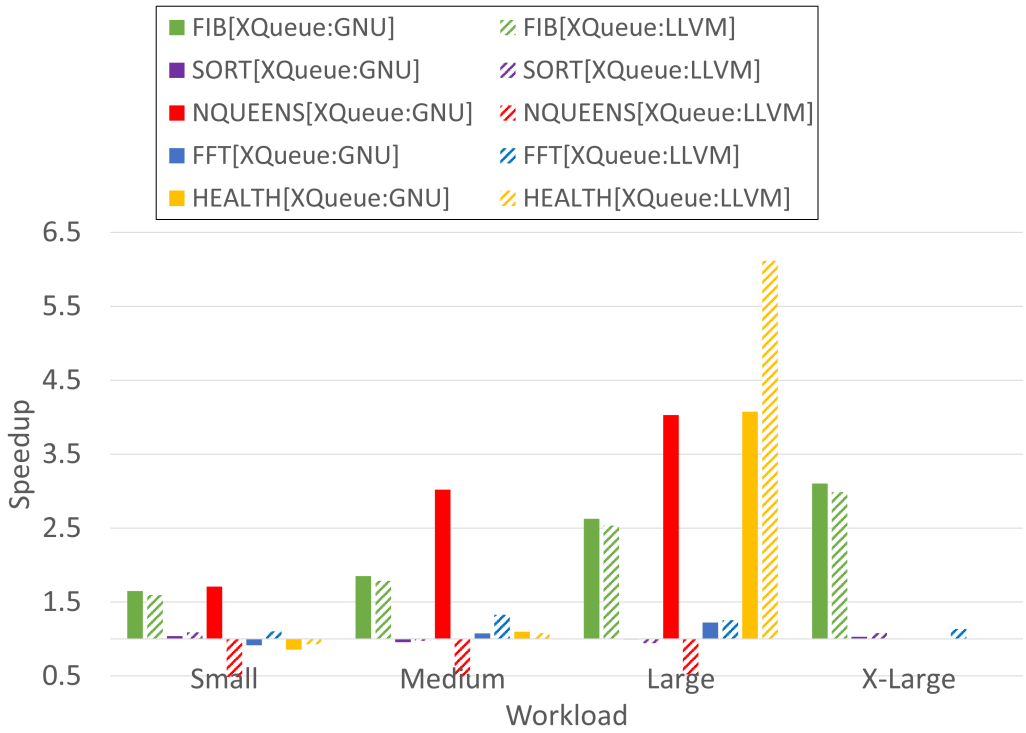


Figure 9 Speedup of XQueue over standard GNU and LLVM OpenMP implementations on the BOTS benchmarks on skylake-192 using 192 threads.

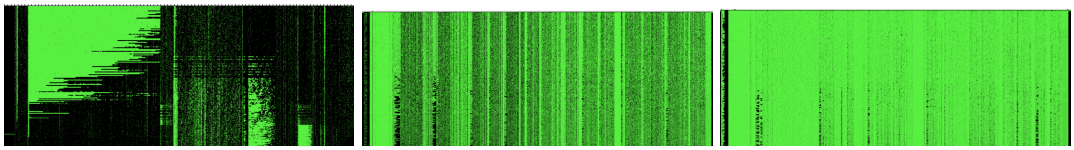


Figure 10 Load balance of FFT on skylake-192 - LLVM+XQueue (left), Native LLVM (middle), GNU(right)

Breadth First Search (BFS) is a fundamental building block of many graph algorithms: it checks the connectivity of the graph from given source vertices, visiting one layer at a time. In order to demonstrate the applicability of XQueue using real-world datasets, we evaluate the BFS application from the GAP Benchmark Suite [12] using social network graphs such as Twitter and Friendster. The original implementation of BFS in the GAP benchmark leverages loop parallelism (LP) to parallelize every level of the tree. We modified the code to use recursive task-based (TP) parallelism with a base condition of 1024 nodes to evaluate XQueue. We also evaluate the extreme case with a base condition of 1 node, which creates several extremely fine-grained tasks. Each data point is the average speedup obtained by running BFS 64 times from pseudo-randomly selected non-zero degree source vertices. The Twitter graph has 61 million nodes and 1.47 trillion directed edges for a degree of 23 where degree is the maximum number of edges connecting a vertex. The Friendster graph has 65 million nodes and 3.61 trillion directed edges for a degree of 55.

Figure 11 shows the speedup achieved for both the test graphs on the skylake-192 using 192 threads. For the Friendster graph with a base case of 1024 nodes, GNU OpenMP scales well up to 24 threads and performance degrades at higher concurrency levels. XQueue performs reasonably well at full scale of 192 threads as compared to GNU and LLVM. XQueue achieves a speedup of 1.4$\times$ for Friendster and 3$\times$ for Twitter graphs over GNU with base case of 1024 nodes. Execution times for LLVM and XQueue are similar for Friendster and for Twitter, XQueue achieves 2.4$\times$ speedup. For the base case of 1 node, while there is no significant performance difference between LLVM and XQueue, GNU's performance suffers significantly (up to 116$\times$ slower) due to the overhead of managing fine-grained tasks. Since real social network graphs are very unbalanced, they result in highly irregular memory accesses and load imbalance. Compared to the original GAP BFS using loop parallelism, XQueue achieves 1.9$\times$ speedup using Friendster and 1.6$\times$ speedup using Twitter with 192 threads, showing promise that the task-based parallel approach can be beneficial for these types of workloads.

Overall, our results show that there is significant room for improvement in existing task-parallel runtimes and higher performance can be achieved by using lock-less techniques. Improving load balancing could yield further performance improvements similar in size to the improvements seen here.

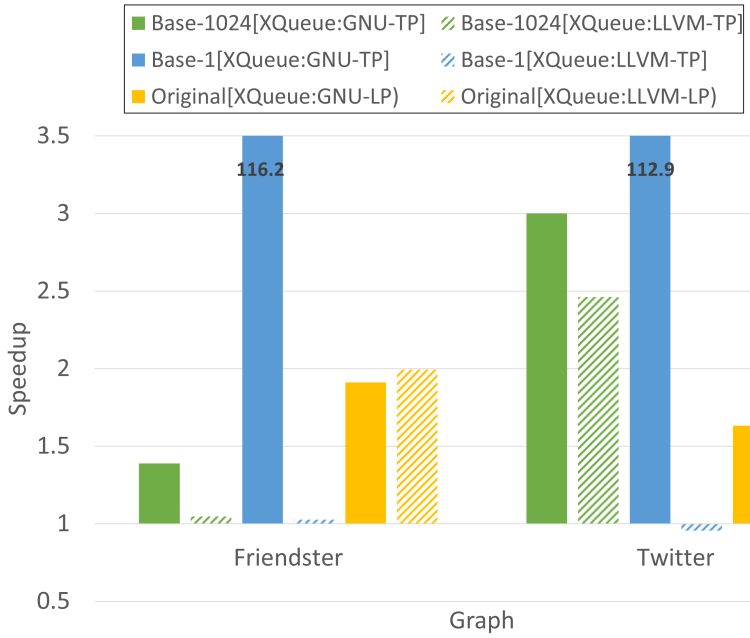


Figure 11 Speedup of XQueue over standard GNU and LLVM OpenMP implementations when applied to Breadth First Search from GAP Benchmark Suite on skylake-192 using 192 threads.

3.5 Conclusion

XQueue is an extremely scalable lock-less MPMC out of order queuing system which can be used in tasking runtimes to overcome the performance limitations due to overhead of synchronization. Evaluation results show that XQueue is scalable up to hundreds of threads of execution with up to 6900× lower latencies and 3300× higher throughput when compared to naive implementations. We integrated XQueue with LLVM OpenMP and were able to achieve up to 6× speedup compared to native LLVM OpenMP and 1× to 4× speedup compared to GNU OpenMP in most cases with up to 116× speedup in some cases on applications from the BOTS benchmark suite and BFS application from the GAP benchmark suite.

CHAPTER 4

X-OpenMP – eXtreme fine-grained tasking using lock-less work stealing

We introduced XQueue, a lock-less concurrent queueing framework for task parallel runtime systems which enables extreme fine-grained task parallelism. This is achieved by reducing the overheads of the underlying concurrent data structures used in runtime systems. We demonstrated performance improvements that could be obtained on modern architectures with hundreds of cores using several benchmarks. However, XQueue framework relies on a static round-robin load balancing strategy for distributing work across processors. While this approach to push work eagerly to other workers can achieve modest load balancing, the lack of dynamic load balancing can severely limit the performance of real-world workloads.

Load balancing is crucial to parallel applications as imbalances quickly lead to sub-optimal execution times. Work stealing is typically used in most parallel runtimes and execution models for load balancing. Work stealing involves stealing work from a random busy worker when a processor runs out of work. Traditional work stealing implementations use lock-based approaches to steal work from concurrent queues. These concurrent data structures do not scale up to hundreds of threads on modern many-core architectures and exhibit significant overheads at high levels of concurrency. Acar et al. explored a lock-less approach for work stealing by implementing an algorithm that can steal work non-atomically [2]. We extend their work

on load balancing along with our prior work on lock-less concurrent parallel framework [80] and propose a dynamic lock-less load balancing mechanism that can provide significant performance improvements using real application workloads.

4.1 Motivation

In task-parallel runtimes, load imbalance is a significant performance limiting factor. Several studies have shown the importance of dynamic load balancing in multi-threaded applications [31, 6]. Dynamic load balancing enables better distribution of work across the processors to achieve efficient performance. In a multi-threaded runtime, typically tasks are executed by a fixed number of workers. Every worker owns a task pool and execute tasks from their pool. Any subtasks that are spawned are inserted into the worker's own task pool. When a worker runs out of tasks, it randomly picks workers to steal tasks from. The amount of load balancing required varies from application to application. Workers can steal a single unit of work at a time, or two units, or half the amount of work from the victim's task pool. Literature has shown that asking two random workers for work is sufficient in most cases to achieve exponential improvement in performance [76].

Figure 12 shows the timeline plot of the Unbalanced Tree Search (UTS) benchmark [83] executed using GNU's implementation of OpenMP. The green dots indicate effective CPU time and the black dots indicate idle time. The plot shows a significant load imbalance for this application where several workers (bottom of the figure) are idle for most of the application run, and other workers are idle for a significant amount of the time. The load imbalance results in a major slowdown in the execution time of the application. The UTS benchmark is designed to understand the efficiency of dynamic load balancing in parallel runtime systems and this plot clearly highlights the imbalance in existing task-based runtime systems. Processors are heavily under-utilized resulting in poor overall performance. The simplest way to achieve load balancing is to distribute work across workers in a round-robin fashion. While this is easy to implement, real-world applications are dynamic in nature with varying computational intensity and complexity. A naive round-robin load balancing approach may not be sufficient for improving the performance of real-world workloads [13].

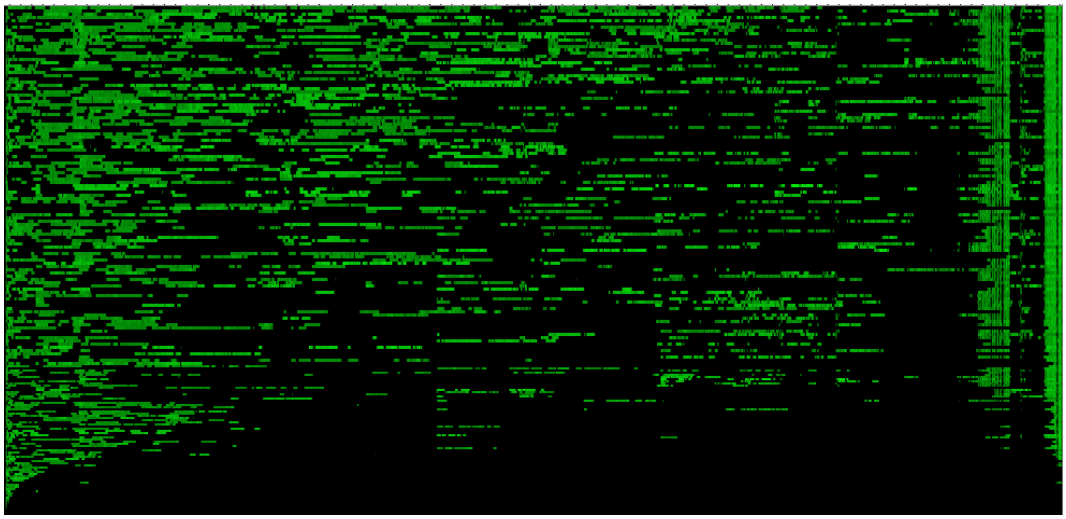


Figure 12 Load Imbalance in Unbalanced Tree Search using 192 threads and GNU OpenMP

Multi-threaded systems use synchronization mechanisms like mutexes, semaphores, spinlocks, or atomic operations [105, 59] to ensure thread safety and correctness. Concurrent data structures are the central building block of multi-threaded execution models and work stealing relies heavily on these implementations. However, traditional synchronization mechanisms do not scale to hundreds of threads. The mutual exclusion required to ensure correctness, consistency, and thread safety leads to serialized concurrent accesses and adds unnecessary overheads. New approaches for concurrent data structures are necessary to push the limits of scalability on modern many-core architectures. Lock-free approaches [93, 24, 73, 72, 74] mitigate these overheads to an extent by using atomic operations which guarantee system-wide progress, but literature has shown it is very difficult to write correct lock-free code [100]. Lock-less non-atomic updates to data structures are significantly faster compared to the atomic variants and are the focus of our work.

One of the challenges of parallel execution models that use traditional work stealing is the potential need for a large number of steals to achieve optimal load distribution. When a runtime is initialized and workers are created, they start looking for work and when no work is found in the local task pool, work stealing is triggered. Studies have shown that several steal requests are generated at the beginning and tail end of the execution [117]. Work stealing implemented using traditional synchronization-based mechanisms tend to have huge overheads for stealing work. Hence, work stealing should

be triggered sparingly and only when necessary to avoid unnecessary overheads.

4.2 X-OpenMP - eXtreme fine-grained tasking runtime

OpenMP is a popular standard for implementing parallel runtime execution models. Task-based parallelism has emerged for exploiting dynamic parallelism from applications on modern many-core and multi-core architectures. We introduce X-OpenMP with the goal of enabling extreme fine-grained parallelism for task-parallel applications. We extend our work on XQueue and implement dynamic load balancing to overcome the limitations of static round-robin load balancing.

4.2.1 Load Balancing

Static round-robin load balancing is limited for dynamically unfolding task graphs due to the inability to load balance during the course of application execution. Most multi-threaded runtime systems [38, 23, 104] use load balancing mechanisms like work stealing and work sharing in order to reduce the overall execution time. Traditional work stealing mechanisms typically use synchronization constructs to safely steal work from the victim's queue. However, since XQueue uses SPSC queues where queue operations are not protected using locks, there is a need to design a lock-less algorithm that can perform dynamic load balancing of tasks using work stealing.

4.2.1.1 Lock-less Work Stealing Using Wait

A mechanism that does not use synchronization is required for implementing work stealing using XQueue. Intel's x86 architectures have a memory model that supports Total Store Ordering (TSO) [96]. TSO guarantees that load and store operations to a memory location are in order as issued by the processor. This memory consistency model provides an opportunity to explore lock-less techniques on x86 architectures for implementing low overhead concurrent data structures and load balancing mechanisms. Prior work has presented an algorithm that does not require atomic read-modify-write operations for shared memory work stealing [2] that works on total store memory architectures like Intel's x86. Processors communicate by reading and writing into memory locations non-atomically. The details of the

original implementation can be found in the technical report [2]. We employed a modified version of this algorithm for work stealing in X-OpenMP to implement dynamic load balancing. The implementation works as follows. The algorithm requires two memory cells per worker where one cell holds a combination of 40-bit round number (representing the round of work stealing) and 24-bit identifier (ID of the worker) packed into a 64-bit word and the other memory cell holds a pointer to the stolen task. Algorithms 2 and 3 present the pseudocode for victim and stealer threads. To perform work stealing, an idle thread (stealer) first randomly picks a victim. As shown in Algorithm 3, the stealer first checks if the victim is accepting requests. This is shown in the first line of the algorithm where the 40-bit round number is extracted by using bit operations and compared with the victim's own round number. The steal request is valid only if the extracted round number is less than the victim's own round number. The stealer then takes a copy of victim's round number and writes its identifier packed with the round number into the victim's 64-bit memory cell. The stealer thread waits in a while loop until the copy of its round number matches the victim's round or a stolen task is received. While waiting, it also writes a steal request to it's own memory cell and leaves it unserved. This self query makes sure no other steal requests come in to this thread since it is idle. When a stolen task is copied by the victim to the stealer's memory cell, the stealer immediately breaks out of the while loop and executes the task. On the other hand, a busy victim looks at it's memory cell during a dequeue operation, as shown in Algorithm 2, extracts the round number from the steal request and compares this round number with its current round number. If it matches, the steal request is valid and the victim dequeues a task from its queue and copies it to the stolen task memory cell of the stealer. The victim increments it's round number to invalidate any steal requests coming in. The round is incremented in 2 scenarios: (1) when a steal request is served and a task is copied to the stealer's stolen task field; and (2) when victims' queues are empty.

The pseudocode presents only the core logic leaving out the complex implementation specific details. The actual implementation also ensures that a stealer is not able to steal requests from other threads while it is waiting to steal a task. Also, this implementation works similarly to traditional work stealing mechanisms where a stealer waits to steal a task from a victim.

The original algorithm in the technical report [2] is implemented for stealing threads and waits forever in the while loop until a steal succeeds or is

Algorithm 2 Work Stealing With Wait - Victim's Logic

$local\_steal\_req \leftarrow thread- > steal\_req$; $round \leftarrow local\_steal\_req \ \&((1 << 40) - 1)$;

if $round == thread- > round$ **then** $ret \leftarrow dequeue(thread\_id, item)$;

if $ret == SUCCESS$ **then** $stealer\_id \leftarrow local\_steal\_req >> 40$;
$threads[stealer\_id]- > stolen\_task \leftarrow item$;
$thread- > round + +$;

Algorithm 3 Work Stealing With Wait - Stealer's Logic

if $(victim- > steal\_req\&((1 << 40) - 1) < victim- > round)$ **then**
$round = victim- > round$;
$victim- > steal\_req = round + (thread\_id << 40)$;

while $round == victim- > round || thread- > stolen\_task \neq NULL$ **do**

if $(victim- > steal\_req\&((1 << 40) - 1)) < round$ **then** $victim- > steal\_req = round + (thread\_id << 40)$;

if $(thread- > stolen\_task \neq NULL)$ **then** return $thread- > stolen\_task$;

return $NULL$;

invalidated. However, in the implementation of X-OpenMP, to ensure the application terminates after executing the DAG, the worker breaks out of the loop after waiting for a certain amount of time. The amount of time a worker waits to steal a task has a direct impact on overall execution time. Due to the static load balancing, a worker waiting to steal a task might get work from other workers and the worker needs to return to executing tasks as soon as possible. In order to achieve better performance, the time a worker waits to steal a task is dynamically adjusted based on the recent activity. The concept is similar to exponential backoff in computer networks where feedback is used to multiplicatively decrease the rate of some process in order to achieve an acceptable rate [66]. In our model, the wait time is controlled by the number of loop iterations, starting with a very small number and doubling every time a steal request fails. If a steal request succeeds, the number of iterations is decreased by a small amount in order to achieve the ideal number of iterations required for stealing. Effectively, the wait time increases exponentially for failed requests and decreases linearly for successful requests with the goal to achieve an optimal wait time. This

approach minimizes the number of failed steal requests while adjusting the wait time to achieve better performance.

4.2.1.2 Lock-less Work Stealing Without Wait (Wait-Free)

While the above algorithm using dynamic wait time works like traditional work stealing algorithms, the communication between workers in XQueue using SPSC queues can be used to implement work stealing without waiting. The benefit of this approach is that it eliminates the wait time while enabling load balancing using steal requests and queue operations. As shown in Algorithm 5, it starts off with the stealer submitting a steal request to a random victim thread by writing a 64-bit word in the victim's memory cell. Instead of waiting in a while loop to receive a task from the victim, the stealer immediately returns to the scheduler and checks it's own queues for tasks. If no tasks are found, it picks another random worker to submit a steal request.

On the victim's side, if a steal request is received, the victim can take action in both enqueue and dequeue operations. Algorithm 4 shows the pseudocode of the dequeue operation. If the victim is trying to dequeue a task and a steal request is received, the victim checks all it's queues for a task, and it enqueues the task into the stealer's auxiliary queue instead of copying it to the stealer's stolen task memory cell. In case of an enqueue operation, if a steal request is received, instead of following a round-robin order for distributing tasks, it enqueues the task into the auxiliary queue of the stealer. If no steal request is found, the enqueue continues in a round-robin fashion across all the workers. This approach of work stealing leverages the existing connections between queues and workers for enqueue and dequeue and does not require sophisticated waiting logic to ensure termination of the application.

Algorithm 4 Wait-Free Work Stealing - Victim's Logic

$local\_steal\_req \leftarrow thread- > steal\_req$; $round \leftarrow local\_steal\_req\&((1ULL << 40) - 1)$;

if $round == thread- > round$ **then** $ret \leftarrow dequeue(thread\_id, item)$;

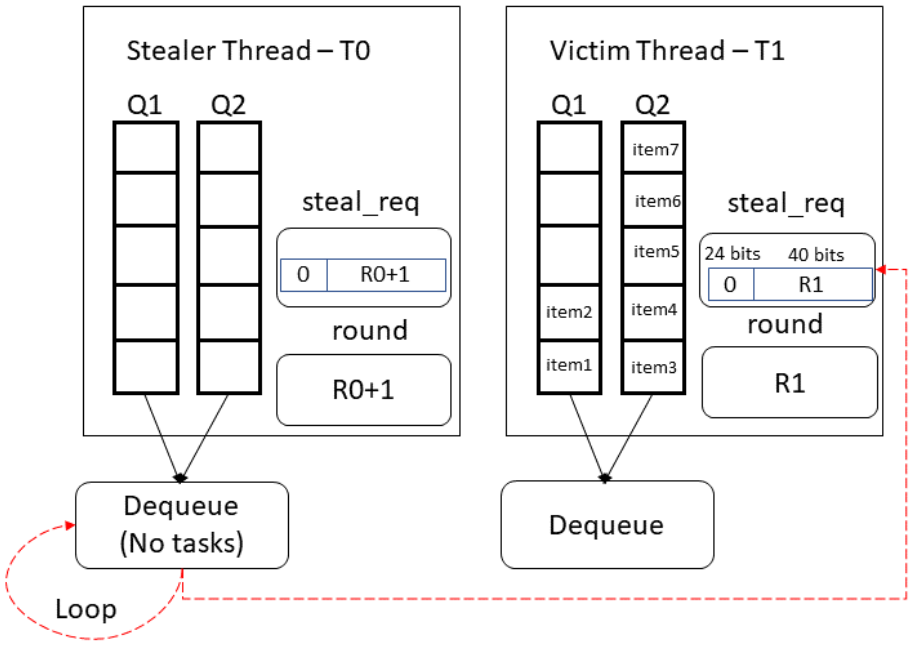
if $ret == SUCCESS$ **then** $stealer\_id \leftarrow local\_steal\_req >> 40$; $threads[stealer\_id]- > enqueue(item)$; $thread- > round++$;

Algorithm 5 Wait-Free Work Stealing - Stealer's Logic

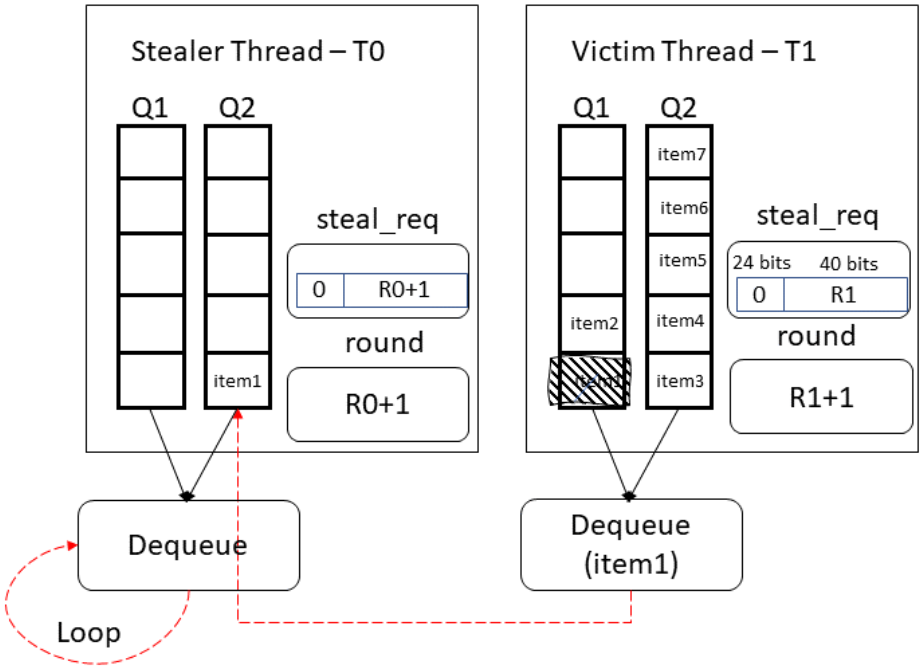
if $(victim->steal\_req \& ((1 << 40) - 1) < victim->round)$ **then** $round = victim->round$;
$victim->steal\_req = round + (thread\_id << 40)$;
return NULL;

It is worth noting that the wait-free work stealing algorithm results in many more steal requests being submitted than the wait-based approach, thereby resulting in more successful steals and better load balancing in terms of the number of tasks. A significant difference between the traditional work stealing approach and the lock-less approaches described above is that in the traditional approach, an idle worker is doing all the work for stealing a task. However, in the case of the lock-less approach, a busy worker is facilitating work stealing by checking it's queues and pushing a task to the stealer. This approach may slightly increase the overhead of tasking, however it is not significant as we will show in the evaluation section. During a dequeue operation, the worker is checking all the queues to dequeue tasks. In the case of dequeue with no steal requests, one task needs to be removed, whereas if there is a steal request, two tasks need to be removed from the queues, one for executing by itself and the other for handing over to the stealer.

The wait-free lock-less work stealing algorithm is shown in the Figure 13. For simplicity, we show two threads and two queues per thread where thread T0 can enqueue into queue Q2 of thread T1 and T1 can enqueue into queue Q2 of T0. In Figure 13-A, the stealer thread T0 checks it's own queues for tasks during dequeue operation. If no tasks are found, T0 writes a steal request into T1's memory cell as shown by the dotted red line. After putting a steal request, T0 checks if the termination condition for the runtime is satisfied and if not, returns back to the dequeue operation which is shown by the dotted red loop for dequeue. Victim thread T1 checks for incoming steal requests during a dequeue operation. If a request is received, thread T1 checks its queues for two tasks, one for executing itself and the other for fulfilling the steal request. Only the stealing part is shown in the figure. Thread T1 dequeues an item and enqueues it to queue Q2 of thread T0. It then increments it's round value to allow other incoming steal requests. Also, thread T0 writes a self query using its own round number incremented by one into it's own steal request memory cell. This tells the other workers that steal requests are not currently being accepted by this worker (as shown in Algorithm 3.



(a) A



(b) B

Figure 13 Wait-free work stealing in action - [A] shows the stealer putting a steal request to the victim [B] shows the victim serving the steal request

4.2.1.3 Considerations

X-OpenMP is designed using lock-less techniques to overcome the high overheads of synchronization at high concurrency levels. This approach requires several SPSC queues per worker to enable concurrent access and to ensure a single thread enqueues and a single thread dequeues at any point in time. Multiple queues in XQueue require more memory per worker as compared to a single queue per worker in other OpenMP implementations, which becomes significant when there are hundred's of workers in the system. It is worth noting that the performance is not sensitive to queue size as is the case with the native LLVM OpenMP. If the queue size is very small, it results in many failing enqueues which in turn results in few workers executing most of the tasks. We evaluated the X-OpenMP approach using varying queue sizes and achieved similar performance with both small and large queues due to the presence of multiple queues.

The original implementation of XQueue uses N^2 queues where N is the number of workers in the system. This is a limitation imposed by the static round-robin load balancing strategy which can limit the scalability of the system. Our implementation of work stealing enables dynamic load balancing which can be used to reduce the number of queues required in order to achieve better performance. One strategy would be to constrain the ring buffer of queues to be within a NUMA zone and load balance dynamically across other NUMA zones. This was proposed by the authors in prior work and our work in dynamic load balancing enables exploration of these options with just minor changes to the current implementation.

4.2.1.4 Scheduling Logic

Our scheduling logic is similar to XQueue with some additional logic for tracking the last successful victim. The worker first checks its own queues for tasks. If no tasks are found, it randomly chooses a victim thread to steal work from. A steal request is submitted to the victim and if the steal is successful, the runtime tracks the victim's ID for future steals. If the steal fails, the saved victim ID is reset and the scheduler randomly picks another victim to steal from. This is an optimization from the native LLVM OpenMP implementation that we adopt for X-OpenMP. This optimization enables efficient work stealing from an overloaded worker.

If some workers are overloaded, instead of stealing one task at a time, multiple tasks can be stolen to load balance quickly and efficiently using less steal requests [117]. The wait-free work stealing approach submits several work stealing requests due to the virtue of its design and we explore the performance by stealing one and two tasks at a time to understand the overall impact on performance.

4.3 Evaluation

We evaluate X-OpenMP using a set of synthetic benchmarks and real-world applications. The microbenchmarks are specifically designed to understand the performance of lock-less techniques described in our work for tasking and load balancing. We evaluate 4 different implementations in X-OpenMP:

(1) XQUEUE-STATIC - uses static round robin load balancing;

(2) XOMP-DYNAMIC-WAIT - uses static load balancing and dynamic wait-based work stealing;

(3) XOMP-DYNAMIC-WAITFREE/ XOMP-DYNAMIC-WAITFREE-STEALONE - uses static load balancing and dynamic wait-free work stealing, stealing one task at a time;

(4) XOMP-DYNAMIC-WAITFREE-STEALTWO - uses static load balancing and dynamic wait-free work stealing, stealing two tasks at a time.

We compare the performance of X-OpenMP (XOMP) with native LLVM OpenMP (OMP) and GNU OpenMP (GOMP). To quantify the performance improvements in real application workloads, we evaluate strassen's matrix multiplication from the BOTS benchmark suite [36], cholesky factorization and symmetric rank-k update routines from the PLASMA linear algebra library [32] and the Unbalanced Tree Search benchmark [83]. All experiments are conducted on an Intel Skylake Server with 192 cores (384 hardware threads) at 2.1GHz with 8 sockets and 8 NUMA zones. This server is part of the Mystic testbed [85]. We compiled all the benchmarks using LLVM Clang version 11.0 and O3 optimization level and ran experiments on Ubuntu 20.04.4.

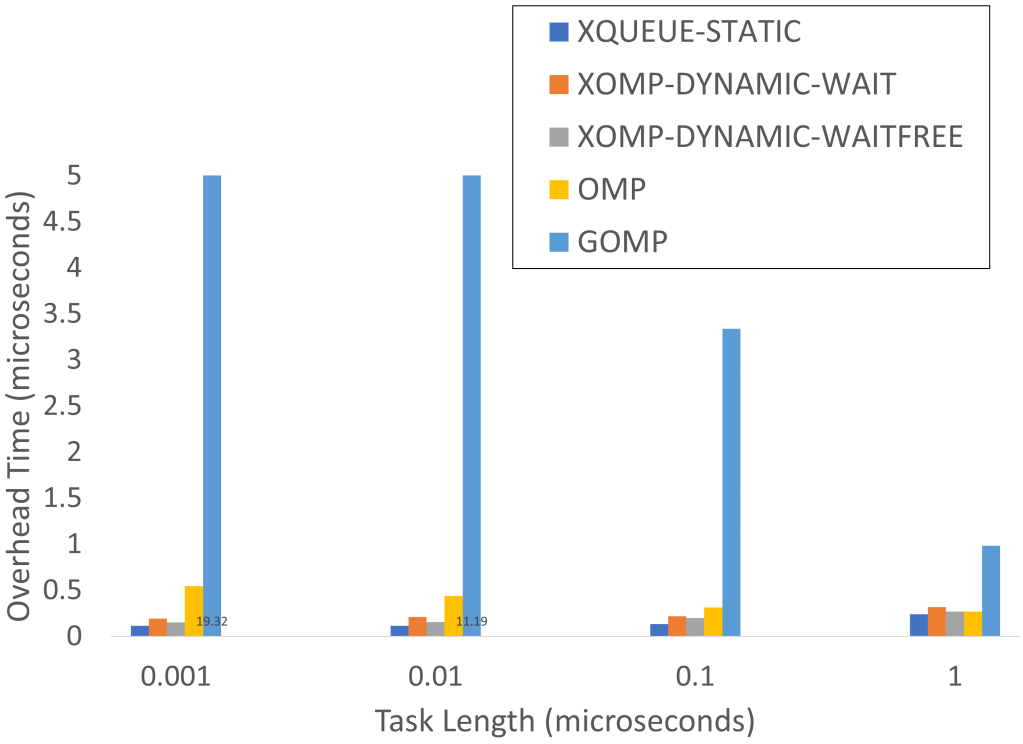


Figure 14 Parallel Tasking Overhead on skylake-192 using 192 threads (lower is better)

4.3.1 Microbenchmarks

To evaluate the overheads of tasking and to explore the scalability of X-OpenMP with extremely fine-grained tasks, we implemented a set of microbenchmarks inspired by the EPCC Benchmark Suite [19]. While the EPCC benchmark suite contains benchmarks for measuring the overheads of tasking and load balancing in OpenMP, these benchmarks are not sufficient for understanding the performance of the lock-less techniques described in this work. For the purposes of evaluation, each microbenchmark runs a loop that increments a variable for a certain number of iterations as a task. The number of iterations is derived based on the delay time specified in the benchmark by running a test loop. We refer to this task as the delay task. For benchmarking X-OpenMP, we designed 3 different microbenchmarks: (1) Tasking overhead - measures the overhead of launching a task of a certain length; (2) Task Distribution - measures how the tasks are distributed across workers when all workers are given an equal number of fixed length tasks; (3) Work Stealing Efficiency - measures the efficiency of work stealing when only the master worker receives all the tasks.

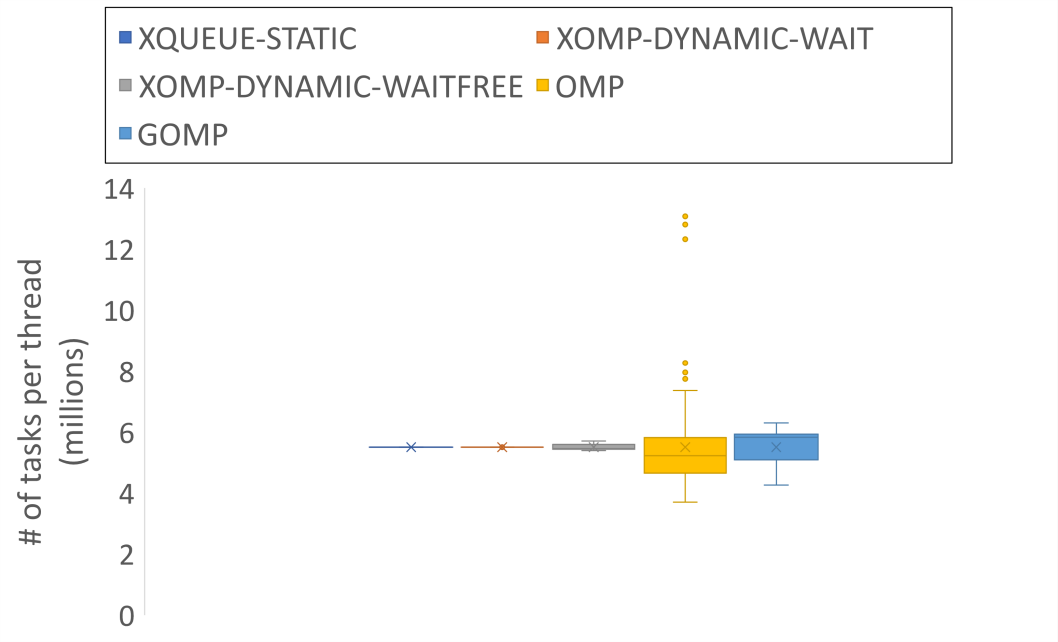


Figure 15 Task Distribution on skylake-192 using 192 threads

Figure 14 shows the overheads of tasking in microseconds for various versions of OpenMP using 192 threads. In this benchmark, each worker processes 8 million delay tasks where each task runs for a fixed length of 0.1 microseconds. The experiment is repeated 20 times and the plot shows the average execution time. The tasking overhead measured for X-OpenMP with static round-robin load balancing is about 110 nanoseconds. The overhead of X-OpenMP with workstealing is about 150 to 200 nanoseconds. In native LLVM OpenMP, the tasking overhead is about 400 nanoseconds. GNU OpenMP exhibits significantly higher overhead for extremely fine-grained tasks at about 20 microseconds for 1 nanosecond tasks, with the overhead going down up to 2 microseconds for 1 microsecond tasks. These results clearly illustrate that the overheads of tasking can be significantly reduced by using lock-less concurrent queuing mechanisms.

Figure 15 shows a box plot of task distribution across workers for 20 runs of 8 million fixed delay tasks using 192 threads. Every worker in the X-OpenMP implementation with static load balancing executes the same number of tasks due to the absence of dynamic load balancing. LLVM and GNU OpenMP versions spend significant time in load balancing depending on the execution speed of each worker. The tasking overhead plays a significant role in triggering work stealing, since higher overhead for pushing tasks implies that the workers are idle for a long time which triggers work stealing even when it is not necessary. X-OpenMP with wait-based and wait-free

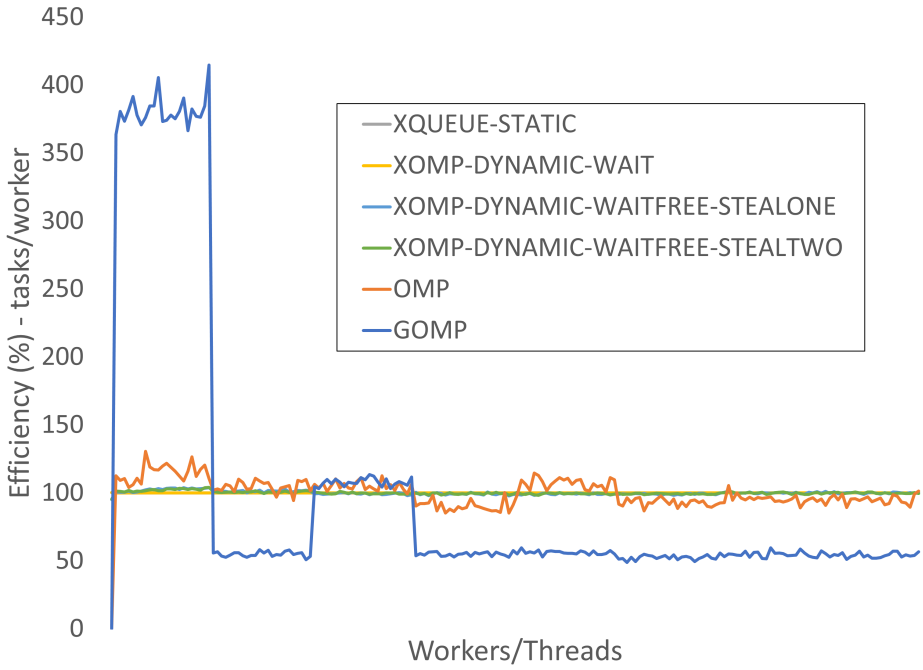


Figure 16 Efficiency of Work Stealing on skylake-192 using 192 threads (closer to 100% is better)

workstealing approaches also steal tasks in order to load balance, however the standard deviation is low compared to the native LLVM and GNU versions. Overall, the execution time is directly correlated with the number of tasks executed by each worker. Compared LLVM and GNU versions, X-OpenMP run about 36% faster in this microbenchmark. This slowdown is due to the overheads of enqueueing and dequeuing in lock-based approaches used in LLVM and GNU versions.

Figure 16 shows the efficiency of work stealing across 192 workers. This benchmark creates an OpenMP parallel region and the master thread runs a for loop which creates 65K delay tasks with 0.1 microsecond delay. This experiment is repeated 20 times and we count the total number of tasks processed per worker. The plot shows the efficiency of each worker based on the task distribution across all the runs. Efficiency is calculated by taking a ratio of actual task count over the ideal task count. The ideal case is when every worker runs an equal number of tasks which implies the efficiency is 100%. The efficiency for all versions of XOMP ranges between 95% and 105% and for the native LLVM OpenMP version, the efficiency ranges between 90% and 120%. GNU OpenMP shows significant variance in the task distribution which is also observed in the overall execution time and it runs about 5X to 10X slower compared to the native LLVM and XOMP versions.

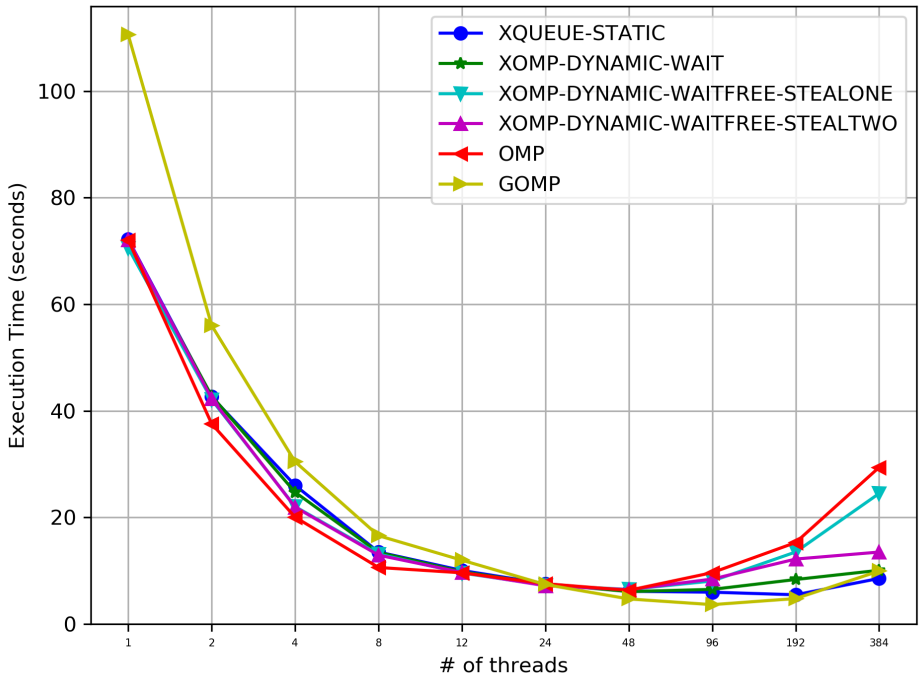


Figure 17 Scaling of Strassen's Matrix Multiplication using 8K matrix on skylake-192 (lower is better)

The main takeaway from this benchmark is that lock-less implementations of work stealing perform similarly to traditional work stealing implementations.

4.3.2 Macrobenchmarks

To demonstrate the behavior of X-OpenMP in real application scenarios, we chose benchmarks which are most studied, fundamental and relevant to real-world HPC applications: a matrix multiplication benchmark, two linear algebra routines, and an unbalanced tree search benchmark. We evaluate these applications on the skylake machine with 192 cores using various versions of X-OpenMP and compare with LLVM and GNU versions.

Strassen's Matrix Multiplication [36, 52] is a parallel algorithm that uses the divide and conquer approach to multiply two square matrices. A large matrix is divided into smaller and smaller matrices by recursion. When the algorithm reaches the base size, it computes the matrix multiplication using a divide and conquer approach. The depth based cutoff value for divide and conquer algorithm is set to 3.

Figure 17 shows the scalability plot for Strassen's matrix multiplication algorithm. The experiment multiplies square matrices of size 8192x8192 us-

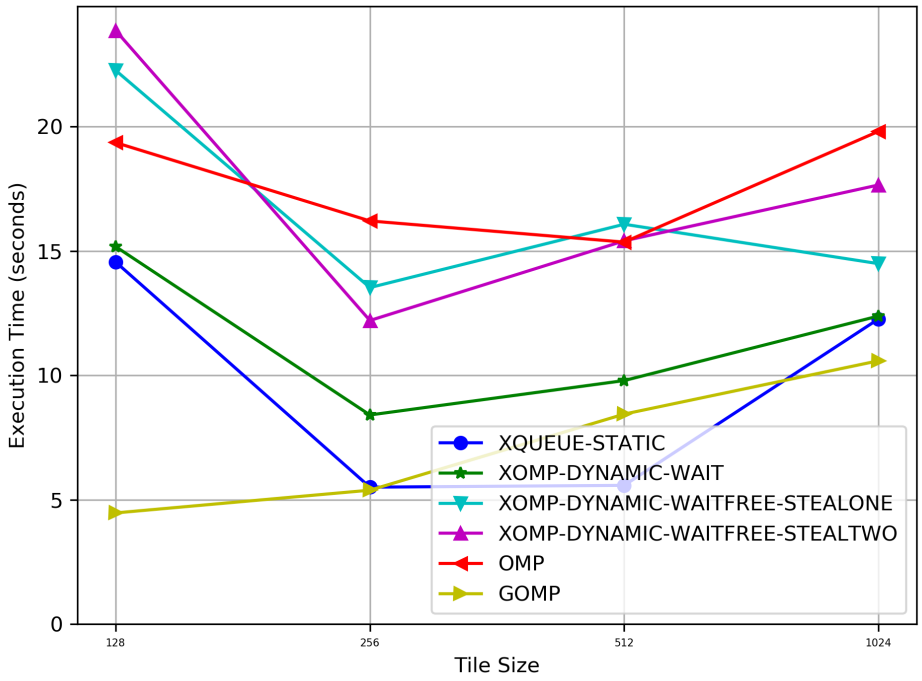


Figure 18 Performance of Strassen's Matrix Multiplication using 8K matrix and varying base sizes on skylake-192 (lower is better)

ing the recursive algorithm and base condition is set to 256 since it gives the fastest execution time for most implementations (see below). The results show that the implementation scales up to 96 threads and then performance degrades. GNU OpenMP is the fastest and runs in 3.6 seconds using 96 threads, followed by XOMP-STATIC which runs in about 5.9 seconds. The native LLVM version runs about 5% slower than X-OpenMP using 96 threads. It is interesting to note that while GNU OpenMP scales well beyond 96 threads, the LLVM OpenMP quickly degrades in performance.

Figure 18 shows the results obtained by running Strassen's algorithm on an 8192x8192 matrix using 192 threads and varying base sizes for the matrix from 128 to 1024. The plot shows the average of three runs. The best performance is achieved using base sizes of 256 and 512 in case of X-OpenMP. It is worth noting that X-OpenMP using static load balancing is sufficient to achieve good performance for this algorithm. Dynamic work stealing induced additional overhead increasing the overall running time for this application, however the behavior is specific to this algorithm.

LLVM OpenMP is much slower compared to the other implementations for Strassen's matrix multiplication using 192 threads. At this concurrency scale, the runtime incurs significant overheads due to wait time and synchroniza-

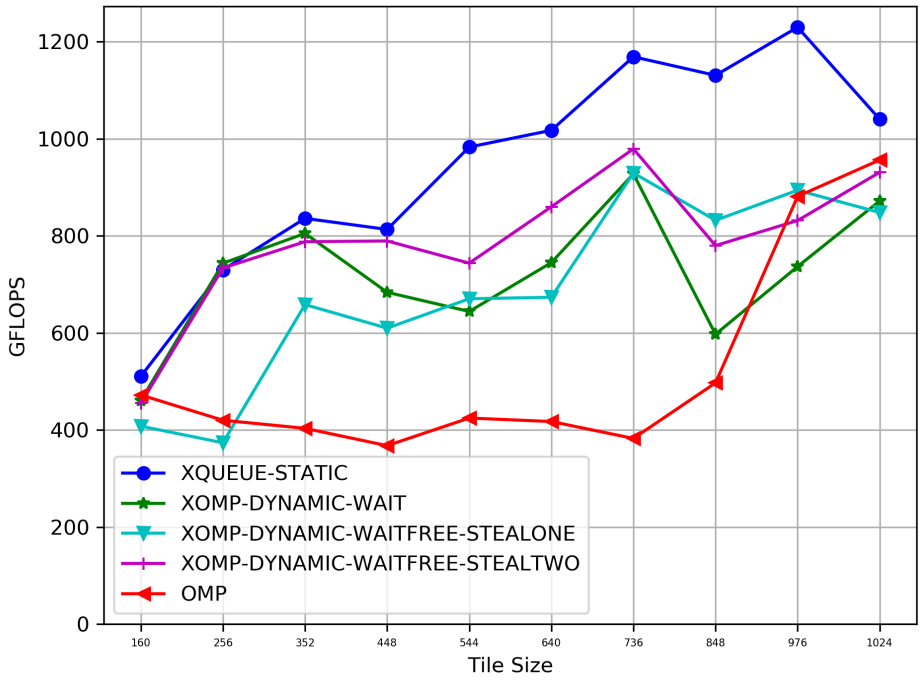


Figure 19 Symmetric Rank Update using 12K matrix on skylake-192 using 96 threads (higher is better)

tion which results in high cycles per instruction rate. This algorithm is also highly memory intensive and memory profile of the application showed high memory pressure on one numa node compared to the others for all the runtimes.

Symmetric Rank-k Update (SYRK) [22] is an important building block of many linear algebra algorithms and included in the Basic Linear Algebra Subprograms (BLAS) specification [34]. The SYRK algorithm computes the upper or lower part of the result of a matrix product where the given matrix is a symmetric matrix. Parallel Linear Algebra Software for Multicore Architectures (PLASMA) numerical library [32] is a dense linear algebra package which implements a full set of BLAS routines using task-based parallelism. PLASMA library uses a tile-based approach for the algorithms where the matrix is divided into square blocks and each tile is typically processed by a task.

Figure 19 shows the results obtained by running DSYRK on skylake-192 using 96 threads and varying tile sizes. The algorithm scales up to 4 sockets and 96 threads on the skylake-192 server. X-OpenMP with static round-robin load balancing achieves the highest floating point operations per second with 1229 GFLOPS at 976 tile size. X-OpenMP with wait-based work

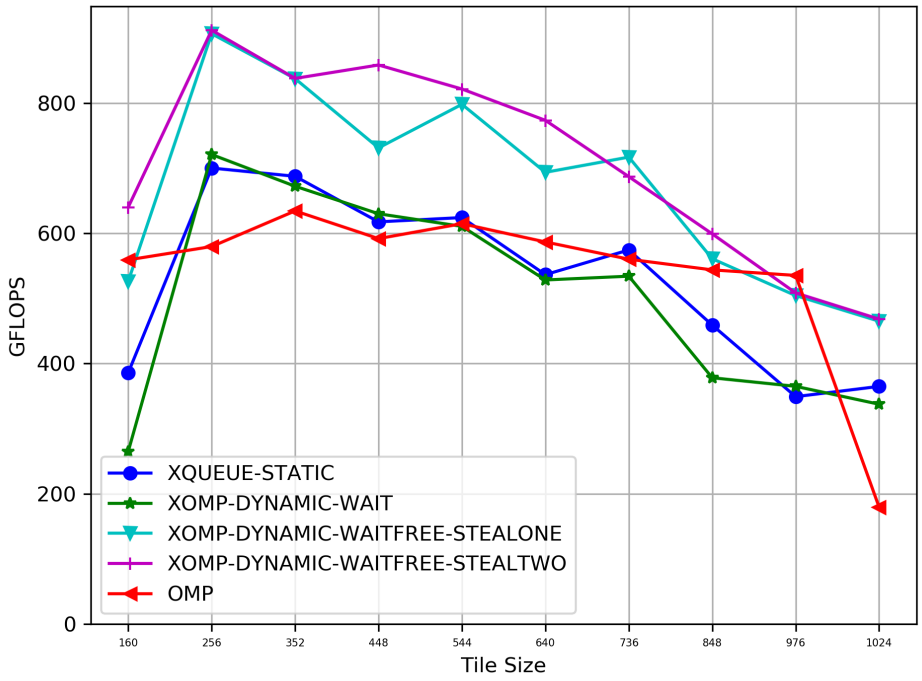


Figure 20 Cholesky Factorization on a 12K matrix on skylake-192 using 96 threads (higher is better)

stealing approach achieves 927 GFLOPS using 848 tile size. X-OpenMP with the wait-free approach and stealing two tasks at a time achieves 979 GFLOPS using 736 tile size. The native LLVM version achieves 956 GFLOPS using 1024 tile size, however it is about 50% slower with smaller block sizes. These results clearly illustrate the importance of low overhead tasking [50] for achieving high performance on modern machines with hundreds of cores.

Cholesky Factorization (POTRF) of a symmetric positive definite matrix is the factorization of the matrix into upper triangular and lower triangular matrices with positive diagonal elements. Several prior works have explored task-based Cholesky factorization algorithms and we evaluate the DPOTRF algorithm from the PLASMA numerical library which is a tile-based implementation using OpenMP tasking. Cholesky factorization uses DPOTRF for factorization of a tile and uses three kernels from the library for the algorithm: DGEMM (general matrix matrix multiplication), DTRSM (for solving a system with a triangular matrix) and DSYRK (for rank-k update of the symmetric matrix).

Figure 20 shows the performance of Cholesky Factorization algorithm on skylake-192 server using 12K matrix, 96 threads and varying tile sizes. The

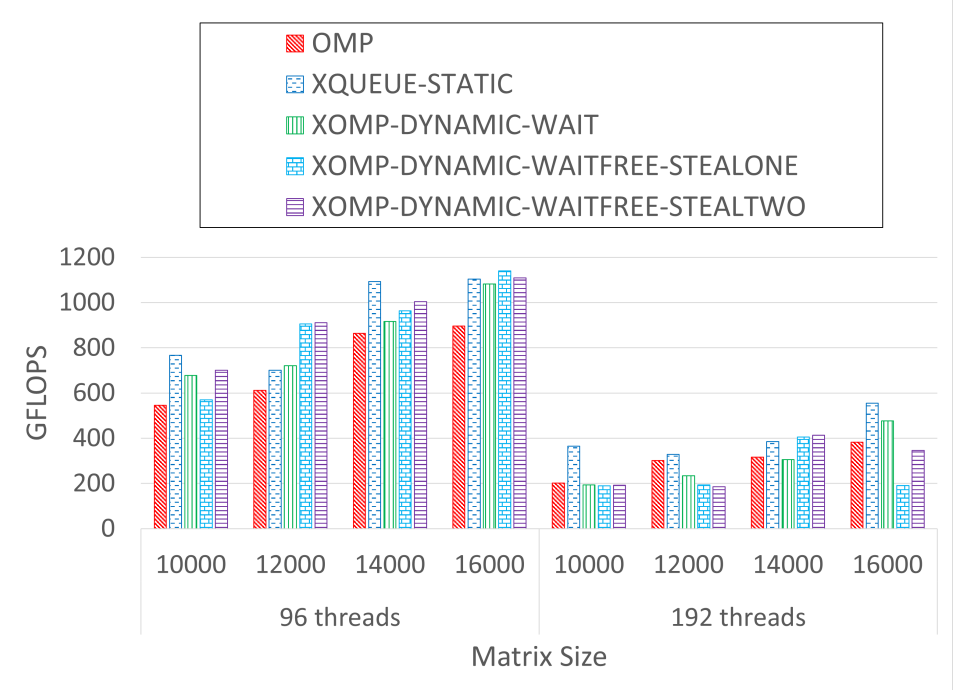


Figure 21 Cholesky Factorization using different matrix sizes and tile size of 256 on skylake-192 run using 96 and 192 threads (higher is better)

highest performance of 911 GFLOPS is achieved using a tile size of 256. X-OpenMP with wait-free work stealing performs best for this algorithm. Stealing two tasks instead of one seems to achieve better performance for some tile sizes and overall the dynamic work stealing highly improves the performance compared to native LLVM OpenMP. It is worth noting that the native LLVM version achieves peak performance using tile size of 352, and all versions of X-OpenMP achieve peak performance using a tile size of 256. This clearly shows that the lightweight tasking and reduced synchronization overheads can help speed up applications using tasks of much finer granularity than is possible in today's runtime systems. This also highlights the potential to explore over decomposition of task-based applications to achieve maximum speed up on modern architectures. The algorithm using GNU OpenMP takes a long time to execute and it results in very low GFLOPS for both DPOTRF and DSYRK algorithms, hence we have not included the results in the plots. We plan to explore the reason further and include the results in the final revision.

Figure 21 shows the results obtained by executing Cholesky Factorization on different matrix sizes on the skylake-192 server. The experiment is performed using 96 threads and 192 threads using native LLVM OpenMP and various X-OpenMP implementations. As mentioned earlier, the current im-

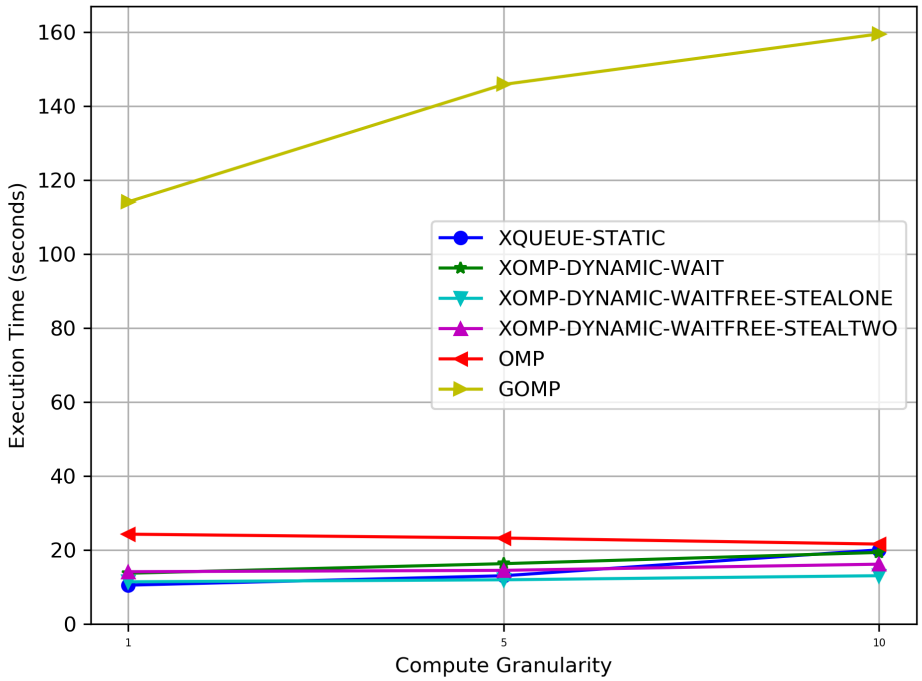


Figure 22 Unbalanced Tree Search using 96 threads on skylake-192 (lower is better)

plementation of this algorithm scales up to 96 threads and the performance drops significantly using 192 threads. X-OpenMP consistently achieves 20% higher performance using 96 threads for all matrix sizes evaluated. X-OpenMP using static load balancing and wait-free based single task work stealing achieve high performance compared to the other implementations of X-OpenMP.

Unbalanced Tree Search (UTS) [83] benchmark is designed to evaluate the performance of dynamic load balancing in task parallel runtime systems. The benchmark implements a version of UTS using OpenMP tasking where workstealing is used to reduce the load imbalance between workers. We chose this benchmark since it requires efficient dynamic load balancing to achieve good performance. The benchmark traverses all the nodes of a tree with a parameterized size and imbalance and reports the total number of nodes in the tree. The benchmark provides sample trees for the purposes of evaluation. We evaluate T3L which is binomial tree with over 100 million nodes with 17844 tree depth and close to 90 million leaf nodes. We report the results of running UTS using 96 threads and 192 threads on skylake-192 server.

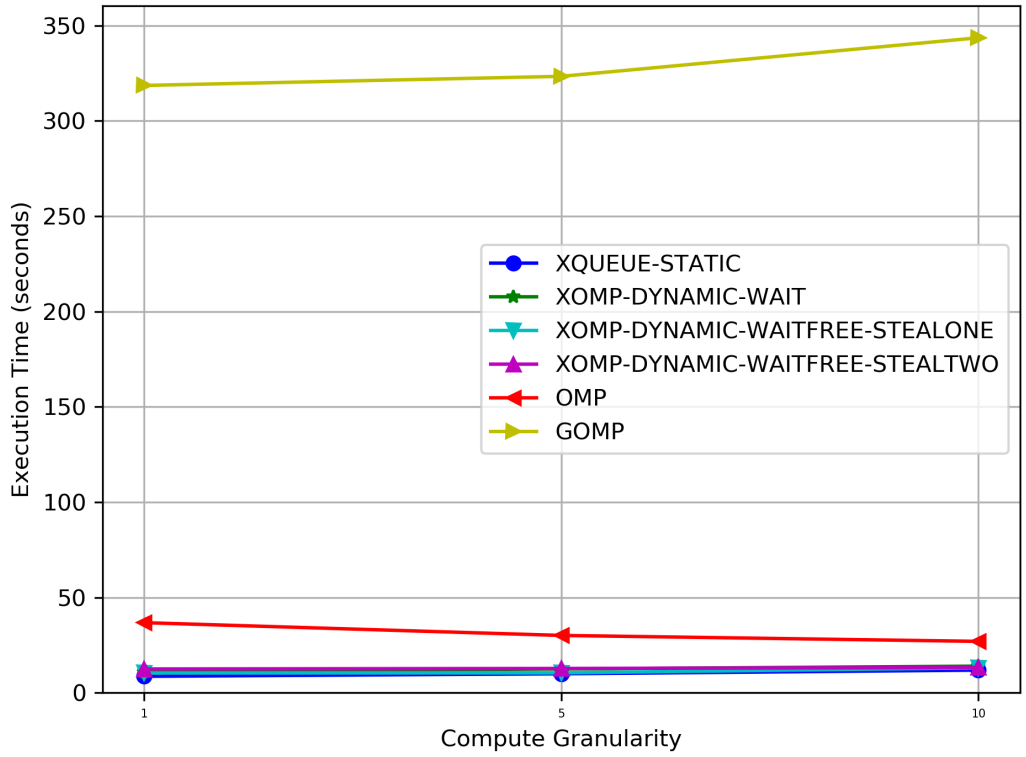


Figure 23 Unbalanced Tree Search using 192 threads on skylake-192 (lower is better)

Figures 22 and 23 show the execution time of T3L using 96 threads and 192 threads on the skylake-192 server. As with the other benchmarks, UTS benchmark also scales up to 4 sockets and 96 threads on this machine using LLVM and GNU OpenMP, X-OpenMP scales up to 192 threads. To understand the impact on performance due to high tasking overheads at high levels of concurrency, we evaluated the application using 96 threads and 192 threads. The plots show execution time of UTS at varying levels of compute granularity. The granularity defines the amount of compute for each task, with 1 being the finest granularity and 10 being the coarsest. For all fine, medium and coarse grain tasks, X-OpenMP with static round robin load balancing achieves the best execution time of 8.6 seconds, 9.9 seconds, and 11.8 seconds respectively using 192 threads. GNU OpenMP incurs significant overheads with this workload with about 40X slowdown across all granularities. Figure 24 shows a part of the timeline plot of one execution of UTS using T3L graph and X-OpenMP. The static round-robin load balancing coupled with dynamic work stealing achieve good task distribution across all the workers. Although the nature of the workload is highly imbalanced, X-OpenMP achieves a reasonable load balance and speed up compared to the other OpenMP implementations. These results showcase the signifi-

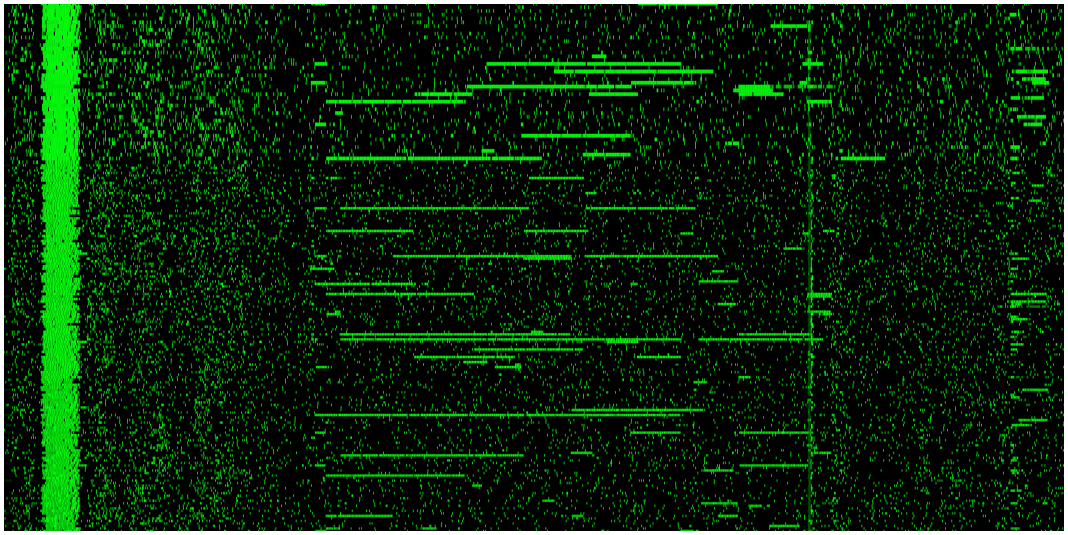


Figure 24 Unbalanced Tree Search using X-OpenMP and 192 threads on skylake-192

cance of better load balancing to achieve improved performance. Using 96 threads, the best execution time is achieved using X-OpenMP with static round robin load balancing at the finest granularity. For medium and coarse granularities, X-OpenMP with wait-free load balancing and stealing one task at a time performs the best at 11.9 seconds and 13.1 seconds. X-OpenMP is 10X faster than GNU OpenMP and 2X faster than LLVM OpenMP using 96 threads.

4.3.3 Results Discussion and Summary

This evaluation showed that static load balancing mechanisms are suitable for some applications, while others require more dynamic approaches. Configuring how many tasks to steal at a time is dependent on the application and the computational complexity of the tasks. If tasks are of similar lengths in terms of execution time, static round-robin load balancing along with stealing one task at a time works well. For highly imbalanced applications, traditional work stealing approaches can incur extremely high overheads due to synchronization at higher concurrency levels. Such applications can benefit from lock-less approaches presented in our work. Most state-of-the-art applications do not scale up to hundreds of threads on modern architectures and the applications must be redesigned to achieve further improvements in performance using extremely fine-grained tasks.

These experimental results clearly demonstrate the performance improvements that can be achieved using lightweight tasking and reduced synchro-

nization overheads. The techniques presented in this work can be used to enhance existing parallel runtime systems to improve the efficiency of fine-grained parallelism on many-core architectures.

4.4 Conclusion

We propose X-OpenMP as a framework to enable extremely fine-grained task parallelism on modern shared memory architectures with hundreds of cores. We extend our prior work on lock-less queuing mechanisms with static load balancing and propose an algorithm for achieving dynamic load balancing using work stealing. The work stealing algorithm in X-OpenMP does not require any atomicity for read, write, and modify operations, and achieves competitive performance with state-of-the-art implementations. X-OpenMP extends LLVM OpenMP using our techniques described in this work. As a result, existing OpenMP applications can run unmodified just by linking against the X-OpenMP library. We evaluate our approach using workloads that are highly prevalent in HPC applications and are crucial for achieving better performance in real-world scenarios. We demonstrate speedups of up to 40X compared to GNU OpenMP and up to 2X compared to the native LLVM OpenMP implementation.

CHAPTER 5

The Template Task Graph (TTG) — an emerging practical dataflow programming paradigm for scientific simulation at extreme scale

Our work in X-OpenMP shows that it is possible to implement task-parallel execution models using lockless techniques to achieve significant speedups on shared memory manycore architectures. However, there are several applications that require dataflow based parallelism [11, 27, 18] to overcome the limitations of fork-join based parallelism by specifying data dependencies at finer granularity and allowing tasks to execute as soon as data dependencies are satisfied. Fork-join parallelism can introduce artificial dependencies in applications that are data-parallel and not task-parallel. Hence, it is imperative to explore this space in order to cover a broader scope of applications which are irregular and hard to speedup using existing parallel execution models.

This work is inspired by the belief that flowgraph programming (FGP) is a superior match for (1) high-performance parallel programming of modern computers with complex (distributed/heterogeneous) memory hierarchies and a large number of, potentially heterogeneous, compute resources, (2) *irregular* (scientific) applications characterized by data-dependent operation streams, and (3) combinations thereof. This belief is reflected by many efforts employing data- and control/work-flow programming to simplify parallel programming as an alternative to traditional bulk-synchronous models. The

advantages of FGP are due to several traits: (1) specification of only *essential* dependencies between operations maximizes exploitable concurrency and opportunities for hiding latency by overlapping data motion and computation, (2) making the data part of the flow, a *dataflow* reduces the need for synchronization, eliminates scheduling delays, and makes operations easier to reuse by eliminating nonessential side effects, and (3) by raising the level of abstraction, programs (often) become easier to write, easier to transform (thereby supporting the development of domain-specific languages), and easier to port. While these advantages are not unique to FGP, achieving them via more conventional means typically involves significant programming costs due to the low-level (and sometimes explicit) management of asynchronous execution. Thus FGP is uniquely positioned to address the tension between programmer productivity and the programming challenges posed by the ever-increasing complexity of hardware and applications.

5.1 TESSE - Task-based Environment for Scientific Simulation at Extreme Scale

TESSE attacks the twin challenges of programmer productivity and portable performance for advanced scientific applications on massively-parallel hybrid systems with complex disjoint memories. Of specific interest are irregular computations which are hard to compose and execute efficiently with mainstream parallel programming paradigms. There are several source of the Irregularity in modern advanced scientific applications. First, there is the irregularity of the data itself: as greater simulation size and/or higher precision are targeted dense data structures (uniform meshes, dense tensors) must be replaced by their data-sparse counterparts (adaptively-refined meshes, block-sparse and ranksparse tensors) to keep the simulation cost tractable. Second, as applications become more complex (e.g., due to multiple physics models being coupled) and as they seek greater concurrency on increasingly heterogeneous hardware, it also becomes necessary to execute multiple (potentially, dissimilar) operations in parallel. The ensuing irregularity and associated resource management and other issues are commonly resolved by static partitioning of processors, or centralized job stealing. Both approaches have been successful as long as the computations involved were large enough to hide the cost of scheduling and/or migration, the two pillars of most of the existing solutions. Unfortunately, as exemplified by the two paradigmatic science applications motivating TESSE (fast tree-based computation on deeply refined numerical meshes, and block-sparse

tensor algebra in many-body quantum simulation) exploiting sparsity usually leads to highly irregular fine-grained computation as well as highly data-dependent data/work flows that are very dynamic in nature. To be able to compose and execute efficiently the irregular computation patterns underlying these and other modern scientific applications in TESSE we adopted the ideas of dataf low and other flow programming approaches to raise the level of abstraction beyond the relatively low-level APIs of modern task-based programming models and runtimes. The key innovation of TESSE is TTG, a flow programming model inspired by earlier innovations such as Flow-Based Programming (FBP) [2]. Unlike the earlier uses of flow programming, the targets of TTG are modern scientific algorithms to be deployed to current and near-future supercomputers, hence the efficient utilization of hardware resources, distributed-memory, and heterogeneity are all first-class concerns. Although the key innovations of TTG are not tied to particular choice of implementation, we implemented TTG as a library in C++ for general applicability and close-to-metal efficiency. TTG can be viewed as marrying the ideas of flow programming models with the key innovations in the PARSEC runtime [3] for compact specification of task DAGs, namely the Parameterized Task Graph (PTG; section V) [4] in which each edge represents a flow of data associated with a parameter identifying the particular data and, equivalently, the receiving task. Simplistically, such a parameter represents a loop index or data structure coordinate (e.g., integer tuple addressing elements of a tensor). Thus, each edge and vertex in PTG encodes several edges and vertices in a DAG of tasks, allowing potentially massive task DAGs to be represented compactly as well as instantiated across a narrow wave front only as needed for execution. The algorithms of dense linear algebra are naturally expressed within the PTG as a workflow over mutable data that fully captures the lifetime of a datum including whether it is created, read, written, modified, or consumed by a given task. This parameterization combined with the deep understanding of what tasks are doing with data helps reduce resource utilization and data motion, and enables efficient placement and scheduling through use of temporal/spatial locality. The TTG extends the idea of PTG by generalizing the notion of parameters to arbitrary types and enabling datadependent selection of task dependencies, which allows to dynamically build the DAG of tasks depending on computations within the predecessor tasks. The TTG implementation replaces the standalone DSL for specifying PTG in PARSEC by a high-level programming API realized as a modern C++ library (the 2017 ISO standard of C++ is used). The use of modern C++ allows for type information to be utilized to ensure correct-

ness when constructing graph as well as for optimizations (e.g. consuming data passed by rvalue references). Lastly, TTG also introduces some concepts from flow programming models, such as programmable terminals, that were not part of PTG. Thus TTG is a major advance of the successful idea of PTG towards general-purpose computation; unfortunately, the general-purpose character of TTG makes it challenging for TTG to retain all of the optimizations feasible within the PTG; making TTG exploit the full capabilities of PARSEC runtime is the focus on the ongoing work.

5.2 Template Task Graph

TTG represents an algorithm as a flowgraph (*template task-graph*, TTG) composed of one or more nodes (*template tasks*) equipped with ordered sets of input and output *terminals* connected by directed *edges*. In the current C++ implementation of *TTG*, template tasks, terminals, and edges are explicitly and strongly typed. Edges encode all possible flows of *messages*. Each message consists of a *task ID* and *data*; this idea builds on the concept of the Parameterized Task Graph (PTG) [27]. The task ID represents the task (instance of a task template) for which the data is intended. Thus, messages in the *TTG* model generally contain both a control part (task ID) and data part, allowing to marry the control-flow and data-flow paradigms. Pure control flow can be implemented by omitting the data part, i.e., by using the null type (void) to represent the data part of the message. Pure dataflow can be implemented analogously by using the null type to represent the task ID.

Once every input terminal of a given template task has received one message with the same value of task ID, a task is created with the data parts of the corresponding messages. Tasks define a *task body*, which is a C++ method that will be executed by the runtime system. *TTG* does not constrain the task bodies in any way (i.e., the tasks can be arbitrary, not necessarily pure, functions) but any side effects may require additional synchronization to avoid data races. During its execution, the task may deliver new messages to zero or more output terminals. Introducing the data dependence into the control flow (i.e., by deciding whether a particular output terminal will receive a message or not, or by making the task IDs of the outgoing messages dependent on the data contents of the input messages) allows to implement general data-dependent task flows in *TTG* seamlessly. Thus, the message flow through a TTG generates a set of tasks representing an

application. Each TTG can be viewed as encoding a set of possible directed acyclic graphs (DAGs) of tasks with the actual DAG executed being dependent on the data flowing through it.

5.2.1 TTG Concepts

Central concepts are:

- TaskId: A unique identifier for each task. For example, if computing on a vector it might be the vector/loop index, or if computing on a database it might be the name of a record, or in a matrix-multiplication algorithm it might be the triplet of integers identifying the tiles being operated upon. The only constraint on the type of TaskId is that it be hashable. The TaskId is used by the runtime to identify the compute resource (process rank, gpu, etc.) for the task using an optional user-defined map, and when a task sends data to a successor the same map is used to route data. This map is thus the primary tool for balancing load and data. A TaskId of type void implies a singleton task on process rank zero.

- Terminal: Each input argument and output result of a (template) task are exposed to the programmer and runtime as a Terminal. A task propagates a result or output value to a successor task by sending the value and the successor's TaskId to the appropriate output Terminal. Broadcast to multiple values of TaskId is supported. By default, an input Terminal is a singleassignment variable, this property being used by the runtime to determine when arguments of a task are available. However, an input Terminal is programmable and, for instance, could perform a reduction operation. If the number of expected input values is fixed, the runtime can determine completion, but with variable length (streaming) data either the user-provided reduction operation or a predecessor task must finalize the argument.

- Edge: Programs are composed by connecting output terminals with input terminals, currently identified by position but by name is planned. Multiple edges can connect to an input terminal, enabling data to come from multiple sources, and an output terminal might connect to multiple successors implying a broadcast operation.

- TemplateTask: This wraps a user-defined function with informal signature void f(TaskId, Arg0, Arg1, ..., OutputTerminals). Again, each input argument is exposed as a Terminal, and OutputTerminals is a tuple of the output terminals (an alternative interface also provides the input arguments as a tuple of references). The task associated with a specific TaskId is instantiated when any input Terminal receives a value, and a task is marked ready for execution when all arguments are finalized. If there are no arguments, the task must be created either manually via a special method (invoke(TaskId))ofthe TemplateTask, or via a pull operation as described below. Most users will instantiate a TemplateTask by invoking the `make_tt` factory function that deduces type information from the signature of the user's function, as illustrated below. However, a user-defined class can derive from the TemplateTaskBase class template using the curiously recurring template pattern that enables the base class to access methods of the derived class. As originally conceived and important for distributed-memory computers, tasks were assumed to only receive data through their input terminals and to have sending data to output terminals as their only side effect. However, there is no constraint on this behavior and work flow over mutable data is also readily composed.

- CompositeTemplateTask: This exposes the same API as TemplateTask but wraps an entire subgraph exposing input and output terminals as selected by the programmer.

- Push versus pull: As described so far, data must be pushed from a task's output terminal into a successor's input terminal. However, many algorithms, such as those operating on pre-existing data structures, can be more easily composed and more efficiently executed by pulling data as needed. This is accommodated by connecting terminals via a pull-Edge. When a task is instantiated, the runtime checks each input terminal to see if its value should be pulled, in which case the necessary predecessor task (the TaskId of which is computed from the current task's TaskId via a user-defined function) is instantiated. This can be done recursively and lightweight operations, such as reading a value from local memory, can be directly invoked to avoid the overhead of task creation.

Given a user function (f) with the required signature (see TemplateTask above), a call to the `make_tt` factory would be used as

```
auto tt = make_tt(f, input_edges, output_edges, task_name,
    input_terminal_names, output_terminal_names);
```

in which `input_edges` and `output_edges` are possibly tuples of edges to connect to each terminal, and `task_name`, `input_terminal_names`, and `output_terminal_names` are optional names for the task and terminals.

5.2.2 Cholesky Decomposition Example Using TTG

To illustrate these concepts, we consider the well-known algorithm for (non-pivoted) Cholesky factorization of a dense tiled matrix used in the standard distributed-memory linear algebra package *ScaLAPACK* [25] and whose TTG implementation will be assessed in subsection 5.3.2. Figure 25 illustrates its template task graph. The algorithm consists of 4 types of tasks: `POTRF` (Cholesky factorization of diagonal tiles), `GEMM` (generalized matrix multiply), `SYRK` (symmetric rank-k matrix update), and `TRSM` (triangular linear system solver). Each task type is represented by a node in TTG, with two additional nodes representing reading of the input data (`INITIATOR`) and writing the output data (`result`).

```
/* Edges with 1-tuple task IDs */
ttg::Edge<Int1, Tile> init_potrf;
/* Edges with 2-tuple task IDs */
ttg::Edge<Int2, Tile> potrf_trsm, trsm_result,
                      trsm_syrk, gemm_trsm;
/* Edges with 3-tuple task IDs, encodes the iteration K */
ttg::Edge<Int3, Tile> trsm_gemm_row, trsm_gemm_col;
auto POTRFOp =
   ttg::make_tt(potrf_fn /* not shown here */,
                /* input edges */ ttg::edges(init_potrf),
                /* output edges */
                ttg::edges(potrf_results,
                           potrf_trsm));
auto trsm_fn =
    [](const Int2& id, const Tile<T>& tile_kk,
       Tile<T>&& tile_mk,
       std::tuple<ttg::Out<Int2, Tile<T>>,
                  ttg::Out<Int2, Tile<T>>,
                  ttg::Out<Int3, Tile<T>>,
                  ttg::Out<Int3, Tile<T>>>& out){
  const auto [I, J] = id;
  const auto K = J;
  /* call LAPACK library's tsrm function */
  TRSM(tile_kk, tile_mk);
  std::vector<Int3> row_ids, col_ids;
  /* ids for gemms row I */
  for (int n = J+1; n < I; ++n)
    row_ids.push_back(Int3(I, n, K));
  /* ids for gemms column I */
  for (int m = I+1; m < NROWS; ++m)
    col_ids.push_back(Int3(m, I, K));
  /* broadcast the result to 4 output terminals:
   * 0: to final output task writing back the tile;
   * 1: to the SYRK kernel;
   * 2: to the gemm tasks on in row I;
```

```
36     * 3: to the gemm tasks in column K; */
37   ttg::broadcast<0, 1, 2, 3>(
38     std::make_tuple(id, Int2(I, K), row_ids, col_ids),
39     std::move(tile_mk), out);
40 };
41 auto TRSMOp = ttg::make_tt(trsm_fn,
42                            /* input edges */
43                            ttg::edges(potrf_trsm, gemm_trsm),
44                            /* output edges */
45                            ttg::edges(trsm_result, trsm_syrk,
46                                       trsm_gemm_row,
47                                       trsm_gemm_col));
48
```

5.2.2 illustrates how the TTG is composed by connecting inputs and outputs of each task template to the edges (represented in C++ by `ttg::Edge`). Note that each output terminal may be attached to one or more input terminals. Each task template is typically composed from a free or lambda function by calling `ttg::make_tt` (Lines 9 and 41). 5.2.2 illustrates also how the `TRSM` task template is implemented. The lambda (or free function) implementing a task body receives as its arguments the task ID (if non-void), input data (if non-void), and the tuple of output terminals (`ttg::Out`; Lines 14–20). The function body performs arbitrary computation on the data and, if needed, "sends" the data to the output terminals via `ttg::send` (if intended to be an input for a single task) or `ttg::broadcast` (if intended to be an input for multiple tasks; Lines 37–39). Since the edges, input, and output terminals are all explicitly parametrized by the type of data they transport the type safety of TTG's edges and task templates is checked at compile time. Note that the graph built by connecting the nodes that represent task types via edges includes cycles and thus does not represent directly the DAG of tasks. It is during the execution, when tasks are instantiated with their task IDs, that the DAG of task is constructed, distributed across processes, by each task instance that discovers a new task instance.

The task ID of a given task does not have to match, or even be of the same type as, the task IDs of its output terminals. For example, the `TRSM` task IDs are represented by a 2-tuple (`Int2`) and produce data that will be used to create tasks with IDs represented by 3-tuples (`Int3`). Immutable data may be shared between tasks while tasks mutating inputs receive private copies, which may be passed on to other operations. Thus, applications need not be concerned with protecting access to data under *TTG*'s control. The safety of side-effects of tasks on data outside the control of *TTG* is under the purview of the application.

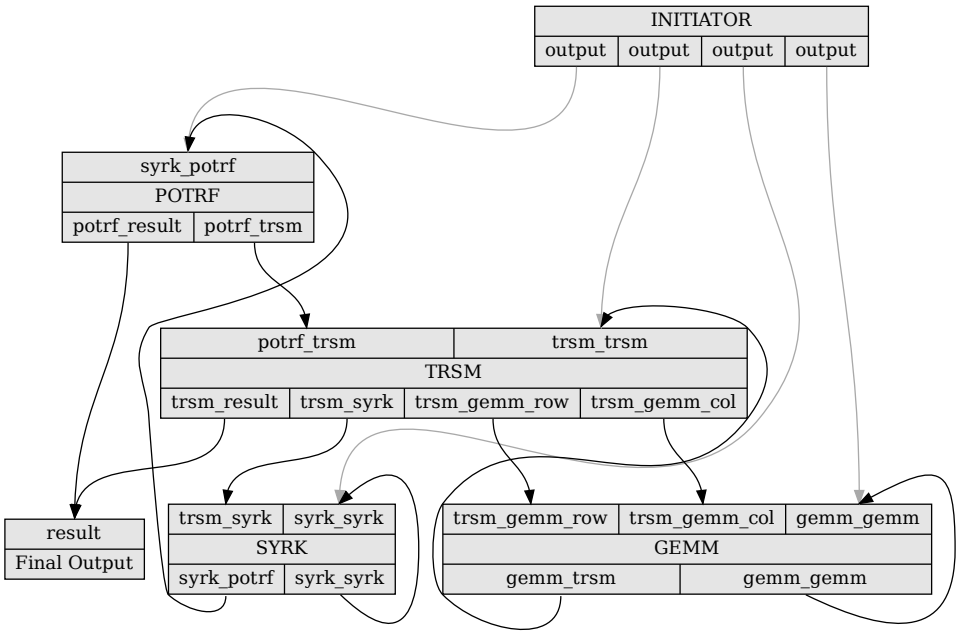


Figure 25 Template task-graph of the tiled Cholesky factorization. The `INITIATOR` operation is responsible for providing input to tasks that have no direct predecessor in the algorithm.

Once a task template receives all inputs needed for a given task ID the task is scheduled for execution. The process on which a given task will be executed is specified by a user-defined function mapping task IDs to process ranks. Note that creation and execution of tasks is entirely abstracted out in *TTG*. Thus, *TTG* can be viewed as a higher-level abstraction for a low-level task runtime. Current implementation of *TTG* can use one of two task runtimes for distributed task execution: *PaRSEC* and *MADNESS*. subsection 5.2.6 will discuss the relevant implementation details.

In recent work, the following features were added to *TTG*:

- the ability to assign *priorities* to tasks by supplying each task template with a *priority map* mapping a task ID to a specific task priority that is provided to the underlying runtime system;
- optimized implementation of `ttg::broadcast`, which appears as a common use case, e.g. in the `TRSM` task template in 5.2.2;
- streaming terminals that can receive not just a single message but a (bounded or unbounded) stream of messages;
- support for C++ data types serializable via general-purpose serialization frameworks, as well as support for RMA data transfers where supported by the runtime;

- an improved implementation of *TTG* over two runtimes, focusing on performance.

Several of these features are discussed in detail below.

5.2.3 Sending and Broadcasting

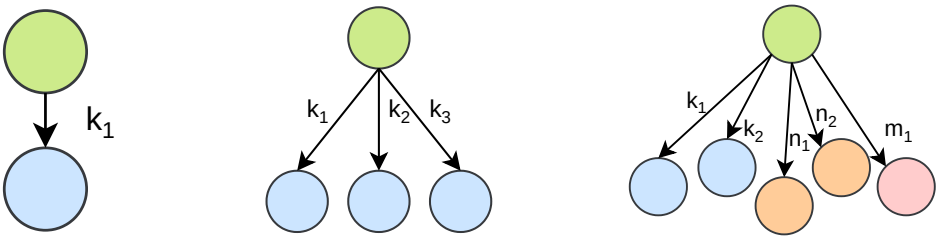
TTG supports several ways to send data out of tasks:

- to a single output terminal accompanied by a single task ID (`ttg::send`; see Figure 26a);
- to a single output terminal accompanied by several task IDs (`ttg::broadcast`; see Figure 26b);
- to multiple output terminals, each accompanied by one or more task IDs (`ttg::broadcast`; see Figure 26c).

The latter is used in the implementation of the `TRSM` task template shown in 5.2.2 (Lines 37–39). The data broadcasting was introduced to optimize data transfers between processes and avoid repeated transfers of the same data.

Note that by default `send` and `broadcast` both copy the argument data; this allows subsequent mutation of the data for sending it to other terminals. Passing data by constant reference indicates that the copying can be bypassed, if possible (e.g., if the lifetime of the object is already tracked by the runtime; see Section 5.2.6). To indicate that the data is no longer going to be used in the task template body the data can be passed by rvalue reference (via `std::move`); for types with efficient rvalue copies this allows to implement efficient (potentially, zero-copy within memory space) data flow through the graph. These customization mechanisms are illustrated in 5.2.3.

```
1 void taskfn(const TaskID& task_id, const MatrixTile& input,
2              tuple<Out<TaskID, MatrixTile>,
3                    Out<TaskID, MatrixTile>,
4                    Out<TaskID, MatrixTile>>& out) {
5   MatrixTile output = compute_output_tile(input);
6   send<0>(task_id, output, out); // new copy required
7   send<1>(task_id, move(output), out); // no copy due to move
8   send<2>(task_id, input, out); // no copy as input is const
9 }
```



(a) Send data accompanied by task ID k_1 to 1 output terminal

(b) Broadcasting data accompanied by task IDs $\{k_1, k_2, k_3\}$ to 1 output terminal

(c) Broadcasting data accompanied by task IDs $\{k_1, k_2\}$, $\{n_1, n_2\}$, $\{m_1\}$ to 3 output terminals

Figure 26 TTG send and broadcast operations.

5.2.4 Streaming Terminals

The original design of *TTG* mandated that each input terminal can receive only a single message for a given task ID. For some types of algorithms this restriction produces task templates with large numbers of input terminals. For example, a 1D Jacobi would only require 3 input terminals: the state of the task at the previous iteration as well as the state of the left and right neighbors. However, a 2D Jacobi requires 5 to 9 inputs (depending if neighbors on the diagonal need to be considered), and a 3D Jacobi quickly becomes un-manageable through explicit input terminals defined as independent variables in the user code. In this work, this restriction was lifted by making all input terminals capable of receiving a stream of messages for every task ID. The input messages are *reduced* (e.g., concatenated) using a user-provided function $U \otimes T \to U$ reducing a pair of values into a single value. Each incoming message is processed in a light-weight manner (i.e., without spawning a task) until either the prescribed number of messages has been received or the input terminal is programmatically "finalized" for the given task ID (see Figure 27). An example for using streaming terminals will be provided later in subsection 5.3.5.

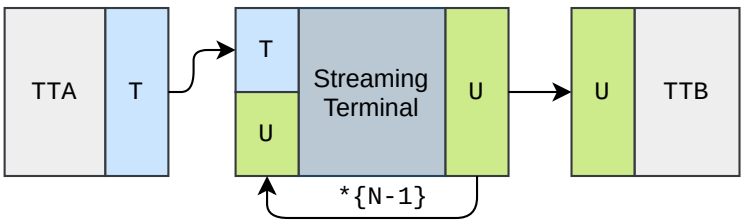


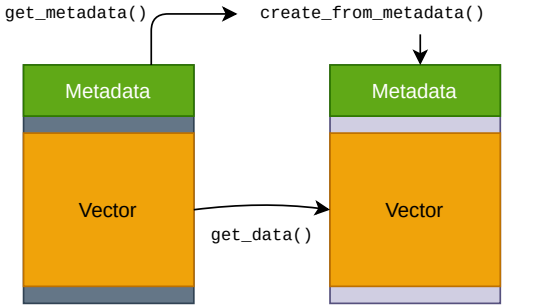
Figure 27 TTG streaming terminal with input `T`, output `U`, and a size of `N`. The reduction operation of the terminal will be called $N-1$ times on input from `TTA` before before a task of `TTB` will be eligible for execution.

5.2.5 Data serialization

Execution of *TTG* programs involving dataflow requires support for serialization of user data types, both for data between memory spaces (such as between host memories of different processes, or between host and device memory for a single process). The original implementation of *TTG* was limited to serialization of data types that were (1) trivially (bitwise) copyable, or (2) were serializable by the *MADNESS* runtime-provided serialization. In this work, *TTG* was extended to support data types serializable via the widely-available Boost.Serialization[1] library. Since stock Boost serialization archives provide a number of default features intended for archival purposes (type versioning, pointer tracking, etc.), they are ill-suited for high-performance applications like *TTG*. Therefore, support for Boost.Serialization-compatible types in *TTG* uses custom archives optimized for high-performance serialization into in-memory buffers. *TTG* also provides type traits that detect serializability of a given type via Boost.Serialization, *MADNESS*, or by `memcpy`, and makes the optimal choice of serialization protocol. Thus several mechanisms of serialization are provided for a given flowgraph.

Unfortunately, the default serialization protocols that *TTG* can exploit necessarily involve multiple copies (object to/from serialization buffer to/from MPI message buffer, etc.). To increase the efficiency of data flow, a split-metadata (`splitmd`) mechanism was implemented in *TTG* in the course of this work. Unlike Boost.Serialization and its sibling *MADNESS* serialization protocols, in which the entire object is serialized and transferred as a whole, `splitmd` is a 2-stage protocol (see Figure 28). First, the object's *metadata* (data fields that are sufficient for allocating an object's representation in memory) are serialized and transferred. In addition, the object's contiguous memory is registered with the communication library. The metadata and registration information combined are typically sufficiently small to utilize the eager protocol commonly found in MPI implementations. On the receiving process, the metadata is used to allocate a new object. In the second phase, the received registration information is used to fetch the data into the contiguous memory of the newly created object using remote memory access (RMA). The RMA capability to *TTG* is typically provided by underlying communication libraries such as LCI [28], UCX [107], or GASnet [15]. It can also be emulated using MPI point-to-point operations or use features proposed

[1] https://www.boost.org/doc/libs/1_77_0/libs/serialization/doc/index.html




```
template<>
struct SplitMetadataDescriptor<MatrixTile> {

  auto get_metadata(const MatrixTile& t) {
    return t.metadata();
  }

  auto get_data(MatrixTile& t) {
    return std::array<iovec, 1>({t.size(),
                                 t.data()});
  }

  auto create_from_metadata(metadata_t& meta) {
    return MatrixTile(meta);
  }
};
```

Figure 28 Schematic depiction of TTG's serialization format for objects containing contiguous data segments (left) and an example implementation for a MatrixTile (right).

for MPI RMA [92]. Once the transfer is complete, the sender is notified to release the source object.

Since the `splitmd` serialization fundamentally requires allocated-but-not-yet-initialized to be a valid state, the `splitmd` is intrusive (i.e., typically requires modification of the type definition and/or implementation). Type traits are used to test at compile time whether a given type supports the `splitmd` protocol. Serialization protocols chosen by *TTG* are selected in this order of preference: `splitmd` (if supported by the *TTG* backend; see subsection 5.2.6), trivial (`memcpy`), Boost.Serialization (if the Boost library is available), `madness` (if the *MADNESS* library is available).

5.2.6 *TTG* Execution Backends

As mentioned before, *TTG* as a programming model is a higher-level abstraction over the underlying low-level task runtime. The current C++ implementation of *TTG* can in principle create tasks using many available task runtimes, e.g., standard C++ (`std::async`) or OpenMP. In practice, however, for optimal resource utilization the implementation details of the task runtime matter greatly even in a shared-memory (host-only) setting. For distributed memory operation, additional features are needed to support seamless data transfers, parallel primitives (collective operations, global termination detection), and resource management; extra support is needed for heterogeneous execution and memory spaces within the node.

The implementation details of *TTG* are collectively referred to as a *TTG backend*. A backend provides the ability to schedule and execute tasks

as well as resource management and coordination for communication and computation in a distributed setting. There are currently two backends supporting the execution of *TTG* applications on shared- and distributed-memory platforms: *MADNESS* or *PaRSEC*. The *MADNESS* backend served as an early proof of concept for *TTG*, with the *PaRSEC* backend targeted to serve as the main vehicle for efficient performance-portable operation on distributed and heterogeneous platforms. The feature set required to implement *TTG* is not unique to these two backends and is available in other runtimes (e.g., UPC++), thus implementation of additional backends for *TTG* should be straightforward.

MADNESS **parallel runtime** started as the foundation for fast integrodifferential numerical calculus with guaranteed precision in up to 6 dimensions, with applications in chemistry and nuclear physics, among others [47]. By now, however, the *MADNESS* parallel runtime has evolved into a powerful general-purpose environment for task-based composition of a wide range of parallel algorithms on distributed data structures as varied as irregular trees in *MADNESS* and the sparse tensors in the *TiledArray* framework [21]. The central elements of the parallel runtime are a) futures for hiding latency and managing dependencies, b) global namespaces with one-sided access, c) remote method invocation in objects in global namespaces, and d) dynamic load balancing and data redistribution. An SPMD model is provided with a single logical main thread per process, a thread pool to execute tasks, and a thread dedicated to serving remote active messages. *MADNESS* can be configured to use its own thread pool implementation, or to use Intel TBB or *PaRSEC*. An application in the *MADNESS* runtime can be viewed as a dynamically constructed DAG, with futures as edges.

PaRSEC **[17]** is a task-based runtime for distributed heterogeneous architectures, capable of dynamically unfolding a concise description of a graph of tasks on a set of resources and satisfying all data dependencies by shepherding data between memory spaces (including between nodes) and scheduling tasks on heterogeneous resources. Compared to many runtime systems that support a single way to represent or discover a DAG of tasks, *PaRSEC* is designed to support many Domain Specific Languages (DSLs) or Application Programming Interfaces (APIs). This makes *PaRSEC* a tool of choice to study different APIs or DSLs for distributed task-based programming.

Multiple components constitute the *PaRSEC* runtime: programming interfaces (DSLs/APIs), schedulers, communication engines and data interfaces. The runtime uses a modular component architecture (MCA), allowing different modules or instances to be dynamically selected during runtime, providing a varied set of capabilities to different instances of the runtime (such as scheduling policies, or support for heterogeneity). A well-defined API for these modules transforms them into black boxes, and allows interested developers or users to implement their own, application specific, policies. The different DSLs share the same runtime, data representation, communication engine, scheduler, cohabiting over the same set of hybrid resources and seamlessly inter-operating in the context of the same application.

Several optimizations were introduced in this work specifically for the *PaRSEC* backend, including improvements to the *PaRSEC* runtime itself: a flexible new interface of the *PaRSEC* runtime system to efficiently organize communication between processes, the use of active messages for control signals, the use of a one-sided communication for asynchronous transfers of data, and the use of completion callbacks for notifications. The `splitmd` serialization protocol is also only available when using the *PaRSEC* backend. Most importantly, the *PaRSEC* backend now owns the data flowing through the *TTG* graph and is in charge of managing its life-cycle and marshaling it across memory space boundaries, such as for avoiding copying when data is passed to `ttg:send` or `ttg::broadcast` by `const` reference.

These additions target improving the efficiency and scalability of the *PaRSEC* backend, but have no impact on the correctness and capability of *TTG*, both current *TTG* backends support the full set of *TTG* features. In fact, all *TTG* programs developed in this work are backend independent, with the backend selection performed at compile time by setting a single preprocessor macro. Since the backend can sometimes have substantial impact on the performance, where warranted the performance will be demonstrated for both backends.

5.3 Benchmarks

A set of paradigmatic algorithms, with varying degree of irregularity in their data and computation traits, was implemented using C++ implementation of the *TTG* programming model. The performance was evaluated against

Table 3 Software configurations

Software	*Hawk*	*Seawulf*
MPI	Open MPI 4.1.1, UCX 1.10.0	Intel MPI 20.0.2
Compiler	GCC 10.2.0	GCC 10.2.0
HWLOC	1.11.9	1.11.12
MKL	19.1.0	20.0.2

reference implementations using traditional programming models or, where available, against existing state-of-the-art implementations.

5.3.1 Test Setup

We performed our evaluation on two systems. The *Hawk* system is a Hewlett Packard Enterprise Apollo[2] installed at the High Performance Computing Center Stuttgart (HLRS) in Stuttgart, Germany, consisting of 5,632 dual-socket 64-core AMD EPYC 7742 nodes equipped with 256 GB main memory and connected through a Mellanox Infiniband HDR 200 fabric. The *Seawulf* system is a Linux cluster installed at StonyBrook University[3] and consists of a variety of nodes equipped with Intel CPUs. In particular, we used up to 32 dual-socket Intel 20-core Xeon Gold 6148 CPUs with 192 GB main memory connected using a Mellanox InfiniBand FDR network. The used software configuration for both systems are listed in Table 3.

5.3.2 Dense Cholesky Factorization

We implemented the dense tiled Cholesky factorization (POTRF) [20] in *TTG* and compared its performance against state-of-the-art implementations *SLATE* [39], *Chameleon*[4] (running on top of *StarPU* [7]), *ScaLAPACK* [25], and *DPLASMA* [16] (running on top of *PaRSEC*). The templated task-graph of the tiled POTRF algorithm is depicted in Figure 25. To demonstrate the competitive performance *TTG* can deliver, we ran two separate scaling experiments: i) weak scaling across a number of nodes; and ii) problem scaling on a fixed number of nodes. In both cases, we used 60 worker threads pinned to a single NUMA domain per node to avoid interference and to work around issues with process binding observed with some of the reference im-

[2] https://www.hlrs.de/systems/hpe-apollo-hawk/
[3] https://it.stonybrook.edu/help/kb/understanding-seawulf
[4] https://project.inria.fr/chameleon/

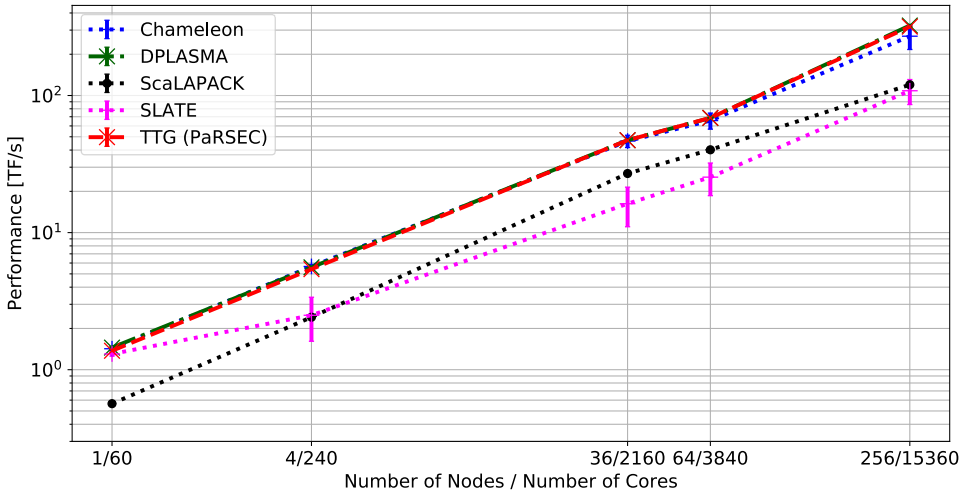


Figure 29 Weak-scaling of `POTRF` on *Hawk*. Each node holds a submatrix of size $30k^2$. The tile size is 512^2.

plementations, leaving 4 cores for the operating system and communication threads.

5.3.2.1 Node-scaling

Figure 29 shows a clear separation between two sets of scalability trends. *ScaLAPACK* and *SLATE* steadily continue to grow their performance but at a slower pace compared with the others, a behavior that can be explained by the sequentiality induced by the compute flow in the Cholesky algorithm without lookahead implemented in these two libraries. On the other side, all task-based versions benefit from the lack of synchronizations of the tile Cholesky implementation, and see a significant growth in performance with the increase in the number of compute resources in this weak-scale setup. Chameleon slightly trails behind the *TTG* and *DPLASMA* despite having the same potential parallelism due to the same tiled Cholesky implementation. A possible explanation is a more efficient communication substrate in *PaRSEC*, including the collective communication, but a more in-depth analysis would be necessary to confirm this.

5.3.2.2 Problem-scaling

Figure 30 shows a similar outcome, two well-separated groups, both asymptotically reaching their peak for this number of processes. Again, the task-based approaches benefit from the lack of synchronizations, and thus a large potential parallelism that once efficiently mapped into the compute re-

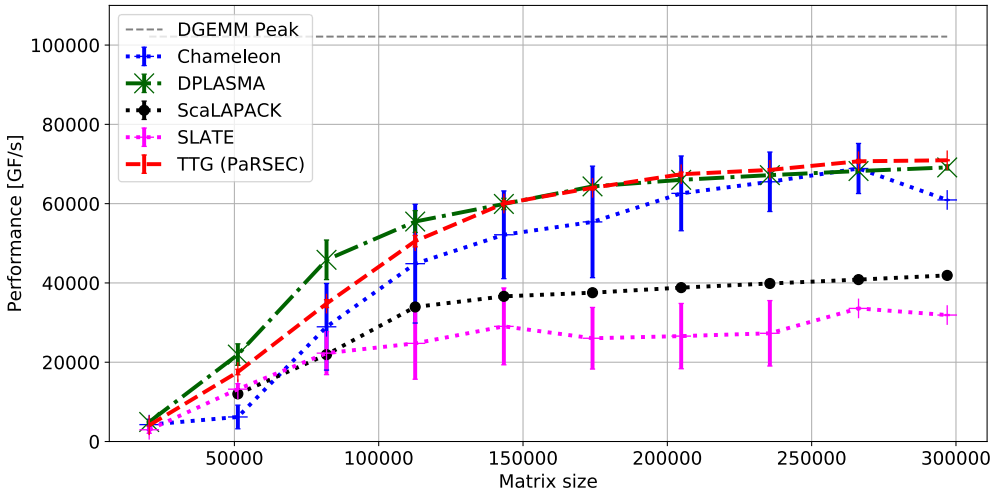


Figure 30 Scaling the matrix size on 64 nodes performing tiled Cholesky factorization with a tile size of 512^2 on *Hawk*.

sources lead to a more efficient execution and to reaching the practical peak for smaller matrix sizes.

5.3.3 Floyd-Warshall All-Pairs-Shortest Path (FW-APSP)

The FW-APSP algorithm finds the shortest path between every pair of vertices in a directed graph. It is among the most fundamental graph algorithms and has several applications in computer networks, logic programming, optimizing compilers, model-checking, social media, transportation, among others.

```
1 void fw_apsp(double **X,int N) {
2   for(k=0; k<N; ++k)
3     for(i=0; i<N; ++i)
4       for(j=0; j<N; ++j)
5         X[i][j]=min(X[i][j], X[i][k]+X[k][j]);
6 }
```

Prior work proposed different optimization techniques to improve the performance of the algorithm. Venkataraman et al. proposed a single-level tiled algorithm to improve the I/O complexity [115]. Javanmard et al. extended it to a recursive multi-level tiled algorithm to run efficiently on distributed-memory machines as well as GPUs [55, 53]. In the recursive multi-level tiled algorithm, the first level of tiling is used to distribute the underlying adjacency matrix among processes and further parallelism and I/O efficiency were achieved by recursive sub-tiling. Nookala et al. [82] implemented a data-flow version of the standard two-way recursive divide-and-conquer FW-APSP algorithm in Intel CnC [18] and compared the performance with a fork-join implementation in OpenMP. They showed that a data-flow im-

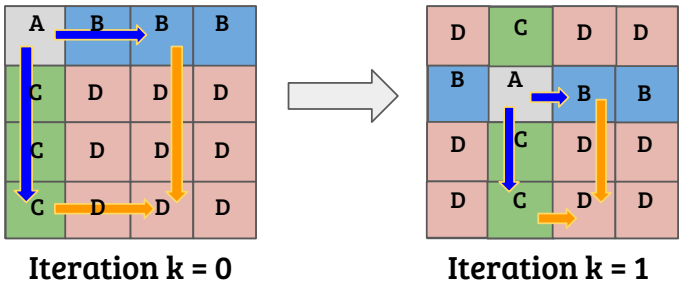


Figure 31 Flow of data among different kernels in blocked FW-APSP algorithm.

plementation outperforms its fork-join counter-part when, due to artificial dependencies, the fork-join implementation fails to generate enough sub-tasks to keep all processors busy and does not have enough data locality to compensate for the lost performance.

As shown in Figure 31, the parametric recursive algorithm has four kernels (A, B, C, and D) that each compute the minimum shortest path within the input tiles of the adjacency matrix. Kernel A is only applied to the tiles on the diagonal, followed by kernels B and C applied to the respective row and column. The results of kernels B and C are used as input for kernel D, which is applied to the panels on both sides of the current row and column. In the multi-level MPI+OpenMP implementation, the exchange of super-tiles along rows and columns is performed using MPI broadcast operations while the application of the operations to the sub-tiles is done using OpenMP tasks. In *TTG*, on the other hand, a single-level 2D block-cyclic distribution of tiles is used and tiles are broadcast to all successor operations independent of other tiles. The MPI+OpenMP implementation of [55] puts significant constraints on the available process configurations by requiring process numbers that are both square and multiples of 2. This constraint was later discussed in [54, 53] and virtual padding is mentioned as a potential solution to this constraint but the distributed-memory implementation was not discussed. While the *TTG* implementation of the benchmark does not have these constraints, in the interest of comparability we decided to run the same configuration for both MPI+OpenMP and *TTG*.

Figure 32 depicts the strong-scaling behavior of both the *TTG* and MPI+OpenMP implementation on a 32k matrix with different block sizes. The data shows that the *TTG* implementation clearly outperforms the MPI+OpenMP implementation up to 16 nodes by a factor of almost 2, with *TTG* running on top of *PaRSEC* further scaling to 64 nodes for block sizes of 64 and 128. *TTG*

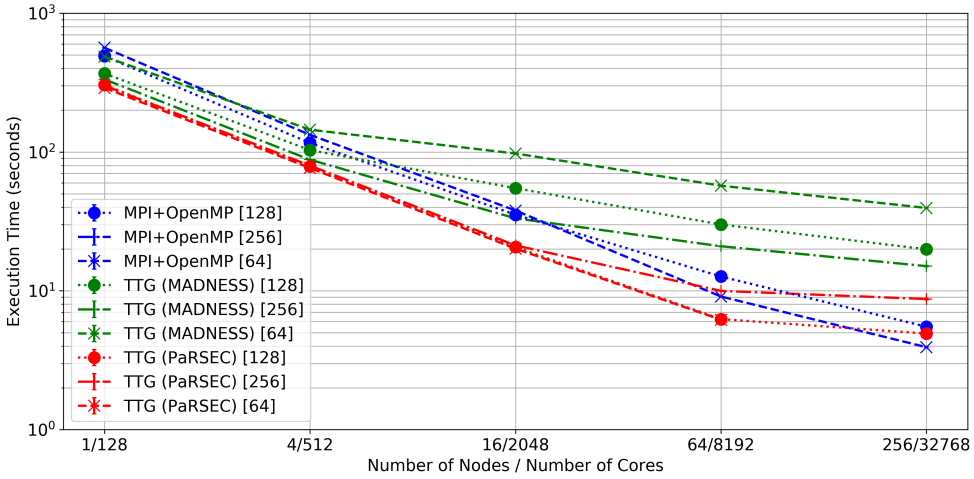


Figure 32 Strong scaling of the Floyd-Warshall benchmark using *TTG* and MPI+OpenMP on Hawk using 16 processes per node, 8 threads each (block sizes in square brackets).

running on top of *MADNESS* benefits from larger tile sizes, presumably due to the lower number of tiles to communicate, but is limited in its scalability.

For *TTG* running on top of *PaRSEC*, smaller block sizes lead to better scalability. At 256 nodes, however, *TTG* using blocks of size 128 reaches its scalability limit: $(\frac{32k}{128}) = 256$ blocks in each dimension distributed across $\sqrt{256 \times 16} = 64$ processes per dimension results in $\frac{256}{64} = 4$ blocks per process, less than the number of threads. Unfortunately, an issue in Open MPI prevented us from running with block sizes of 64 with *TTG* on top of PaRSEC on 256 nodes. However, we expect *TTG* to further scale to 256 nodes once this issue is resolved.

Figure 33 shows the strong-scaling behavior on SeaWulf using a 32K matrix with block sizes 128 and 256. *TTG* implementations outperform the MPI+OpenMP implementation on up to 32 nodes by a factor of 4. *TTG* with *MADNESS* performs similar to the *PaRSEC* version with 256 tile size as compared to 128 tile size due to less communication with larger tiles. The running time for benchmarks with 64 tile size exceeded the time-limit and hence are not included in the plot.

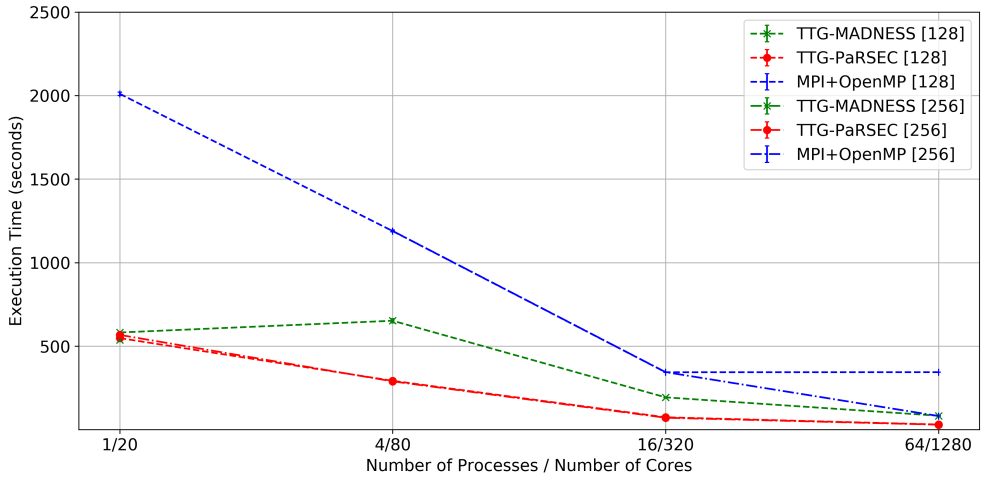


Figure 33 Strong scaling of the Floyd-Warshall benchmark using *TTG* and MPI+OpenMP on SeaWulf using 2 processes per node, 20 threads each (block sizes in square brackets).

5.3.4 Block-Sparse GEMM

As a first irregular application, we considered a block-sparse matrix-matrix multiplication (`bspmm`). The arguments are matrices tiled in blocks of irregular dimensions, with a significant subset of blocks empty. The bspmm algorithm we implemented follows a 2D SUMMA strategy [114], adapted to the task-based representation, similarly to [49]. The *TTG* for bspmm introduces control tasks and edges to implement three control-flow feedback loops in the application: two to limit the amount of parallel broadcasts of tiles of the input matrices, and introduce some ordering of communications, and a third to constrain the task scheduler and improve data re-use by forcing the work to focus on a subset of GEMM tasks that work on the same subset of data.

The resulting TTG is depicted in Figure 34. Tasks of type `ReadSpA/B` load the tiles from memory and inject them into the flowgraph. The tiles are broadcast to remote nodes via the tasks of type `BcastA/B`, and stored on each node in the tasks of type `LStoreA/B`, to avoid additional communications. There is a control-flow feedback loop from `LStoreA/B` to the `ReadSpA/B` to control how many of these communications can happen in parallel. Then, tiles flow to the main computational kernel in tasks of type `MultiplyAdd` through local broadcast tasks of type `LBcastA/B`. There is another feedback control loop through tasks of type `Coordinator` that wait until multiple `MultiplyAdd` tasks are completed before it allows tasks of type `LBcastA/B` to continue broadcasting local work. This reduces the choices of the scheduler and

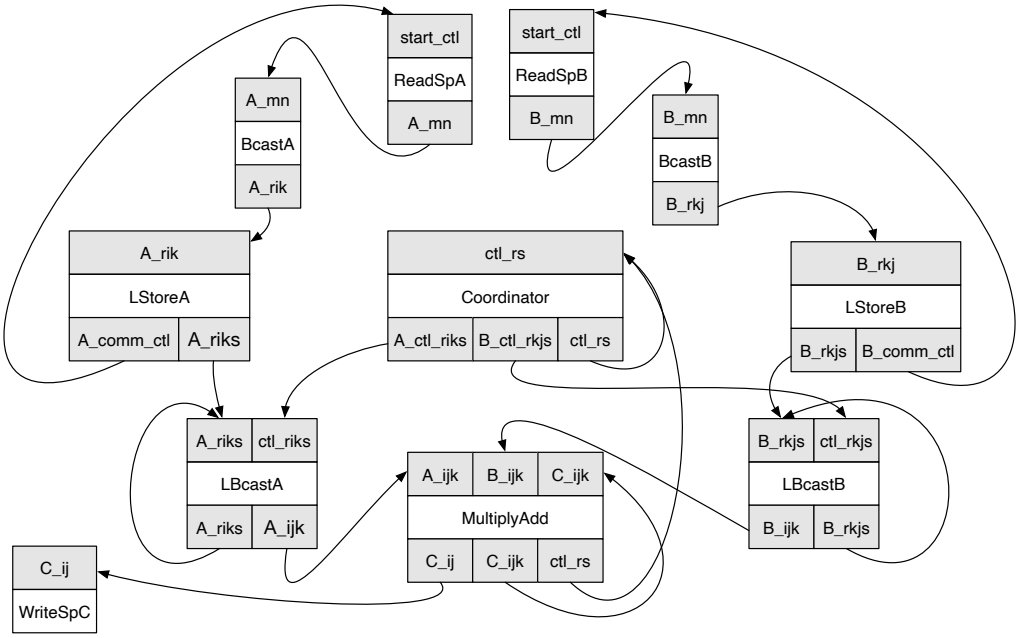


Figure 34 Template task-graph of the block-sparse matrix-matrix multiply algorithm.

forces it to focus on a subset of GEMM tasks that work on the same subset of data. Both feedback loops are implemented using streaming terminals discussed in subsection 5.2.4.

Compared to the previous applications, BSPMM is irregular and requires dynamic decisions: the DAG of tasks that must be executed depends on each input problem, and there is no universal data placement and scheduling strategy that can guarantee optimal performance without adapting these decisions to each input problem. The task-based approach of *TTG* delegates the dynamic scheduling decision to the underlying runtime system, creating some adaptability. Data placement remains heuristical, based on a 2D block cyclic distribution to balance the load, and the additional control flow is here to manage the high degree of parallelism of the problem.

To evaluate the performance of the `bspmm` implementation, we used the matrix representation of the Yukawa integral operator ($\exp(-r_{12}/5)/r_{12}$) in the cc-pVDZ-RIFIT Gaussian atomic orbital basis for the main protease of the SARS-CoV-2 virus in complex with the N3 inhibitor [58] (total of 2,500 atoms; this size is representative of target problem sizes in biomedical applications). The size of the matrix is 140,440; rows/column panels corresponding to each of the 2,500 atoms are grouped into tiles such that the size of each tile does not exceed the target tile size of 256. Tiles of the matrix with the

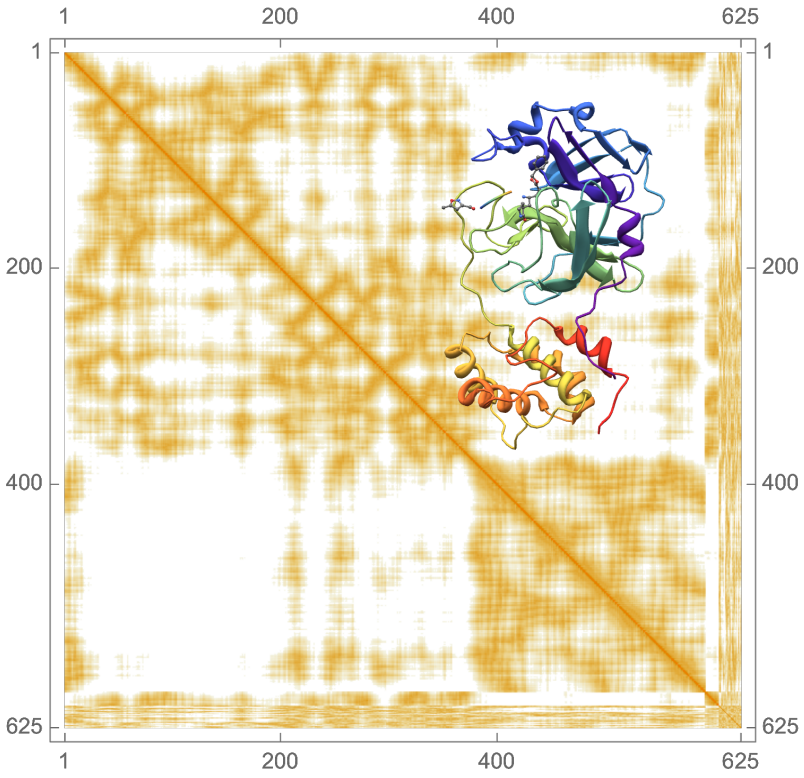


Figure 35 Nonzero blocks of the block-sparse Yukawa potential matrix used for the `bspmm` benchmark (see text for details).

per-element Frobenius norm of less than 10^{-8} are discarded. We compute the square of the resulting block-sparse matrix A (Figure 35) using the `bspmm` implementation.

We compare the *TTG* implementation with the Distributed Block Compressed Sparse Row library (DBCSR [67]). DBCSR is part of the CP2K quantum chemistry and solid state physics program package; it implements a 2.5D communication-reducing SUMMA algorithm [99] and focuses on block-sparse matrix-matrix multiplication of matrices with a relatively large occupation.

Figure 36 shows the performance obtained for an increasing number of nodes for the Yukawa potential matrix multiplication. From 8 to 128 nodes, DBCSR and both *TTG* backends all exhibit very similar performance, with a linear strong scaling. The *TTG* implementation with both backends stop scaling at this size and for this matrix, while the DBCSR one continues. The *TTG* implementation over the *PaRSEC* backend shows a high variability at 128 nodes, with some runs significantly slower than others, and a peak at the same speed as the *TTG* implementation over the *MADNESS* backend

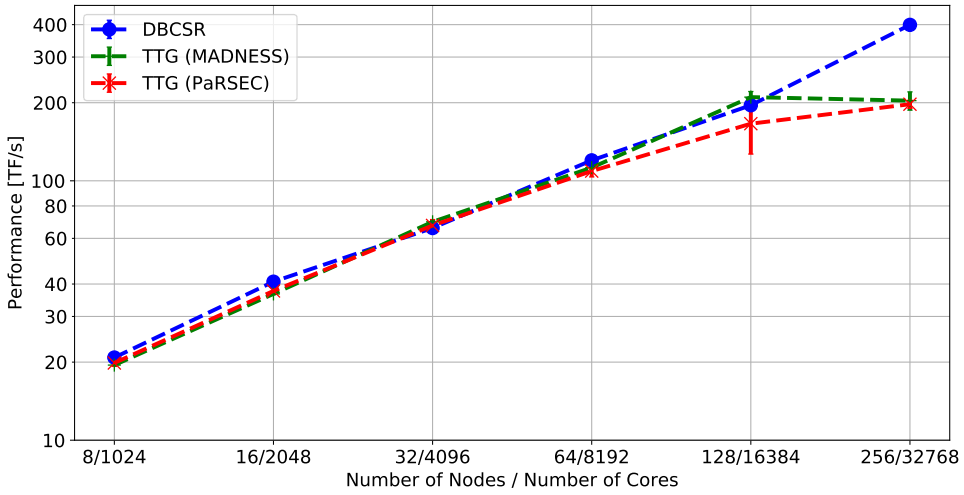


Figure 36 Strong scaling of block-sparse GEMM.

and the DBCSR one. We are investigating this instability, that we observe only for the specific communication pattern for 128 nodes.

At 256 nodes, each process holds only a few tiles of the product matrix, and communications become the dominant factor of the execution. The 2.5D SUMMA algorithm [99] implemented in DBCSR continues to scale due to its ability to leverage greater cross-section bandwidth compared to the 2D SUMMA variant that was implemented in *TTG*. We expect that by converting the current 2D SUMMA *TTG* implementation to 2.5D SUMMA we will be able to at least match the strong-scaling performance of DBCSR.

5.3.5 Multi-Resolution Analysis (MRA)

This benchmark computes adaptively the order-10 multiwavelet [4, 3] representation of 3-D Gaussian functions (exponent $30,000$) to precision of 10^{-8} with Gaussian centers distributed randomly in a $[-6,6]^3$ volume. This random distribution leads to substantial clustering and hence load imbalance that is only partially addressed by overdecomposition using a *task ID map* that randomly distributes function tree nodes (and their children) across processes at some target level of refinement. Empirically, the load imbalance is offset by the reduction of communication.

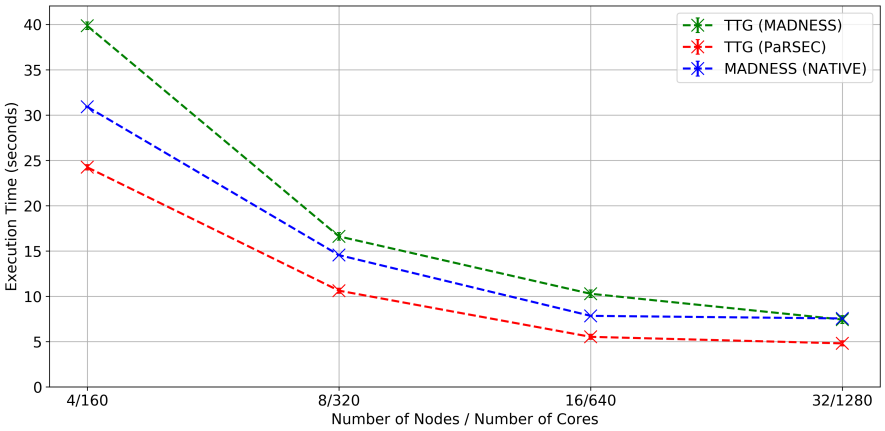
The MRA computation on each function commences by adaptively projecting into the multiwavelet basis by recurring down until the local representation error is below the truncation threshold. The resulting data structure is a 3D spatial tree that extends down about 6 levels of adaptive dyadic refinement. Subsequently, the fast wavelet transform (compression) and

inverse transform (reconstruction) are performed and the norm of the function is also computed for verification purposes. Work and data flow down the tree in the projection and reconstruction steps, and flows up the tree for compression. In the compression operation, a parent node needs coefficients from its $2^3 = 8$ children. The code is templated by the number of dimensions, making this a perfect use case of streaming terminals so that a single terminal can process children in arbitrary dimensions. Prior to streaming terminals, the example had to employ complex C++ templates to manage a variable and potentially large number of terminals. The native *MADNESS* implementation computes on each tree in parallel, but there is an explicit barrier after each computational step (projection, compression, reconstruction, norm) as the in-memory data structure is completed. In contrast, the *TTG* implementation eliminates all inessential barriers and streams data through the entire DAG and never stores an explicit representation of all trees. The transition between algorithms that ascend and descend implies that there is a moment for each tree for which all data is stored (as arguments of pending tasks), but computation on other trees proceeds independently in the *TTG* implementation.

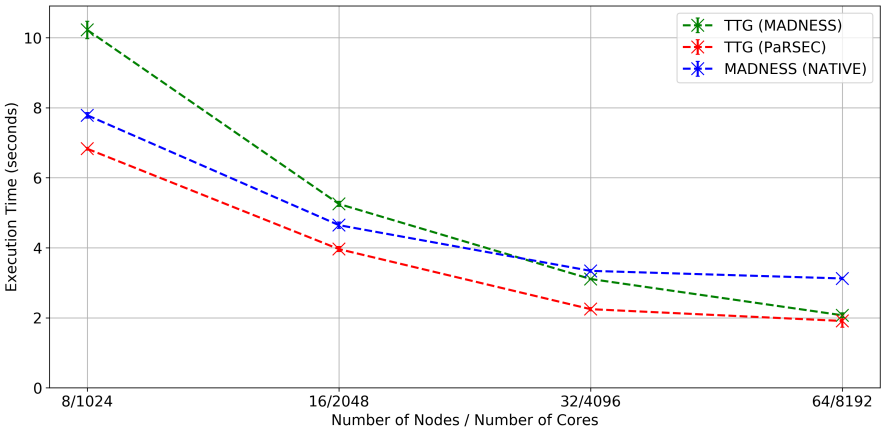
```
reduce_leaves_tt->template set_input_reducer<0>(
  /* the reduction operator */
  [](FunctionReconstructedNode<T,K,NDIM> &&a,
     FunctionReconstructedNode<T,K,NDIM> &&b)
  {
    a.neighbor_coeffs[a.key.childindex()] = a.coeffs;
    a.is_neighbor_leaf[a.key.childindex()] = a.is_leaf;
    a.neighbor_sum[a.key.childindex()] = a.sum;
    a.neighbor_coeffs[b.key.childindex()] = b.coeffs;
    a.is_neighbor_leaf[b.key.childindex()] = b.is_leaf;
    a.neighbor_sum[b.key.childindex()] = b.sum;
    return a;
  },
  1 << NDIM /* the number of reductions to perform */
);
```

The streaming terminal feature is essential for expressing the MRA numerical calculus algorithms, such as the compress operation, in a manner independent of the number of dimensions d. Since the number of inputs to a compress task is 2^d, changing d would require changing the flowgraph. 5.3.5 shows how streaming terminal can be used to implement accumulation of the input node data sent to the compress task. Each compress task expects exactly 2^d inputs, hence the size of the stream expected by the input terminal can be passed directly to the `set_input_reducer` method.

Figures 37a and 37b show the results of strong-scaling MRA using *TTG* and native *MADNESS* on *Seawulf* up to 32 nodes and on *Hawk* up to 64 nodes.



(a) Strong scaling MRA: 4 to 32 nodes with 120 functions on *Seawulf*, using 2 processes per node with 20 threads each.



(b) Strong scaling MRA: 8 to 64 nodes with 400 functions on *Hawk*, using 8 processes per node with 16 threads each.

Figure 37 Strong scaling MRA on *Seawulf* and *Hawk*.

TTG over *PaRSEC* clearly outperforms *TTG* over *MADNESS* and native *MADNESS* on both machines. The benchmark uses plain-old-data (POD) structures for node data and the performance of *TTG* over *MADNESS* suffers due to data copies and high communication overhead as compared to the efficient communication in *TTG* over *PaRSEC* which avoids unnecessary copying of data. The native *MADNESS* implementation scales up to 32 nodes on both machines. However, it reaches the scalability limit due to the existence of barriers at every step of the computation and re-allocation of data. We are investigating methods for reducing the communication overheads in *TTG* over *MADNESS*.

5.4 Conclusion

Template Task Graph is a new flowgraph programming model that aims to lower the complexity of performance-portable parallel programming of (especially, irregular) complex applications by abstracting many details of the underlying task scheduling and execution as well as associated data and resource management. In this paper, we presented the current status of TTG's C++ distributed-memory implementation using two task-based runtime systems (*MADNESS* and *PaRSEC*). We evaluated these implementations over four paradigmatic applications, ranging from the most regular and compute intensive to applications whose execution is data dependent and memory bound. These evaluations show high performance and scalability, on par and sometimes exceeding the performance of state of the art implementations in other programming paradigms. We presented in detail how the features of the language are exploited by the implementations to reduce memory copies and increase data management and communication. It must also be noted that the development cost, while a subjective measure, was certainly significantly lower compared with the state-of-the-art applications, and done by outsiders of the representative domain.

CHAPTER 6

Related Work

In this chapter, we talk about existing work in the areas of many task computing, concurrent data structures and parallel runtime systems and how they differ from our work.

6.1 Many Task Computing

In the last years, the use of scheduler based on many-core or heterogeneous architectures for general or for specific applications has been widely studied [121, 78]. S. Yamagiwa et al. [121] propose a GPGPU streaming based on distributed computing environment; S. Nakagawa et al. [78] provide a new middleware capable of out-of-order execution of works and data transfers using stream processing. Other works [41, 118] follow a similar strategy based on streaming to minimize data transfers overhead. S. Kato et al. [61] introduce *TimeGraph*, a GPU scheduler composed by two different GPU scheduling policies which allow to interrupt the low priority tasks execution in order to execute higher priority tasks within a real-time multi-tasking environments for video applications. Similar to the previously mentioned works and considering that the GPUs in a cluster are not usually fully utilized, Duato et al. [33] present their *rCUDA*, a middleware that enables CUDA remoting over a commodity network by allowing to use CUDA-compatible GPUs installed in a remote computer, as, they were installed in the computer where the application is being executed. Also, V. J. Jiménez et

al. [57] present a sort of predictive runtime scheduling which supports several scheduling algorithms in order to choose the appropriate platform (Multicore, GPU, ...) in which the algorithm would be better executed, resulting in almost fully usage of CPU/GPU-like systems, with a peak time reduction of 40% with respect to only using the GPU. Basically most the aforementioned works take advantage of overlapping memory transfers among CPU and GPU memories with single kernel executions.

With the aim of exploiting MTC on many-core, other authors [68, 63] have studied the efficiency of this new feature. *Merged task*, maybe the first MTC approach on GPUs, allows us to run several independent kernels over the same GPU simultaneously. It was presented by M. Guevara et al. [44] and P. Valero-Lara et al. [111]. Posteriorly, C. Gregg et al. [42] and K. Zhang et al. [122] included a scheduler which can select the best matching among tasks before running. Additionally, P. Valero-Lara et al. [110] applied this strategy to different GPU architectures to obtain the most convenient architectural features for running concurrent kernels. After that, in [113], it is proposed a new heterogeneous (CPU-GPU) scheduler in which groups of independent blocks of tasks were efficiently managed to fully use CPU-GPU and reduce the overhead of memory transfers. More recently, S. Krieder et al. [64] presented *GeMTC*, a CUDA based framework which allows MTC workloads to run efficiently on NVIDIA's GPUs. P. Nookala et al. [81] adapted this framework (*GeMTC*) to efficiently use the particular features of Intel Xeon Phi and evaluate MTC applications on Intel accelerators. The above mentioned works relate to our early work using Intel Xeon Phis which motivated us to explore parallel runtime systems for fine-grained tasking in general.

6.2 Concurrent queues

Several researchers have proposed concurrent queue implementations. Scogland et al. [94] presented the characterization of various concurrent queues on many-core architectures and proposed a high-throughput queue specifically engineered for many-core architectures. Schweizer et al. [93] performed detailed analysis of x86 atomic instructions on various architectures and discovered that atomics prevent instruction level parallelism and that latency depends on architectural properties such as the coherence state of the accessed cache lines. Scott et al. [73] proposed a lock-free queue algorithm for machines that provide atomic primitives. Cache-friendly concurrent

lock-free queue (CFCLF) [72] is a lock-free queue that employs a matrix for the queue structure, reducing core-to-core communication overhead and making it cache efficient. BQ [74] is a lock-free queue that exploits batching to gain better performance. Morrison et al. [77] proposed a concurrent non-blocking linearizable FIFO queue using atomic FAA that outperforms CAS based implementations by up to 2×.

6.3 Parallel runtime systems

Most parallel runtime systems and execution models, such as OpenMP [14], Charm++ [60], and Swift/T [120], use concurrent queues for sharing data between threads or processes. OpenMP's task construct [8] enables task-based parallelism. Charm++ demonstrates about 10-20% improvement in performance by using optimization techniques like lock-free queues, CPU affinity, and memory management [70]. Recently, Cpp-taskflow [104] emerged as an alternative to OpenMP task parallelism for C++.

Numerous efforts to provide a similar level of abstraction via a fine-grain task-based dataflow programming exist, adding to those that have transitioned from a grid-based workflow toward a task-based environment. Some of the recent task-based runtimes like Legion [11], *StarPU* [7], HPX [48], CnC [18], *OmpSs* [35], DASH [91], *PaRSEC* [17] and *MADNESS* [47] act as an intermediary between the hardware resources and a programming paradigm, language or API to isolate application developers from the underlying hardware. Some of these programming interfaces have nascent support for distributed execution, e.g., recent versions of the OpenMP specification [84] introduce the *task* and *depend* clauses which can be employed to express control flow graphs. OpenMP is widely used and supports homogeneous, shared memory systems, and its *target* extension to support accelerators is quickly gaining traction. A limitation of the OpenMP model is that distributed memory and inter-node communication need to be explicitly implemented with the use of an external communication library.

In *OmpSs*, tasks are discovered by a single thread and executed by worker threads. The model allows nesting of tasks in individual nodes to relieve the main thread; however it may suffer from scalability issues on large scale distributed systems.

HPX aims to overcome these challenges by replacing explicit communications and synchronizations with asynchronous communication between

nodes and lightweight control objects, allowing applications to exploit fine-grained parallelism within the context of a global address space.

Legion, on the other hand, describes logical regions of data and uses those regions to express the dataflow and dependencies between tasks, and defers to its underlying runtime, REALM [103], the scheduling of tasks, and data movement across distributed heterogeneous nodes.

To the best of our knowledge, we are the first to explore lock-less strategies in concurrent programming where data can be carefully manipulated to avoid the use of locks. Furthermore, existing runtime systems have not focused on the efficient support of fine-grained tasks, resulting in sub-optimal application execution, a problem that will only get worse with larger many-core architectures.

6.4 Load Balancing

Several researchers have proposed various load balancing mechanisms [31, 45]. Blumofe and Leiserson et al. introduced work stealing and proved that it is superior to work sharing [13]. Quintin et al. proposed hierarchical work stealing for exploiting data locality to achieve speed up compared to classical work stealing algorithms [87]. Various parameters of work stealing have been explored in the literature and Michael et al. showed that two random choices for work stealing exponentially improves performance and is sufficient to achieve good load balancing [76]. Several applications implement their own load balancing mechanisms in order to achieve ideal performance on various architectures. Unbalanced Tree Search benchmark [83] implements a work stealing mechanism for efficient dynamic load balancing and by varying key work stealing parameters, the authors expose important tradeoffs between the granularity of load balance, the degree of parallelism, and communication costs. Recently Shiina et al. introduced “Almost Deterministic Work Stealing" which addresses the issue of data locality by making scheduling almost deterministic [97]. All mechanisms proposed in the literature for multi-threaded runtimes rely on concurrent data structures and synchronization mechanisms for achieving dynamic load balancing. In contrast, our work explores lock-less techniques for achieving comparable dynamic load balancing by using non-atomic memory updates.

6.5 Flowgraph Programming

Flowgraphs, while ubiquitous as general models of computation (e.g., in compilers), have recently become featured as first-class concepts in programming models and languages aimed at high performance. Control-flow graph models include Taskflow [51], CUDA graphs [26]; TensorFlow [1] and Dask [90] APIs support dataflow graphs; Intel TBB [65] includes support for both control flow and dataflow graphs; CnC [18] and Legion [11] can support control or dataflow graphs through data partitioning and mapping. The most direct influence on *TTG* was Parametrized Task Graph, a programming model supported by *PaRSEC* in which computation is represented as flows of tuple-indexed data through an operation graph. Almost all of these programming models are implemented as C++ libraries. Most implementations limit the support for flowgraphs to shared memory setups, or use explicit communications transformed in tasks to simulate the flowgraph in a distributed setting. The Hume flowgraph DSL focuses on real-time embedded systems [46]. The S-NET DSL [43] is an orchestration language of tasks, strictly decoupling implementation and parallelism.

CHAPTER 7

Conclusion and Future Work

We believe that X-OpenMP creates the opportunity to transparently accelerate many applications with fine-grained parallelism. Our work in this area of task-based parallel runtime systems creates new avenues for exploring lock-less techniques in the HPC space. We plan to evaluate real-world scientific applications in Computational Biology, Materials Science, Computational Chemistry, and Astrophysics using X-OpenMP to demonstrate the performance improvements achievable in parallel runtimes as a step towards exascale goals. We also plan to explore applications that can be over-decomposed into many finer-grained tasks by rethinking the algorithms to achieve improved performance using the techniques presented in this paper. We also would like to investigate the applicability of lock-less programming techniques in GNU OpenMP [101], the Swift/T workflow system [120], as well as the Parsl parallel programming library [9] in order to further broaden the applications that could take advantage of the proposed techniques.

With respect to *TTG*, future work will consider extensions to simplify data injection in the DAG of tasks, to better manage memory and network utilization, to provide some degree of Quality-of-Service with regard to the computation and communication scheduling, and to support heterogeneous platforms. We are also interested in adding more runtime backends to *TTG* in future with X-OpenMP being one of the choices. By enabling efficient support of fine-grained parallelism across the growing range of scales seen

in modern and future hardware, we believe this work will enhance the productivity of parallel programmers.

Timeline

- Revise X-OpenMP paper based on reviews from PACT'22 - June/July 2022.
- Journal Extension to XQueue for submission to TPDS or TOC - Soon.
- Extend XQueue work by integrating lockless queues into GNU OpenMP for submission to IPDPS'23 or TPDS/TOC - November 2022
- Explore the opportunities for integrating XQueue into Parsl - July/August 2022.
- Release first version of TTG to public after enabling GPUs for applications using PaRSEC - Soon.
- Plan to graduate - December 2022.

References

[1] Martın Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] Umut A. Acar et al. *Atomic Read-Modify-Write Operations are Unnecessary for Shared-Memory Work Stealing*. Research Report. Sept. 2013.

[3] B. Alpert. “Sparse Representation of Smooth Linear Operators”. PhD thesis. Yale University, 1990.

[4] B. Alpert et al. “Adaptive Solution of Partial Differential Equations in Multiwavelet Bases”. In: *Journal of Computational Physics* 182.1 (2002), pp. 149–190. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.2002.7160. URL: https://www.sciencedirect.com/science/article/pii/S0021999102971603.

[5] S. Arnautov et al. “FFQ: A Fast Single-Producer/Multiple-Consumer Concurrent FIFO Queue”. In: IEEE, 2017.

[6] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. “Thread scheduling for multiprogrammed multiprocessors”. In: *Theory of computing systems* 34.2 (2001), pp. 115–144.

[7] C. Augonnet et al. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *Conc. Comp. Pract. Exper.* 23 (2011), pp. 187–198.

[8] Eduard Ayguadé et al. “The Design of OpenMP Tasks”. In: *TPDS’09*. Vol. 20. 23. IEEE, 2009.

[9] Y. Babuji et al. “Parsl: Pervasive Parallel Programming in Python”. In: *HPDC’19*. New York, NY, USA: ACM, June 2019.

[10] Samy Al Bahra. *Concurrency Kit*. 2011. URL: http://concurrencykit.org/.

[11] Michael Bauer et al. “Legion: Expressing locality and independence with logical regions”. In: *Supercomputing*. 2012. ISBN: 9781467308069. DOI: 10.1109/SC.2012.71.

[12] Scott Beamer, Krste Asanović, and David Patterson. “Direction-optimizing breadth-first search”. In: *SC’12*. Nov. 2012, pp. 1–10.

[13] Robert D Blumofe and Charles E Leiserson. “Scheduling multithreaded computations by work stealing”. In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[14] OpenMP Architecture Review Board. *OpenMP®: Support for the OpenMP language*. URL: https://openmp.llvm.org/.

[15] Dan Bonachea and Paul H. Hargrove. “GASNet-EX: A High-Performance, Portable Communication Library for Exascale”. In: (Oct. 2018). DOI: 10.25344/S4QP4W.

[16] George Bosilca et al. “Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA”. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (2011). DOI: 10.1109/ipdps.2011.299.

[17] George Bosilca et al. “PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability”. In: *Comp in Sc. and Eng.* 99 (2013), p. 1. DOI: 10.1109/MCSE.2013.98.

[18] Zoran Budimlić and Kathleen Knobe. “CnC: A Dependence Programming Model”. In: *Proceedings of the Sixth Workshop on Data-Flow Execution Models for Extreme Scale Computing*. DFM’16. Haifa, Israel: ACM, 2016. ISBN: 978-1-4503-6199-6.

[19] J Mark Bull, Fiona Reid, and Nicola McDonnell. “A microbenchmark suite for OpenMP tasks”. In: *International Workshop on OpenMP*. Springer. 2012, pp. 271–274.

[20] Alfredo Buttari et al. “A class of parallel tiled linear algebra algorithms for multicore architectures”. In: *Parallel Computing* 35.1 (2009), pp. 38–53. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2008.10.002.

[21] J Calvin and EF Valeev. *TiledArray: A massively-parallel, block-sparse tensor framework written in C++*. https://github.com/ValeevGroup/tiledarray. 2018.

[22] Justus A Calvin, Cannada A Lewis, and Edward F Valeev. “Scalable task-based algorithm for multiplication of block-rank-sparse matrices”. In: *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. 2015, pp. 1–8.

[23] William Carlson et al. *Introduction to UPC and Language Specification*. Tech. rep. CCS-TR-99-157. IDA Center for Computing Sciences, May 1999.

[24] Milind Chabbi et al. “An efficient abortable-locking protocol for multi-level NUMA systems”. In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2017, pp. 61–74.

[25] J. Choi et al. “ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers”. In: *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, Oct. 1992. DOI: 10.1109/FMPC.1992.234898.

[26] *CUDA Programming Guide - CUDA Graphs*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs. 2021.

[27] Anthony Danalis et al. “PTG: An abstraction for unhindered parallelism”. In: *Proceedings of WOLFHPC’14* (2014), pp. 21–30. DOI: 10.1109/WOLFHPC.2014.8.

[28] H. Dang et al. “A Lightweight Communication Runtime for Distributed Graph Analytics”. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018. DOI: 10.1109/IPDPS.2018.00107.

[29] S. R. Das and R. M. Fujimoto. “A Performance Study of the Cancelback Protocol for Time Warp”. In: *SIGSIM Simul.* Vol. 23. 1. 1993, pp. 135–142.

[30] E. W. Dijkstra. “Solution of a problem in concurrent programming control”. In: *CACM 1965*. Vol. 8. 1965, p. 569.

[31] J. Dinan et al. “Scalable work stealing.” In: *SC’09*. 2009.

[32] Jack Dongarra et al. “PLASMA: Parallel linear algebra software for multicore using OpenMP”. In: *ACM Transactions on Mathematical Software (TOMS)* 45.2 (2019), pp. 1–35.

[33] J. Duato et al. “Performance of CUDA virtualized remote GPUs in high performance cluster”. In: *the 40st International Conference on Parallel Processing (ICPP)* (2011), pp. 365–374.

[34] Iain S Duff, Michael A Heroux, and Roldan Pozo. “An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum”. In: *ACM Transactions on Mathematical Software (TOMS)* 28.2 (2002), pp. 239–267.

[35] A. Duran et al. “A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks”. In: *Intl. Journal of Parallel Programming* 37.3 (2009), pp. 292–305. DOI: 10.1007/s10766-009-0101-1.

[36] Alejandro Duran et al. “Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP.” In: *ICPP’09*. 2009, pp. 124–131.

[37] Hadi Esmaeilzadeh et al. “Dark Silicon and the End of Multicore Scaling”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA ’11. San Jose, California, USA: ACM, 2011, pp. 365–376. ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000108. URL: http://doi.acm.org/10.1145/2000064.2000108.

[38] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. “The Implementation of the Cilk-5 Multithreaded Language”. In: *PLDI'98*. 1998, pp. 212–223. DOI: 10.1145/277650.277725.

[39] Mark Gates et al. “SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library”. In: *Supercomputing*. SC '19. Association for Computing Machinery, 2019. DOI: 10.1145/3295500.3356223.

[40] David Geer. “Industry Trends: Chip Makers Turn to Multicore Processors”. In: *Computer* 38.5 (2005), pp. 11–13. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/MC.2005.160.

[41] J. Gómez-Luna et al. “Performance models for CUDA streams on NVIDIA GeForce series”. In: *J. Parallel Distrib. Comput.* 72.9 (2012), pp. 1117–1126.

[42] C. Gregg et al. “Fine-Grained Resource Sharing for Concurrent GPGPU Kernels”. In: *In Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar)* (2012).

[43] Clemens Grelck, Jukka Julku, and Frank Penczek. “Distributed S-Net: Cluster and Grid Computing without the Hassle”. In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*. 2012. DOI: 10.1109/CCGrid.2012.140.

[44] M. A. Guevera et al. “Enabling Task Parallelism in the CUDA Scheduler”. In: *In Proceedings of the Workshop on Programming Models for Emerging Architectures (PMEA), in conjunction with the ACM/IEEE/IFIP International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2009).

[45] Yi Guo et al. “Work-first and help-first scheduling policies for terminally strict parallel programs”. In: *Proc. of the 23rd IEEE International Parallel and Distributed Processing Symposium*. Vol. 10. 2009.

[46] Kevin Hammond and Greg Michaelson. “Hume: A Domain-Specific Language for Real-Time Embedded Systems”. In: *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*. GPCE '03. Erfurt, Germany: Springer-Verlag, 2003, pp. 37–56. ISBN: 3540201025.

[47] Robert J. Harrison et al. “MADNESS: A Multiresolution, Adaptive Numerical Environment for Scientific Simulation”. In: *SIAM J. Sci. Comput.* 38.5 (2016), S123–S142.

[48] T. Heller, H. Kaiser, and K. Iglberger. “Application of the ParalleX execution model to stencil-based problems”. In: *Computer Science - Research and Development* 28.2-3 (2013), pp. 253–261. ISSN: 18652034. DOI: 10.1007/s00450-012-0217-1.

[49] Thomas Herault et al. “Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure”. In: *35th IEEE International Parallel and Distributed Processing Symposium IPDPS*. IEEE, 2021. DOI: 10.1109/IPDPS49936.2021.00062.

[50] Michael A Heroux et al. *Advancing Scientific Productivity through Better Scientific Software: Developer Productivity and Software Sustainability Report*. Tech. rep. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2020.

[51] Tsung-Wei Huang et al. "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System". In: *IEEE TPDS* (2021), pp. 1–1. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2021.3104255. URL: https://ieeexplore.ieee.org/document/9511796/ (visited on 10/15/2021).

[52] Steven Huss-Lederman et al. "Implementation of Strassen's algorithm for matrix multiplication". In: *Supercomputing'96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. IEEE. 1996, pp. 32–32.

[53] Mohammad Mahdi Javanmard. "Parametric Multi-Way Recursive Divide-and-Conquer Algorithms for Dynamic Programs". PhD thesis. State University of New York at Stony Brook, 2020.

[54] Mohammad Mahdi Javanmard et al. "Deriving parametric multi-way recursive divide-and-conquer dynamic programming algorithms using polyhedral compilers". In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 317–329.

[55] Mohammad Mahdi Javanmard et al. "Toward efficient architecture-independent algorithms for dynamic programs". In: *International Conference on High Performance Computing*. Springer. 2019.

[56] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 9780124104143, 9780124104945.

[57] V. J. Jiménez et al. "Predictive Runtime Code Scheduling for Heterogeneous Architectures". In: *the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)* (2009), pp. 19–33.

[58] Zhenming Jin et al. "Structure of Mpro from SARS-CoV-2 and Discovery of Its Inhibitors". In: *Nature* 582.7811 (June 2020), pp. 289–293. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-2223-y. URL: http://www.nature.com/articles/s41586-020-2223-y (visited on 10/15/2021).

[59] Michael L. Scott John M. Mellor-Crummey. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". In: *TOCS'91*. 1991.

[60] L. Kalé and S. Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based on C++". In: *OOPSLA'93*. 1993.

[61] S. Kato et al. "TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments". In: *In Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)* (2011).

[62] Max Khiszinsky. *C++ library of lock-free containers and safe memory reclamation schema*. 2006. URL: http://libcds.sourceforge.net/.

[63] J. Kreutz. “DGEMM-Tiled matrix multiplication with Cuda”. In: *Jülich Forchungszentrum* (2013). URL: http://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/advanced-gpu/adv-gpu-cuda-DGEMM.pdf?__blob=publicationFile.

[64] Scott J. Krieder et al. “Design and Evaluation of the Gemtc Framework for GPU-enabled Many-task Computing”. In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. HPDC ’14. Vancouver, BC, Canada: ACM, 2014, pp. 153–164. ISBN: 978-1-4503-2749-7. DOI: 10.1145/2600212.2600228. URL: http://doi.acm.org/10.1145/2600212.2600228.

[65] Alexey Kukanov and Michael J Voss. “The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks.” In: *Intel Technology Journal* 11.4 (2007).

[66] Byung-Jae Kwak, Nah-Oak Song, and Leonard E Miller. “Performance analysis of exponential backoff”. In: *IEEE/ACM transactions on networking* 13.2 (2005), pp. 343–355.

[67] Alfio Lazzaro et al. “Increasing the Efficiency of Sparse Matrix-Matrix Multiplication with a 2.5D Algorithm and One-Sided MPI”. In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. Lugano Switzerland: ACM, June 2017, pp. 1–9. ISBN: 978-1-4503-5062-4. DOI: 10.1145/3093172.3093228. URL: https://dl.acm.org/doi/10.1145/3093172.3093228 (visited on 10/15/2021).

[68] J. Lima et al. “Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs”. In: *24rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (2012), pp. 75–82.

[69] Robert Lucas et al. *DOE advanced scientific computing advisory subcommittee (ASCAC) report: top ten Exascale research challenges*. Tech. rep. USDOE Office of Science (SC)(United States), 2014.

[70] Chao Mei et al. “Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study”. In: *TeraGrid’10*. 2010.

[71] X. Meng et al. “A cache-friendly concurrent lock-free queue for efficient inter-core communication”. In: *ICCSN’17*. IEEE, 2017.

[72] Xianghui Meng et al. “A cache-friendly concurrent lock-free queue for efficient inter-core communication”. In: *ICCSN’17*. 2017.

[73] Maged M. Michael and Michael L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms”. In: *PODC’96*. 1996.

[74] Gal Milman et al. “BQ: A Lock-Free Queue with Batching”. In: *SPAA’18*. 2018, pp. 99–109. DOI: 10.1145/3210377.3210388.

[75] Konstantina Mitropoulou et al. “Lynx: Using OS and Hardware Support for Fast Fine-Grained Inter-Core Communication”. In: *ICS’16*. 2016.

[76] Michael Mitzenmacher. “The power of two choices in randomized load balancing”. In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (2001), pp. 1094–1104.

[77] Adam Morrison and Yehuda Afek. “Fast Concurrent Queues for x86 Processors”. In: *PPOPP’13*. PPoPP, 2013, pp. 103–112. URL: https://dl.acm.org/citation.cfm?id=2442527.

[78] S. Nakagawa, F. Ino, and K. Hagihara. “A middleware for efficient stream processing in CUDA”. In: *Computer Science - Research and Development* 16 (2010), pp. 197–204.

[79] Poornima Nookala and Ioan Raicu. “XTASK - eXTreme fine-grAined concurrent taSK invocation runtime”. In: *Illinois Institute of Technology, Department of Computer Science, PhD Oral Qualifier*. 2017.

[80] Poornima Nookala et al. “Enabling Extremely Fine-grained Parallelism via Scalable Concurrent Queues on Modern Many-core Architectures”. In: *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2021, pp. 1–8.

[81] Poornima Nookala et al. “Evaluating the Support of MTC Applications On Intel Xeon Phi Many-Core Accelerators”. In: *International Conference on Cluster Computing*. CLUSTER ’15. 2015.

[82] Poornima Nookala et al. “Understanding Recursive Divide-and-Conquer Dynamic Programs in Fork-Join and Data-Flow Execution Models”. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2021, pp. 407–416.

[83] Stephen Olivier et al. “UTS: An unbalanced tree search benchmark”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2006, pp. 235–250.

[84] OpenMP Architecture Review Board. *OpenMP Application Programming Interface. Version 5.2.* Tech. rep. Nov. 2021. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[85] Alexandru Iulian Orhean et al. “MYSTIC: Programmable Systems Research Testbed to Explore a Stack-WIde Adaptive System fabriC”. In: *GCASR’19*. 2019.

[86] Artur Podobas, Mats Brorsson, and Vladimir Vlassov. “Scheduling for improved data-driven task performance with fast dependency resolution”. In: *IWOMP’14*. Salvador, Brazil: Springer, Sept. 2014, pp. 45–57. DOI: 10.1007/978-3-319-11454-5_4.

[87] Jean-Noël Quintin and Frédéric Wagner. “Hierarchical work-stealing”. In: *European Conference on Parallel Processing*. Springer. 2010, pp. 217–229.

[88] Ioan Raicu, Ian T Foster, and Yong Zhao. “Many-task computing for grids and supercomputers”. In: *2008 workshop on many-task computing on grids and supercomputers*. IEEE. 2008, pp. 1–11.

[89] Ian Rickards et al. *LIBLFDS*. 2009. URL: http://www.liblfds.org/.

[90] Matthew Rocklin. “Dask: Parallel computation with blocked algorithms and task scheduling”. In: *Proceedings of the 14th python in science conference*. Vol. 130. 2015, p. 136.

[91] Joseph Schuchart and José Gracia. “Global Task Data-Dependencies in PGAS Applications”. In: *High Performance Computing*. Springer International Publishing, 2019.

[92] Joseph Schuchart et al. *Quo Vadis MPI RMA? Towards a More Efficient Use of MPI One-Sided Communication*. 2021. arXiv: 2111.08142 [cs.DC].

[93] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. “Evaluating the cost of atomic operations on modern architectures”. In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 445–456.

[94] Thomas RW Scogland and Wu-chun Feng. “Design and evaluation of scalable concurrent queues for many-core architectures”. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. 2015, pp. 63–74.

[95] Larry Seiler et al. “Larrabee: A Many-core x86 Architecture for Visual Computing”. In: *ACM SIGGRAPH 2008 Papers*. SIGGRAPH ’08. Los Angeles, California: ACM, 2008, 18:1–18:15. ISBN: 978-1-4503-0112-1. DOI: 10.1145/1399504.1360617. URL: http://doi.acm.org/10.1145/1399504.1360617.

[96] Peter Sewell et al. “x86-TSO: a rigorous and usable programmer’s model for x86 multiprocessors”. In: *Communications of the ACM* 53.7 (2010), pp. 89–97.

[97] Shumpei Shiina and Kenjiro Taura. “Almost deterministic work stealing”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–16.

[98] M. Själander, M. Martonosi, and S. Kaxiras. *Power-Efficient Computer Architectures: Recent Advances*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, Dec. 2014. ISBN: 978-1-62705-645-8.

[99] Edgar Solomonik and James Demmel. “Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms”. In: *Link.Springer.Com*. 2011. DOI: 10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2.

[100] Herb Sutter. *The Trouble with Locks*. 2005. URL: http://www.drdobbs.com/cpp/the-trouble-with-locks/184401930.

[101] GCC team. *GOMP: An OpenMP Implementation for GCC*. URL: https://gcc.gnu.org/projects/gomp/.

[102] TOP500.org. *TOP500 List June 2015*. URL: http://www.top500.org/list/2015/06/.

[103] Sean Jeffrey Treichler. “Realm: Performance Portability through Composable Asynchrony”. PhD thesis. Stanford University, 2014.

[104] Guannan Guo Tsung-Wei Huang Chun-Xun Lin and Martin Wong. “Cpp-Taskflow: Fast task-based parallel programming using modern c++”. In: *IPDPS’19* (2019), pp. 974–983.

[105] David Tudor, Rachid Guerraoui, and Vasileios Trigonakis. “Everything you always wanted to know about synchronization but were afraid to ask.” In: *SOSP’13*. 2013.

[106] “Intel® 64 and IA-32 Architectures Software Developer’s Manual”. In: (2018). URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[107] Unified Communication Framework Consortium. *UCX: Unified Communication X API Standard v1.6*. Unified Communication Framework Consortium. 2019. URL: https://github.com/openucx/ucx/wiki/api-doc/v1.6/ucx-v1.6.pdf (visited on 04/09/2019).

[108] Pedro Valero et al. “A GPU-based implementation of the MRF algorithm in ITK package”. English. In: *The Journal of Supercomputing* 58.3 (2011), pp. 403–410. ISSN: 0920-8542. DOI: 10.1007/s11227-011-0597-1. URL: http://dx.doi.org/10.1007/s11227-011-0597-1.

[109] P. Valero-Lara. “Multi-GPU acceleration of DARTEL (early detection of Alzheimer)”. In: *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*. Sept. 2014, pp. 346–354. DOI: 10.1109/CLUSTER.2014.6968783.

[110] P. Valero-Lara and F. L. Pelayo. “Analysis in performance and new model for multiple kernels executions on many-core architectures”. In: *IEEE International Conference on Cognitive Informatics (ICCI*CC)* (2013), pp. 189–194.

[111] P. Valero-Lara and F. L. Pelayo. “Towards a More Efficient Use of GPUs”. In: *Computational Science and Its Applications (ICCSA) Workshops* (2011), pp. 3–9.

[112] Pedro Valero-Lara. “A GPU Approach for Accelerating 3D Deformable Registration (DARTEL) on Brain Biomedical Images”. In: *Proceedings of the 20th European MPI Users’ Group Meeting*. EuroMPI ’13. Madrid, Spain: ACM, 2013, pp. 187–192. ISBN: 978-1-4503-1903-4. DOI: 10.1145/2488551.2488592. URL: http://doi.acm.org/10.1145/2488551.2488592.

[113] Pedro Valero-Lara and Fernando L Pelayo. “Full-Overlapped Concurrent Kernels”. In: *Architecture of Computing Systems. Proceedings, ARCS 2015-The 28th International Conference on*. VDE. 2015, pp. 1–8.

[114] R. A. Van De Geijn and J. Watts. “SUMMA: scalable universal matrix multiplication algorithm”. In: *Concurrency: Practice and Experience* 9.4 (1997). DOI: 10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291096-9128%28199704%299%3A4%3C255%3A%3AAID-CPE250%3E3.0.CO%3B2-2.

[115] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. “A blocked all-pairs shortest-paths algorithm”. In: *Journal of Experimental Algorithmics (JEA)* 8 (2003), pp. 2–2.

[116] Junchang Wang et al. “B-Queue: Efficient and Practical Queuing for Fast Core-to-Core Communication”. In: *IJPP* 41.1 (Feb. 2013), pp. 137–159. ISSN: 1573-7640. DOI: 10.1007/s10766-012-0213-x.

[117] Ke Wang et al. “Exploring the design tradeoffs for extreme-scale high-performance computing system software”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.4 (2015), pp. 1070–1084.

[118] B. van Werkhoven et al. “Performance Models for CPU-GPU Data Transfers”. In: *CCGRID* (2014), pp. 11–20.

[119] Michael Wilde et al. “Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers”. In: *Journal of Physics: Conference Series* 180 (July 2009), p. 012046. DOI: 10.1088/1742-6596/180/1/012046.

[120] Justin M. Wozniak et al. “Swift/T: scalable data flow programming for many-task applications”. In: *PPOPP’13*. 2013.

[121] S. Yamagiva and L. Sousa. “Design and implementation of a stream-based distributed computing platform using graphics processing units”. In: *4th Int. Conf. Computing Frontiers (CF’07)* (2007), pp. 197–204.

[122] Yao Zhang, Jonathan Cohen, and John D. Owens. “Fast Tridiagonal Solvers on the GPU”. In: *SIGPLAN Not.* 45.5 (Jan. 2010), pp. 127–136. ISSN: 0362-1340. DOI: 10.1145/1837853.1693472. URL: http://doi.acm.org/10.1145/1837853.1693472.