EXTREME FINE-GRAINED PARALLELISM ON MODERN MANY-CORE ARCHITECTURES

BY

POORNIMA NOOKALA

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved __________________________
Adviser

Chicago, Illinois
December 2022

ACKNOWLEDGEMENT

First and foremost, I am incredibly grateful to my supervisor Dr. Ioan Raicu for his continuous support and invaluable guidance throughout my PhD. I am also extremely thankful to Dr. Robert Harrison for his treasured support and for advising my research work at Stony Brook University. Dr.Raicu and Dr.Harrison have been monumental in shaping my PhD and I can't thank them enough.

I would like to offer special thanks to my dissertation committee: Dr. Kyle Hale, Dr. Stefan Muller and Dr. Jia Wang. I would like to extend my sincere thanks to Dr. Kyle Hale and Dr. Stefan Muller from Illinois Institute of Technology and Dr. Peter Dinda from Northwestern University for helping me develop the core idea behind the thesis. I would also like to express my gratitude to Dr. Kyle Chard for help in polishing my papers. In addition, I would like to thank all my colleagues and collaborators for their time spent and efforts made in countless meetings, brainstorming sessions, and running experiments. This dissertation could not have been completed without the consistent encouragement and support from my husband Ram Karri. My appreciation also goes out to my children, my family and friends for their encouragement and support all through my studies.

I would like to acknowledge the support from National Science Foundation (NSF) for this work under grants OAC-2107548/2107283, CCF-1757964, CCF-1461260, CCF-1757970, CNS-1730689 at Illinois Institute of Technology and OAC-1931387 and ACI-1450344 at Stony Brook University.

AUTHORSHIP STATEMENT

I, Poornima Nookala, attest that the work in this thesis is substantially my own.

In accordance with the disciplinary norm of Computer Science (see IIT Faculty Handbook - https://web.iit.edu/general-counsel/faculty-handbook, Appendix S), the following collaborations occurred in the thesis:

I am extremely grateful to Dr. Ioan Raicu for his guidance and continuous support throughout my PhD journey. I will be forever grateful to Dr. Robert Harrison for giving me the opportunity to be part of the TTG team and for the guidance and contributions of Dr. Edward F. Valeev from University of Tennessee, Dr. George Bosilca, Dr. Thomas Herault and Dr. Joseph Schuchart from the University of Tennessee at Knoxville for making TTG possible. Dr. Mahdi Javanmard, Dr. Martin Kong, Dr. Rezaul Chowdhury and Zafar Ahmad have been extremely helpful in exploring the limitations of fork-join parallelism. I would also like to thank Dr. Pavan Balaji from Meta for the brainstorming sessions for improvising my ideas on XQueue.

Last, but not the least, I would like to thank my fellow PhD student Alexandru Orhean, and Caleb Lehman and Jonathan Anderson (who were part of the Research Experience for Undergraduates (REU) program) for their contributions to this work. I would also like to thank Serapheim Dimitropoulos, Karl Stough, Pedro Valero-Lara, Fernando L Pelayo and Johan Jansson for their contributions to my early work in Many Task Computing.

The work in this thesis would not have been possible without the inputs and guidance from everyone listed here. I am extremely grateful to all!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 the fork-join model is suitable for on-node parallelism, the use of joins and synchronization induces artificial dependencies which can lead to under utilization of resources. Data-flow based parallelism is crucial to overcome the limitations of fork-join parallelism by specifying dependencies at a finer granularity. It is also crucial for parallel runtime systems to support heterogeneous platforms to better utilize the hardware resources that are available in modern day supercomputers. 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 can address the issues of performance portability 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.

CHAPTER 1
INTRODUCTION

The Department of Energy (DOE) has reported that *"Scientific productivity is one of the top ten exascale research challenges"* [3]. The scientific computing community is facing unprecedented changes in computer architectures that has fueled the emergence of the many-core computing architecture. Today's high-end computing systems have 100s of processors and GPUs have 1000s of threads of execution. In a recent report [4], 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 of a 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 [5]. 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 [6].

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 [7]. 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 of performance/cost and an efficient power consumption design [8, 9, 10]. 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 [11], many of the most powerful supercomputers today are based on platforms that combine multi-core and many-core processors with data parallel accelerators. These many-core architectures have the potential to address the needs of computation-hungry scientific applications at the node level, but they are difficult to program due to an increasing level of parallelism that requires programmers to have a deep understanding of hardware, parallel constructs, and associated synchronization mechanisms.

Shared memory parallelism can be expressed in various forms mainly loop-based parallelism and task-based parallelism. Typically, in loop-based parallelism, loops are divided into chunks of equal size and executed concurrently by different threads managed by a parallel runtime system. Task-based parallelism evolved in parallel runtime systems to support irregular parallelism in applications where the parallelism cannot be expressed by loops. In task-based parallelism, an application is decomposed into dependent or independent tasks to form a task graph which can be executed by different threads in the system. Furthermore, task-based parallelism can be categorized based on the way task graphs are expressed: control-flow and data-flow. In control-flow, the dependent tasks can only be triggered after completion of the parent task, whereas in data-flow, the tasks are ready to run when input data

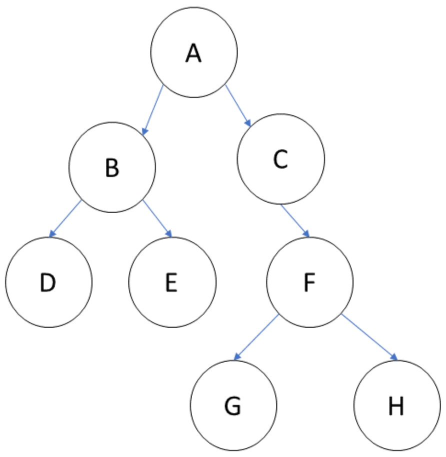


Figure 1.1. Directed Acyclic Graph (DAG)

for a task becomes available. We explore both control-flow and data-flow based task parallelism in this work.

1.1 Task-Based Parallelism

Task-based parallelism is a simple paradigm for shared memory parallelism in which a computation is broken-down into a set of inter-dependent tasks which can then be executed concurrently on various cores. When a task is created by some processor or 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 Graphs (DAGs) which 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 1.1 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 [12, 13, 14, 15, 16, 17] rely on such task parallelism. 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.

1.2 Many-Task Computing

Many-Task Computing (MTC) [18] 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.

1.3 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 [1] that incorporates earlier work on the Larrabee architecture [19]. 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 coprocessor 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 [20, 21] 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 [1]. 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 [1]. Unlike previous GPUs, the L2 cache is kept fully coherent by a global-distributed tag directory.

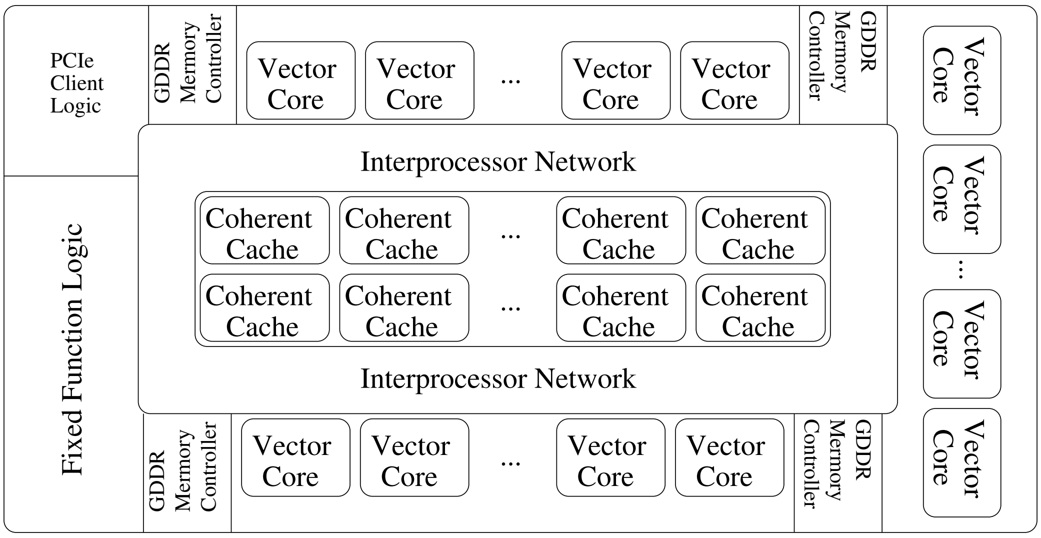


Figure 1.2. Micro-architecture of the Entire MIC coprocessor [1].

Due to the foundations of Intel architecture, the coprocessor can be programmed in several different ways. Here, we introduce two different approaches, OpenMP and SCIF (Intel's Symmetric Communications Interface). OpenMP uses offloading approach for offloading computations from the host to the accelerator. The SCIF implementation runs natively on the accelerator and accepts jobs from Clients running on the host. There are several advantages and disadvantages between the two methods. 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. Frameworks that use offloading mode (*OpenMP*), do not necessarily take advantage of the DMA-features of the hardware they run on while on *SCIF* you are guaranteed that if you are using Remote Memory Access (RMA). That is not to say that *OpenMP* does not come with any advantages over *SCIF*. Quite the opposite, the advantages of offloading are pretty significant for the framework that was implemented for this project. The low-level C code needed for the *SCIF* implementation is relatively a lot more complex when compared with *pragma* directives provided by *OpenMP*. In addition, using *SCIF* implies that the framework must have at least one of its parts running natively on the Phi as the endpoint. In order to do that the developer needs to set up an application to run natively on the Phi and involves a lot of configuration. Using *OpenMP* with the offloading capabilities provided by the MIC, all this configuration is taken care of.

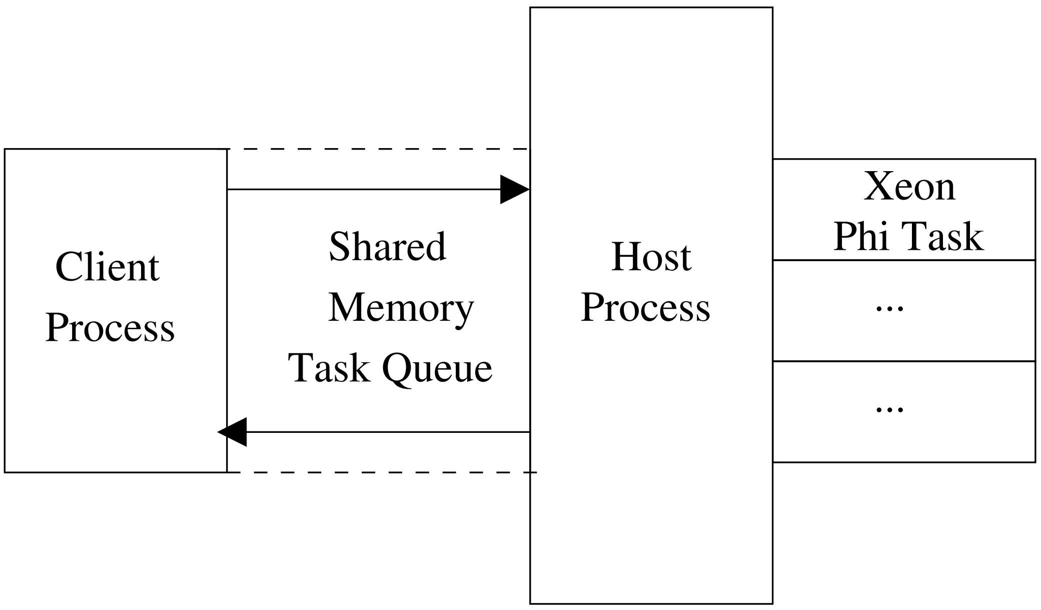


Figure 1.3. Micro-architecture of the Entire MIC coprocessor [1].

The OpenMP version of the framework as shown in Figure 1.3 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. This approach was chosen to provide the same feature set as GeMTC [22] while taking advantage of asynchronous offloading capabilities of OpenMP.

SCIF implementation employs a Client Server architecture where clients send their tasks to the Phi from the host and the server, which runs natively on the Phi, accepts the jobs. After submitting the job, the clients can request the result and the server will deliver it to them when the task has finished processing and is placed on the results queue of the framework. The whole procedure is non-blocking for the server who can handle multiple requests and submissions at the same time. That functionality is implemented with *epoll()* for handling connections that are later passed to threads [23] that push or dequeue tasks from the queues. The *SCIF* socket-like API is used for communications between the server and the clients. It comes as a shared library named **libmtcq*. This library includes all the functionality that handles incoming and outgoing queues of tasks, pushing jobs and distributing tasks to workers. It is also completely parametrizable in terms of queue sizes, worker threads, and application threads. Since the Xeon Phi does not have the hierarchical architecture of SMXs and Warps nor the concept of application kernels that you generally see in GPGPUs, everything is implemented with standard Pthreads. There is a parametrizable number of master threads that dequeues tasks from the incoming queue. If the task is a parallel application, which is the case most of the time, then the master thread will assign the task to the specified number of worker threads. Else if it is sequential only one thread will be assigned and the master thread will go back to dequeue more jobs. Each queue is implemented as a finite buffer from the Producer-Consumer model which means that it uses a single mutex and two semaphores to

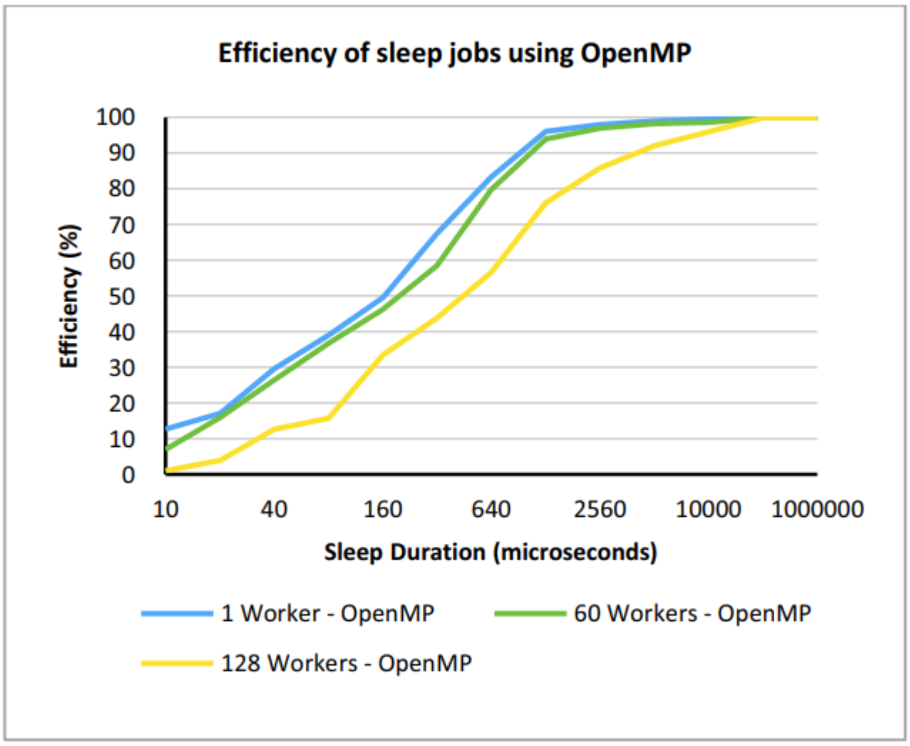


Figure 1.4. Efficiency of Sleep workload using MIC by varying concurrency

ensure that no deadlocks or data-races arise.

All of our experiments were run on the Midway High Performance Computing Cluster at University of Chicago. Our testing host is an Intel Sandy Bridge with 32 cores at 2.6 Ghz and 32 GB of RAM. It has 2 Xeon Phis attached to it.

We performed experiments using a synthetic sleep workload with various sleep length tasks. As seen in figure 1.4, preliminary results show that efficiency reaches higher 90s for task lengths at 1 msec when using 1 worker on the host, 2 msec for 60 workers and 5 msec for 128 workers. This clearly shows that this framework using OpenMP performs better than GeMTC on Xeon Phi which reaches higher efficiency only at 5 ms. To reduce the overhead of multithreading, we took an approach of creating threads on the Phi before offloading tasks which reduced the overall execution time considerably. Also, figure 1.5 shows the comparison of sleep workload efficiency

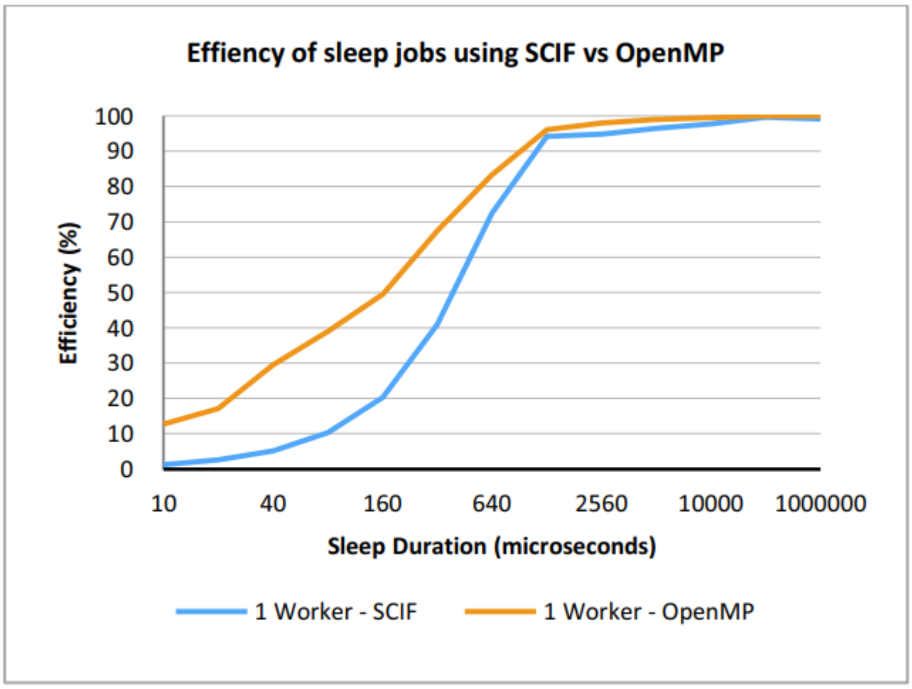


Figure 1.5. Efficiency comparison of sleep workload using OpenMP and SCIF

between OpenMP and SCIF. Both implementations achieve 90% efficiency with sleep duration of 640 microseconds.

To enable running MTC workloads on Xeon Phi, we designed a framework that not only sends and executes tasks on Xeon Phi but also ensures that these tasks are isolated from each other and can run in parallel. We implemented both OpenMP as well as SCIF-based frameworks and were able to run MTC workloads on Xeon Phi.

1.4 Dissertation Overview 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 use concurrent data structures like queues to hold a bag of tasks. In Chapter 2, we shifted our focus to analyzing the performance of concurrent queues and synchronization mechanisms to understand the overheads of managing tasks in task-based runtime systems. In Chapter 3, we propose a lock-less concurrent out of order queueing mechanism, XQueue, with static round-robin load balancing aimed at reducing the overheads of concurrent data structures in parallel runtime systems. We integrate the lock-less mechanism into LLVM's implementation of OpenMP to test our idea on real applications written using OpenMP. In Chapter 4, we introduce X-OpenMP which extends XQueue-enabled LLVM OpenMP and implements dynamic load balancing using lock-less work stealing. X-OpenMP library can be used to transparently accelerate applications written in OpenMP just by linking against our library. In Chapter 5, we analyze the limitations of fork-join programming model in OpenMP and compare with data-flow programming models. We show that data-flow programming models are important for expressing fine-grained parallelism in applications. In Chapter 6, we present a new data-flow based programming model TTG that aims to bridge the gap between programmer productivity and performance portability. In Chapter 7, we discuss the related work and in Chapter 8, we conclude.

CHAPTER 2

HOW BAD IS CONCURRENT QUEUE PERFORMANCE ACROSS MANY THREADS?

A queue is a data structure that allows insertion of items using enqueue operation and removal of items using dequeue operation. The operations are performed in a first-in first-out (FIFO) order. A concurrent queue allows multiple producers to insert items and multiple consumers to remove items from the queue by protecting the operations using synchronization mechanisms. Concurrent queues permeate computer systems and networks. For example, Hadoop MapReduce [24] uses scheduling queues to organize applications and share resources between the running applications. Apache Spark [25] uses queues for FIFO job scheduling and can concurrently execute jobs submitted by various threads. Database Management Systems and Information Retrieval Systems often have a query execution engine that is responsible for selecting the proper selection algorithms to implement a given query plan, deciding whether intermediate results are materialized or pipelined and executing the resulting physical query plan as a parallel program. Today's TPUs [26] can be used to accelerate neural network performance; however, the input data pipeline needs to be efficient to extract data asynchronously for achieving peak performance. Multiple concurrent queues can be used for data pipelining by fetching data ahead of time thereby improving resource utilization and increasing the overall throughput.

Our focus on concurrent queues is more specifically motivated by intra-node task parallelism on modern and future parallel computers. Task parallelism is an important paradigm for shared-memory parallelism in which computation is broken down into a set of inter-dependent tasks which can then be executed concurrently on various cores. Task dependencies and data dependencies are used to control the flow of tasks through the runtime system. The execution models of many parallel languages use such task parallelism. For example, OpenMP [27] has evolved to a task-centric model where even parallel loops are compiled to fine granularity tasks with dependencies which the run-time must dynamically schedule to the available resources. When a task is enabled by some thread, it is conceptually queued for execution by a future available thread. Software dataflow languages [14] similarly have a runtime that executes a dynamically unfolding task graph with scheduling via concurrent queues.

To achieve strong scaling and high effective levels of parallelism, today's parallel languages and execution models are also moving to finer and finer granularity tasks. One reason for this is that as core counts grow on the node, applications need to support over-decomposition (many more tasks than cores) in order to improve performance, hide latency caused by blocking on dependencies, and otherwise achieve maximum speedup [28]. This and other drivers produce the same outcome: tasks and their dependencies need to be managed at the sub-microsecond granularity. As the need for extremely low latency, high throughput concurrent queues grows, achieving them is simultaneously becoming increasingly difficult because the node itself is scaling.

Of particular interest here are single producer, single consumer (SPSC) and multiple producer, multiple consumer (MPMC) concurrent queues. The queue itself contains tasks, typically in the form of pointers (to task objects). A concurrent SPSC queue allows a single producer to enqueue while a single consumer simultaneously dequeues. An MPMC concurrent queue allows multiple simultaneous producers and consumers to queue and dequeue. While each producer and consumer is typically a software thread, the interesting case for latency and throughput is when these are mapped to non-overlapping hardware threads (logical cores) and scheduled simultaneously. We elaborate on these queues in Section 2.2, and show that simple approaches to their concurrency, while perfectly adequate at small scales, quickly fall apart as we consider larger nodes.

Threads running concurrently in this manner can interleave instructions in many ways and a shared data structure needs to be carefully protected to avoid races. Concurrent SPSC and MPMC queues are no exception and require that their state (e.g. head and tail) be protected with a synchronization mechanism. Various synchronization mechanisms [29] exist, including mutual exclusion locks (mutexes), spinlocks, semaphores, and atomic primitives.

A second approach to concurrent queues is avoid separate synchronization by embedding race-avoidance directly into the data structure design itself. This also has the benefit of avoiding the possibility of deadlock due to misuse of synchronization primitives (e.g., lock acquisition in different orders along different codepaths). ***lock-free*** data structures achieve this through the use of atomic primitives, such as CAS (compare-and-swap) and Fetch-And-Add (FAA). Several libraries internally use lock-free techniques [30, 31, 32], but the literature has show that it is difficult to write lock-free code that is correct [33, 34, 35, 36]. Even more compelling are ***lock-less*** data structures [37], which not only avoid the use of locks, but also can avoid the need for atomic operations under certain conditions. Both lock-free and lock-less programming are challenging due instruction and memory access reordering that occurs due to the compiler and the hardware, and the need to observe the memory consistency model that is actually provided between logical cores.

How well do the basic primitives on which traditionally synchronized and lock-free/lock-less SPSC and MPMC queues build actually perform? To address this question, we evaluated the scalability of a wide range of primitives in terms of latency of throughput. A large, diverse set of hardware was used, including numerous variants of x64 from Intel and AMD, two generations of Intel Xeon Phi, ARM, and IBM Power9 (details in Table 2.1). Machines with as many as eight sockets and 384 logical cores (hardware threads) were considered.

This work has the following contributions:

1. We motivate scalable SPSC and MPMC constructs from the needs of fine-grained task parallelism.
2. We provide a detailed performance study of synchronization primitives, including mutexes, semaphores, spin locks, and atomic fetch-and-add operations, on today's largest shared-memory systems from Intel, AMD, IBM, and ARM. Systems with up to 8 sockets and 384 hardware threads are included.

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 sharing 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.

In order to show the scalability and performance of MPMC queues compared to SPSC queues, we selected five diverse systems (see Table 2.1) from the Mystic Testbed [38] 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

Table 2.1. 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

work) can be found in Table 2.1.

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 [39]. Listing 1 shows the implementation of enqueue and dequeue operations for an SPSC queue. 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

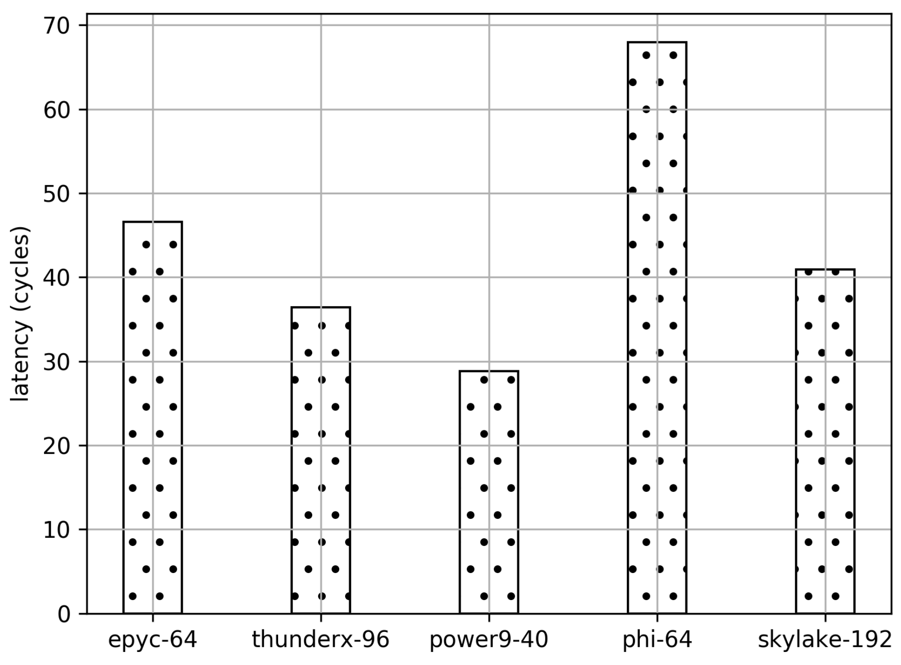


Figure 2.1. Average latency of enqueue/dequeue operations on SPSC queue

size is set to the number of samples for the purposes of this evaluation. Results in Figure 2.1 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 2.2 represents the throughput, which is the rate at which items are being processed by the queue. For throughput experiments, 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

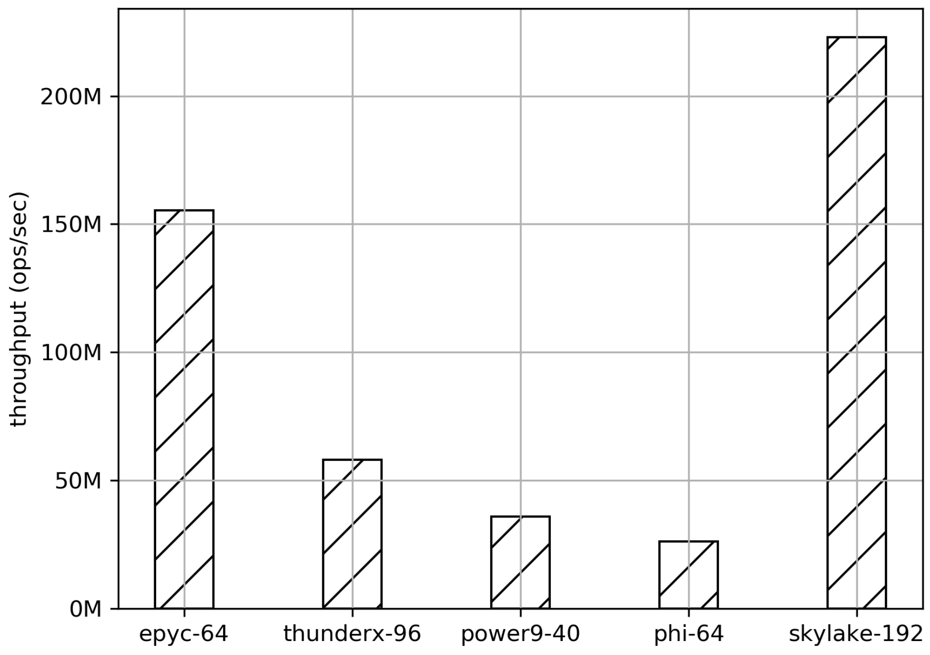


Figure 2.2. Average throughput of enqueue/dequeue operations in millions(M) on SPSC queue

cannot be used with more than one producer and one consumer.

Listing 2 shows the implementation of enqueue and dequeue operations for a multiple producer multiple consumer queue. 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 numbers 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 processors can result in better cache utilization, thereby reducing costly

```c
void enqueue(int item, struct queue *q)
{
    q->tasks[(++q->rear) % q->capacity] = item;
    return;
}

int dequeue(struct queue *q)
{
    int item = q->tasks[(++q->front) % q->capacity];
    return item;
}
```

Listing 1. Single Producer Single Consumer Queue Operations

```c
void enqueue(int item, struct queue *q)
{
    sem_wait(&q->spaces_sem);
    pthread_mutex_lock(&q->lock);
    q->tasks[(++q->rear) % q->capacity] = item;
    pthread_mutex_unlock(&q->lock);
    sem_post(&q->task_sem);

    return;
}

int dequeue(struct queue *q)
{
    int item;
    sem_wait(&q->task_sem);
    pthread_mutex_lock(&q->lock);
    item = q->tasks[(++q->front) % q->capacity];
    pthread_mutex_unlock(&q->lock);
    sem_post(&q->spaces_sem);

    return item;
}
```

Listing 2. Multiple Producer Multiple Consumer Queue Operations

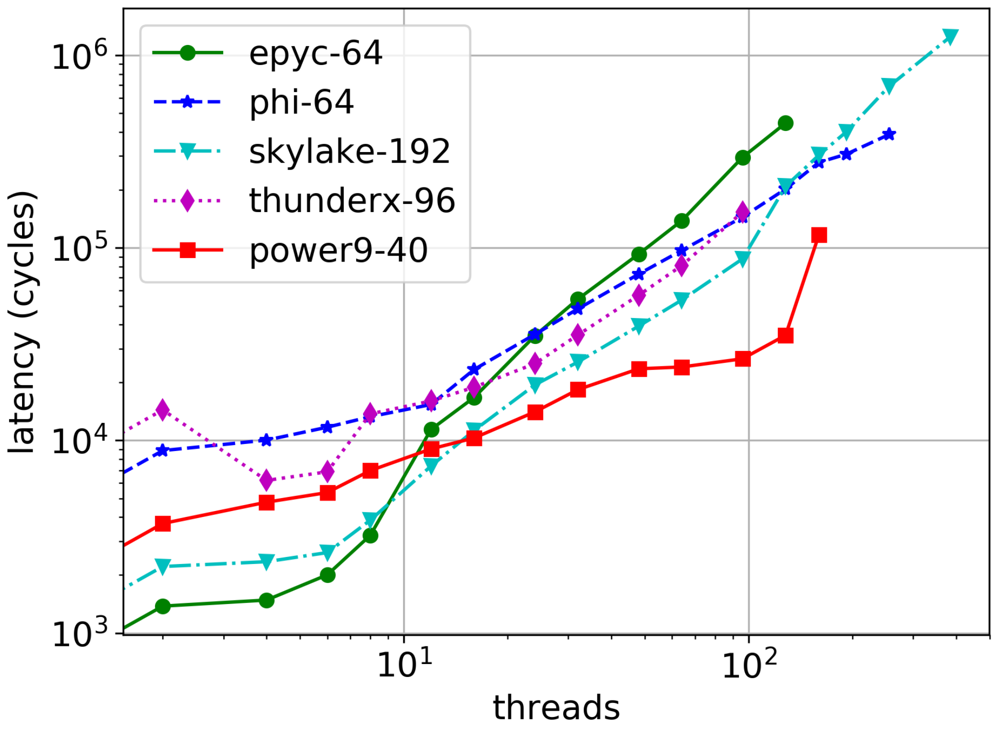


Figure 2.3. Average latency of enqueue/dequeue operations on a lock-based queue. This graph is shows that simple lock-based queues don't scale beyond 8 threads on any modern processors.

memory accesses. This thread placement is a result of tests performed by pinning producer to core 0 and 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 2.3 and 2.4 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× loss of performance). The fastest MPMC queue throughput at any scale reached just 5 million operations/sec. These

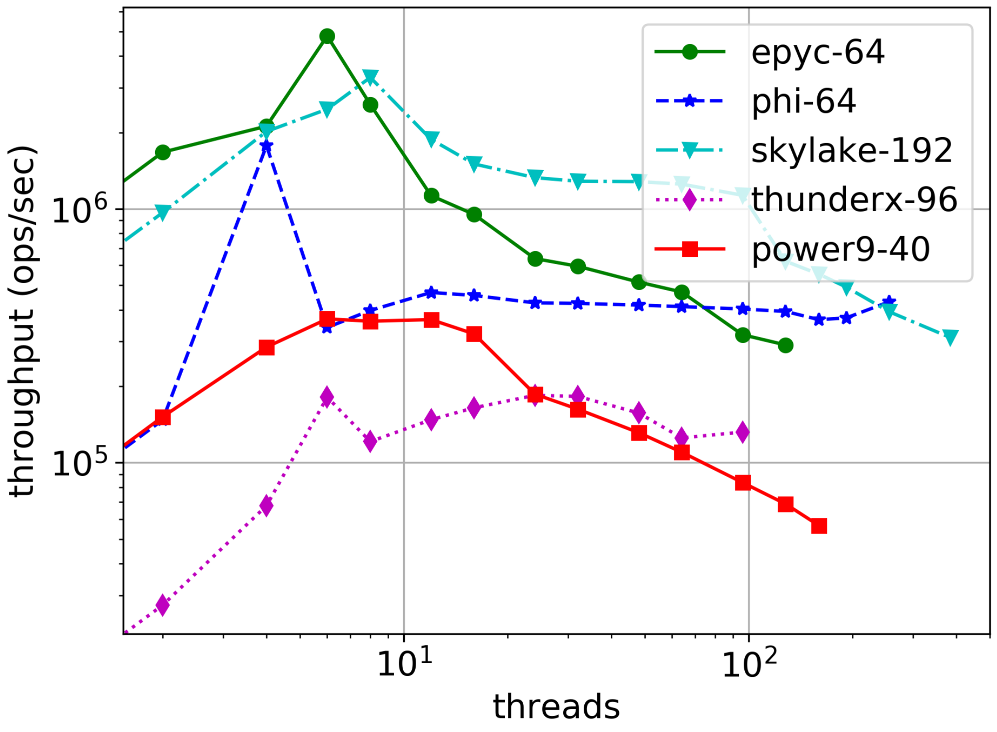


Figure 2.4. Average throughput of enqueue/dequeue operations on lock-based queue. This graph shows that the throughput of a simple lock-based queue plateaus beyond 8 threads on modern processors

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.

2.2 Analysis of Synchronization Mechanisms

This section conducts a detailed performance study [40] 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 Testbed, Software Stack, and Timing Mechanisms.

Testbed: Table 2.1 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.

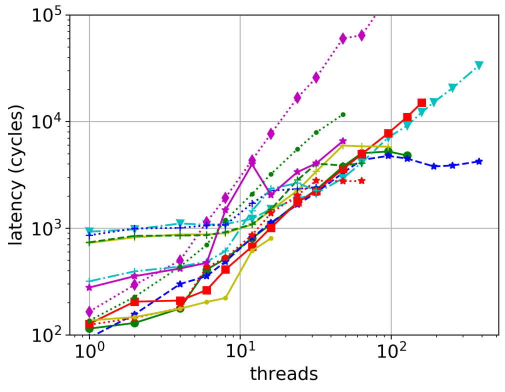
The primary focus here is to analyze the cost of low-level thread synchroniza-

```
for (int i = 0; i < NUM_SAMPLES_PER_THREAD; i++)
{
    lock();
    counter++;
    unlock();
}
```

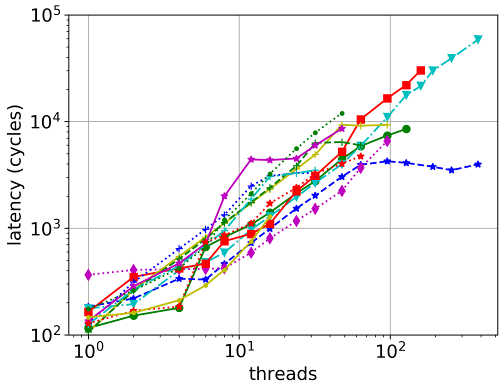
Listing 3. Single Producer Single Consumer Queue Operations

tion mechanisms and for this purpose, we benchmarked *pthread_mutex*, *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 'ldxr/stxr' which are load/store exclusive instructions used for implementing atomic read, modify, write operations. These benchmarks are obtained by running a loop of 1 billion operations and collecting the aggregate of the results. As shown in Listing 3, each iteration acquires the lock, increments a shared integer and releases the lock, excluding fetch-and-add which performs an increment operation atomically.

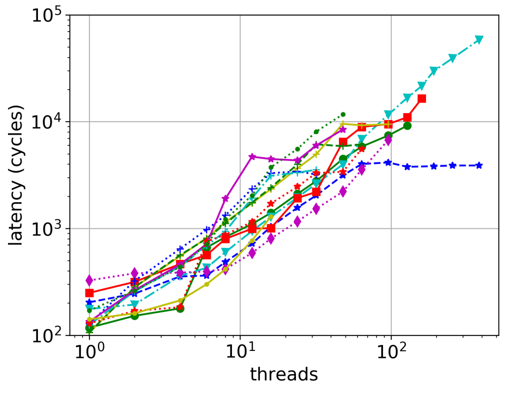
Figure 2.5 shows that all synchronization mechanisms exhibit higher latencies due to contention at higher levels of concurrency. There are many factors that impact the cycle counts like cache coherence, communication latency between cores on same and different sockets, interrupts, cache misses, etc. 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 system 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



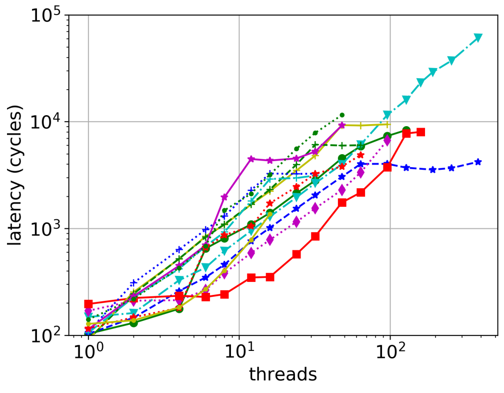
(a) Atomic Fetch-and-add



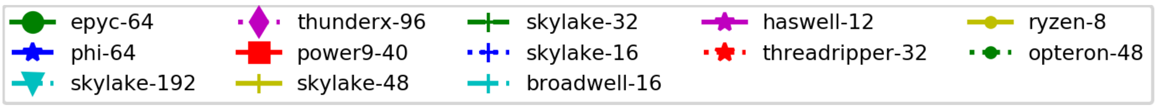
(b) Mutex



(c) Semaphore



(d) Spinlock



(e) Legend

Figure 2.5. 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.

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 [41].

2.3 Summary

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 [42]. 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. From the results presented in Section 2.2, it is clear that having a single lock across all threads is not scalable and severely limits parallelism across many threads. Using more locks is a better alternative which will result in less contention and allow for more fine-grained parallelism. However, more locks also means more CPU cache flushes to maintain cache coherency, which could adversely impact the performance of locking code. A balanced locking scheme with fewer locks (or no locks if possible) can support an ever increasing amount of parallelism in concurrent programming. The key idea behind this work originates from the significant loss in performance of MPMC queues as compared to SPSC queues as observed in Section 2.1.

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 or data flow carefully.

We make the following contributions in this chapter:

1. We design and implement XQueue [39], a lock-less, relaxed-order MPMC queue that uses multiple queues for improved locality without using locks or atomic operations. We demonstrate the scalability of XQueue using microbenchmarks measuring latency as low as 110 cycles and throughput as high as 1 billion ops/sec across today's largest shared-memory systems from Intel, AMD, IBM, and ARM up to 192-core scales. These numbers represent 6900X lower latencies and 3300X higher throughput compared to existing MPMC queue implementations.

2. We integrate XQueue into LLVM OpenMP and evaluate the performance improvements on 6 unmodified applications (Fib, FFT, Multisort, NQueens, Health and Strassen) from the Barcelona OpenMP Task Suite (BOTS) as well as the breadth first search (BFS) application from the GAP benchmark suite, Gaussian Elimination algorithm and Symmetric Rank Update kernel from the PLASMA numerical library [43]. We show that the combination of XQueue and LLVM OpenMP is capable of delivering better scalability for fine-grained task-parallel workloads with up to 6× speedup compared to native LLVM OpenMP and 1× to 4× speedup compared to GNU OpenMP in most cases, and up to 116× speedup in some cases.

3.1 XQueue: Lock-less Queueing Mechanism for Task-Parallel Runtime Systems

We introduce XQueue [39], a novel lock-less MPMC, out-of-order queuing mechanism that can scale up to hundreds of threads. XQueue uses B-queue [44] 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 [45, 46, 47] and we aim to demonstrate that XQueue can be built with any fast and scalable SPSC queue.

Figure 3.1 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

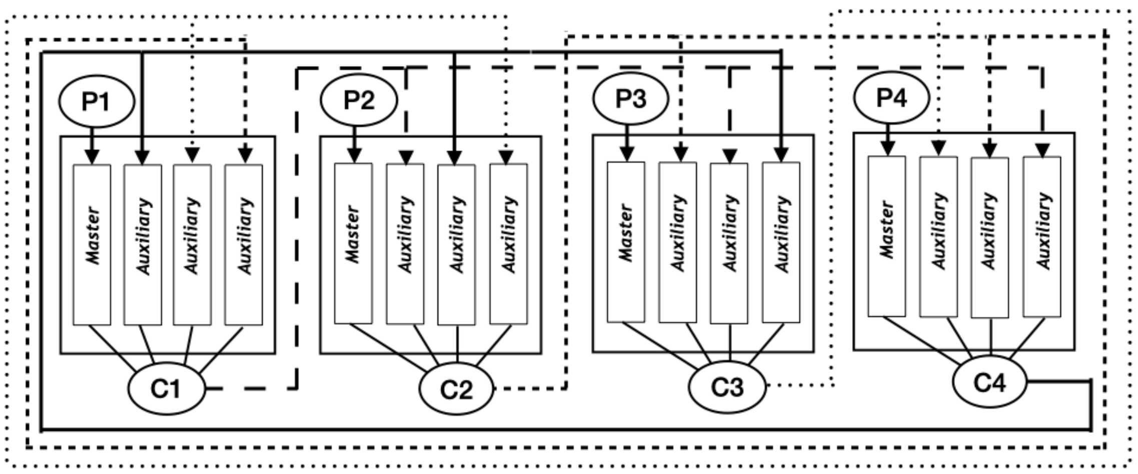


Figure 3.1. Architecture of XQueue on a 4-core machine with 4 queues per consumer.

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 3.1. 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 Listing **??**. 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 3.1) 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

```
Input: $id \leftarrow coreId$;
Input: $next \leftarrow nextCoreId$;
1  while 1 do
2      $ret \leftarrow dequeueFromMaster(id, item)$;
3      if $ret = SUCCESS$ then
4          $retItem \leftarrow processItem(item)$;
5          if $retItem \neq NULL$ then
6              $enqueueToAuxiliary(next, retItem)$;
7      $ret \leftarrow dequeueFromAuxiliary(id, item)$;
8      if $ret = SUCCESS$ then
9          $retItem \leftarrow processItem(item)$;
10         if $retItem \neq NULL$ then
11             $enqueueToAuxiliary(next, retItem)$;
12     $next \leftarrow (next + 1)\%numCores$;
13     if $next == id$ then
14         $next{+}{+}$;
```

3.1.1 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 [48, 49]. 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 3.1, 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 3.2.

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.

3.1.2 XQueue Integration with the OpenMP Runtime. In order to extend our research to real systems, we integrated XQueue into OpenMP [50] 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) [51], LLVM OpenMP [50], 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 [52].

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-enabled OpenMP implementation, every thread checks its 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. For the applications we evaluate, this hinting mechanism gives better performance while maintaining good load balancing. We have used physical cores available on the machine for this evaluation by setting OMP_PLACES environment variable to 'cores' and OMP_PROC_BIND to 'close', since not all applications can benefit from using hardware threads.

3.2 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 2.1); 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.

3.2.1 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 [16] with a separate queue per worker.

Cilk deque is implemented as part of Cilk 5 multi-threaded language [16] and uses a shared-memory, mutual-exclusion protocol called the THE protocol[53] 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.

3.2.2 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 3.2 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 3.2.2 compares the latency of XQueue (SPSC) with Cilk Deque and MPMC queues on skylake-192. Here, Cilk Deque/MPMC is a single queue shared

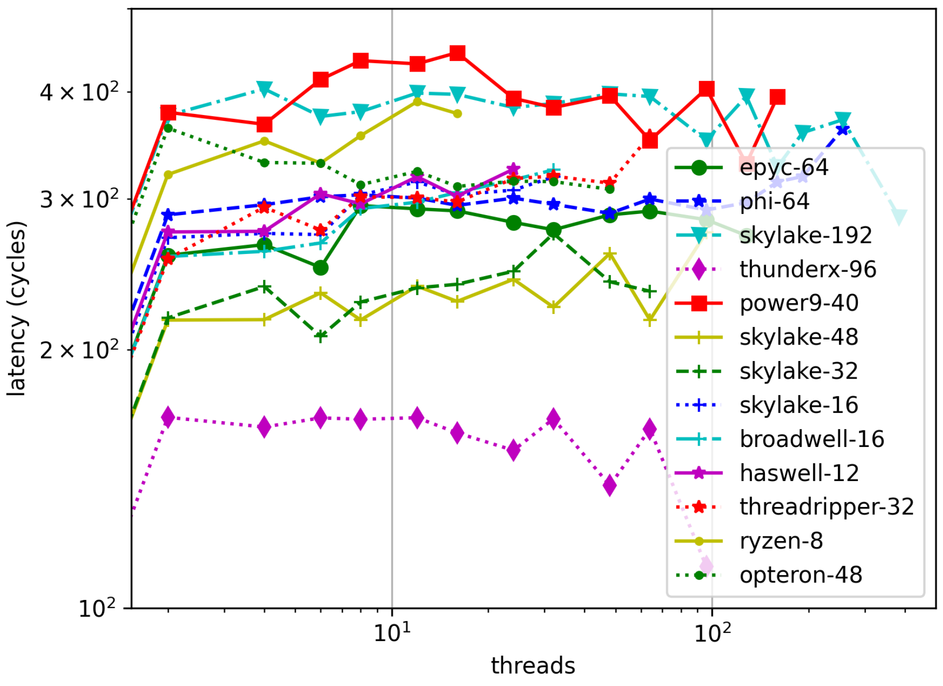


Figure 3.2. Average latency of enqueue/dequeue operations using XQueue (SPSC)

across all the workers. With 192 producers/consumers, latency of MPMC queue is 13× 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.

Figure 3.2.2 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

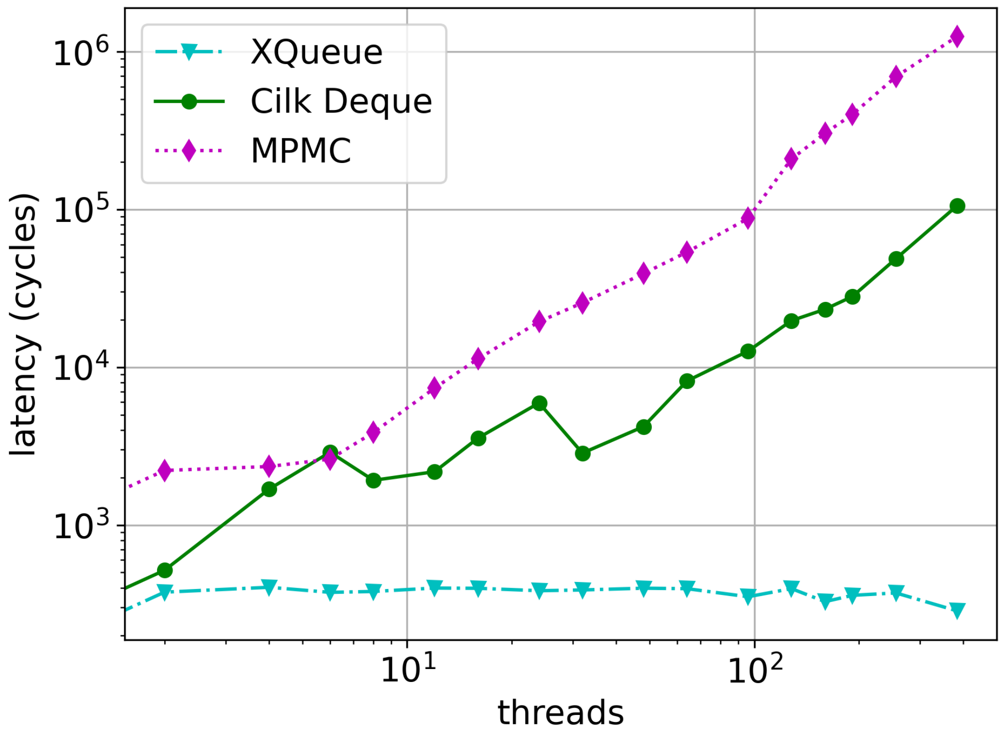


Figure 3.3. Latency Comparison

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. As noted in Section 2.2 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

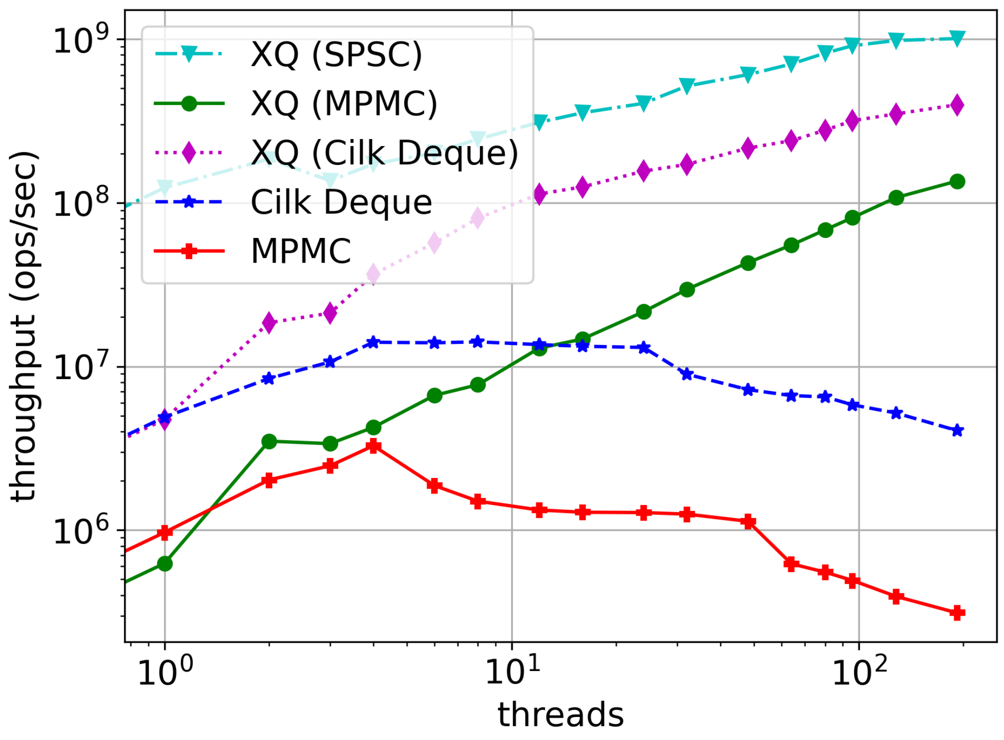


Figure 3.4. Throughput Comparison

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.

3.2.3 Queue Length Study. While the above microbenchmarks use homogeneous tasks which run for the same amount of time, real applications typically have a mix of fine-grained and coarse-grained tasks. We designed a benchmark to explore this scenario. In this benchmark, the master worker receives $N \times 10$ tasks (where N is the number of workers) which run for 1 second each. The ideal time to execute these tasks with perfect load balancing is 10 seconds. Also, every worker receives 8 million delay tasks of length 0.1 microseconds, which should ideally execute within a second. The idea behind this synthetic benchmark is to simulate starvation where load imbalance exists between workers and some workers can become idle for long periods of time. In case of X-OpenMP, steal requests need to be handled by busy workers to facilitate work stealing and this benchmark is designed to understand how the runtime behaves. Figure 3.5 shows the results obtained by running this benchmark. The plot shows execution time by varying queue sizes in the runtime implementations to understand the impact of queue size when a single worker is overloaded with work. Please note that the queue sizes couldn't be matched for X-OpenMP and LLVM implementations since the latter requires powers of two for queue length. The execution time of this synthetic benchmark for LLVM OpenMP shows that the performance of the runtime is sensitive to queue size in such scenarios and starvation can occur. XQueue and X-OpenMP can handle such load imbalances well due to the round-robin load balancing where long-running tasks get distributed to all the workers. This experiment also clearly shows that X-OpenMP is not sensitive to the size of the queue due to the existence of multiple queues per worker. The queue sizes can be kept small, thereby requiring less memory for the underlying runtime and achieve good performance.

3.2.4 Data Locality Study. In order to understand the performance of XQueue on NUMA architectures, we evaluate the STREAM [54] memory benchmark. It is a synthetic benchmark written in C and parallelized using OpenMP work sharing. The benchmark measures sustainable memory bandwidth by performing basic operations on large vectors. The vectors are big enough (10 million double precision floats) to fit in the cache memory and over 90% of the data is accessed from the main memory during the computation. We modified this benchmark to implement task-parallel STREAM Triad where the main loop is divided into chunks and every chunk becomes a task during execution.

We evaluate the benchmark using XQueue and compare with LLVM and

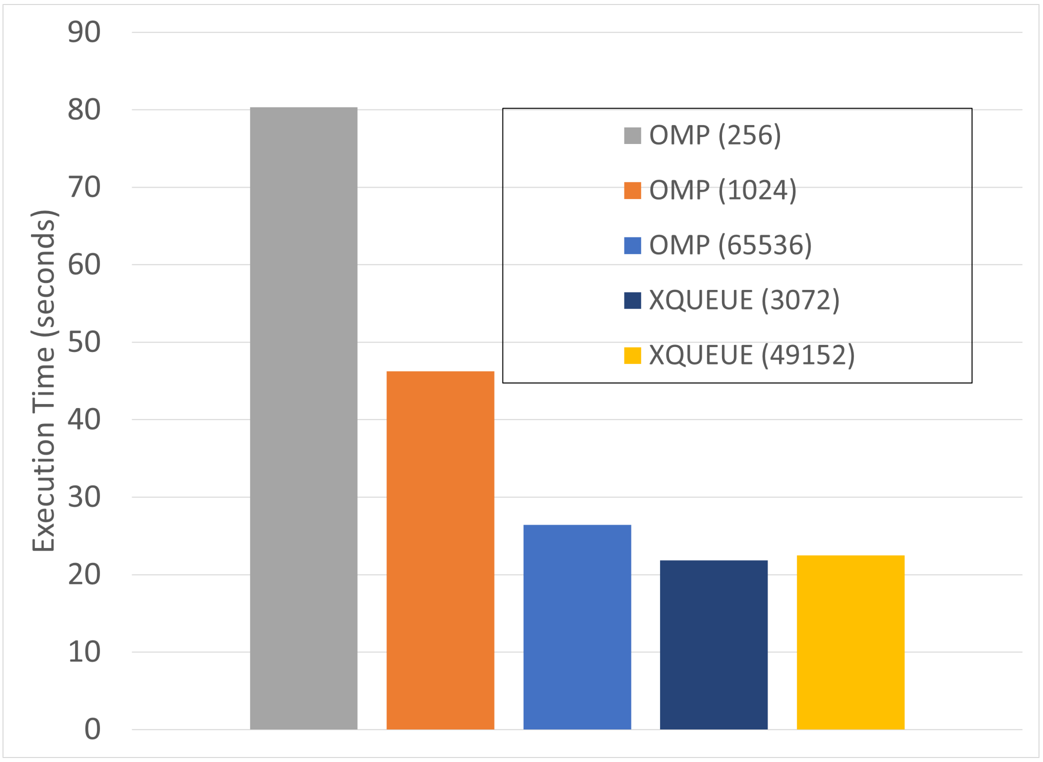


Figure 3.5. Execution Time (using 192 threads) of a synthetic benchmark with a mix of fine-grained and coarse-grained tasks and varying queue sizes (legend shows queue sizes in parentheses).

GNU implementations of OpenMP. Figure 3.6 shows the bandwidth achieved on the skylake-192 machine by running the STREAM benchmark at varying concurrency levels. The peak memory bandwidth measured using Intel's memory latency checker tool [55] is about 110 GB/s within a single NUMA node and 16 GB/s across nodes. The task-based version of STREAM benchmark distributes tasks across all workers and achieves around 75% of the measured peak bandwidth across nodes. Results show that XQueue achieves 40-60% more bandwidth as compared to the work stealing based LLVM's implementation. GNU OpenMP achieves significantly lower performance compared to XQueue and LLVM implementations.

3.2.5 Macro-benchmark Performance Results. To quantify the improvements in real application workloads, we evaluate the speedup achieved using XQueue-

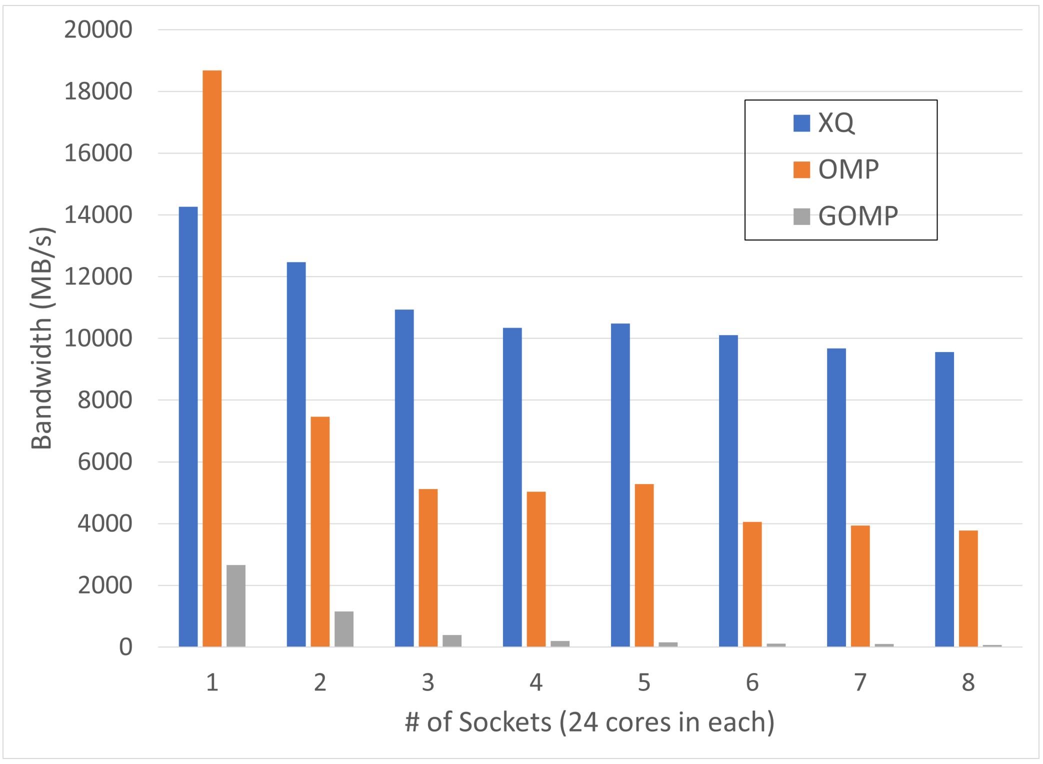


Figure 3.6. Memory Bandwidth obtained using STREAM Triad benchmark.

enabled LLVM OpenMP as compared to the native LLVM OpenMP and GNU OpenMP libraries. We evaluate six out of nine applications from the BOTS benchmark suite [56]: Fibonacci, FFT (Fast Fourier Transform), Multisort, NQueens, Health and Strassen's Matrix Multiplication, the Gaussian Elimination algorithm implemented using OpenMP tasking and Symmetric Rank Update (dsyrk) kernel from the PLASMA numerical library [43]. Results are shown in Figure 3.7. We also evaluate the breadth first search application from the GAP benchmark suite [57] with real-world social network graphs such as those from Friendster and Twitter. Results are shown in Figure 3.10. The application workloads are summarized in Table 3.1.

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

Table 3.1. 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

tasks, and thus exposes the limits of a tasking runtime in terms of granularity. Figure 3.7 shows the results obtained on skylake-192. OpenMP with XQueue achieves 3× 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× speedup for the largest problem size. However, the performance drops by 50% at 192 threads. As shown in Figure 3.7, 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 heavily 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 [58]. 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× speedup compared to LLVM variant and 4× 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 3.8 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× to 1.2×) due to the smaller overheads incurred by avoiding locks.

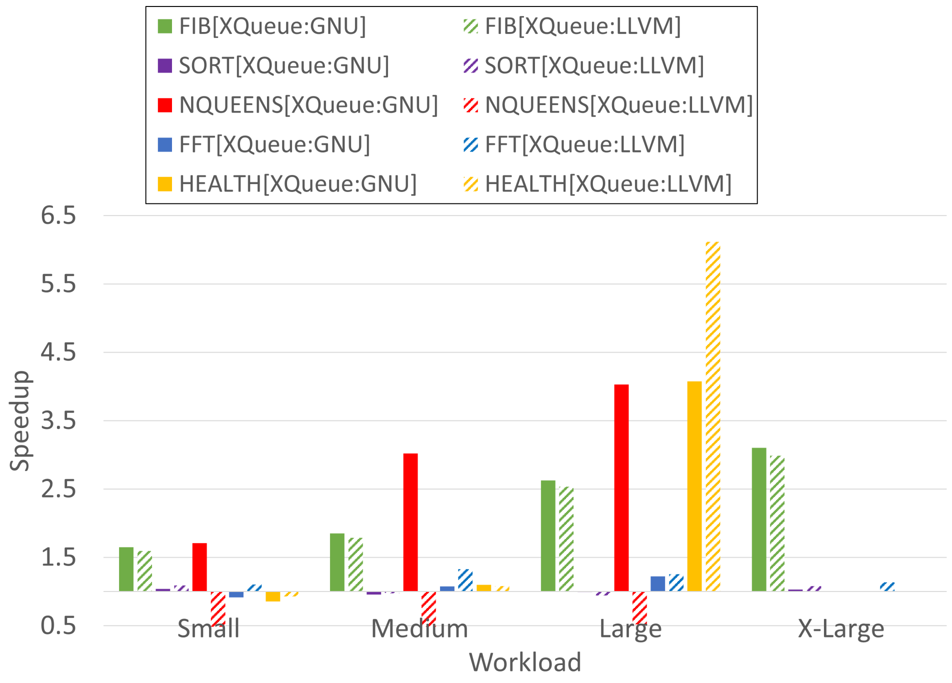
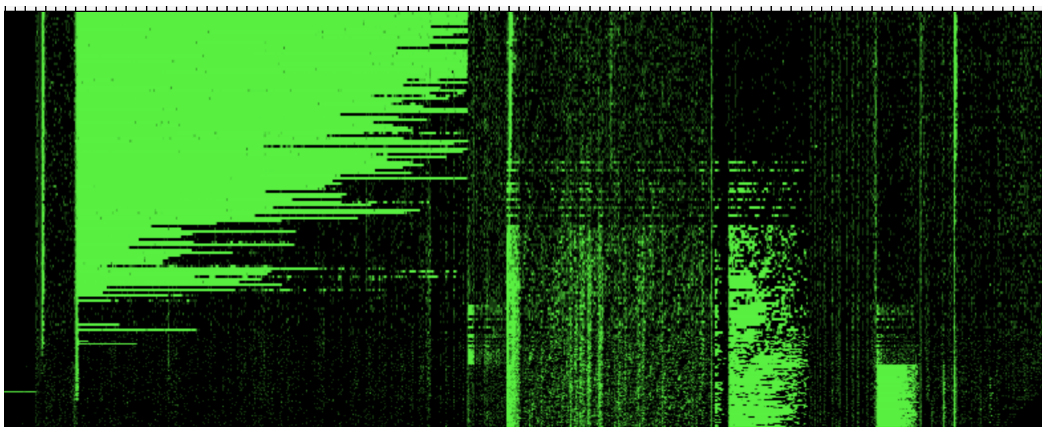
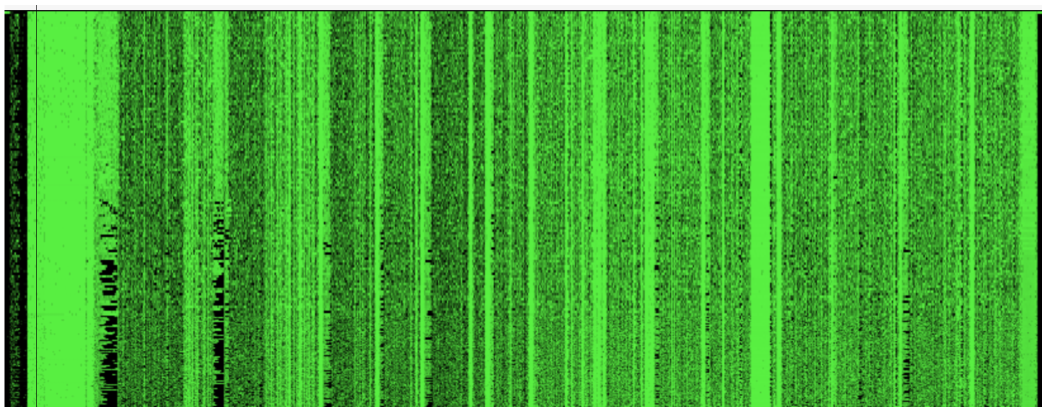


Figure 3.7. Speedup of XQueue over standard GNU and LLVM OpenMP implementations on the BOTS benchmarks on skylake-192 using 192 threads.

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 3.7 shows that the XQueue OpenMP 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.

Strassen's Matrix Multiplication uses Strassen's algorithm to multiply


(a) XQueue-enabled LLVM OpenMP


(b) Native LLVM OpenMP

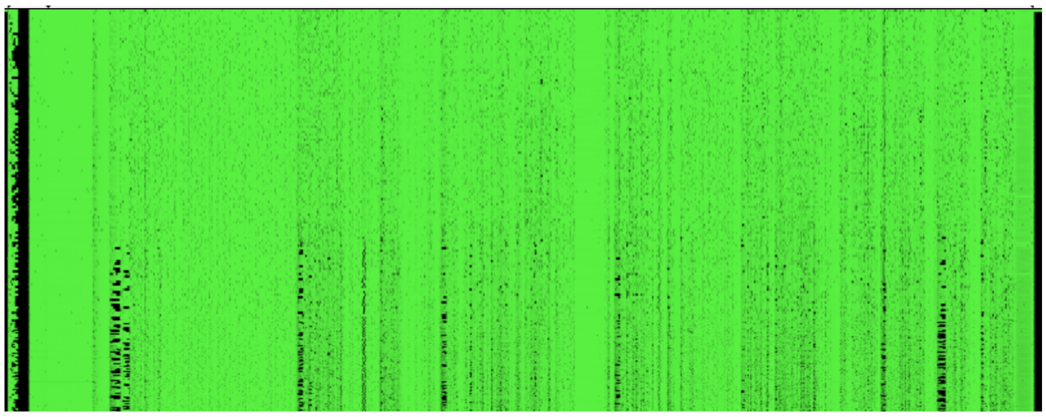

(c) GNU OpenMP

Figure 3.8. Load balance of FFT on skylake-192

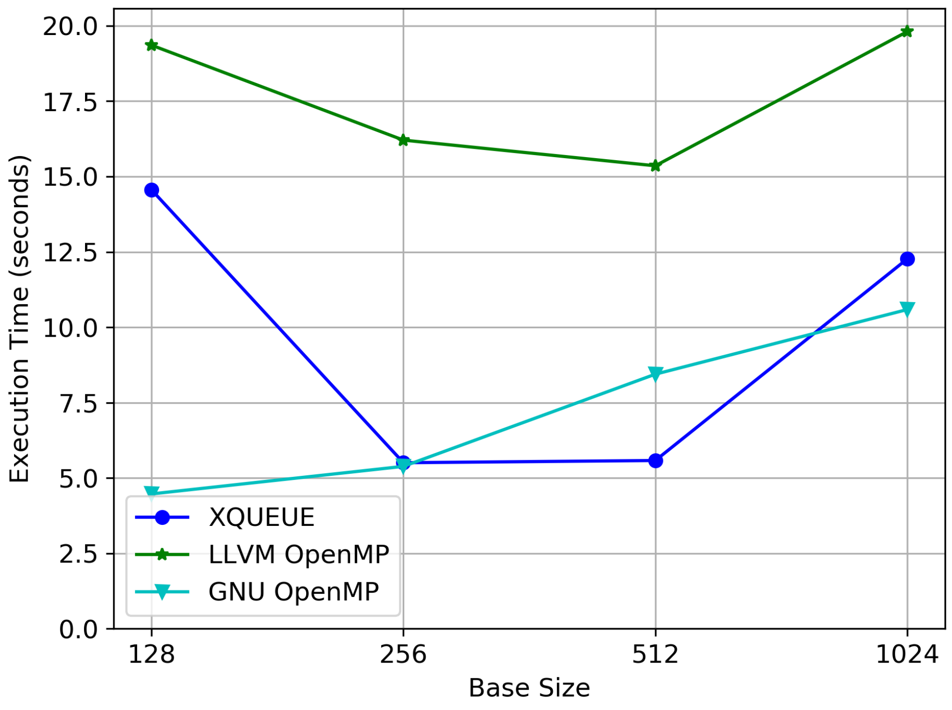


Figure 3.9. Performance of Strassen's Matrix Multiplication benchmark using 8K matrices on skylake-192 using 192 threads and varying tile sizes (lower is better).

two square matrices. It uses a recursive parallel divide-and-conquer approach where the matrix is divided into smaller and smaller matrices at every step. When the base condition is reached, matrix multiplication is computed using a sequential divide and conquer approach. It is faster than standard matrix multiplication for large matrices.

Figure 3.9 shows the execution time of strassen's algorithm using 8K matrix size on skylake-192 system with 192 threads and varying base sizes. LLVM's implementation is 40 to 60% slower than XQueue for all the base sizes. GNU's implementation achieves the best running time of 4.4 seconds with 128 base size. It is interesting to note that naive round-robin load balancing in XQueue is able to achieve significant speedup compared to LLVM at high concurrency level of 192 threads. These results clearly show the significance of low synchronization overheads to achieve better performance at high concurrency levels.

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 [57] 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 3.10 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× for Friendster and 3× 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× 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× 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× speedup using Friendster and 1.6× speedup using Twitter with

192 threads, showing promise that the task-based parallel approach can be beneficial for these types of workloads.

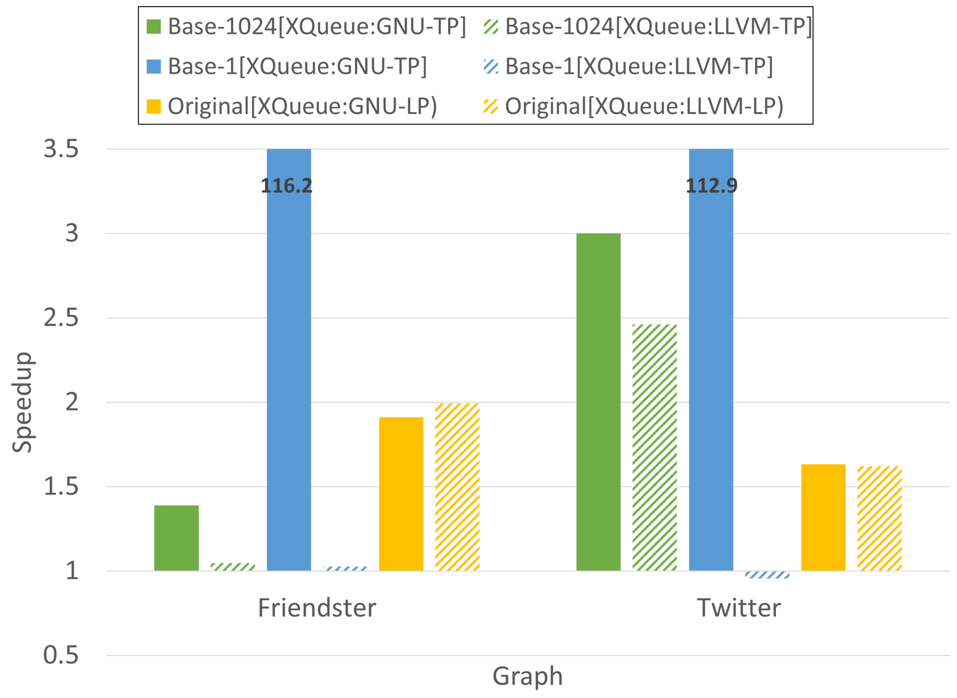


Figure 3.10. Speedup of XQueue over standard GNU and LLVM OpenMP implementations when applied to Breadth First Search from GAP Benchmark Suite on skylake-192 using 192 threads.

Gaussian Elimination (GE) algorithm has several applications in Linear Algebra and one of the most important applications is solving a linear system of equations. We have implemented the algorithm without pivoting. In prior work, we presented an analytical model for understanding the performance of this algorithm [59]. Figure 3.11 shows the results obtained by executing GE algorithm using 16K matrix size on skylake-192 machine using 192 threads pinned to cores. Best running time of this algorithm is achieved using XQueue and 128 base size. As analyzed in our prior work, these block sizes perfectly fit in L2 cache on the skylake machine and achieve best performance. With smaller base sizes, the number of tasks is much higher and due to the cost of creating and managing the tasks, the performance drops by about 10% for the base size of 64 and about 40% for 32 base size in

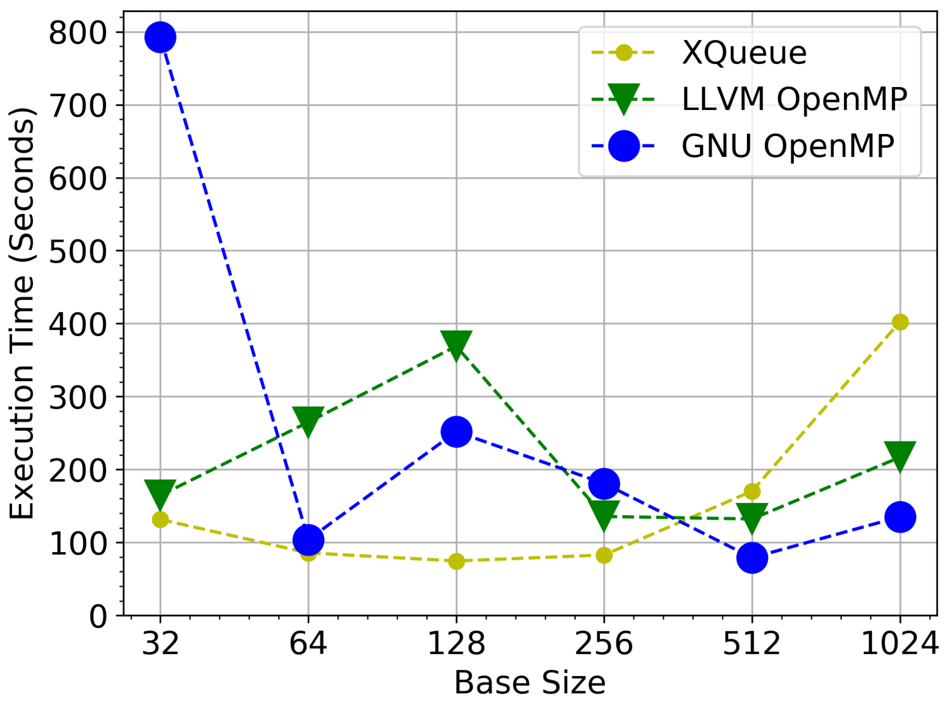


Figure 3.11. Performance of Gaussian Elimination algorithm using 16K matrix on skylake-192 using 192 threads and varying base sizes (lower is better).

case of XQueue. The performance at smaller base sizes is also impacted by the lack of dynamic load balancing in XQueue, however it is worth noting that XQueue achieves the fastest running time inspite of the static load balancing. GNU's implementation significantly loses performance with extremely fine-grained tasks such as at base size 32. This behavior is consistent with other benchmarks evaluated in this paper and is an area that can be improved.

Symmetric Rank Update is part of the Basic Linear Algebra Subprograms (BLAS) specification [60] and is another important building block of several linear algebra applications. For a given symmetric matrix, the algorithm computes updates the upperlower part of the result matrix with the matrix product. The algorithm is implemented as part of the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) numerical library [43]. PLASMA is a dense linear algebra library

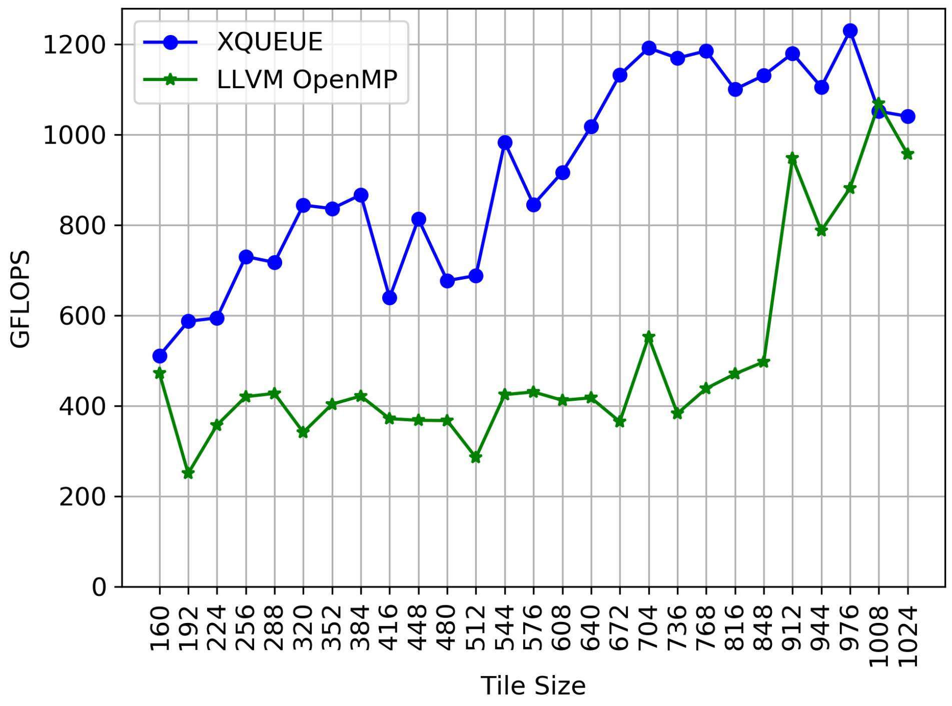


Figure 3.12. Performance of Symmetric Rank Update (DSYRK) on skylake-192 using 192 threads and varying tile sizes (lower is better).

which implements a full set of BLAS routines using task-based parallelism. The algorithm uses a tile-based approach where the matrix is divided into square blocks and each tile is typically processed by a task.

Figure 3.12 shows the performance achieved in GFLOPS by running the DSYRK kernel using 12K matrix and 96 threads on the skylake-192 server. This algorithm only scales up to 4 sockets. XQueue clearly achieves highest performance of 1230 GFLOPS with a tile size of 976 even with it's naive round-robin load balancing scheme. LLVM's implementation achieves best performance of 1000 GFLOPS with a tile size of 1008. These results shows evidence that higher performance can be achieved by using finer-grained tasks if the parallel runtime systems has low overheads for tasking.

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 presented in this paper. The results also demonstrate the need for exploring the opportunities for finer-grained parallelism in existing algorithms for achieving higher performance on modern machines at concurrency levels of hundreds of threads. Improving load balancing could yield further performance improvements similar in size to the improvements seen here.

3.3 Summary

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.

In our previous work, we explored various lock-based work stealing approaches [61]. Since XQueue is built using lock-less approach, we cannot use the traditional work stealing algorithms for dynamic load balancing. In the next part of this work, we investigate lock-less work stealing [62] as a scalable mechanism for dynamic load balancing with the aim to improve the current deterministic load balancing, broaden the applicability of XQueue, and achieve better performance on modern machines with hundreds of cores.

CHAPTER 4

X-OPENMP : EXTREME FINE-GRAINED TASKING USING LOCK-LESS WORK STEALING

In Chapter 3, we introduced XQueue [39], 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 [62]. We extend their work on load balancing along with our prior work on lock-less concurrent parallel framework [63] and propose a dynamic lock-less load balancing mechanism that can provide significant performance improvements using real application workloads.

The main contributions of this chapter are:

1. We introduce X-OpenMP library [64] and propose a work stealing algorithm

that enables lightweight tasking and dynamic load balancing using lock-less techniques.

2. We integrate our approach into LLVM's OpenMP implementation which allows existing applications written using OpenMP to leverage the lightweight tasking proposed in this work.

3. We evaluate X-OpenMP using micro benchmarks, numerical kernels and unbalanced trees and demonstrate significant performance improvements using our approach.

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 [48, 65]. 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 [66].

Figure 4.1 shows the timeline plot of the Unbalanced Tree Search (UTS) benchmark [67] 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)

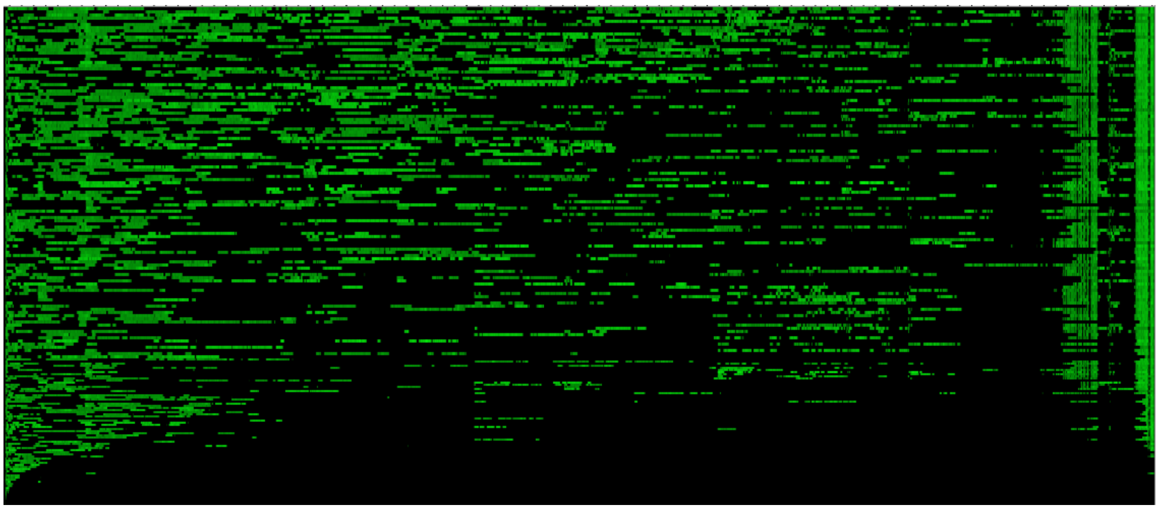


Figure 4.1. Load Imbalance in Unbalanced Tree Search using 192 threads and GNU OpenMP

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 [68].

Multi-threaded systems use synchronization mechanisms like mutexes, semaphores, spinlocks, or atomic operations [69, 70] 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 [71, 72, 73, 74, 75] 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 [76]. Lockless 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 [77]. 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 [64] 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 [16, 78, 17] 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) [79]. 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 [62] 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 [62]. 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 [62] is implemented for stealing threads and waits forever in the while loop until a steal succeeds or is invalidated. However, in the implementation of X-OpenMP, to ensure the application terminates

Algorithm 2: Work Stealing With Wait - Victim's Logic

```
Data: $local\_steal\_req \leftarrow thread{-}>steal\_req$;
Data: $round \leftarrow local\_steal\_req\ \&((1 << 40) - 1)$;
1 if $round == thread{-}>round$ then
2     $ret \leftarrow dequeue(thread\_id, item)$;
3     if $ret == SUCCESS$ then
4         $stealer\_id \leftarrow local\_steal\_req >> 40$;
5         $threads[stealer\_id]{-}>stolen\_task \leftarrow item$;
6     $thread{-}>round++$;
```

Algorithm 3: Work Stealing With Wait - Stealer's Logic

```
1 if $(victim{-}>steal\_req\&((1 << 40) - 1) < victim{-}>round)$ then
2     $round = victim{-}>round$;
3     $victim{-}>steal\_req = round + (thread\_id << 40)$;
4     while $round == victim{-}>round || thread{-}>stolen\_task \neq NULL$
        do
5         if $(victim{-}>steal\_req\&((1 << 40) - 1)) < round$ then
6             $victim{-}>steal\_req = round + (thread\_id << 40)$;
7         if $(thread{-}>stolen\_task \neq NULL)$ then
8             return $thread{-}>stolen\_task$;
9     return $NULL$;
```

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 [80]. 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 its 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

```
Data: $local\_steal\_req \leftarrow thread{-}>steal\_req$;
Data: $round \leftarrow local\_steal\_req \& ((1ULL << 40) - 1)$;
1 if $round == thread{-}>round$ then
2     $ret \leftarrow dequeue(thread\_id, item)$;
3     if $ret == SUCCESS$ then
4         $stealer\_id \leftarrow local\_steal\_req >> 40$;
5         $threads[stealer\_id]{-}>enqueue(item)$;
6     end
7     $thread{-}>round{+}{+}$;
8 end
```

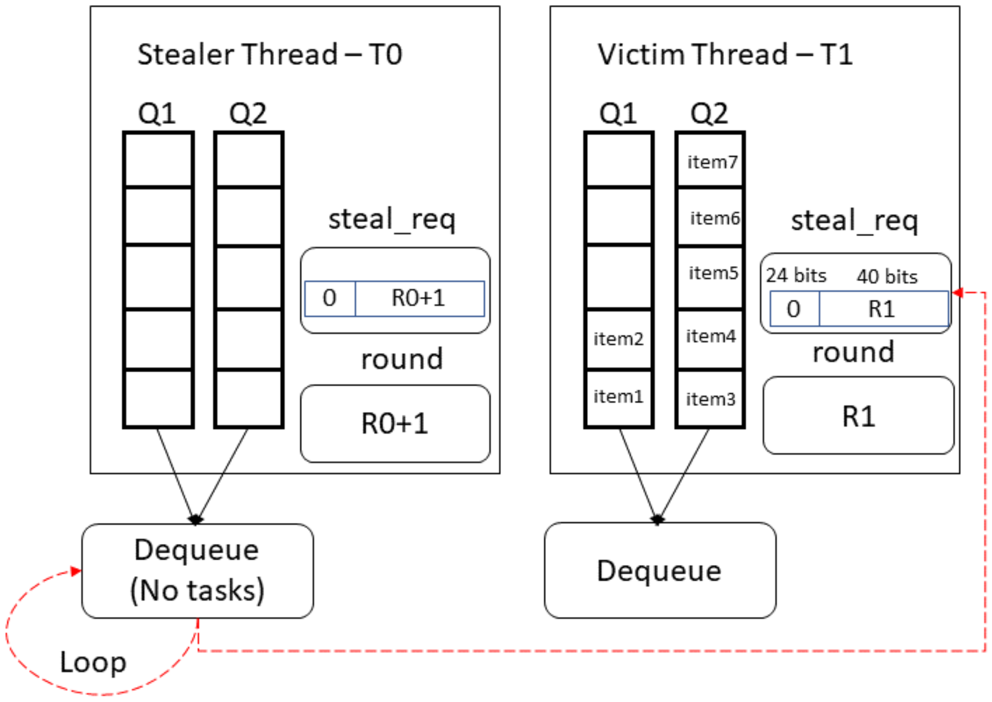
It is worth noting that the wait-free work stealing algorithm results in many more steal requests being submitted than the wait-based approach, thereby resulting in more successful steals and better load balancing in terms of the number of tasks. A significant difference between the traditional work stealing approach and the lock-less

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

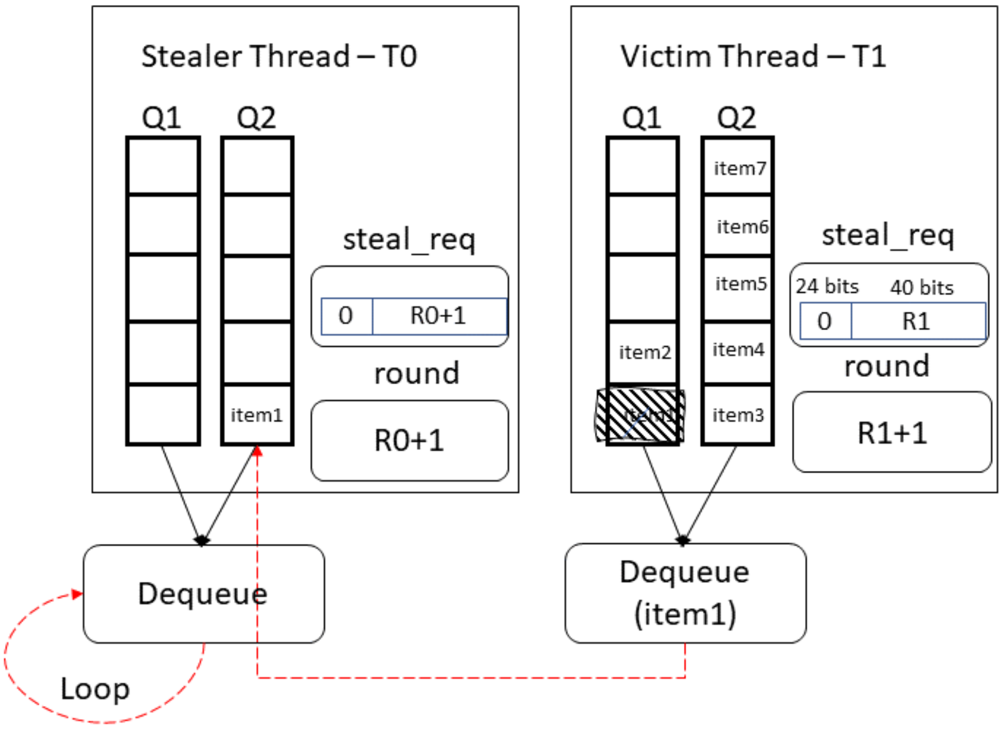
```
1 if ($victim- > steal\_req \& ((1 << 40) - 1) < victim- > round$) then
2     $round = victim- > round$;
3     $victim- > steal\_req = round + (thread\_id << 40)$;
4     return NULL;
5 end
```

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 4.2. 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 4.2-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.



(a) A



(b) B

Figure 4.2. 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

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.

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 [77]. 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 Performance 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 [56], cholesky factorization and symmetric rank-k update routines from the PLASMA linear algebra library [43] and the Unbalanced Tree Search benchmark [67]. 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 [38]. We compiled all the benchmarks using LLVM Clang version 11.0 and O3 optimization level and ran experiments on Ubuntu 20.04.4.

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 [81]. 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

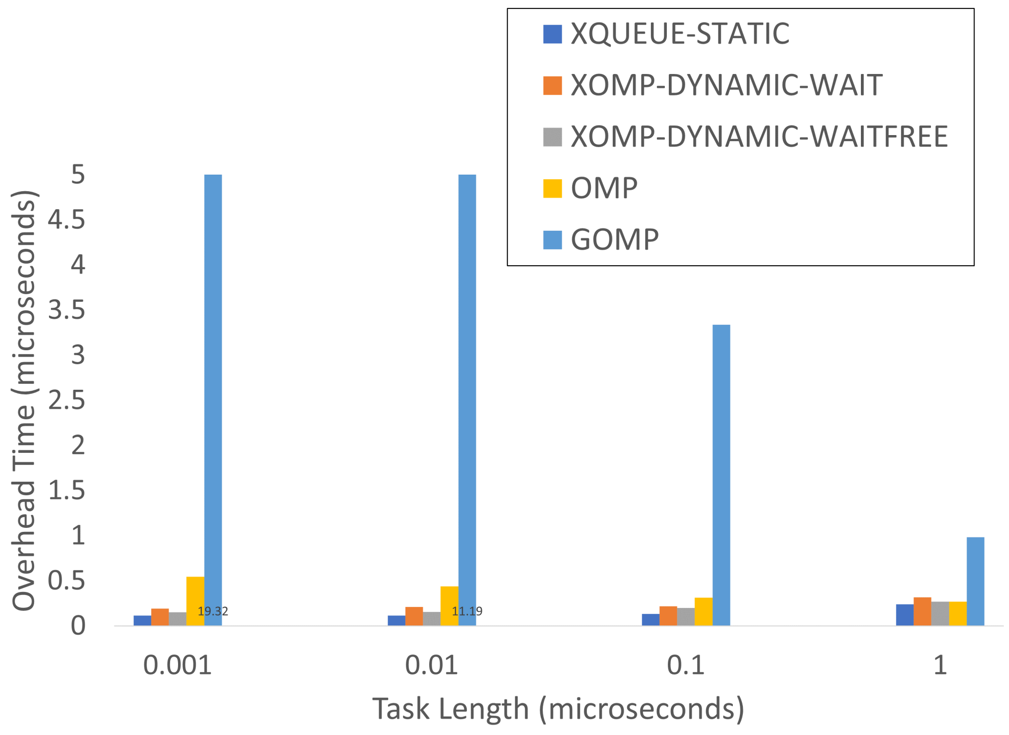


Figure 4.3. Parallel Tasking Overhead on skylake-192 using 192 threads (lower is better)

master worker receives all the tasks.

Figure 4.3 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 0.001 to 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

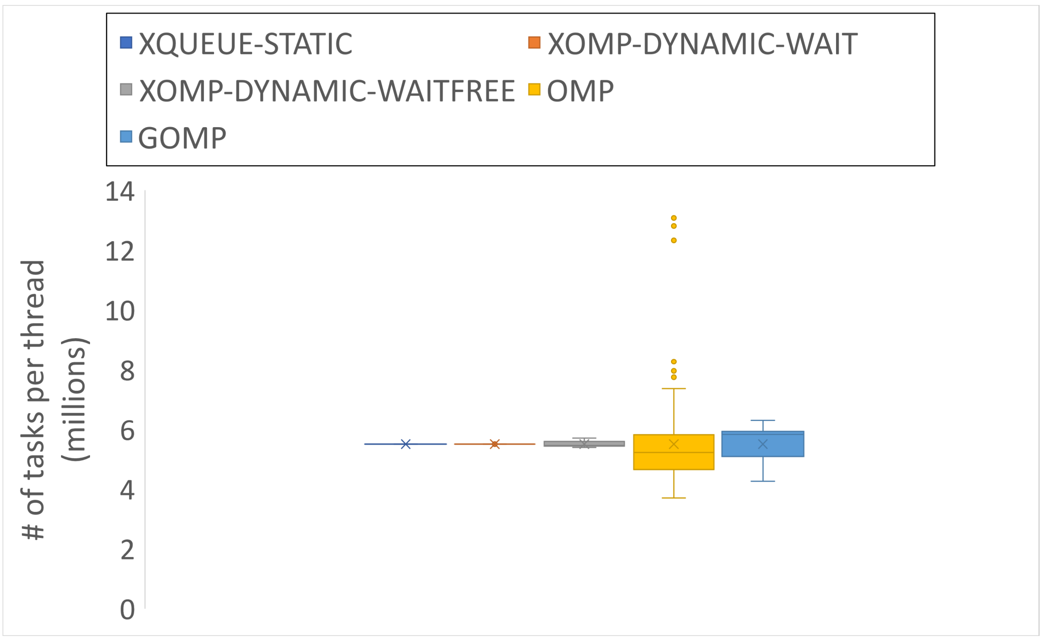


Figure 4.4. Task Distribution on skylake-192 using 192 threads

concurrent queuing mechanisms.

Figure 4.4 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 no-wait 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 runs 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.

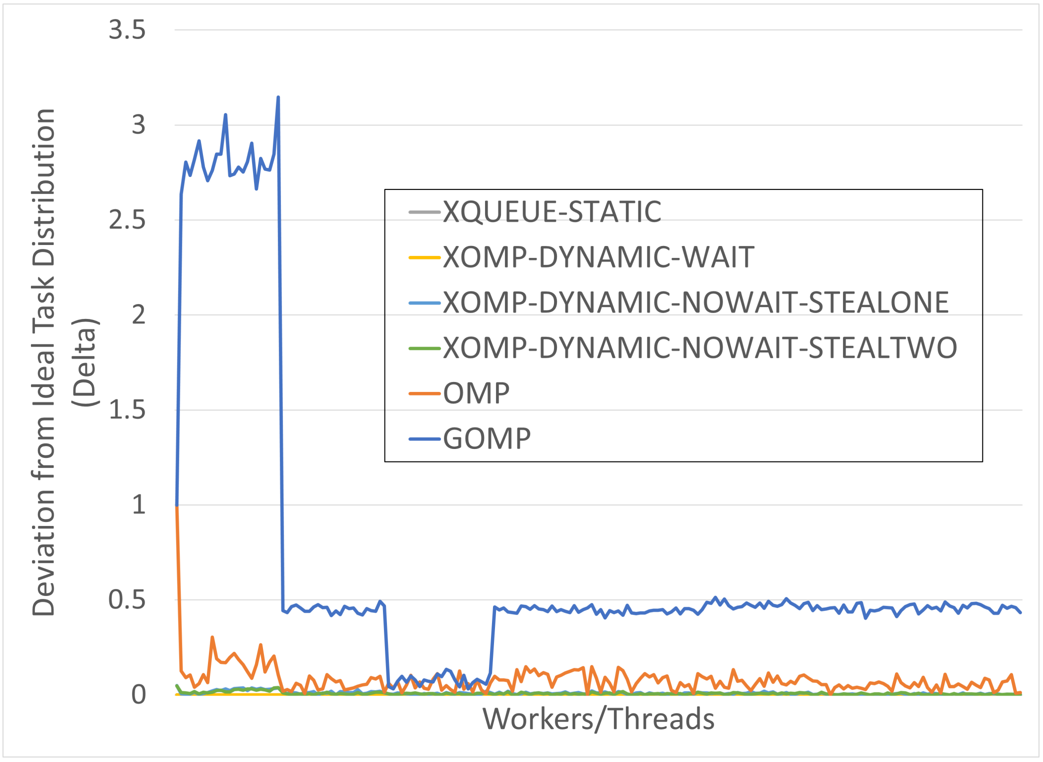


Figure 4.5. Delta of Task Distribution using Work Stealing on skylake-192 using 192 threads (lower is better)

Figure 4.5 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 deviation from the ideal case (delta) of each worker based on the task distribution across all the runs. The delta metric of each worker is calculated using the formula:

$$Delta_{worker_i} = \frac{|Tasks_{worker_i} - Tasks_{ideal}|}{Tasks_{ideal}}$$

The ideal case is when every worker runs an equal number of tasks which implies the delta is zero. The delta for all versions of X-OpenMP is very close to zero and for the native LLVM OpenMP version, the delta ranges between 0.0005 and 0.99. 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 X-OpenMP versions. The main takeaway from this benchmark is that lock-less implementations of work stealing perform similar to traditional work stealing implementations.

While the above microbenchmarks use homogeneous tasks which run for the same amount of time, real applications typically have a mix of fine-grained and coarse-grained tasks. We designed a benchmark to explore this scenario. In this benchmark, the master worker receives $N \times 10$ tasks (where N is the number of workers) which run for 1 second each. The ideal time to execute these tasks with perfect load balancing is 10 seconds. Also, every worker receives 8 million delay tasks of length 0.1 microseconds, which should ideally execute within a second. The idea behind this synthetic benchmark is to simulate starvation where load imbalance exists between workers and some workers can become idle for long periods of time. In case of X-OpenMP, steal requests need to be handled by busy workers to facilitate work stealing and this benchmark is designed to understand how the runtime behaves. Figure 3.5 shows the results obtained by running this benchmark. The plot shows execution time by varying queue sizes in the runtime implementations to understand the impact of queue size when a single worker is overloaded with work. Please note that the queue sizes couldn't be matched for X-OpenMP and LLVM implementations since the latter requires powers of two for queue length. The execution time of this synthetic benchmark for LLVM OpenMP shows that the performance of the runtime is sensitive to queue size in such scenarios and starvation can occur. XQueue and X-OpenMP can handle such load imbalances well due to the round-robin load balancing where long-running tasks get distributed to all the workers. This experiment also clearly shows that X-OpenMP is not sensitive to the size of the queue due to the existence of multiple queues per worker. The queue sizes can be kept small, thereby requiring less memory for the underlying runtime and achieve good performance.

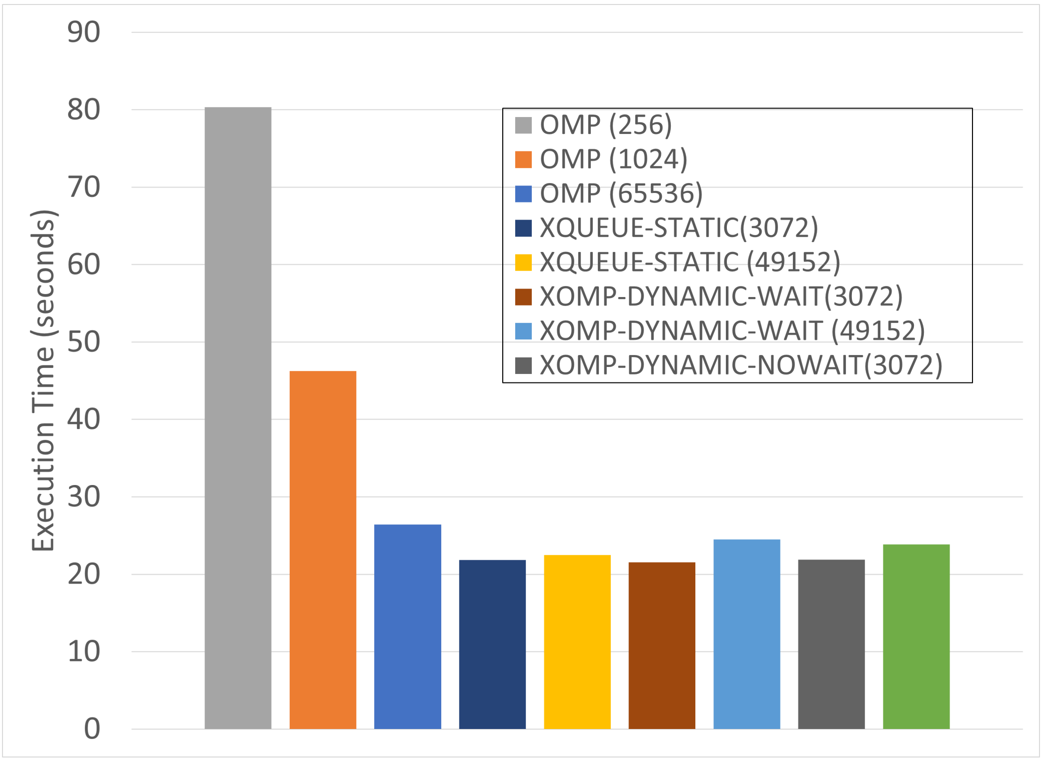


Figure 4.6. Execution Time (using 192 threads) of a synthetic benchmark with a mix of fine-grained and coarse-grained tasks and varying queue sizes (legend shows queue sizes in parentheses).

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 [**56**, **82**] 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

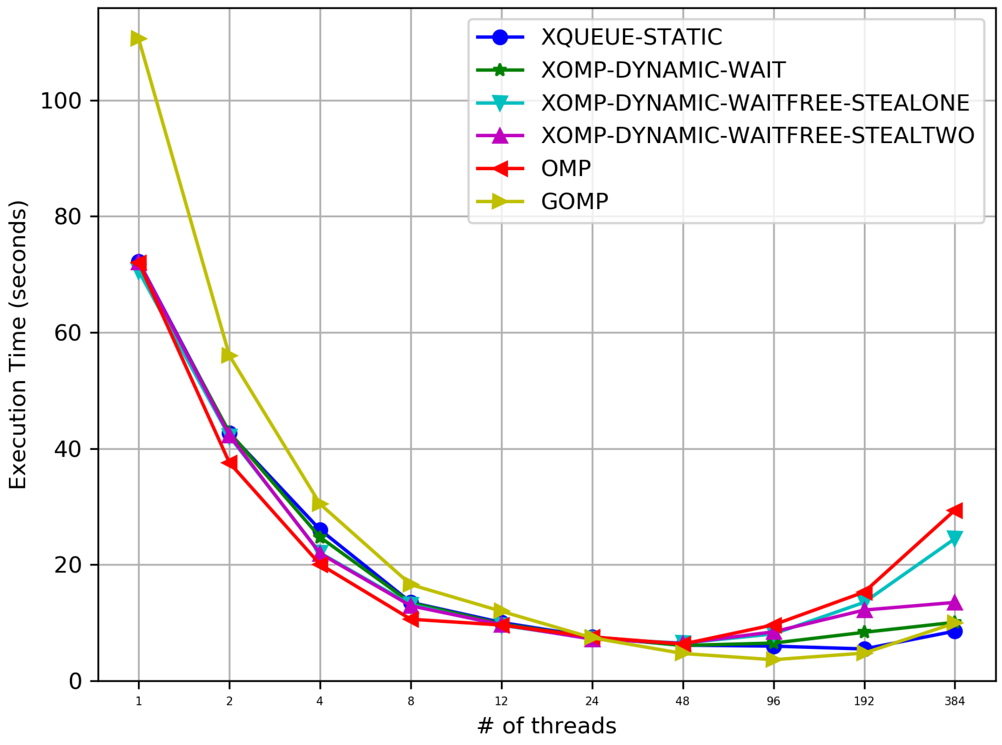


Figure 4.7. Scaling of Strassen's Matrix Multiplication using 8K matrix on skylake-192 (lower is better)

approach. The depth based cutoff value for divide and conquer algorithm is set to 3.

Figure 4.7 shows the scalability plot for Strassen's matrix multiplication algorithm. The experiment multiplies square matrices of size 8192x8192 using 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 4.8 shows the results obtained by running Strassen's algorithm on an

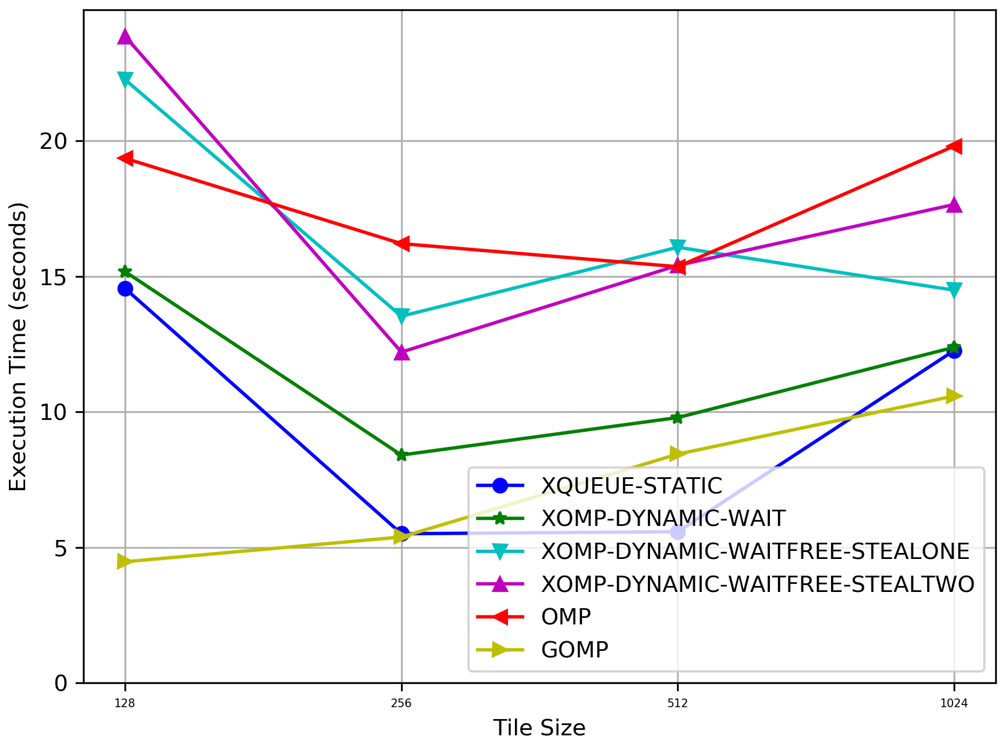


Figure 4.8. Performance of Strassen's Matrix Multiplication using 8K matrix and varying base sizes on skylake-192 (lower is better)

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 synchronization 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) [83] is an important building block of many linear algebra algorithms and included in the Basic Linear Algebra Subprograms (BLAS) specification [60]. 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 [43] 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.

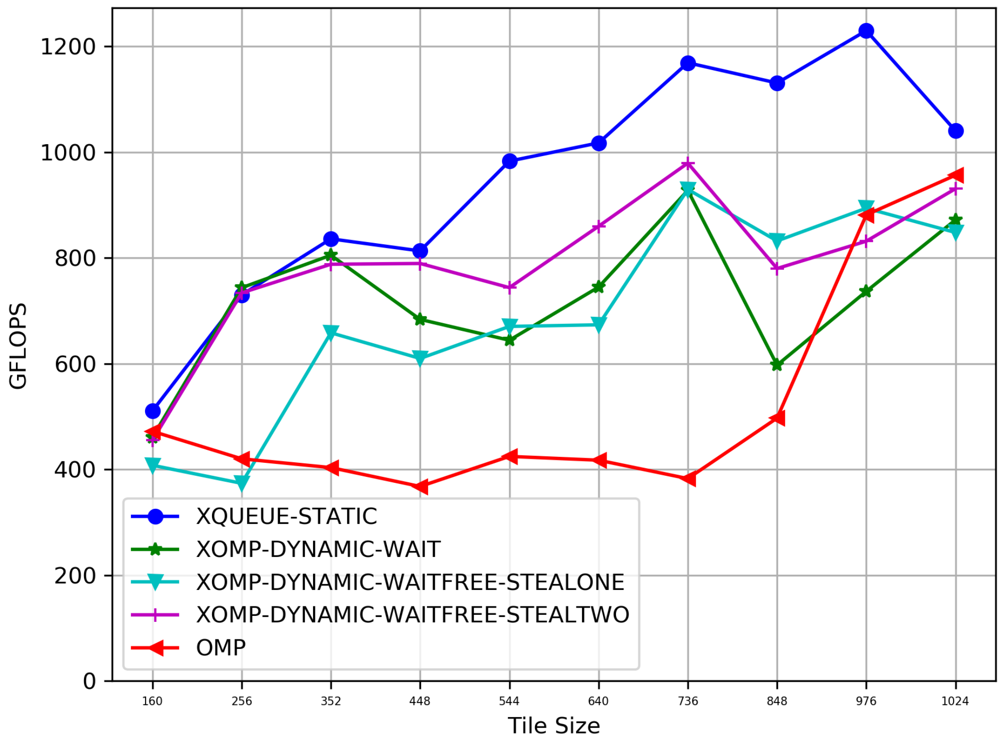


Figure 4.9. Symmetric Rank Update using 12K matrix on skylake-192 using 96 threads (higher is better)

Figure 4.9 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 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 [4] 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 4.10 shows the performance of Cholesky Factorization algorithm on skylake-192 server using 12K matrix, 96 threads and varying tile sizes. The 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

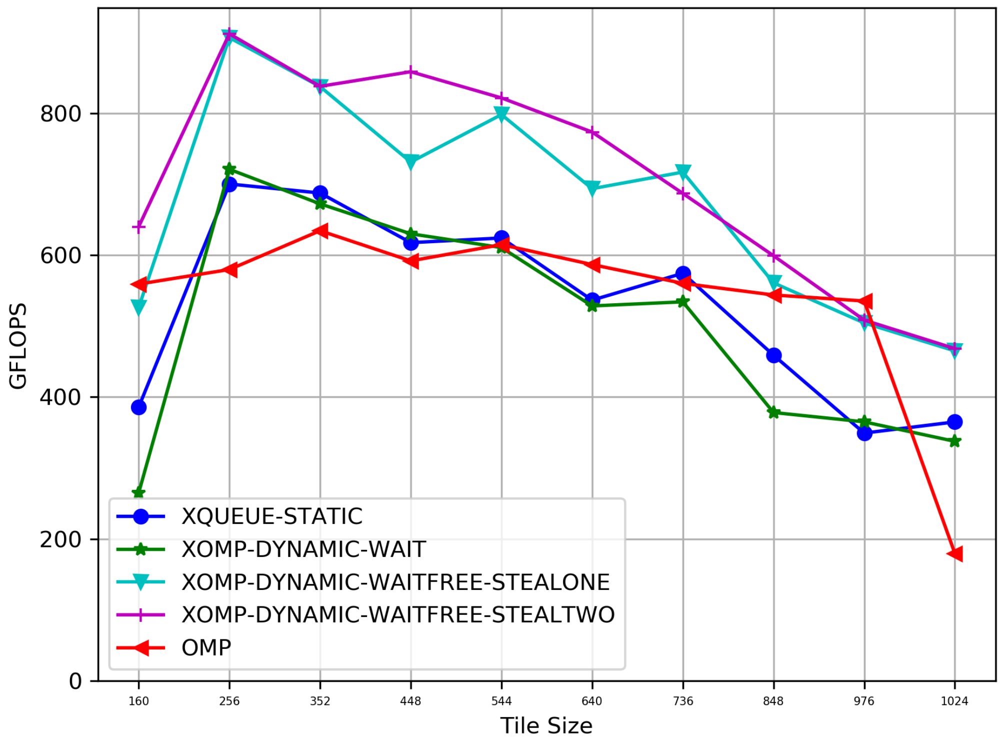


Figure 4.10. Cholesky Factorization on a 12K matrix on skylake-192 using 96 threads (higher is better)

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 4.11 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 implementation 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

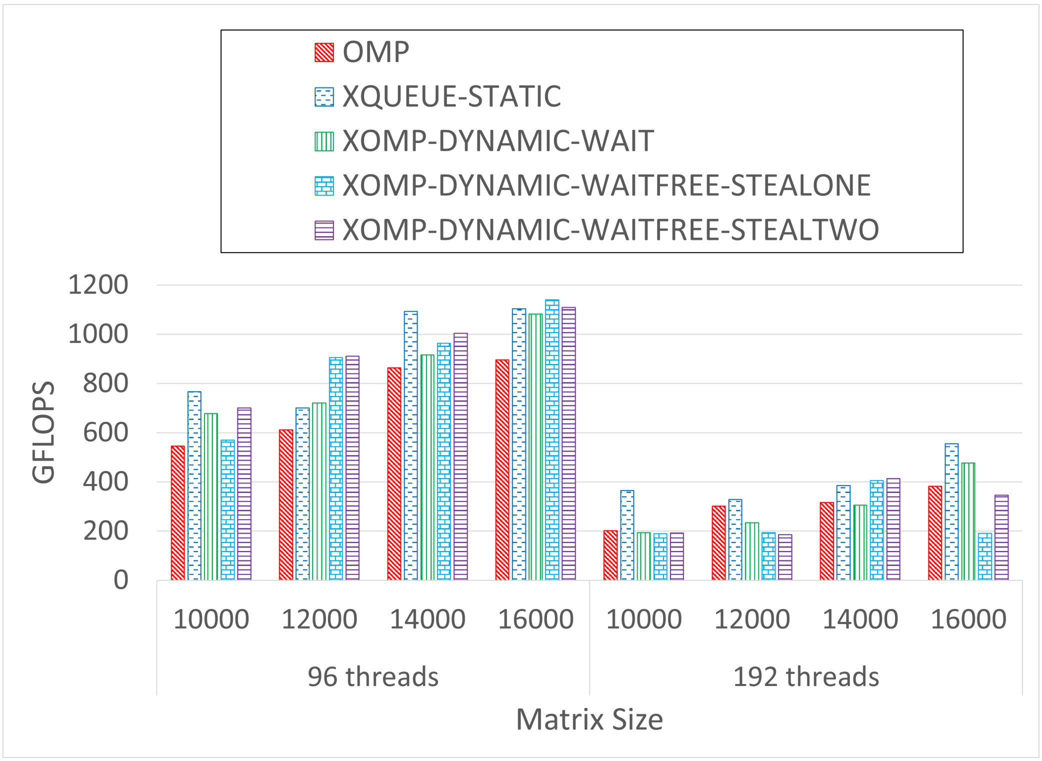


Figure 4.11. Cholesky Factorization using different matrix sizes and tile size of 256 on skylake-192 run using 96 and 192 threads (higher is better)

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) [67] 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.

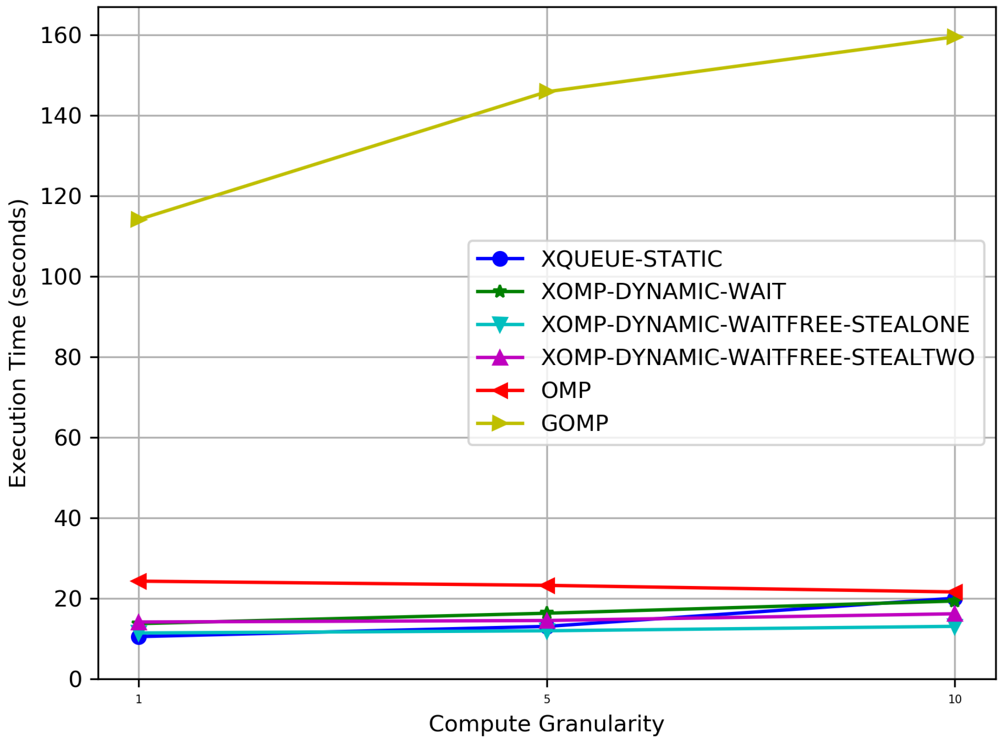


Figure 4.12. Unbalanced Tree Search using 96 threads on skylake-192 (lower is better)

Figures 4.12 and 4.13 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

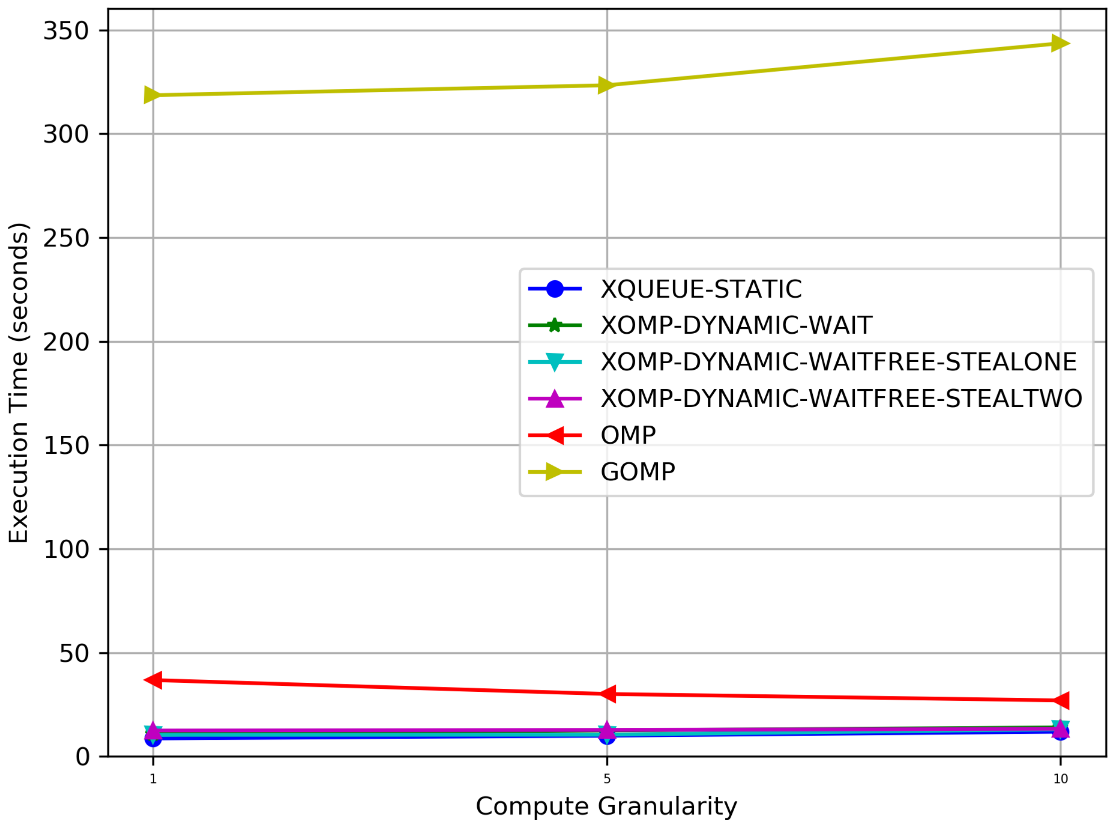


Figure 4.13. Unbalanced Tree Search using 192 threads on skylake-192 (lower is better)

this workload with about 40X slowdown across all granularities.

Figure 4.14 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 significance 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

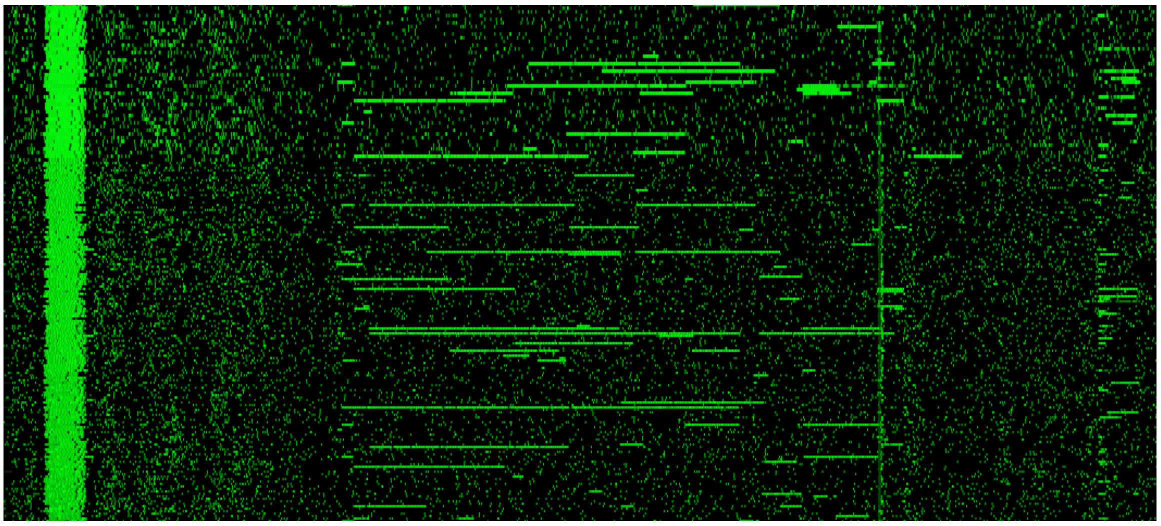


Figure 4.14. Unbalanced Tree Search using X-OpenMP and 192 threads on skylake-192

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. 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 synchronization 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 Summary

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

EFFICIENT EXECUTION OF DYNAMIC PROGRAMS USING DATA-FLOW BASED PARALLEL PARADIGM

While our prior work explored light-weight tasking mechanisms in fork-join based parallel runtime systems, the fork-join model imposes limitations for expressing parallelism in certain scenarios. We explore the limitations of fork-join based parallelism by using two-way recursive divide-and-conquer algorithms as compared to data-flow based parallelism. On shared-memory multicore machines, classic two-way recursive divide-and-conquer algorithms are implemented using common fork-join based parallel programming paradigms such as Intel Cilk+ or OpenMP. However, in such parallel paradigms the use of joins for synchronization may lead to artificial dependencies among function calls which are not implied by the underlying DP recurrence. These artificial dependencies can increase the span asymptotically, and thus reduce parallelism. From a practical perspective, they can lead to resource under-utilization, i.e., threads becoming idle. To eliminate such artificial dependencies, task-based runtime systems and data-flow parallel paradigms (e.g., Concurrent Collections (CnC), PaRSEC, and Legion) have been introduced. Such parallel paradigms and runtime systems overcome the limitations of fork-join parallelism by specifying data dependencies at finer granularity and allowing tasks to execute *as soon as* dependencies are satisfied. In this work [59], we investigate how the performance of data-flow implementations of recursive divide-and-conquer based DP algorithms compare with fork-join implementations. We have designed and implemented data-flow versions of DP algorithms in Intel CnC and compared the performance with fork-join based implementations OpenMP. Our experimental results show that the data-flow based implementations of classic two-way DP algorithms provide a competing performance in comparison with the corresponding fork-join implementations.

Dynamic Programming (DP) is an algorithm design technique that recursively decomposes a problem into smaller overlapping subproblems. It solves each unique overlapping subproblem exactly once and stores its result into the memory (DP table) for further reuse. Theoretically and practically, DP improves the performance of a recursive solution by preventing solving the repeating subproblems when they are encountered later [84, 85, 86]. DP algorithms can be viewed as trading off space-efficiency for reduced computation time [85]. DP is considered as one of the building blocks in solving a variety of combinatorial optimization problems [87]. It has numerous applications in different research and engineering areas, including computational biology [88], molecular modeling [89], etc.

The most common approach to implement DP algorithms is to use a loop-based program that populates the results into the underlying DP table cells iteratively. The recurrence relation of the DP specification enforces the correct ordering of storage and retrieval of the results of the subproblems. Such implementations often have good spatial locality and prefetching optimizations can be applied to gain further performance. However, they do not perform efficiently due to the lack of temporal locality. As a result, to overcome the shortcomings of the loop-based DP algorithms, researchers proposed tiled or blocked algorithms [90, 91, 92, 93] as well as standard 2-way recursive divide-&-conquer algorithms [94, 95]. Recursive divide-&-conquer DP algorithms are, unlike the tiled programs, cache oblivious [96, 95] and cache adaptive [97, 94]. Because of the heterogeneous nature of many modern supercomputers, standard 2-way (or any fixed r-way) recursive divide-&-conquer algorithms may suffer from the lack of performance portability and performance scalability on such supercomputers. Such important limitations led to the introduction and development of parametric r-way recursive divide-&-conquer DP algorithms (r-way $\mathcal{R}$-DP) to run efficiency on different architectures such as GPUs and distributed-memory parallel machines [98, 99, 100, 101, 102]

5.1 Motivation

On shared-memory multicore machines, classic 2-way algorithms have been implemented by fork-join based parallel programming paradigms such as Intel Cilk+ or OpenMP. However, in such parallel paradigms the use of joins for synchronization may create artificial dependencies among function calls which are not implied by the underlying DP recurrence. These artificial dependencies can increase the span asymptotically, and thus reduce parallelism [103, 104]. From a practical perspective, they can lead to resource underutilization, i.e., threads becoming idle. Due to such an important limitation, several researchers introduced task-based runtimes [105, 106, 107, 108, 109] and data-flow parallel paradigms [110, 111, 112, 113]. Such runtimes and paradigms which follow the data-flow model of execution and point-to-point synchronization, overcome the limitations of fork-join parallel paradigms. Data dependencies between tasks can be specified at finer granularity and tasks can execute as soon as the data becomes available (i.e., when dependencies are satisfied). In this chapter, we investigate the application and efficiency of running DP algorithms on Intel Concurrent Collections (CnC) [114] which is one of the pioneering implementations of the data-flow based parallel paradigm. We compare the results with implementations in OpenMP. Considering different execution parameters (e.g., algorithmic properties such as recursive base size as well as machine configuration such as the number of physical cores, etc), we explain in what scenarios data-flow based implementation outperforms the fork-join based implementation. Considering Gaussian Elimination without pivoting (GE) algorithm, we provide an analytical model approximating the execution time of a DP computation. To summarize, followings are the **key contributions** of this work:

- By summarizing some of the important differences of fork-join based and data-flow based parallel paradigms, we explain how a standard 2-way recursive divide-

&-conquer DP (2-way $\mathcal{R}$-DP) algorithm is specified and developed in OpenMP and Intel CnC. We explain how the CnC runtime executes the program [59].

- We explain how the use of joins for synchronizations in the fork-join (OpenMP) implementations of $\mathcal{R}$-DP algorithms introduces artificial dependencies which leads to increase in span, reduction in parallelism and resource underutilization. We explain how data-flow implementation can resolve the issue.
- We design, implement and analyze three important DP benchmarks in OpenMP and Intel CnC: Gaussian Elimination without Pivoting, Smith-Waterman Local Alignment, and Floyd Warshall's All Pairs Shortest Path. We summarized the lessons learned from the experiments. We compared the experimental results and explained in what scenarios, each of the parallel paradigms outperform the other.
- Due to the importance of data movement cost in the memory hierarchy, in order to understand it better, for GE benchmark, we design an analytical model which correctly predicts the trend in data movement cost obtained from experimental results. The model can be easily extended to the other DP algorithms.

This chapter is organized as follows. Sec. 5.2 provides a background on CnC model. Using the GE algorithm as a running example, Sec. 5.3 explains fork-join based implementation (in OpenMP) as well as data-flow based implementation (in Intel CnC) of the recursive divide-&-conquer DP algorithms. Experimental results are provided in Sec. 5.4. This section explains under what circumstances data-flow based implementation outperforms the fork-join based implementation and vice versa. Sec. 5.5 concludes the chapter by summarizing the key points and mentioning the future work.

5.2 Background

Concurrent Collections (CnC) [111] is a data-flow based parallel programming model (originated from TStreams [115]). Different forms of parallelism, (including task, data, loop, pipeline and tree) can be expressed using this model. The important aspect of CnC is the idea of separation of concerns between application logic and parallel implementation. A CnC program/specification can be viewed as a communication means (or an interface) between the *domain expert*[1] and the *tuning expert*[2]. This separation of concerns simplifies the task of the domain expert, as writing a program in this language does not require any reasoning about parallelism or any knowledge of a target architecture [116]. The domain expert does not specify how operations are scheduled. The tuning expert (who can also be the domain expert) does not need to have an understanding of the domain (e.g., physics, chemistry, etc). S/he maps the CnC specification to a specific target architecture to be executed efficiently. Knobe and Burke [117] have introduced a tuning language within the CnC paradigm. Through the concept of affinity groups[3], the tuning language enables the tuning expert to map the implicit parallelism in the CnC domain specification on a target platform.

The three main CnC concepts are step collections, item collections (or data collections) and tag collections (or control collections). The CnC program is specified as a graph of collections, communicating with one another. More precisely, a CnC specification is a graph whose nodes are either step collections, item collections, or tag

[1]Whose interests and expertise in the application domain (e.g., finance, genomics, numerical analysis, etc) who does not necessarily have expertise in parallel programming and performance tuning.

[2]Whose interests and expertise are in performance and parallel programming.

[3]An affinity group is a set of computation that the tuner recommends to be executed closely (in time and space).

```
/* tag collection myCtrl prescribes step collection
  myStep */
<myCtrl >::(myStep );
/* step collection myStep consumes items from item
  collection myData, produces item to myData and
  puts tags into the tag collection myCtrl */
[myData] --> (myStep) --> [myData],<myCtrl>;
```

Figure 5.1. Simple CnC Specification

collections, and the edges among them represent *producer*, *consumer*, and *prescription* dependencies. These edges enforce the partial order among the operations [116]. The relationships among the graph components are specified *statically* but during the execution, for each static collection, a set of *dynamic* instances are generated. A step collection corresponds to a specific computation (specified by the domain expert). A tag collection is the main concept for control flow of the program. Each tag collection is prescribed to a step collection, which means that putting tags into a tag collection will cause CnC runtime to generate an instance of the corresponding step collection, which will *eventually* execute with that tag instance as an input. Step collections dynamically read and write data through putting/getting items into/from item collections. From this perspective, the item collection can be considered as a placeholder for intermediate (or final) results produced and consumed by instances of the step collections. Figure 5.1 shows a simple example of a CnC specification [111].

In the listing 5.1, paired parentheses represent step collections, paired square brackets represent item collections and finally tag collections are represented by paired angled brackets.

A graphical representation of the above program is shown in Fig. 5.2. In this representation, ovals represent step collections, rectangles represent item collections and Hexagons represent tag collections. The term *env* in the figure is the environment, which is the world *outside of the CnC program* and can be other threads or processes. They can put item(s) into (input) item collections and also trigger the computation by putting tag(s) into tag collections.

An instance of collections is identified by a unique tag. Item collections are an associative container that are indexed by the unique tags. The tags usually have meaning within the application, e.g., in a 2-D tiled computation, the tag $\langle i, j \rangle$ can represent the coordinates of a tile. CnC preserves the *dynamic single assignment*[4] property which is used in the proof of determinism of CnC programs[5] [111]. The C++ implementation performs dynamic (run-time) checks to ensure that the execution adheres to the single assignment rule. Budimlić et al. have argued that CnC programs are Turing Complete [111] though they do not claim the absence of deadlocks. However, due to the deterministic property of CnC, deadlocks are straightforward to identify and fix.

CnC has different implementations in different languages including C++, Java, .NET, and Haskell. Since we use C++ implementations for our benchmarks, our focus will be on the C/C++ implementation of CnC which has two variations [116]. One is the X10-based [118] implementation from the Habanero project at Rice University. The other one is Intel's Concurrent Collections [114] which uses Intel's Threading Building Blocks (TBB).There are several advantages in using TBB. (1) through TBB tasks, fine-grained parallelism is supported[6]. (2) TBB contains a rich set of efficient concurrent containers, including vectors and hash-maps. More importantly, (3) TBB provides a scalable concurrent memory allocator [119]. High performance memory

[4]Due to this property, the item with index (or tag) i, once written, cannot be overwritten.

[5]As long as step collections themselves are deterministic.

[6]TBB tasks are user-level function objects which are scheduled by TBB's work-stealing scheduler.

allocation is considered as one of the important bottlenecks in parallel computing.

The CnC implementation on TBB uses an object-oriented design methodology [116]. The runtime provides class definitions for the three collections (`TagCollection`, `StepCollection`, and `ItemCollection`). A CnC graph contains objects that represent the step collections, item collections, tag collections and their relationships. A user-defined C++ functor represents a step collection. When a tag is put in a tag collection, an instance of the prescribed step collection is created and mapped to a TBB task. The TBB task can be spawned immediately upon prescription or delayed until all the data dependencies are satisfied. Thus, the `Get` operation on an item collection can be blocking or non-blocking. In case of a blocking `Get`, if the item to be retrieved is not ready, the step collection instance is aborted and put in a separate list associated with the failed `Get` to be re-executed later. When an item becomes available, all the steps in the list, waiting for that item, get triggered to be executed. Data items are created and retrieved by calling the `Put` and `Get` methods of `ItemCollection`. The items are maintained in a TBB concurrent hash-map which are accessed by indices/tags.

5.3 Classic 2-way $\mathcal{R}$-DP Algorithms: Fork-Join based and Data-flow based Implementations

5.3.1 Overview. In this section, we explain two different implementations of the classic 2-way recursive divide-&-conquer DP algorithms (2-way $\mathcal{R}$-DP) [95, 94]: fork-join implementation in OpenMP and data-flow implementation in Intel CnC. In this section, we explain how data-flow implementation eliminates the artificial dependencies which exist in the corresponding fork-join implementation. Elimination of such artificial dependencies leads to having finer-grained barriers in the execution of the algorithm. In Section 5.4, we discuss under what scenarios, having finer-grained barriers in the data-flow implementation can lead to a better parallelism and more

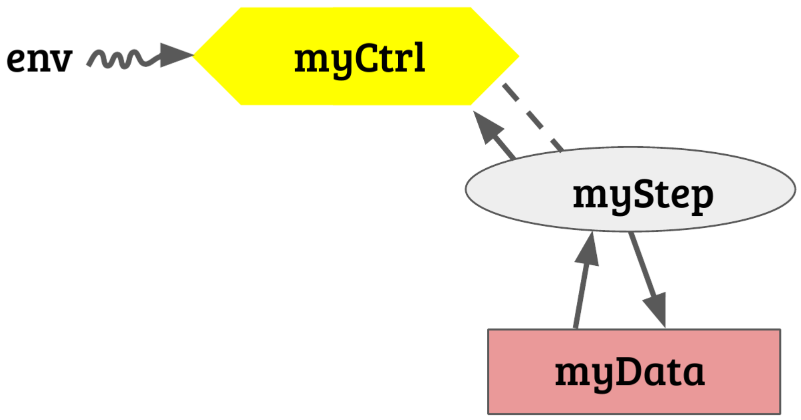


Figure 5.2. A graphical representation of the CnC program.

```
void I_GE(double **C,int N)
 for(k=0; k < N-1; ++k)
 for(i=0; i < N; ++i)
 for(j=0; j < N; ++j)
  if (i>k && j>=k) C[i][j]-=(C[i][k]*C[k][j])/C[k][k];
```

Listing 4. Loop-based serial implementation of GE

efficiency over the corresponding fork-join implementation.

For the rest of the chapter, we use Gaussian Elimination without pivoting (GE) [86] as a running example which has DP-like structure. The GE algorithm has applications in Linear Algebra. It solves systems of linear equations and performs LU decomposition of symmetric positive-definite or diagonally dominant real matrices. A system of $(n-1)$ linear equations with $(n-1)$ unknowns $(x_1, x_2, ..., x_{n-1})$ is represented by a $(n \times n)$ matrix C. In such a matrix, the rth row represents the equation $\sum_{j=0}^{n-1}(C[r,j] \times x_j) = C[r,n]$. Listing 4 shows the loop-based serial implementation of the GE algorithm.

5.3.2 Fork-Join based Implementation of $\mathcal{R}$-DP.

Due to poor temporal locality of the loop-based implementation, which leads to poor I/O efficiency, researchers have introduced recursive divide-&-conquer algorithms (2-way $\mathcal{R}$-DP). Such algorithms are theoretically and experimentally proven to be I/O efficient [94, 95]. Fig. 5.3 shows part of the classic 2-way $\mathcal{R}$-DP version of the GE algorithm. Computation starts with function A_{GE}. Function A_{GE} recursively calls itself for updating the top-left submatrix. Then it calls functions B_{GE} and C_{GE} in parallel to update the top-right and the bottom-left submatrices, respectively. Then function D_{GE} is called to partially update the bottom-right submatrix and finally function A_{GE} completes updates of the bottom-right submatrix. Functions B_{GE}, C_{GE}, and D_{GE} have similar recursive specifications. It has been shown that such an algorithm can be automatically generated from its corresponding loop-based code [94].

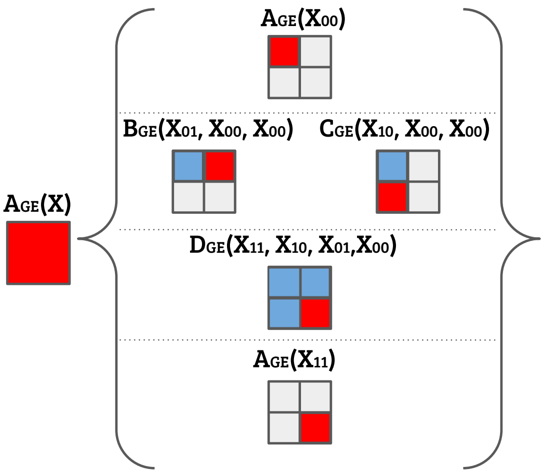


Figure 5.3. Function A of 2-way $\mathcal{R}$-DP GE algorithm.

Listing 5 shows the OpenMP implementation of function A_{GE} depicted in Fig. 5.3. The fork-join based implementation provided in Listing 5 comes with a structural property that limits its performance: synchronization points among the recursive function calls, enforced by `#pragma omp taskwait` in OpenMP or `cilk_sync` in Intel Cilk+, in the join sections of the program create *artificial dependencies* among

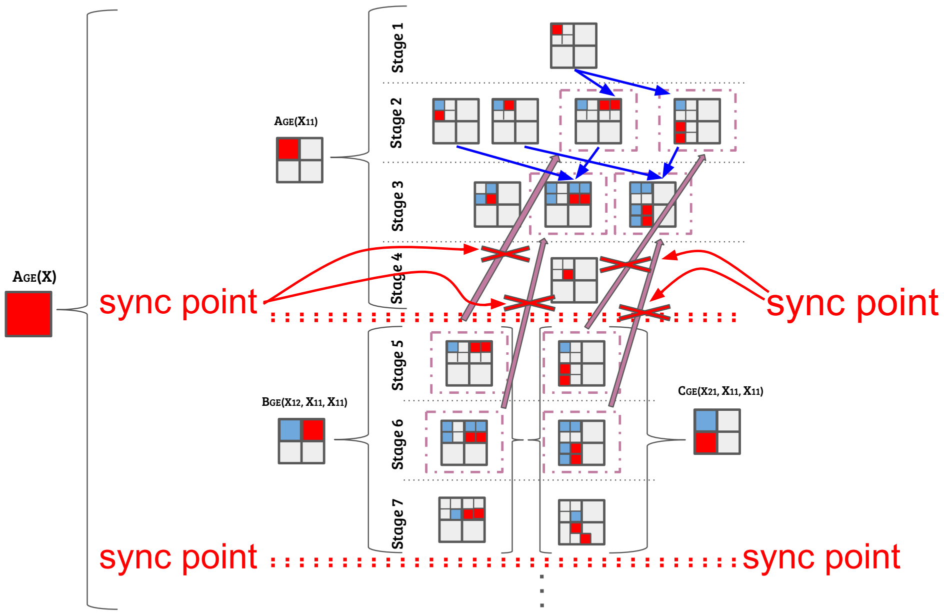


Figure 5.4. Barriers prevent further potential parallelism

the function calls which are not implied by the underlying DP recurrence. These artificial dependencies also exist in the sub-function calls and hence increase the span asymptotically [103, 104] and thus reduce the parallelism. From a practical perspective, they can lead to *resource underutilization*, i.e., threads becoming idle. Fig. 5.4 illustrates the problem. In Fig. 5.4 synchronization points prevent function calls in stages 5 and 6 to be executed in stages 2 and 3. The same problem exists in recursive functions B_{GE}, C_{GE}, and D_{GE}.

This problem has been identified and researchers have proposed *algorithmic solutions within the fork-join model* to resolve this problem statically [103, 104, 98, 102, 99, 101]. However, the proposed solutions are too complicated to develop, analyze, implement, and generalize. For example, they require hacking into a parallel runtime [104] or coming up with timing functions that are not straightforward [103]). On the other hand, task-based runtimes and data-flow parallel models provide a straightforward way to express the algorithms yet easily resolving the inefficiency introduced

```
void func_A(double** X, int input_sz, int block_sz,
            int base_sz, int i_lb, int j_lb, int k_lb)
 if(block_sz <= base_sz) // base case part
  for(int k = k_lb; k < k_lb+block_sz; ++k)
  for(int i = i_lb; i < i_lb+block_sz; ++i)
  for(int j = j_lb; j < j_lb+block_sz; ++j)
   if(k < i && k < j)
    X[i][j] -= (X[i][k] * X[k][j])/X[k][k];
  return;
 // recursive part
 block_sz /= 2;
 // Updating X11 (top-left sub-matrix)
 func_A(X,input_sz,block_sz,base_sz,i_lb,j_lb,k_lb);
 // In parallel, updating X12, and X21 sub-matrices
 #pragma omp task
 func_B(/*Updating X12, reading from X11*/);
 #pragma omp task
 func_C(/*Updating X21, reading from X11*/);
 #pragma omp taskwait
 func_D(/*Updating X22, reading from X11, X12, X21*/);
 func_A(/*Updating X22*/);
```

Listing 5. OpenMP version of function A in $\mathcal{R}$-*DP* GE algorithm.

by synchronization points in the fork-join model. They overcome this limitation as follows. Data dependencies can be specified directly at *finer granularity* and tasks get executed *as soon as its data dependencies are satisfied*. We explain Intel CnC implementation of the same algorithm in the next section.

5.3.3 Data-Flow based Implementation of $\mathcal{R}$-*DP*. To implement the 2-way $\mathcal{R}$-*DP* in Intel CnC, by considering the recursive specification of function calls, we figure out the data dependencies among them. For example, from the specification of function A_{GE}, we can conclude that functions B_{GE} and C_{GE} depend on the output of A_{GE}. Similarly, function D_{GE} depends on the output of functions A_{GE}, B_{GE}, and C_{GE}.

```
struct GEContext: public CnC:context<GEContext> {
 double* dp_table; int input_sz, base_sz;
 typedef pair<pair<int,int>,pair<int,int>> CollectionT;
 // defining step/tag/item collections, X in {A,B,C}
 CnC::step_collection<FunctionX> funcX_step;
 CnC::tag_collection<CollectionT> funcX_tags;
 CnC::item_collection<CollectionT,bool> funcX_outputs;
 // constructor, containging CnC graph information
 GEContext(double* dp_t, int p_sz, int b_sz)
 : dp_table(dp_t), input_sz(p_sz),
   base_sz(b_sz),funcX_step(*this) {
    // prescribing/producing/consuming relationships
    // X in {A, B, C}
    funcX_tags.prescribe(funcX_step,*this);
    funcX_step.produces(funcX_outputs);
    // consumes, defined based on the data dependencies
    funcB_step.consumes(funcA_outputs);
    funcB_step.consumes(funcD_outputs);
    // ...
 }
}
```

Listing 6. Intel CnC graph description of $\mathcal{R}$-DP GE algorithm.

The CnC program has four step collections with one for each of the functions, four tag collections with one for prescribing each of the step collections, and four item collections. The item collections are used as means of synchronization among the step collections to enforce *fine-grained data dependency* among the instances of step collections. The high level structure of the CnC graph or program is depicted in Listing 6.

In Listing 6, tag collections are templated by `CollectionT` which is $pair < pair < int, int >, pair < int, int >>$. This data structure contains the information which is needed for the functions to execute correctly. For example, for function B_{GE} which updates the tile $[I, J]$ of size b by reading from the tile $[I, K]$, the tag is

`<<I,J>,<K,b>>`. Item collections are templated by `<CollectionT,bool>`, which is a mapping from the tile information `<<I,J>,<K,b>>` to Boolean indicating whether the tile has been updated completely (and it is ready to be used by other functions). For example, function B_{GE} puts the mapping (`<<`I_0`,` J_0`>,<`K_0`,` b_0`>>` $\rightarrow$ `true`) to the item collection `funcB_outputs` after completing the update on tile $[I_0, K_0]$. Such put will trigger the execution of all other functions waiting for this tile.

Step collections are templated by C/C++ structs `FunctionA`, ..., `FunctionD`. Each of these structs has a method called `execute` that takes the tag information `execInfo` as the first argument as well as the GE context `ctx` as the second argument.

Based on the data dependencies among the kernels we complete the implementation of the `execute` method in each of the structs. As an example, we explain method `FunctionD::execute` and others are implemented similarly. The implementation has been provided in Listing 7.

If the execution of function D_{GE} reaches its base case, it updates the tile/block with coordinate $[I, J]$ by *first* reading from the tiles/blocks with coordinate $[I, K]$, $[K, J]$, and $[K, K]$ which are produced by kernels C, B, and A, respectively. These three *read-write dependencies* can be enforced by using blocking `get` method on the item collections `funcC_outputs`, `funcB_outputs`, and `funcA_outputs`. Additionally, since it is updating the tile $[I, J]$, for $K > 0$, we need to ensure that the previous call to D_{GE} has finished its update on tile $[I, J]$. So, in order to enforce this *write-write* dependency, we use blocking `get` method on the item collection `funcD_outputs`. If all the dependencies are met, the kernel updates the tile/block and put the item `<<I,J>,<K,b>>`$\rightarrow$ `true` in the item collection `funcD_outputs`. Otherwise, if the function has not yet reached the base case, based on the recursive specification of D_{GE}, for *each of* the recursive function calls defined in its specification, irrespective

```
struct FunctionD {
 /* Updating tile X by reading the tiles updated by
    kernels C, B, and A */
 int execute(const CollectionT& execInfo,
             GEContext& ctx) const {
  int I = execInfo.first.first,
      J = execInfo.first.second,
      K = execInfo.second.first,
      block_sz = execInfo.second.second;
  bool v;
  if(block_sz <= ctxt.base_sz) { // base case
   // checking write-write dependency
   if(K > 0)
    {ctx.funcD_outputs.get({{I,J},{K-1,block_sz}},v);}
    // checking read-write dependencies
    ctx.funcA_outputs.get({{K,K},{K,block_sz}},v);
    ctx.funcB_outputs.get({{K,J},{K,block_sz}},v);
    ctx.funcC_outputs.get({{I,K},{K,block_sz}},v);
    // All dependencies OK, executing the base case
    ge_iterative_kernel(ctx.input_sz, block_sz,
                        I, J, K, ctxt.dp_table);
    ctx.funcD_outputs.put({{I,J},{K,block_sz}}, true);
  }
  else { // recursive part
   int tile_sz = block_sz/2;
   for(int kk = 0; kk < 2; ++kk)
   for(int ii = 0; ii < 2; ++ii)
   for(int jj = 0; jj < 2; ++jj)
    ctx.funcD_tags.put({{I*2+ii,J*2+jj},
                        {K*2+kk,tile_sz}});
  }
  return CnC::CNC_Success;
 }
};
```

Listing 7. Struct `functionD` in CnC implementation of $\mathcal{R}$-*DP* GE algorithm.

of their data dependencies[7], it puts tags into the tag collection `funcD_tags` to trigger their executions.

5.3.4 Improving Intel CnC performance through Tuners.

Intel CnC provides tuners that can pass hints to the runtime system on how to improve performance [114]. One of them is the pre-scheduling tuner which enforces the execution of a step on the same thread that puts the prescribing tag, only after all the data dependencies are satisfied. This can improve performance by avoiding re-scheduling of a step due to unavailability of the items. Another way of improving the performance is to manually pre-declare all the dependencies, before the actual execution of updates in the algorithm. In this way, the underlying scheduler can trigger tasks when all the items are available. We have evaluated both approaches in order to tune the $\mathcal{R}$-DP computations and better understand the behavior.

5.4 Experimental Results

We implement the three following benchmarks in Intel CnC and OpenMP: **(1) Gaussian Elimination without Pivoting (GE).** Section 5.3.1 contains a detailed explanation of this benchmark. It is noteworthy that the GE with partial pivoting does not have a DP-like structure [120] and going beyond DP algorithms is part of the future works. **(2) Smith-Waterman Local Alignment (SW).** The SW algorithm is used to determine the similarity between two DNA (or amino acid) sequences [121]. **(3) Floyd Warshall's All Pairs Shortest Path (FW-APSP).** For each pair of vertices in a directed graph, the FW-APSP algorithm computes the cost of the shortest path [86, 122].

5.4.1 Experimental Setup.

The testbed for our experiments includes AMD Epyc

[7]Note that all the data dependencies are enforced using the blocking `get` method

and Intel Skylake processors which are part of the Mystic testbed [38]. The AMD Epyc 7501 machine has 2 sockets with 32 cores each, 8 NUMA zones, 32K L1, 512K L2 and 8192K L3 caches, 130GB RAM and per socket memory bandwidth of 170 GiB/s. The Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz machine has 8 sockets with 24 cores per socket, 8 NUMA zones, 32K L1, 1024K L2, 33792K L3, 768GB RAM and a theoretical memory bandwidth of 119 GiB/s.

For our Intel CnC implementations, we used Intel CnC version 1.0.1 and compiled using gcc version 7.5.0, with the following flags:

```
-std=c++11 -O3 -march=native -mavx2 -lcnc -lrt -ltbb -ltbbmalloc.
```

We have optimized the algorithms by eliminating branches in the innermost loop to enable vectorization. Naive implementation of SW uses ($\mathcal{O}(n^2)$) space and we have optimized the algorithm to consume ($\mathcal{O}(n)$) space for improving performance. GNU OpenMP implementations are used in the benchmarks with `OMP_PLACES =cores` and `OMP_PROC_BIND=close`. For Intel CnC experiments, we set `CNC_NUM_THREADS` to 64 on AMD Epyc and 192 on Intel Skylake servers.

5.4.2 Performance Results. The goal of our evaluation is to characterize the behavior of $\mathcal{R}$-DP computations under a data-flow execution model. With this in mind, we designed $3{\times}2{\times}4{\times}4 = 96$ experiments, which include three benchmarks (GE, SW and FW-APSP) to explore on two multicore machines, while varying the problem parameters (problem size and base-case size). Our experiments show that even though $\mathcal{R}$-DP is meant to enhance the program's locality, controlling and characterizing the behavior in a data-flow model remains challenging. For each $\mathcal{R}$-DP benchmark, we implemented 4 versions:

- (***Native-CnC***) A base CnC program without scheduling hints.

- (***Tuner-CnC***) A CnC program with task scheduling hints by using CnC tuners (discussed in Sec. 5.3.4).
- (***Manual-CnC***) A manually pre-scheduled CnC program (discussed in Sec. 5.3.4).
- (***OMP-Tasking***) An $\mathcal{R}$-*DP* program using OpenMP tasking.

It is worth mentioning that we also implemented the benchmarks using non-blocking `get` approach [116] and noticed that the non-blocking `get` implementation is profitable only for smaller block sizes. However, the best overall performance is obtained by using blocking `get` approach.

Overall, our validation shows some high-level conclusions. First, $\mathcal{R}$-*DP* data-flow programs incur large runtime overheads on small block sizes. Second, large base case sizes reduce potential run-time task scheduling options.

Figures 5.5 and 5.6 show the execution time of the GE benchmark on the two machines. To understand the behavior of GE on these machines, due to the importance of the data movements in the memory hierarchy [123], we have developed an analytical model to estimate the overall cost of cache misses and the data movements.

As the first step, we will compute the total number of tasks generated by the recursive divide-and-conquer algorithm for GE. Observe that if the base case size is set to 1×1, the total number of times the base case is reached will be equal to the number of assignments made by the looping implementation of GE, which is: $\sum_{k=0}^{n-1} \sum_{i=k+1}^{n} \sum_{j=k+1}^{n} (1) = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$. Now, if we coarsen the base case matrices to $m \times m$, clearly, the number of times such base cases will be reached, i.e., the number of base case tasks generated by the recursive algorithm, will be:

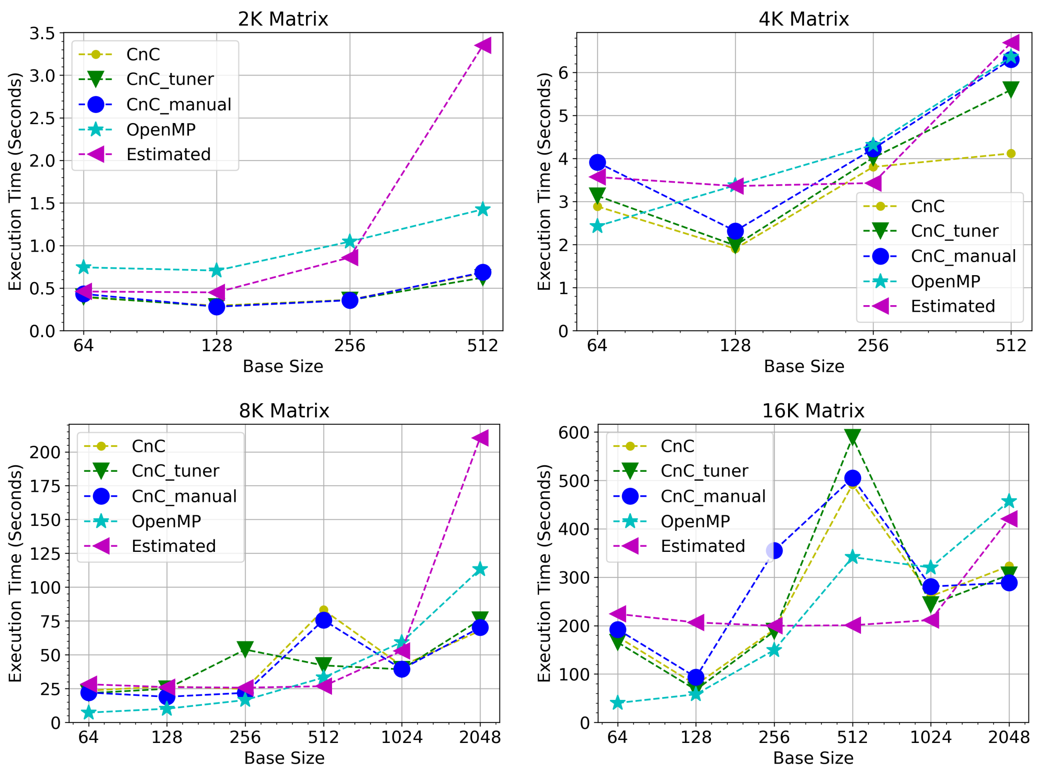


Figure 5.5. Execution time of Gaussian Elimination on EPYC-64

$$\frac{1}{3}\left(\frac{n}{m}\right)^3 + \frac{1}{2}\left(\frac{n}{m}\right)^2 + \frac{1}{6}\left(\frac{n}{m}\right) \tag{5.1}$$

Assuming a fair distribution of the tasks to the processors and among all the cores, we know that the total number of tasks per processor is $\left\lceil \frac{total\ number\ of\ tasks}{number\ of\ \ processors} \right\rceil$.

Once reached, a base case task that works on matrices of size $m \times m$ will perform between $\frac{1}{3}m^3+\frac{1}{2}m^2+\frac{1}{6}m$ (inside `func_A`) and $\sum_{k=0}^{m-1} \sum_{i=0}^{m} \sum_{j=0}^{m}(1) = (m+1)^2 m$ (inside `func_D`) assignments.

Considering the base-case implementation of the GE algorithm, which is the serial implementation (in the Listing 4), we can compute the maximum number of cache misses as follows. We know that the triply nested loop executes up to $\forall_{k=0}^{m-1}\forall_{i=0}^{m}\forall_{j=0}^{m}$ iterations, while accessing memory cells $C[i][j]$, $C[i][k]$, $C[k][j]$, and $C[k][k]$. Then,

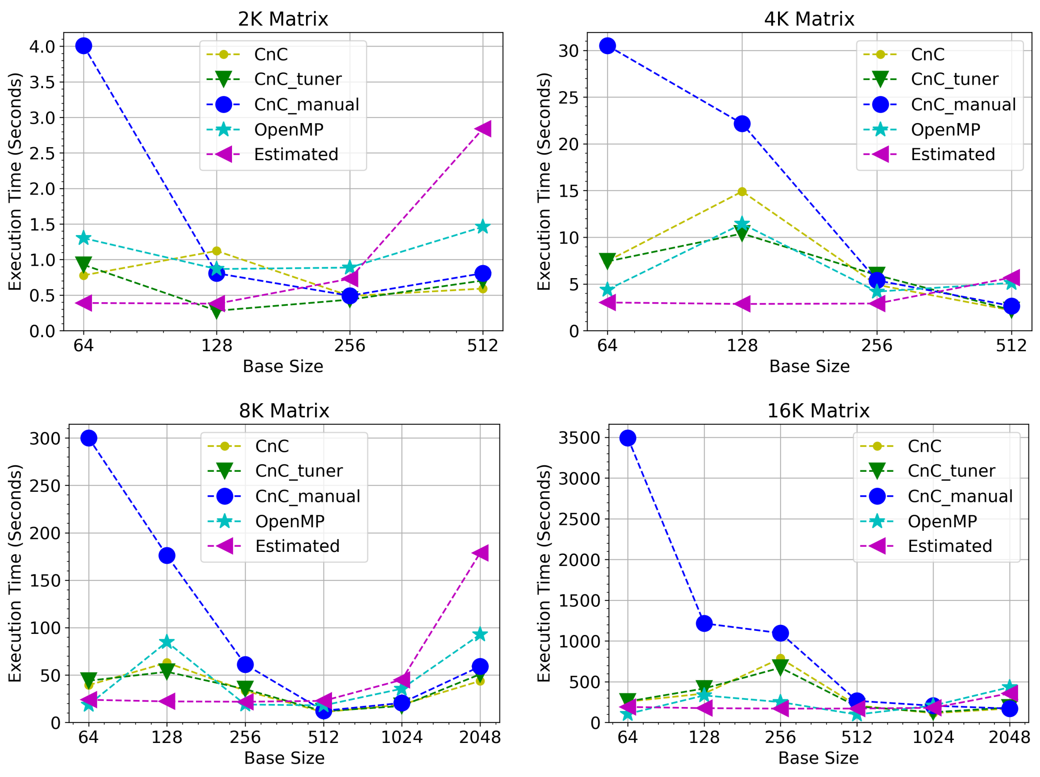


Figure 5.6. Execution time of Gaussian Elimination on SKYLAKE-192

we proceed to count the total number of memory elements accessed for each distinct array reference, divided by the cache line size L, and add them up to get an upper bound on the total number of cache misses assuming that the cache cannot hold more than three cache lines and thus has very limited temporal locality. The bound is obtained as follows:

$$\begin{aligned} &2\left(\sum_{k=0}^{m-1}\sum_{i=0}^{m}\left\lceil\frac{\sum_{j=0}^{m}8}{L}\right\rceil\right)+\left(\sum_{k=0}^{m-1}\sum_{i=0}^{m}1\right)\\ &+\left(\sum_{k=0}^{m-1}1\right)=m\big(1+(m+1)\big(1+\left\lceil\tfrac{8m+8}{L}\right\rceil\big)\big) \end{aligned} \tag{5.2}$$

The first term in the summation above accounts for the maximum number of cache misses incurred when accessing $C[i][j]$ and $C[k][j]$, the second term accounts for $C[i][k]$, and the third one for $C[k][k]$. Given this, the total number of cache misses for each cache $L1$, $L2$, and $L3$, is approximated by adding up all the cache miss penalties at each level of cache. Figures 5.5 and 5.6 show the cost estimated using this model. The model assumes the recursion and looping overheads to be zero.

The ratio of the maximum cache misses estimated by the analytical model over the actual cache misses (i.e., $\frac{estimated\ max\ cache\ misses}{actual\ cache\ misses}$) provides an interesting measure of temporal locality. The larger this ratio the higher the temporal locality. For the GE benchmark with the problem size $8K \times 8K$, we captured the actual cache misses using the PAPI library [124] on SKYLAKE, and calculated this ratio. Table 5.1 shows the ratios for different base case sizes. Considering the sizes of $L2$ and per-core $L3$ cache share (which are 1MB and 32MB, respectively), we observe that for the $L2$ and $L3$ caches, this ratio sharply drops for the base cases larger than 128×128 and 1024×1024, respectively. These two base cases (128×128 for $L2$ and 1024×1024 for $L3$) reflect the largest blocks (more specifically, three such blocks storing double precision floats) that can fit into the $L2$ and $L3$ cache for GE on SKYLAKE.

Another important observation is that the execution times are significantly lower with hardware prefetching turned off for the CnC version. This is due to the coarse-grained data-flow irregularity not allowing full usage of prefetched data, i.e. the prefetcher bringing in data expected to be used, while (CnC) data-flow dependencies essentially flushing the cache immediately after, causing unnecessary overheads.

The analytical model does not take into account the load imbalance due to the data dependencies between the tasks causing the model to underestimate the cost. However, in some cases, using maximum cache misses to calculate the estimated cost causes the model to overestimate. The model also ignores overhead of scheduling of large number of tasks which significantly increases the execution time in case of *Manual-CnC*.

Base Size	L2 Cache	L3 Cache
64	107.61	294.50
128	240.63	660.02
256	38.38	1637.20
512	7.97	5793.74
1024	6.13	8247.60
2048	5.96	127.06

Table 5.1. Ratio of the maximum estimated cache misses over the actual cache misses for the GE benchmark with problem size $8K \times 8K$ on SKYLAKE.

Another important observation from the figures is that for GE and FW-APSP benchmarks, for a fixed computation resource, as we increase the size of the input, the fork-join implementation (i.e,. OpenMP) outperforms the data-flow implementations (i.e., intel CnC). This is due to the fact that for the smaller problem size, because of the artificial dependencies that exists in the fork-join implementation, there are not enough tasks generated by the OpenMP to keep all the processors busy and does not have enough data locality. As a result, we have resource underutilization issue. However, as the problem size gets larger, in spite of the existence of the artificial dependencies, OpenMP is capable of generating enough tasks to feed all the processors and we have less resource underutilization.

Figures 5.7 and 5.8 show the execution time of SW benchmark on EPYC-64 and SKYLAKE-192 systems. Regarding the SW benchmark, the issue of artificial dependencies are so problematic that even for bigger problem sizes, still data-flow implementation outperforms. The main reason is the artificial dependencies in the fork-join implementation prevents the wavefront parallelism among the tasks, where tasks operate on tiles along diagonals of the input matrix[8].

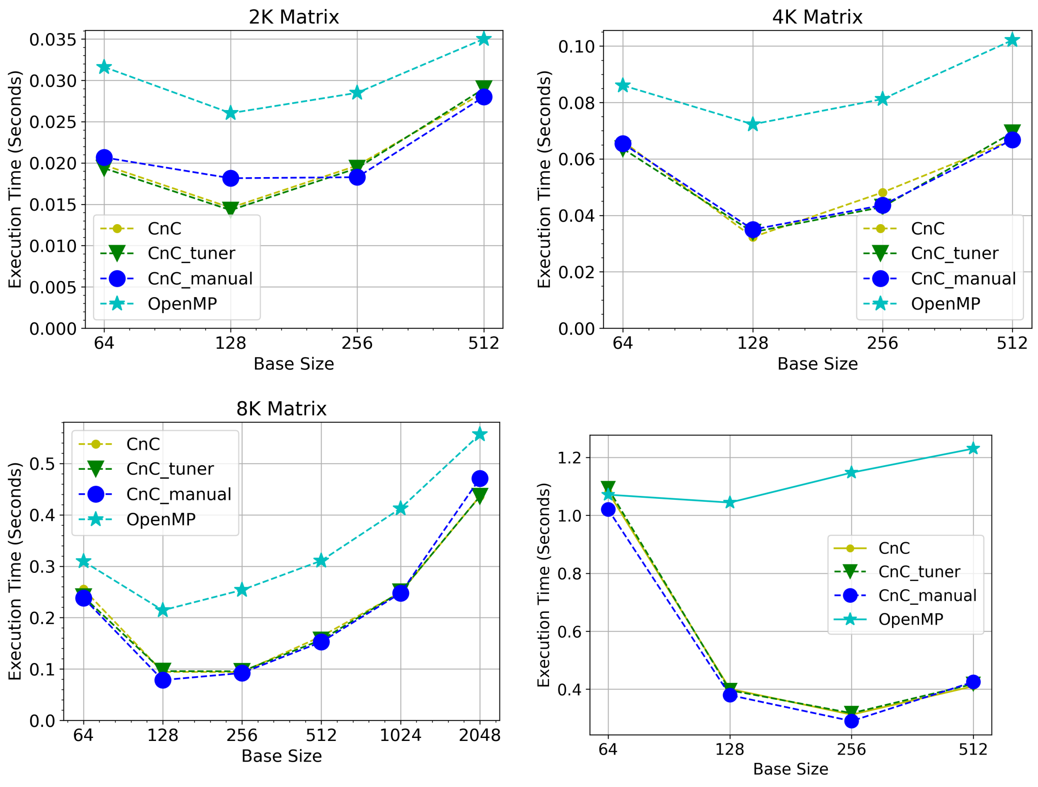


Figure 5.7. Execution time of Smith-Waterman on EPYC-64

However, data-flow implementation can easily benefit from the wavefront parallelism as data dependencies are specified at a finer granularity and there is no coarse-grain barrier synchronization for every wavefront computation.

Figures 5.9 and 5.10 show the results obtained for FW-APSP benchmark. The analytical model described above for GE also applies to FW-APSP since both the same computational complexity as GE ($\mathcal{O}(n^3)$) and similar data access patterns.

[8]In fork-join implementation, there is a barrier synchronization for every wavefront computation

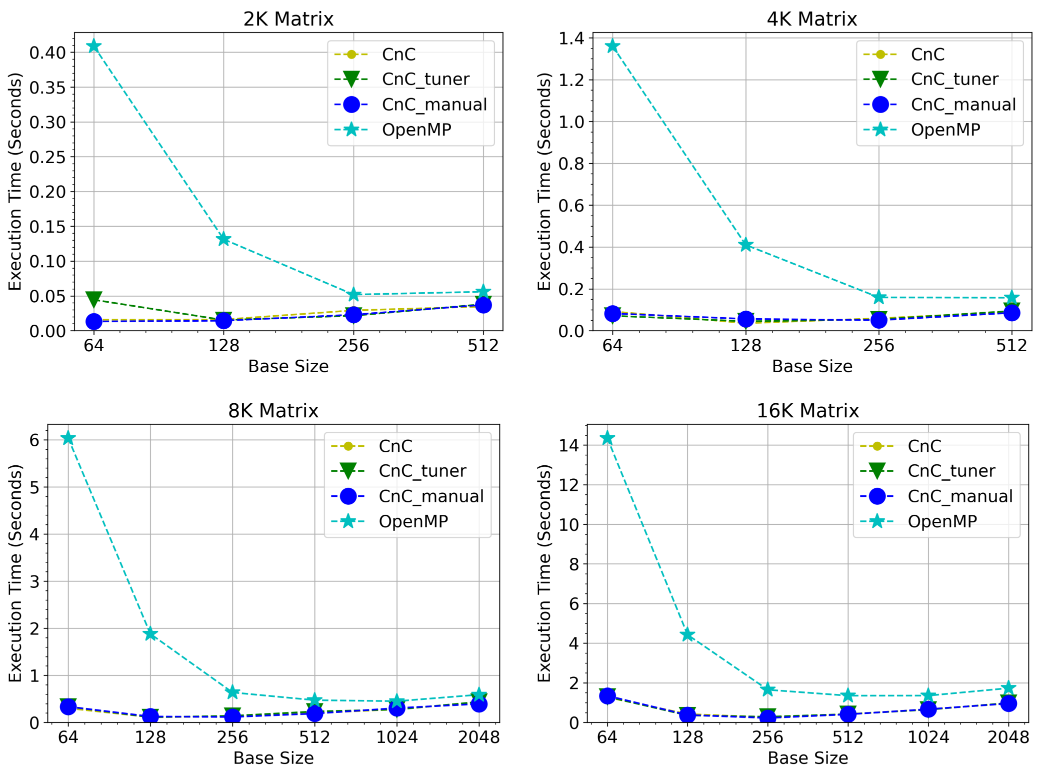


Figure 5.8. Execution time of Smith-Waterman on SKYLAKE-192

Best running time is achieved with block size of 128 and 256 for all the variations of Intel CnC as well as OpenMP.

While fine-grained scheduling and task placement has not been explored in this work, we believe that leveraging other Intel CnC tuners such as `compute_on` and other forms of tasks pre-scheduling can lead to large performance improvements. Such tuner can effectively allow to pin specific tasks to execution locations (cores), thereby minimizing potential inter-core and inter-NUMA data movement.

5.5 Summary

In this work, we discussed the two different paradigms of parallel programming on shared-memory multicore machines: fork-join based and data-flow based parallel paradigms. Focusing on DP algorithms, we explained the major performance bottle-

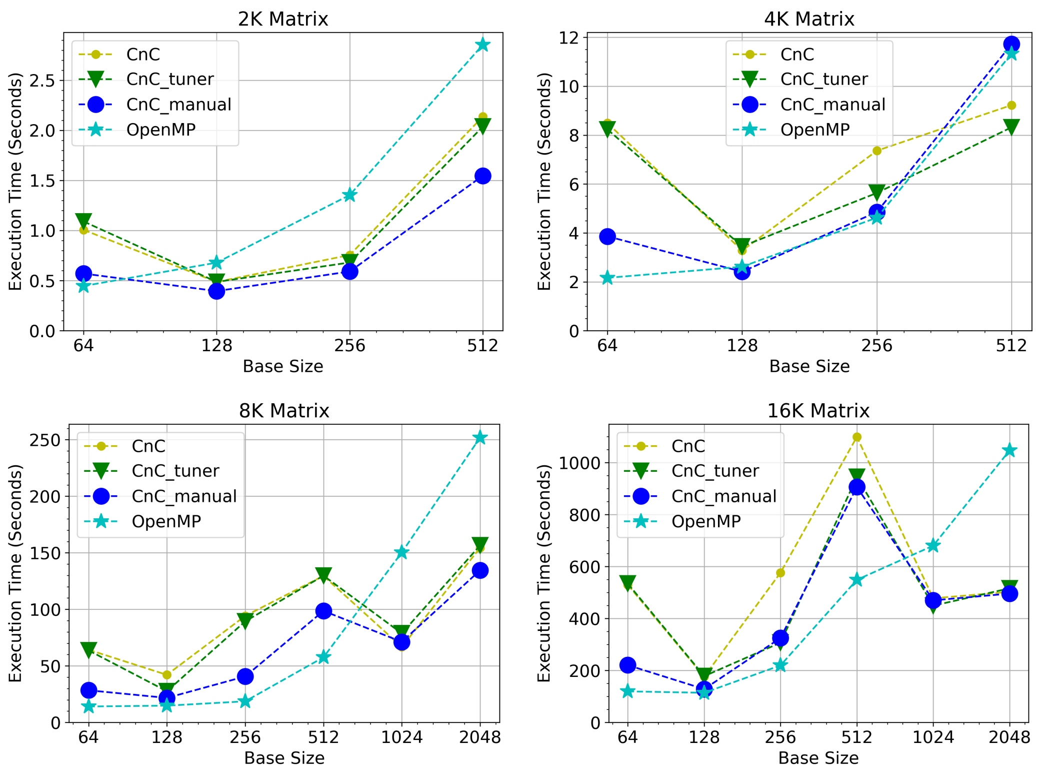


Figure 5.9. Execution time of Floyd Warshall's Algorithm on EPYC-64

neck that exists in fork-join based model: Joins (e.g., `cilk_sync` in Intel Cilk+ or `#pragma omp taskwait` in OpenMP) in synchronization points introduce artificial dependencies which are not implied by the underlying DP recurrence. These artificial dependencies can increase the span asymptotically, and thus reduce parallelism. We also explained how this performance issue can easily be eliminated by using data-flow based parallel paradigms. Our experimental results indicate the fact that the data-flow based implementations outperform the fork-join based implementations.

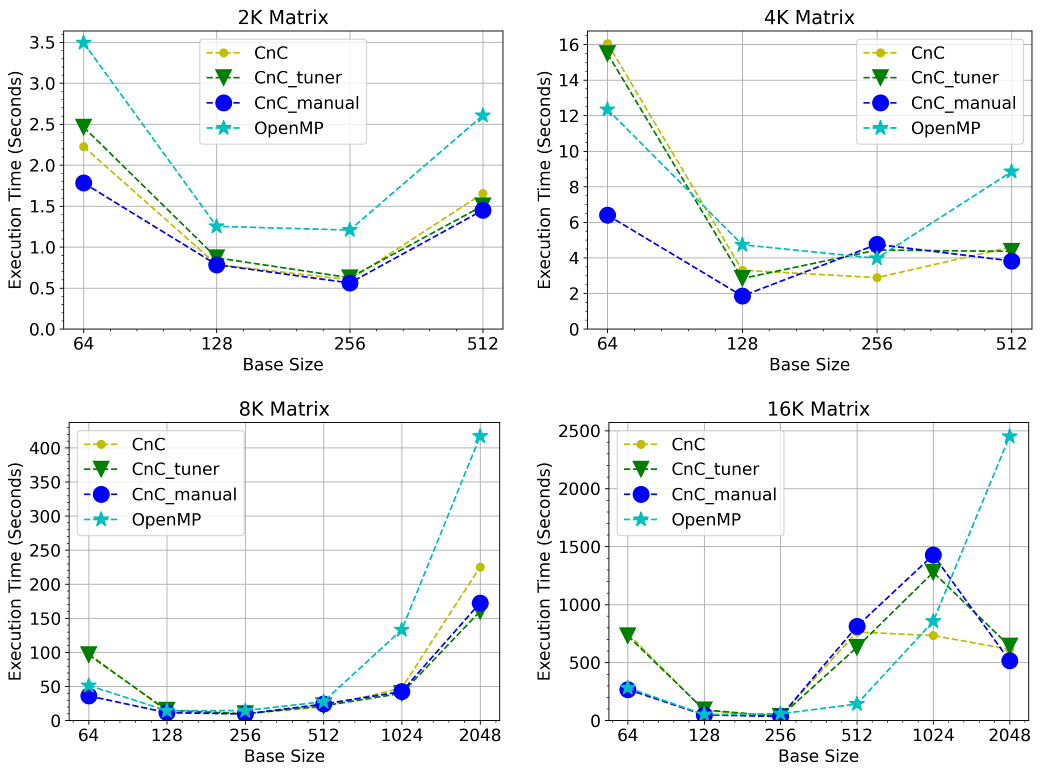


Figure 5.10. Execution time of Floyd Warshall's Algorithm on SKYLAKE-192

CHAPTER 6

THE TEMPLATE TASK GRAPH (TTG): AN DATAFLOW PROGRAMMING PARADIGM FOR IRREGULAR SCIENTIFIC APPLICATIONS

Our work in comparing data-flow to fork-join parallel programming shows that data-flow parallel programming paradigm overcomes several limitations of the fork-join programming model by allowing applications to specify data dependencies at a finer granularity [125, 126, 127]. 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. Several data-flow based programming models exist today, one of them is Intel Concurrent Collections (CnC) [127]. In our prior work with Intel CnC, we identified several limitations for expressing dataflow programs using the programming model: (1) a successor task can be invoked by sending a `tag`, however the task itself needs to call `get` to actually obtain the data. Get can be synchronous or asynchronous, however this limits optimizations that can be applied in the runtime layer to optimize data movement. (2) While development cost is a subjective measure, it requires a substantial coding effort to write data flow programs using CnC. There is a need for a productive parallel programming language that can bridge the gap between productivity and programmability on heterogenous architectures.

The main contributions of this work are:

- Introduction of *TTG*, a C++ API for implementing data-flow parallel programs. Description of key features of *TTG* that serve as a backbone for implementing heterogeneous execution of programs written in *TTG* [113, 2, 128].
- The evaluation of their implementation over different applications, and the comparison of the *TTG* implementation of these applications with the state of the

art implementation in other programming paradigms.

6.1 Motivation

This work is inspired by the advantages of data-flow based parallel programming paradigm. The main advantages are: (1) specification of essential dependencies between tasks maximizing the exploitable concurrency (2) reduced synchronization due to flow of data between tasks. Also, today's data-flow based programming languages [127, 12] involve significant programming costs due to explicit management of data and execution throughout the program run.

This work is done as a part of the TESSE [113] project. It is a combined team effort and this dissertation includes the work I accomplished in this project. TESSE stands for Task-based Environment for Scientific Simulation at Extreme Scale. This project aims to address the twin challenges of programmer productivity and portable performance for advanced scientific applications on modern day heterogeneous hardware. Our focus is on irregular computations that are difficult to compose and execute efficiently with current parallel programming paradigms.

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. TESSE leverages dataflow programming approach to be able to compose and execute irregular computations efficiently on heterogeneous architectures. The key innovation of TESSE is TTG, a dataflow programming model inspired by earlier innovations such as Flow-Based Programming (FBP).

6.2 Template Task Graph

TTG is implemented as a library using C++ and template metaprogramming. *TTG* marries the ideas of flow programming models with the key innovations in the PARSEC runtime [129, 112] for template-based compact specification of task graphs namely Parameterized Task Graph (PTG) [130] in which edge represents the flow of data which identifies the receiving task. The main goal of *TTG* is to support compact specification of task graphs for efficient distributed execution of dynamic and irregular applications. Existing task-based programming interfaces only support shared-memory parallel environments very effectively, supporting a few distributed memory environments and they do so by either discovering the entire DAG which limits the scalability or introduce explicit communications between operations which increases the complexity of programming. *TTG* aims to address these issues by providing higher-level abstractions and supporting multiple runtimes to manage task creation and execution. *TTG* stands out in the following areas: (1) specifying 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, makes operations easier to reuse by eliminating the nonessential side effects, and (3) raising the level of abstraction (by abstracting the details of scheduling, underlying resources), programs (often) become easier to write, easier to transform (thereby supporting the development of domain specific languages), and easier to port.

TTG [113, 2] represents an algorithm as a data-flow graph (*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) [130]. 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.

6.2.1 TTG Concepts. This section describes the key concepts of *TTG*.

- TaskId: A unique identifier for each task. For example, if computing in a loop, it can be the loop identifier, if computing on text, it can be a string identifying the line of the text, if computing on a 3D array, it can be the triplet of indices to identify the element in the array. The TaskId is also used by the underlying runtime for mapping tasks on to processors based on a user-defined process map.

- 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 single assignment 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: An Edge represents connection between the terminals. Output terminal of a task can be connected to the input terminal of another task which forms an edge. Currently terminals are identified by zero-based indexing. Multiple terminals can be connected to a single terminal and vice versa. Recursion can also be implemented by connecting the output terminal of a task to its own input terminal.

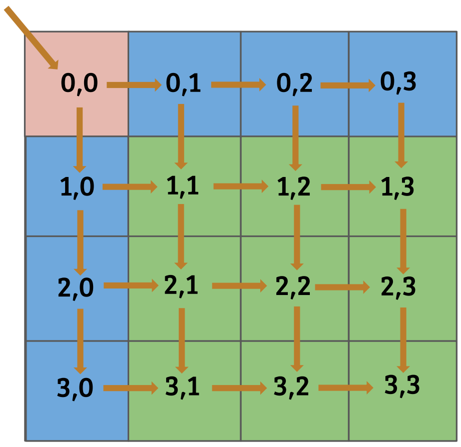
- TemplateTask: This wraps a user-defined function with informal signature void f(TaskId, Arg0, Arg1, ..., OutputTerminals). Again, each input argument is exposed as a Terminal, and OutputTerminals is a tuple of the output terminals (an alternative interface also provides the input arguments as a tuple 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 received. If there are no arguments, the task must be created either manually via a special method (invoke(TaskId))of the TemplateTask, or via a pull operation which which is described below.

- Push versus pull: As seen 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.

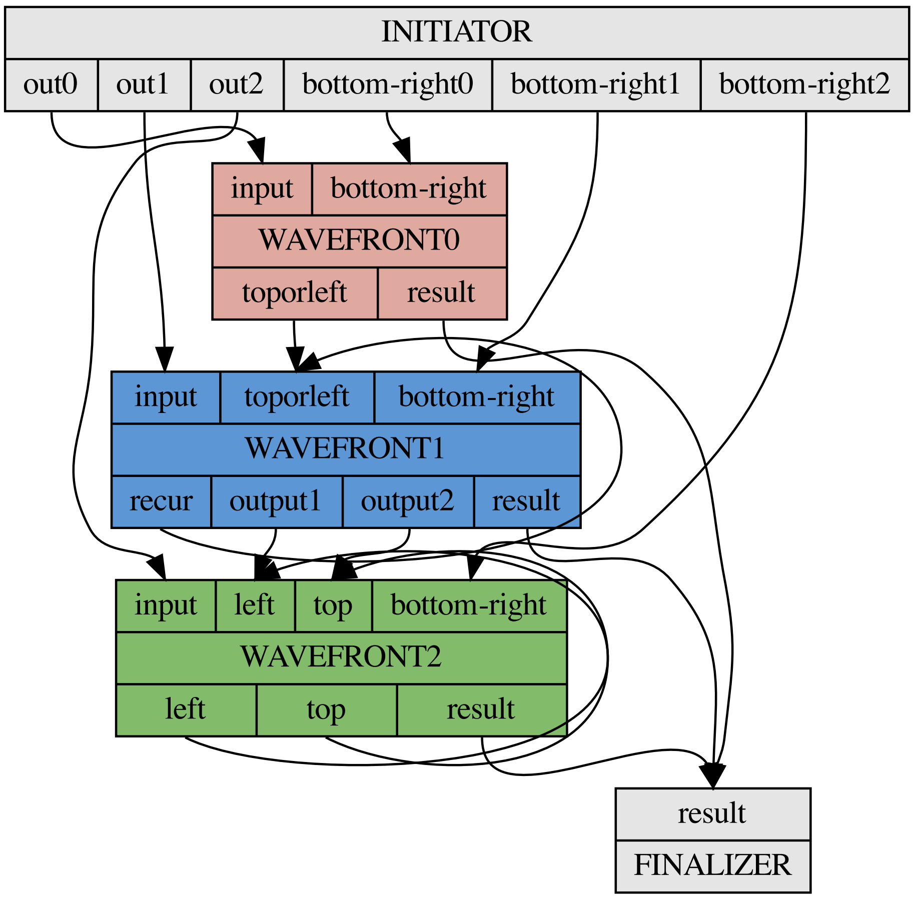
6.2.2 Wavefront Traversal Example Using *TTG*. To illustrate the *TTG* concepts, we consider the well-known algorithm for wavefront traversal and whose TTG implementation is shown in listings 8 and 9 [113].

Figure 6.1(b) illustrates its template task graph. Input to this algorithm is a 2D matrix which is divided into blocks. The algorithm consists of separate template tasks with varying number of input blocks for handling starting, corner and middle cases of the input matrix during wavefront traversal. 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`).

The input is an $N \times N$ matrix divided into blocks, and we reuse a 2D example provided by Cpp-Taskflow [17] with a 5-point stencil for which computation on block $B[i][i]$ requires data from all four neighbors $B[i-1][j]$, $B[i][j-1]$, $B[i+1][j]$, $B[i][j+1]$



(a) Flow of computation



(b) graph using dataflow

Figure 6.1. 2D Wavefront Computation

```
ttg::Edge<pair<int,int>,BlockMatrix<double>> input0("input0"),
                    input1("input1"),
                    input2("input2"),
                    toporleft("toporleft"),
                    output1("output1"),
                    output2("output2"),
                    result("result");
Edge<Key, std::vector<BlockMatrix<double>>> bottom_right0("bottom_right0"),
                    bottom_right1("bottom_right1"),
                    bottom_right2("bottom_right2");

auto i = initiator(m, input0, input1, input2,
                bottom_right0, bottom_right1, bottom_right2);
auto s0 = make_wavefront0(stencil_computation<double>,
                n_brows, n_bcols, input0, toporleft, bottom_right0, result);
auto s1 = make_wavefront1(stencil_computation<double>,
                n_brows, n_bcols, input1, toporleft,
                bottom_right1, output1,
                output2, result);
auto s2 = make_wavefront2(stencil_computation<double>,
        n_brows, n_bcols, input2, output1,
        output2, bottom_right2, result);
auto res = make_result(r2, result);

```

Listing 8. Select elements of the C++ code specifying the TTG implementation of wavefront traversal in 6.1(b)

but only has task dependencies on $B[i-1][j]$ and $B[i][j-1]$. Figure 6.1(a) shows the task dependencies between the blocks — blocks with same color can run concurrently. Computation starts at the top-left and sweeps the grid diagonally.

Listings 8 and 9 illustrate 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`. Listing 9 shows how the `wavefront` task template is implemented. The free function (or lambda) 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`; 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. Since the edges, input, and output terminals are all explicitly parameterized 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.

Every template task can have a different type for the TaskId. Also, the application user does not need to worry about data synchronization since protecting access is under *TTG*'s control. *TTG* can take advantage of the C++ language features for optimizing the data motion.

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 speci-

```
template <typename funcT, typename T>
auto make_wavefront0(const funcT& func, int MB, int NB,
            Edge<Key, BlockMatrix<T>>& input,
            Edge<Key, BlockMatrix<T>>& toporleft,
            Edge<Key, std::vector<BlockMatrix<T>>>& bottom_right,
            Edge<Key, BlockMatrix<T>>& result) {
  auto f = [func, MB, NB](const Key& key, const BlockMatrix<T>& input,
        const std::vector<BlockMatrix<T>>& bottom_right,
        std::tuple<Out<Key, BlockMatrix<T>>, Out<Key, BlockMatrix<T>>>& out) {
    auto [i, j] = key;
    int next_i = i + 1;
    int next_j = j + 1;

    BlockMatrix<T> res = func(i, j, MB, NB, input,
            input, input, bottom_right[0], bottom_right[1]);

    send<0>(Key(i, next_j), res, out);
    send<0>(Key(next_i, j), res, out);

    send<1>(Key(i, j), res, out);
  };

  return make_tt(f, edges(input, bottom_right),
            edges(toporleft, result), "wavefront0", {"input", "bottom_right"},
            {"toporleft", "result"});
}
```

Listing 9. Implementation of one of template tasks for the wavefront traversal

fied 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*.

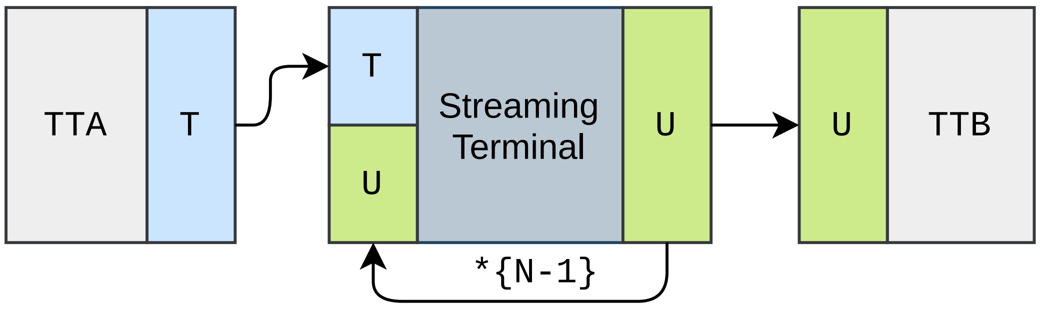


Figure 6.2. TTG streaming terminal with input `T`, output `U`, and a size of `N`. The reduction operation of the terminal will be called $N-1$ times on input from `TTA` before before a task of `TTB` will be eligible for execution [2]

6.2.3 Streaming Terminals. Each input terminal could receive only one message for a given task ID under the original design of *TTG*. Due to this restriction, some algorithms produce task templates with many 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, we listed this restriction 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 $U \otimes T \rightarrow U$ 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 programatically "finalized" for the given task ID (see 6.2). An example for using

```
std::function<const Key (const Key&)> get_inputindex_func =
    [n_brows, n_bcols](const Key& key) {
      return key;
  };

auto container_keymap = [local_row_count](const Key &key) {
                              return key.first / local_row_count;
                            };

Edge<Key, BlockMatrix<double>> block("block", true,
    {m, get_inputindex_func, container_keymap});

```

Listing 10. Creating Pull Terminals

streaming terminals will be provided later in 6.3.3.

6.2.4 Pull Terminals as Generator Terminals. As described above, terminals can be marked as push or pull based on whether the data is pushed by a task to its successor task or pulled into a terminal by a task after its creation. In *TTG*, every operation with different number of input dependencies requires a separate template task. However, some applications require data from various sources purely for computation, and do not necessarily imply a data dependency for execution of the task. The original model of *TTG* only allowed for data to be `PUSH`ed to subsequent tasks via terminals. In order to push data from different sources for computation, a 'reader' task is required which when run, instantiates all the tasks it can push data to, which is not required and can pose resource limitation issues. Tasks should be created when the data dependencies are satisfied. If data can be `PULL`ed by a task when necessary, we can defer running the "reader" tasks until needed.This functionality enables important optimizations to directly call the reader source to access the data without tasking overhead. Data can be pulled greedily(at task creation time) or lazily (when all other inputs are available) to control resource utilization.

Every pull task requires a template task to be defined. `EDGE` can be used to define pull terminals as shown below in Listing 10. A pull template task can contain 0 or 1 pull terminals as input. The `Key` of the puller task is sent as input to the pull task to invoke the task. If the pull task itself has a pull terminal as input, this would walk through the DAG in a reverse manner until necessary data is pulled. *TTG* can invoke pull tasks using a callback mechanism. The callback is registered when an `Edge` is created. As shown in listing 10, registering a pull template task as a data source requires a `Container` to be passed into the `Edge` constructor along with a mapper function that maps the `TaskId` to the index into the container as well as a `keymap` for identifying the process to locate the data.

6.2.5 Pure Tasks. Pure tasks are a variation of pull terminals where the template task can return data instead of calling a `send` function to push the data to the successor task. Pure tasks can be seen as a natural way of writing functions and the runtime internally calls `send` to forward the data to the requested successor. This functionality has several use cases: (1) can be used to implement distributed data structures (2) enables split phase implementation required for devices like GPU where a `send` operation cannot be directly called from a GPU kernel, but after the kernel completes execution, the host code can make the decision of where to send the data.

6.2.6 *TTG* Execution Backends. *TTG* is a higher level of abstraction over tasking runtimes. The current implementation of *TTG* creates tasks which can be handled by any low-level tasking runtimes in general. The runtime that manages the task scheduling and resource management is referred to as a *TTG backend.* Currently *TTG* works with two backends that can run on both shared-memory and distributed-memory platforms: *PaRSEC* and *MADNESS*. 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** [131] is a general-purpose numerical environment for reliable and fast scientific simulation. It enables massive parallelism for dense and sparse tensor algebra and 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 [132]. *MADNESS* provides an SPMD model 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 **[133]** is a widely adopted runtime in scientific applications. *PaRSEC* is designed to support many Domain Specific Languages (DSLs) or Application Programming Interfaces (APIs): `PTG` and `DTD`. `PTG` is similar to TTG in that you specify the task classes and their inputs as terminals and data flows through them. It is less flexible, because one cannot send from within a task, so dynamic refinement etc is quite tedious to do. `DTD` on the other hand is like OpenMP tasks but in a global scope. And it requires discovery of the full global task graph, which limits scalability. It has a C frontend, so it is easier to use. TTG combines aspects of both: the scalable task discovery of `PTG` with the the easier to use API and provides the highest flexiblity of all of the available DSLs and APIs. *TTG* reuses the scheduler, communication and termination detection features of *PaRSEC*.

This work in *TTG* helped improve the efficiency and scalability of the *MADNESS* and *PaRSEC* backends, but without impacting the correctness and capability of *TTG*, and support a full set of *TTG* features. A nice feature of *TTG* is that all programs can be developed independent of the runtime backend and can be linked to the required runtime as required.

6.3 Benchmarks

We implemented a set of algorithms, with varying degree of irregularity in their data and computation traits, using C++ implementation of the *TTG* programming model. The performance was evaluated against reference implementations using traditional programming models or, where available, against existing state-of-the-art implementations.

6.3.1 Test Setup. We performed our evaluation on two systems. The *Hawk* system is a Hewlett Packard Enterprise Apollo[9] 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[10] 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 6.1.

6.3.2 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.

[9]https://www.hlrs.de/systems/hpe-apollo-hawk/

[10]https://it.stonybrook.edu/help/kb/understanding-seawulf

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

Table 6.1. Software configurations

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.

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 [90]. Javanmard et al. extended it to a recursive multi-level tiled algorithm to run efficiently on distributed-memory machines as well as GPUs [98, 101]. 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. [59] implemented a data-flow version of the standard two-way recursive divide-and-conquer FW-APSP algorithm in Intel CnC [127] and compared the performance with a fork-join implementation in OpenMP. They showed that a data-flow implementation outperforms its fork-join counter-part when, due to artificial dependencies, the fork-join implementation fails to generate enough subtasks to keep all processors busy and does not have enough data locality to compensate for the lost performance.

As shown in 6.3, the parametric recursive algorithm has four kernels (A, B,

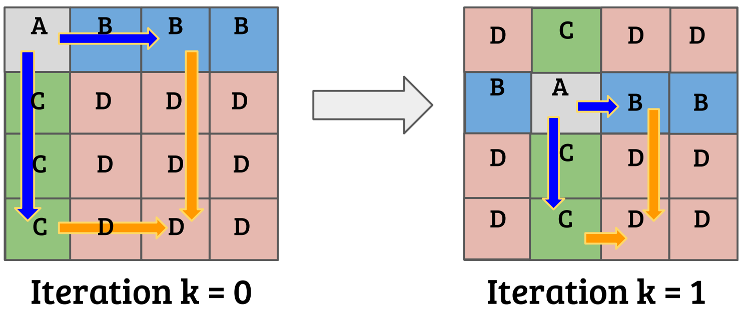


Figure 6.3. Flow of data among different kernels in blocked FW-APSP algorithm.

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 [98] 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 [99, 101] 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*.

6.4 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* 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.

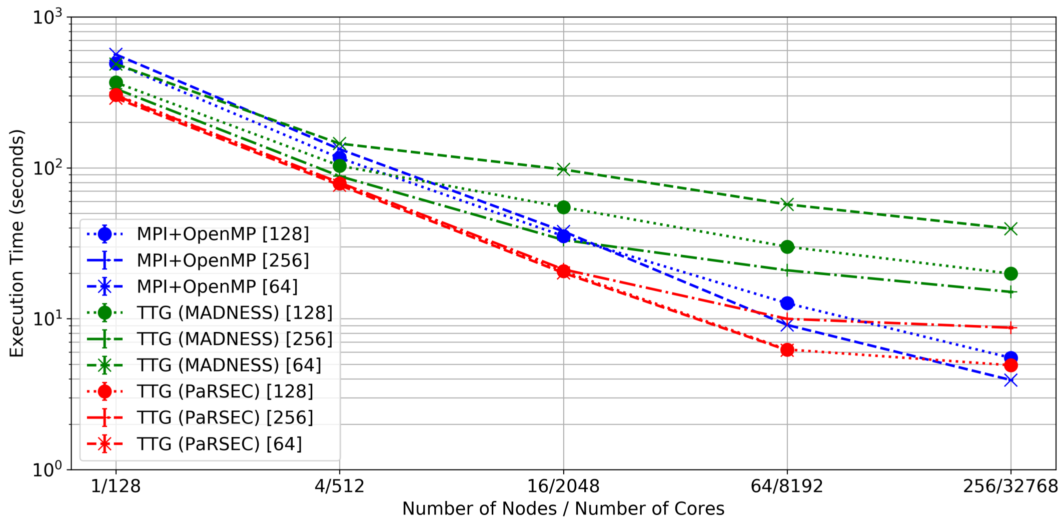


Figure 6.4. Strong scaling of the Floyd-Warshall benchmark using *TTG* and MPI+OpenMP on Hawk using 16 processes per node, 8 threads each (block sizes in square brackets).

Figure 6.5 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

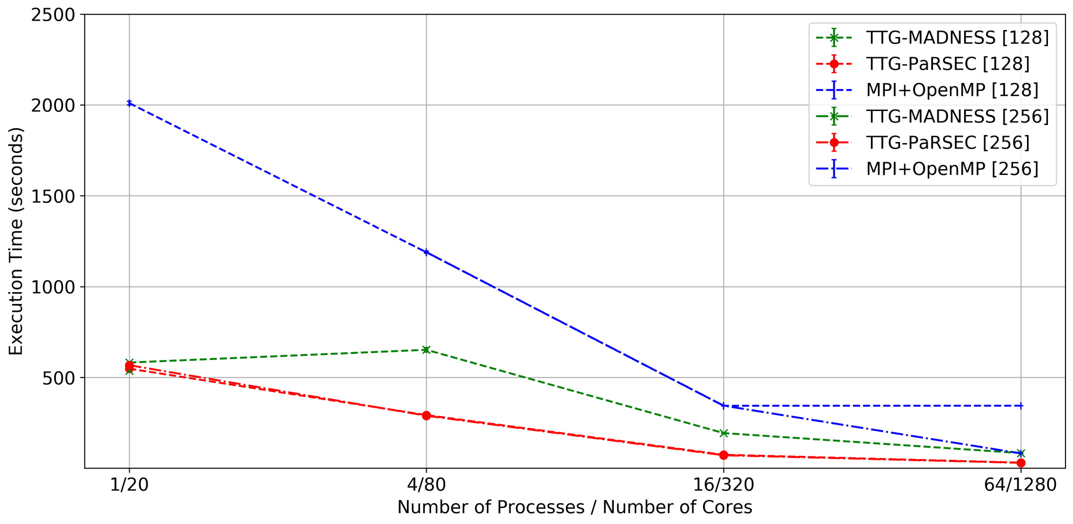


Figure 6.5. Strong scaling of the Floyd-Warshall benchmark using *TTG* and MPI+OpenMP on SeaWulf using 2 processes per node, 20 threads each (block sizes in square brackets).

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.

6.3.3 Multi-Resolution Analysis (MRA). This benchmark computes adaptively the order-10 multiwavelet [134, 135] 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 over-decomposition 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.

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. Listing 11 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 6.6 and 6.7 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. *TTG* over *PaRSEC* clearly outperforms *TTG* over *MADNESS* and native *MADNESS* on both

```
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 */
);
```

Listing 11. Accumulation of child nodes using a streaming terminal on input terminal 0 of the `reduce_leaves_tt` task template in the MRA benchmark.

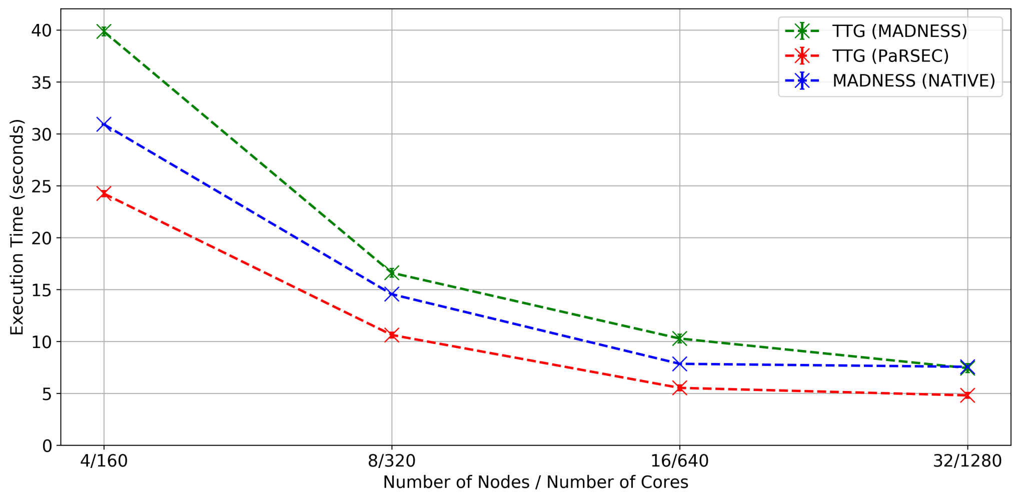


Figure 6.6. Strong scaling MRA: 4 to 32 nodes with 120 functions on *Seawulf*, using 2 processes per node with 20 threads each.

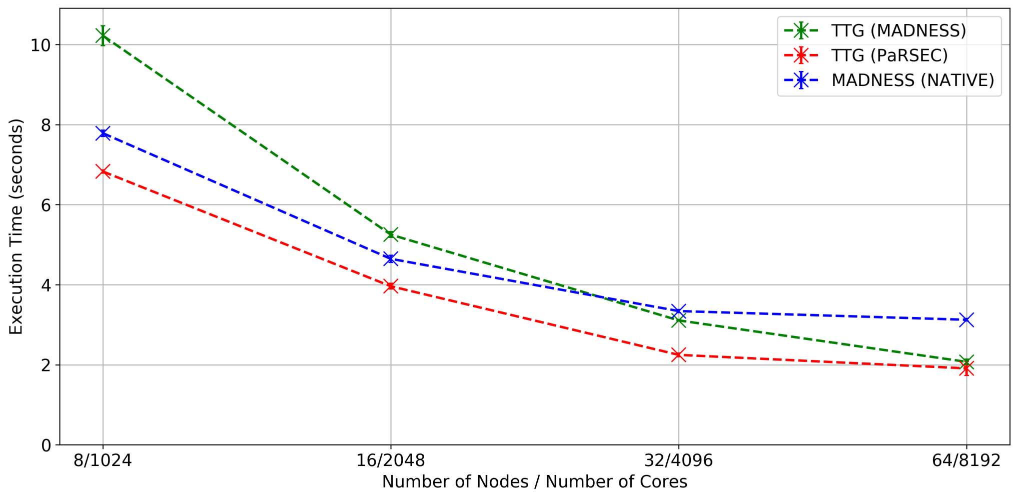


Figure 6.7. Strong scaling MRA: 8 to 64 nodes with 400 functions on *Hawk*, using 8 processes per node with 16 threads each.

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*.

6.4 Summary

Template Task Graph is an emerging 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. As a part of this work, we introduced streaming terminals, pull terminals and pure tasks functionality which lays the foundation for supporting heterogenous architectures using *TTG*. Our evaluations show high performance and scalability, on par and sometimes exceeding the performance of state of the art implementations in other programming paradigms.

CHAPTER 7
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.

7.1 Many Task Computing

In the recent years, the use of scheduler based on many-core or heterogeneous architectures for general or for specific applications has been widely studied [136, 137]. S. Yamagiwa et al. [136] propose a GPGPU streaming based on distributed computing environment; S. Nakagawa et al. [137] provide a new middleware capable of out-of-order execution of works and data transfers using stream processing. Other works [138, 139] follow a similar strategy based on streaming to minimize data transfers overhead. S. Kato et al. [140] 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 multitasking 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. [141] 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. [142] 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 of 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 [143, 144] 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. [145] and P. Valero-Lara et al. [146]. Posteriorly, C. Gregg et al. [147] and K. Zhang et al. [148] included a scheduler which can select the best matching among tasks before running. Additionally, P. Valero-Lara et al. [149] applied this strategy to different GPU architectures to obtain the most convenient architectural features for running concurrent kernels. After that, in [150], 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. [23] presented *GeMTC*, a CUDA based framework which allows MTC workloads to run efficiently on NVIDIA's GPUs. P. Nookala et al. [151] 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.

7.2 Concurrent queues

Several researchers have proposed concurrent queue implementations. Scogland et al. [152] 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. [71] 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) [74] 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 [75] is a lock-free queue that exploits batching to gain better performance. Morrison et al. [153] proposed a concurrent nonblocking linearizable FIFO queue using atomic FAA that outperforms CAS based implementations by up to 2×.

7.3 Parallel runtime systems

Most parallel runtime systems and execution models, such as OpenMP [50], Charm++ [154], and Swift/T [14], use concurrent queues for sharing data between threads or processes. OpenMP's task construct [155] 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 [156]. Recently, Cpp-taskflow [17] 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 [125], *StarPU* [157], HPX [158], CnC [127], *OmpSs* [159], DASH [160], *PaRSEC* [133] and *MADNESS* [131] 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 [12] 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 [161], 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.

7.4 Load Balancing

Several researchers have proposed various load balancing mechanisms [48, 49]. Blumofe and Leiserson et al. introduced work stealing and proved that it is superior to work sharing [68]. Quintin et al. proposed hierarchical work stealing for exploiting data locality to achieve speed up compared to classical work stealing algorithms [162]. 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 [66]. Several applications implement their own load balancing mechanisms in order to achieve ideal performance on various architectures. Unbalanced Tree Search benchmark [67] 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 [163]. 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.

7.5 Dataflow Programming

Sbirlea et al. introduced an intermediate graph representation for macro-dataflow programs (DFGR). It is an extension to the CnC model [164]. DFGR enables programmers to express programs at a high level with data-flow graphs as an intermediate representation. DFGR graphs consist of *step* nodes for computation and *item* nodes for data, which are partitioned into collections by unique tag. DFGR improves the efficiency by expressing what items are read and written to each step (through tag functions [165]). It is used as an abstraction to map the application for extreme-scale systems and run on heterogeneous architectures including GPUs/FPGAs, distributed-memory clusters, etc.

Later, they have proposed a polyhedral compiler framework, Data-Flow Graph Language (DFGL) which uses DFGR to represent dependencies [166]. The framework applies polyhedral analysis on dependencies to perform two important legality checks (single assignment rule and potential deadlocks) as well as applying automatic loop transformation, tiling, and code generation of parallel loops with coarse-grained and fine-grained synchronizations. DFGL framework compiles the input graph program into Habanero-C, which is an extension to C language built on top of CnC. The framework uses the ROSE compiler [167] to also generate OpenMP-4 compatible code, including task-level parallelism. They used Smith-Waterman, Cholesky factorization, Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) and some stencil kernels from PolyBench as their benchmarks. Their experimental results show that the DFGL versions optimized by their framework can deliver up to 6.9× performance improvement relative to OpenMP versions of the benchmarks.

There are several experimental studies that have been done illustrating performance and scalability of the CnC model. Chandramowlishwaran et al. [168] evaluated two dense linear algebra algorithms: (1) asynchronous-parallel Cholesky factorization and (2) "higher-level" partly-asynchronous generalized eigensolver for dense symmetric matrices. For both benchmarks, they showed that their CnC implementations match or exceed the Intel Math Kernel Library (MKL) implementation. They also compared their CnC implementations with other parallel models including ScaLAPACK with shared-memory MPI, OpenMP, Cilk+, and PLASMA 2.0, on Intel Harpertown, Nehalem, and AMD Barcelona systems. For the C++ implementation, Budimlić et al. [111] has used Dedup, a benchmark from PARSEC benchmark suite [169] and compared the performance of CnC implementation and pthread implementation. They showed that the CnC implementation outperforms the pthread implementation for two reasons. First, in the pthread implementation, the load imbalance exists between the stages of the computation. Second, the pthread implementation, unlike the CnC implementation, has data locality (to a thread) issue. They also considered Cholesky Factorization as another case study and showed speed-up with respect to increase in the number of threads. Liu and Kulkarni implemented the proxy application, LULESH in CnC model and have shown that with step fusion and tiling optimizations, the implementation outperforms the original implementation with good scalability (38× speed up) for up to 48 processor machines [170].

7.6 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 [171], CUDA graphs [172]; TensorFlow [173] and Dask [174] APIs support dataflow graphs; Intel TBB [175] includes support for both control flow and dataflow graphs; CnC [127] and Legion [125] 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 [176]. The S-NET DSL [177] is an orchestration language of tasks, strictly decoupling implementation and parallelism.

CHAPTER 8
CONCLUSION AND FUTURE WORK

In this work, we explored task-based parallel programming in shared-memory and distributed-memory environments. We presented approaches to achieve lightweight tasking in today's parallel runtime systems and designed X-OpenMP as a prototype. 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 High Performance Computing 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.

With emerging heterogeneous architectures, there exists a significant gap between programmer productivity and programmability today. It takes significant effort to write a single program that can execute seamlessly on CPUs, GPUs, FPGAs, etc. We presented *TTG* with a goal to bridge this gap of programmer productivity and programmability. We have extended prior work in *TTG* and added the ability for various types of terminals and pure tasks which serve as a foundation for supporting heterogenous architectures in the near future. Several

Our work in X-OpenMP opens up avenues for exploring lightweight tasking in several tasking runtimes and domains. We 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 Intel TBB [175], GNU OpenMP [51], as well as the Parsl parallel programming library [13] in order to further broaden the applications that could take advantage of the proposed techniques.

With respect to *TTG*, a big body of current work is about enabling *TTG* with *MADNESS* backend to run on GPUs. We are looking into implementing data structures for efficient execution on NVIDIA as well as AMD GPUs using Unified Memory as the initial target. Future work will also 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 options.

This high-risk/high-reward research is geared towards yielding transformative improvements in the ease and efficiency of programming parallel machines at every scale. This project has made contributions that realized productive, implicitly parallel high-level languages optimized for single node deployments with many-core architectures to support fine-grained parallelism measured in cycles. Scientist should be able to write a program once, run it at any suitable scale, and have it seamlessly use the most appropriate granularity for each component of the hardware. We anticipate that the innovations shown through this work is broadly applicable to improving existing parallel programming systems such as OpenMP, OneAPI, and Parsl, in terms of efficiency in executing fine grained parallelism. Target hardware included Intel/AMD x86, ThunderX/2 ARM, and IBM Power9. By enabling efficient support of fine-grained parallelism across the growing range of hardware and scales seen in modern and future architectures, we believe this work will enhance the productivity of parallel programmers for developing performance-portable applications as processor architectures push towards ever growing number of processor cores and hardware threads.

BIBLIOGRAPHY

[1] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2013.

[2] J. Schuchart, P. Nookala, M. M. Javanmard, T. Herault, E. F. Valeev, G. Bosilca, and R. J. Harrison, "Generalized flow-graph programming using template task-graphs: Initial implementation and assessment," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 839–849, IEEE, ©[IEEE. Reprinted, with permission, from Joseph Schuchart, Mohammad Mahdi, Thomas Herault, Edward F Valeev, George Bosilca, Robert J Harrison, Generalized Flow-Graph Programming Using Template Task-Graphs: Initial Implementation and Assessment, IEEE International Parallel and Distributed Processing Symposium 2022], 2022.

[3] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, and J. Dongarra, "DOE advanced scientific computing advisory subcommittee (ASCAC) report: top ten Exascale research challenges," tech. rep., USDOE Office of Science (SC)(United States), 2014.

[4] M. A. Heroux, L. McInnes, D. Bernholdt, A. Dubey, E. Gonsiorowski, O. Marques, J. D. Moulton, B. Norris, E. Raybourn, and S. Balay, "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.

[5] D. Geer, "Industry trends: Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[6] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[7] M. Själander, M. Martonosi, and S. Kaxiras, *Power-Efficient Computer Architectures: Recent Advances*. Synthesis Lectures on Computer Architecture, Morgan and Claypool Publishers, Dec. 2014.

[8] P. Valero-Lara, "Multi-gpu acceleration of dartel (early detection of alzheimer)," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pp. 346–354, Sept 2014.

[9] P. 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, (New York, NY, USA), pp. 187–192, ACM, 2013.

[10] P. Valero, J. L. Sánchez, D. Cazorla, and E. Arias, "A gpu-based implementation of the mrf algorithm in itk package," *The Journal of Supercomputing*, vol. 58, no. 3, pp. 403–410, 2011.

[11] TOP500.org, "TOP500 List June 2015."

[12] OpenMP Architecture Review Board, "OpenMP Application Programming Interface. Version 5.2.," tech. rep., Nov. 2021.

[13] Y. Babuji, A. Woodard, B. Clifford, Z. Li, D. S. Katz, R. Chard, R. Kumar, L. Lacinski, J. Wozniak, I. Foster, M. Wilde, and K. Chard., "Parsl: Pervasive parallel programming in python," in *HPDC'19*, (New York, NY, USA), ACM, June 2019.

[14] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. L. Lusk, and I. T. Foster, "Swift/t: scalable data flow programming for many-task applications," in *PPOPP'13*, 2013.

[15] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, S. Kenny, K. Iskra, P. Beckman, and I. Foster, "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers," *Journal of Physics: Conference Series*, vol. 180, p. 012046, Jul 2009.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *PLDI'98*, pp. 212–223, 1998.

[17] G. G. Tsung-Wei Huang, Chun-Xun Lin and M. Wong, "Cpp-Taskflow: Fast task-based parallel programming using modern c++," *IPDPS'19*, pp. 974–983, 2019.

[18] I. Raicu, I. T. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *2008 workshop on many-task computing on grids and supercomputers*, pp. 1–11, IEEE, 2008.

[19] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," in *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, (New York, NY, USA), pp. 18:1–18:15, ACM, 2008.

[20] P. Nookala, S. Dimitropoulos, K. Stough, and I. Raicu, "Evaluating the support of mtc applications on intel xeon phi many-core accelerators," in *2015 IEEE International Conference on Cluster Computing*, pp. 510–511, IEEE, ©[2015] IEEE. Reprinted, with permission, from [Serapheim Dimitropoulos, Karl Stough, Ioan Raicu, Evaluating the support of mtc applications on intel xeon phi many-core accelerators, IEEE International Conference on Cluster Computing 2015], 2015.

[21] P. Valero-Lara, P. Nookala, F. L. Pelayo, J. Jansson, S. Dimitropoulos, and I. Raicu, "Many-task computing on many-core architectures," *Scalable Computing: Practice and Experience*, vol. 17, no. 1, pp. 32–46, 2016.

[22] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, "Design and evaluation of the gemtc framework for gpu-enabled many-task computing," HPDC'14, 2014.

[23] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, "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, (New York, NY, USA), pp. 153–164, ACM, 2014.

[24] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04*, (San Francisco, CA), pp. 137–150, 2004.

[25] M. Zaharia, R. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *CACM*, vol. 59, pp. 56–65, 11 2016.

[26] K. Sato, C. Young, and D. Patterson, "An in-depth look at google's first tensor processing unit (tpu)," 2017.

[27] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, 1998.

[28] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The case for tiny tasks in compute clusters," in *HotOS'13*, 2013.

[29] D. R. Butenhof, *Programming with POSIX threads.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[30] M. Khiszinsky, "C++ library of lock-free containers and safe memory reclamation schema," 2006.

[31] I. Rickards, J. Donner, S. Vigna, W. Brown, and C. via the C Programming Forum, "Liblfds," 2009.

[32] S. A. Bahra., "Concurrency kit," 2011.

[33] H. Sutter, "Lock-free code: A false sense of security," 2008.

[34] H. Sutter., "Writing lock-free code: A corrected queue," 2008.

[35] H. Sutter., "Writing a generalized concurrent queue," 2008.

[36] H. Sutter., "The trouble with locks," 2005.

[37] R. Rodrigues and S. Bhogavilli, "Lockless queues," May 2012. Patent No. US8443375B2, Filed Dec 14th., 2009, Issued May. 14th., 2012.

[38] A. I. Orhean, A. Ballmer, T. Koehring, K. Hale, X.-H. Sun, O. Trigalo, N. Hardavellas, S. Kapoor, and I. Raicu, "Mystic: Programmable systems research testbed to explore a stack-wide adaptive system fabric," in *GCASR'19*, 2019.

[39] P. Nookala, P. Dinda, K. C. Hale, K. Chard, and I. Raicu, "Enabling extremely fine-grained parallelism via scalable concurrent queues on modern many-core architectures," in *29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8, IEEE, ©[2021] IEEE. Reprinted, with permission, from [Peter Dinda, Kyle C Hale, Kyle Chard, Ioan Raicu, Enabling Extremely Fine-grained Parallelism via Scalable Concurrent Queues on Modern Many-core Architectures, 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems 2021], 2021.

[40] P. Nookala, P. Dinda, K. C. Hale, K. Chard, and I. Raicu, "Xqueue - a lockless queueing mechanism for task-parallel runtime systems (under review)," in *Transactions on Emerging Topics in Computing (TETC)*, IEEE, 2023.

[41] "Intel® 64 and ia-32 architectures software developer's manual," 2018.

[42] P. Nookala and I. Raicu, "Xtask - extreme fine-grained concurrent task invocation runtime," in *Illinois Institute of Technology, Department of Computer Science, PhD Oral Qualifier*, 2017.

[43] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, P. Wu, I. Yamazaki, A. YarKhan, M. Abalenkovs, N. Bagherpour, *et al.*, "Plasma: Parallel linear algebra software for multicore using openmp," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 2, pp. 1–35, 2019.

[44] J. Wang, K. Zhang, X. Tang, and B. Hua, "B-queue: Efficient and practical queuing for fast core-to-core communication," *IJPP*, vol. 41, pp. 137–159, Feb 2013.

[45] K. Mitropoulou, V. Porpodas, X. Zhang, and T. M. Jones, "Lynx: Using os and hardware support for fast fine-grained inter-core communication," in *ICS'16*, 2016.

[46] X. Meng, X. Zeng, X. Chen, and X. Ye, "A cache-friendly concurrent lock-free queue for efficient inter-core communication," in *ICCSN'17*, IEEE, 2017.

[47] S. Arnautov, P. Felber, C. Fetzer, and B. Trach, "Ffq: A fast single-producer/multiple-consumer concurrent fifo queue," IEEE, 2017.

[48] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing.," in *SC'09*, 2009.

[49] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "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.

[50] O. A. R. Board, "Openmp®: Support for the openmp language."

[51] G. team, "Gomp: An openmp implementation for gcc."

[52] A. Podobas, M. Brorsson, and V. Vlassov, "Scheduling for improved data-driven task performance with fast dependency resolution," in *IWOMP'14*, (Salvador, Brazil), pp. 45–57, Springer, September 2014.

[53] E. W. Dijkstra, "Solution of a problem in concurrent programming control," in *CACM 1965*, vol. 8, p. 569, 1965.

[54] J. D. McCalpin *et al.*, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19-25, 1995.

[55] "Intel® memory latency checker v3.9a." https://www.intel.com/content/www/us/en/deve
memory-latency-checker.html. Accessed: 2022-10-25.

[56] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade´, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp.," in *ICPP'09*, pp. 124–131, 2009.

[57] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *SC'12*, pp. 1–10, Nov 2012.

[58] S. R. Das and R. M. Fujimoto, “A performance study of the cancelback protocol for time warp,” in *SIGSIM Simul.*, vol. 23, pp. 135–142, 1993.

[59] P. Nookala, Z. Ahmad, M. M. Javanmard, M. Kong, R. Chowdhury, and R. Harrison, “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)*, pp. 407–416, IEEE, ©[2015] IEEE. Reprinted, with permission, from [Ahmad, Zafar and Javanmard, Mohammad Mahdi and Kong, Martin and Chowdhury, Rezaul and Harrison, Robert, Understanding Recursive Divide-and-Conquer Dynamic Programs in Fork-Join and Data-Flow Execution Models, IEEE International Parallel and Distributed Processing Symposium Workshops 2021], 2021.

[60] I. S. Duff, M. A. Heroux, and R. Pozo, “An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 239–267, 2002.

[61] C. Lehman, P. Nookala, and I. Raicu, “Scalable load-balancing concurrent queues in modern many-core architectures,” in *SC19*, (Denver, CO), ACM, 2019.

[62] U. A. Acar, A. Charguéraud, S. Muller, and M. Rainey, “Atomic Read-Modify-Write Operations are Unnecessary for Shared-Memory Work Stealing,” research report, Sept. 2013.

[63] P. Nookala, P. Dinda, K. C. Hale, K. Chard, and I. Raicu, “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)*, pp. 1–8, IEEE, 2021.

[64] P. Nookala, K. Chard, and I. Raicu, “X-openmp – extreme fine-grained tasking using lock-less work stealing (under review),” in *Transactions on Parallel and Distributed Systems (TPDS)*, IEEE, 2023.

[65] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, “Thread scheduling for multiprogrammed multiprocessors,” *Theory of computing systems*, vol. 34, no. 2, pp. 115–144, 2001.

[66] M. Mitzenmacher, “The power of two choices in randomized load balancing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[67] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, “Uts: An unbalanced tree search benchmark,” in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 235–250, Springer, 2006.

[68] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[69] D. Tudor, R. Guerraoui, and V. Trigonakis, “Everything you always wanted to know about synchronization but were afraid to ask.,” in *SOSP’13*, 2013.

[70] M. L. S. John M. Mellor-Crummey, “Algorithms for scalable synchronization on shared-memory multiprocessors,” in *TOCS’91*, 1991.

[71] H. Schweizer, M. Besta, and T. Hoefler, “Evaluating the cost of atomic operations on modern architectures,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 445–456, IEEE, 2015.

[72] M. Chabbi, A. Amer, S. Wen, and X. Liu, “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*, pp. 61–74, 2017.

[73] M. M. Michael and M. L. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” in *PODC’96*, 1996.

[74] X. Meng, X. Zeng, X. Chen, and X. Ye, “A cache-friendly concurrent lock-free queue for efficient inter-core communication,” in *ICCSN’17*, 2017.

[75] G. Milman, A. Kogan, Y. Lev, V. Luchangco, and E. Petrank, “Bq: A lock-free queue with batching,” in *SPAA’18*, pp. 99–109, 2018.

[76] H. Sutter, “The trouble with locks,” 2005.

[77] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, “Exploring the design tradeoffs for extreme-scale high-performance computing system software,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 1070–1084, 2015.

[78] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, “Introduction to upc and language specification,” Tech. Rep. CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[79] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, “x86-tso: a rigorous and usable programmer’s model for x86 multiprocessors,” *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, 2010.

[80] B.-J. Kwak, N.-O. Song, and L. E. Miller, “Performance analysis of exponential backoff,” *IEEE/ACM transactions on networking*, vol. 13, no. 2, pp. 343–355, 2005.

[81] J. M. Bull, F. Reid, and N. McDonnell, “A microbenchmark suite for openmp tasks,” in *International Workshop on OpenMP*, pp. 271–274, Springer, 2012.

[82] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull, “Implementation of strassen’s algorithm for matrix multiplication,” in *Supercomputing’96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pp. 32–32, IEEE, 1996.

[83] J. A. Calvin, C. A. Lewis, and E. 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*, pp. 1–8, 2015.

[84] R. Bellman, “The theory of dynamic programming,” tech. rep., Rand corp santa monica ca, 1954.

[85] S. S. Skiena, *The algorithm design manual: Text*, vol. 1. Springer Science & Business Media, 1998.

[86] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[87] V. T. Paschos, *Concepts of Combinatorial Optimization, Volume 1.* John Wiley & Sons, 2012.

[88] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge university press, 1997.

[89] J. Klepeis, M. Ierapetritou, and C. Floudas, "Protein folding and peptide docking: A molecular modeling and global optimization approach," *Computers & chemical engineering*, vol. 22, pp. S3–S10, 1998.

[90] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A blocked all-pairs shortest-paths algorithm," *Journal of Experimental Algorithmics (JEA)*, vol. 8, pp. 2–2, 2003.

[91] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *PLDI'1991*, pp. 30–44, 1991.

[92] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM TOPLAS*, vol. 18, no. 4, pp. 424–453, 1996.

[93] F. Irigoin and R. Triolet, "Supernode partitioning," in *POPL'1988*, pp. 319–329, 1988.

[94] R. Chowdhury, P. Ganapathi, J. J. Tithi, C. Bachmeier, B. C. Kuszmaul, C. E. Leiserson, A. Solar-Lezama, and Y. Tang, "Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs," *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–12, 2016.

[95] R. A. Chowdhury and V. Ramachandran, "Cache-oblivious dynamic programming," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pp. 591–600, 2006.

[96] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS'1999*, IEEE, 1999.

[97] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley, "Cache-adaptive algorithms," in *SODA'14*, 2014.

[98] M. M. Javanmard, P. Ganapathi, R. Das, Z. Ahmad, S. Tschudi, and R. Chowdhury, "Toward efficient architecture-independent algorithms for dynamic programs," in *International Conference on High Performance Computing*, Springer, 2019.

[99] M. M. Javanmard, Z. Ahmad, M. Kong, L.-N. Pouchet, R. Chowdhury, and R. Harrison, "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*, pp. 317–329, 2020.

[100] M. M. Javanmard, Z. Ahmad, J. Zola, L.-N. Pouchet, R. Chowdhury, and R. Harrison, "Efficient execution of dynamic programming algorithms on apache spark," in *CLUSTER'2020*, pp. 337–348, IEEE, 2020.

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

[102] M. M. Javanmard, P. Ganapathr, R. Das, Z. Ahmad, S. Tschudi, and R. Chowdhury, "Toward efficient architecture-independent algorithms for dynamic programs: poster," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pp. 413–414, 2019.

[103] R. Chowdhury, P. Ganapathi, Y. Tang, and J. J. Tithi, "Provably efficient scheduling of cache-oblivious wavefront algorithms," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 339–350, 2017.

[104] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury, "Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 205–214, 2015.

[105] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE, 2012.

[106] M. Kong, L.-N. Pouchet, P. Sadayappan, and V. Sarkar, "Pipes: a language and compiler for task-based programming on distributed-memory clusters," in *SC'2016*, pp. 456–467, IEEE, 2016.

[107] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[108] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: the new adventures of old x10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pp. 51–61, 2011.

[109] G. Bosilca, D. Genet, R. J. Harrison, T. Herault, M. M. Javanmard, S. Brook, C. Peng, and E. Valeev, "Tensor contraction on distributed hybrid architectures using a task-based runtime system," 2018.

[110] A. Pop and A. Cohen, "Openstream: Expressiveness and data-flow compilation of openmp streaming programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–25, 2013.

[111] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[112] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[113] G. Bosilca, R. J. Harrison, T. Herault, M. M. Javanmard, P. Nookala, and E. F. Valeev, "The template task graph (ttg)-an emerging practical dataflow programming paradigm for scientific simulation at extreme scale," in *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pp. 1–7, IEEE, ©[IEEE. Reprinted, with permission, from George Bosilca, Robert J Harrison, Thomas Herault, Mohammad Mahdi Javanmard, Edward F Valeev, The Template Task Graph (TTG)-an

emerging practical dataflow programming paradigm for scientific simulation at extreme scale, IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2) 2020], 2020.

[114] "Intel (r) concurrent collections for c/c++." https://icnc.github.io/.

[115] K. Knobe and C. D. Offner, "Tstreams: A model of parallel computation (preliminary report)," tech. rep., Technical Report HPL-2004-78, HP Labs, 2004.

[116] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari, "Multi-core implementations of the concurrent collections programming model," in *CPC'09: 14th International Workshop on Compilers for Parallel Computers*, 2009.

[117] K. Knobe and M. G. Burke, "The tuning language for concurrent collections," in *16th Workshop on Compilers for Parallel Computing*, 2012.

[118] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[119] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg, "Mcrt-malloc: a scalable transactional memory allocator," in *Proceedings of the 5th international symposium on Memory management*, pp. 74–83, 2006.

[120] R. A. Chowdhury and V. Ramachandran, "The cache-oblivious Gaussian Elimination Paradigm: theoretical framework, parallelization and experimental evaluation," *TCS*, vol. 47, no. 4, pp. 878–919, 2010.

[121] T. F. Smith, M. S. Waterman, *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[122] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[123] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *IISWC'2013*, pp. 56–65, IEEE, 2013.

[124] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, pp. 157–173, Springer, 2010.

[125] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Supercomputing*, 2012.

[126] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An abstraction for unhindered parallelism," *Proceedings of WOLFHPC'14*, pp. 21–30, 2014.

[127] Z. Budimlić and K. Knobe, "CnC: A Dependence Programming Model," in *Proceedings of the Sixth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, DFM'16, ACM, 2016.

[128] J. Schuchart, P. Nookala, T. Herault, E. F. Valeev, and G. Bosilca, "Pushing the boundaries of small tasks: Scalable low-overhead data-flow programming in ttg," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 117–128, ©[IEEE. Reprinted, with permission, from Joseph Schuchart, Thomas Herault, Edward F Valeev, George Bosilca, Pushing the Boundaries of Small Tasks: Scalable Low-Overhead Data-Flow Programming in TTG, IEEE International Conference on Cluster Computing (CLUSTER) 2022], 2022.

[129] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[130] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "Ptg: an abstraction for unhindered parallelism," in *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pp. 21–30, IEEE, 2014.

[131] R. J. Harrison, G. Beylkin, F. A. Bischoff, J. A. Calvin, G. I. Fann, J. Fosso-Tande, D. Galindo, J. R. Hammond, R. Hartman-Baker, J. C. Hill, J. Jia, J. S. Kottmann, M. Y. Ou, L. E. Ratcliff, M. G. Reuter, A. C. Richie-Halford, N. A. Romero, H. Sekino, W. A. Shelton, B. E. Sundahl, W. S. Thornton, E. F. Valeev, Á. Vázquez-Mayagoitia, N. Vence, and Y. Yokoi, "MADNESS: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S123–S142, 2016.

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

[133] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Comp in Sc. and Eng.*, vol. 99, p. 1, 2013.

[134] B. Alpert, G. Beylkin, D. Gines, and L. Vozovoi, "Adaptive Solution of Partial Differential Equations in Multiwavelet Bases," *Journal of Computational Physics*, vol. 182, no. 1, pp. 149–190, 2002.

[135] B. Alpert, *Sparse Representation of Smooth Linear Operators.* PhD thesis, Yale University, 1990.

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

[137] S. Nakagawa, F. Ino, and K. Hagihara, "A middleware for efficient stream processing in cuda," *Computer Science - Research and Development*, vol. 16, pp. 197–204, 2010.

[138] J. Gómez-Luna, J. González-Linares, J. I. Benavides, and N. Guil, "Performance models for cuda streams on nvidia geforce series," *J. Parallel Distrib. Comput.*, vol. 72, no. 9, pp. 1117–1126, 2012.

[139] B. van Werkhoven, J. Maassen, F. Seinstra, and H. Bal, "Performance models for cpu-gpu data transfers," *CCGRID*, pp. 11–20, 2014.

[140] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," *In Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)*, 2011.

[141] J. Duato, J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti, "Performance of cuda virtualized remote gpus in high performance cluster," *the 40st International Conference on Parallel Processing (ICPP)*, pp. 365–374, 2011.

[142] V. J. Jiménez, L. Vilanova, I. Gelado, G. F. M. Gil, and N. Navarro, "Predictive runtime code scheduling for heterogeneous architectures," *the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, pp. 19–33, 2009.

[143] J. Lima, T. Gautier, N. Maillard, and V. Danjean, "Exploiting concurrent gpu operations for efficient work stealing on multi-gpus," *24rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 75–82, 2012.

[144] J. Kreutz, "Dgemm-tiled matrix multiplication with cuda," *Jülich Forchungszentrum*, 2013.

[145] M. A. Guevera, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling task parallelism in the cuda scheduler," *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.

[146] P. Valero-Lara and F. L. Pelayo, "Towards a more efficient use of gpus," *Computational Science and Its Applications (ICCSA) Workshops*, pp. 3–9, 2011.

[147] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent gpgpu kernels," *In Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2012.

[148] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the gpu," *SIGPLAN Not.*, vol. 45, pp. 127–136, Jan. 2010.

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

[150] P. Valero-Lara and F. L. Pelayo, "Full-overlapped concurrent kernels," in *Architecture of Computing Systems. Proceedings, ARCS 2015-The 28th International Conference on*, pp. 1–8, VDE, 2015.

[151] P. Nookala, S. Dimitropoulos, K. Stough, and I. Raicu, "Evaluating the support of mtc applications on intel xeon phi many-core accelerators," in *International Conference on Cluster Computing*, CLUSTER '15, 2015.

[152] T. R. Scogland and W.-c. Feng, "Design and evaluation of scalable concurrent queues for many-core architectures," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pp. 63–74, 2015.

[153] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *PPOPP'13*, (New York, NY, USA), pp. 103–112, PPoPP, 2013.

[154] L. Kalé and S. Krishnan, “Charm++: A portable concurrent object oriented system based on c++,” in *OOPSLA'93*, 1993.

[155] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, “The design of openmp tasks,” in *TPDS'09*, vol. 20, IEEE, 2009.

[156] C. Mei, G. Zheng, F. Gioachin, and L. V. Kalé, “Optimizing a parallel runtime system for multicore clusters: A case study,” in *TeraGrid'10*, 2010.

[157] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, “StarPU: A unified platform for task scheduling on heterogeneous multicore architectures,” *Conc. Comp. Pract. Exper.*, vol. 23, pp. 187–198, 2011.

[158] T. Heller, H. Kaiser, and K. Iglberger, “Application of the ParalleX execution model to stencil-based problems,” *Computer Science - Research and Development*, vol. 28, no. 2-3, pp. 253–261, 2013.

[159] A. Duran, R. Ferrer, E. Ayguade, R. M. Badia, and J. Labarta, “A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks,” *Intl. Journal of Parallel Programming*, vol. 37, no. 3, pp. 292–305, 2009.

[160] J. Schuchart and J. Gracia, “Global Task Data-Dependencies in PGAS Applications,” in *High Performance Computing*, Springer International Publishing, 2019.

[161] S. J. Treichler, *Realm: Performance Portability through Composable Asynchrony.* PhD thesis, Stanford University, 2014.

[162] J.-N. Quintin and F. Wagner, “Hierarchical work-stealing,” in *European Conference on Parallel Processing*, pp. 217–229, Springer, 2010.

[163] S. Shiina and K. Taura, “Almost deterministic work stealing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2019.

[164] A. Sbirlea, L.-N. Pouchet, and V. Sarkar, “Dfgr an intermediate graph representation for macro-dataflow programs,” in *2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, pp. 38–45, IEEE, 2014.

[165] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, “Mapping a data-flow programming model onto heterogeneous platforms,” *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 61–70, 2012.

[166] A. Sbîrlea, J. Shirako, L.-N. Pouchet, and V. Sarkar, “Polyhedral optimizations for a data-flow graph language,” in *Languages and Compilers for Parallel Computing*, pp. 57–72, Springer, 2015.

[167] “The pace compiler project.” http://pace.rice.edu/.

[168] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, “Performance evaluation of concurrent collections on high-performance multicore computing systems,” in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, IEEE, 2010.

[169] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81, 2008.

[170] C. Liu and M. Kulkarni, "Optimizing the lulesh stencil code using concurrent collections," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pp. 1–10, 2015.

[171] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," *IEEE TPDS*, pp. 1–1, 2021.

[172] "CUDA programming guide - CUDA graphs." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmlcuda-graphs, 2021.

[173] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[174] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th python in science conference*, vol. 130, p. 136, 2015.

[175] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in intel threading building blocks.," *Intel Technology Journal*, vol. 11, no. 4, 2007.

[176] K. Hammond and G. 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, (Berlin, Heidelberg), p. 37–56, Springer-Verlag, 2003.

[177] C. Grelck, J. Julku, and F. 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.