Enabling Extremely Fine-grained Parallelism via Scalable Concurrent Queues on Modern Many-core Architectures

1st Poornima Nookala
Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois, United States
pnookala@hawk.iit.edu

2nd Peter Dinda
Department of Computer Science
Northwestern University
Evanston, Illinois, United States
pdinda@northwestern.edu

3rd Kyle C. Hale
Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois, United States
khale@cs.iit.edu

4th Kyle Chard
Department of Computer Science
University of Chicago
Chicago, Illinois, United States
chard@uchicago.edu

5th Ioan Raicu
Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois, United States
iraicu@cs.iit.edu

***Abstract*—Enabling efficient fine-grained task parallelism is a significant challenge for hardware platforms with increasingly many cores. Existing techniques do not scale to hundreds of threads due to the high cost of synchronization in concurrent data structures. To overcome these limitations we present XQueue, a novel lock-less concurrent queuing system with relaxed ordering semantics that is geared towards realizing scalability up to hundreds of concurrent threads. We demonstrate the scalability of XQueue using microbenchmarks and show that XQueue can deliver concurrent operations with latencies as low as 110 cycles at scales of up to 192 cores (up to 6900× improvement compared to traditional synchronization mechanisms) across our diverse hardware, including x86, ARM, and Power9. The reduced latency allows XQueue to provide orders of magnitude (3300×) better throughput that existing techniques. To evaluate the real-world benefits of XQueue, we integrated XQueue with LLVM OpenMP and evaluated five unmodified benchmarks from the Barcelona OpenMP Task Suite (BOTS) as well as a graph traversal benchmark from the GAP benchmark suite. We compared the XQueue-enabled LLVM OpenMP implementation with the native LLVM and GNU OpenMP versions. Using fine-grained task workloads, XQueue can deliver 4× to 6× speedup compared to native GNU OpenMP and LLVM OpenMP in many cases, with speedups as high as 116× in some cases.**

***Index Terms*—concurrent data structures, fine-grained parallelism, lock-free, lock-less, queues, parallel runtime, tasks**

I. Introduction

Task parallelism is an increasingly important class of parallelism in which computation is broken into a set of inter-dependent tasks which may be executed concurrently on various cores. The execution models of many parallel languages and libraries [1]–[6] rely on such task parallelism. For example, OpenMP [1] has evolved to a task-centric model to enable parallelization of applications where units of work is generated dynamically. When a task is created by some thread, it is conceptually queued for execution by a future available thread. Software dataflow languages [2], [3] similarly include a runtime that executes a dynamically unfolding task graph in which tasks are scheduled via concurrent queues. To achieve strong scaling and high levels of parallelism, today's parallel languages and execution models are moving to tasks with finer granularity. One reason for this is that as core counts per node increase, applications need to support over-decomposition in order to improve performance, hide latency caused by blocking operations, and achieve maximum speedup. This and other drivers produce the same outcome: tasks and their dependencies need to be managed at sub-microsecond timescales.

Queues are an integral component of tasking runtime systems and as task granularity decreases, execution performance is increasingly dependent on queue performance. 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). Threads running concurrently 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, tail and data) be protected with a synchronization mechanisms, such as mutual exclusion locks (mutexes), spinlocks, semaphores, or atomic primitives.

A second approach to concurrent queues is to avoid separate synchronization by incorporating race-avoidance directly into the data structure itself. This also has the benefit of avoiding common concurrency bugs (e.g., deadlocks) due to misuse

This work was supported in part by the National Science Foundation (NSF) under grants 2107548/2107283, CCF-1757964, CNS-1730689, CNS-1763612, CNS-1718252, CCF-2028958, CCF-2028851, CNS-1763743 and CCF-2119069.

of synchronization primitives. ***Lock-free*** data structures use atomic primitives, such as Compare-and-Swap (CAS) and Fetch-and-Add (FAA), to push the burden down to hardware and achieve synchronization at a finer granularity. Several libraries internally use lock-free techniques [7]–[9], but the literature has shown that it is difficult to write correct lock-free code [10]. Even more compelling are ***lock-less*** data structures [11], 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 to instruction and memory access reordering imposed by the compiler and the hardware, and the need to account for the memory consistency model supported by both.

In order to address the aforementioned issues, we design, implement, and evaluate XQueue, a novel lock-less concurrent queuing system with relaxed ordering semantics. XQueue is not a general-purpose MPMC queue, but rather a task queuing system aimed to improve the task management overhead in parallel runtime systems.

We make the following contributions:

1) We design and implement XQueue, 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.
2) We integrate XQueue into LLVM OpenMP and evaluate the performance improvements on 5 unmodified applications (Fib, FFT, Multisort, NQueens, and Health) from the Barcelona OpenMP Task Suite (BOTS) as well as the breadth first search (BFS) application from the GAP benchmark suite. 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.

II. Motivation

Concurrent data structures have to deal with data synchronization and communication between threads. Synchronization mechanisms like mutexes, semaphores, and spinlocks are known to have significant overhead and can easily become the bottleneck to achieving high performance.

An SPSC array-based queue provides the lowest latency the microarchitecture can provide for enqueue and dequeue operations when both operations do not occur simultaneously since they do not require data synchronization or thread-to-thread communication and can benefit from data locality. A popular approach to implement parallelism in applications is to use concurrent queues for sharing work among various threads; MPMC queue is the most commonly used data structure in such cases. Thread contention on shared data, synchronization overheads, cache coherence effects, and cache misses are some of the many factors that can significantly impact the performance of MPMC queues and limit scalability. In order to show the scalability and performance of MPMC queues compared to SPSC queues, we selected five diverse systems (see Table I) from the Mystic Testbed [12] that represent different architectures with large core counts. We conducted experiments on Ubuntu 18.04.3 and compiled using LLVM Clang version 11.0.0 with O3 optimization level and $-march = native$.

TABLE I: 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

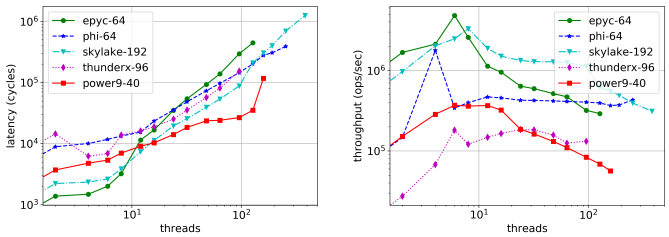
We measure the latency and throughput of a simple SPSC array-based circular queue to identify baseline performance on the latest many-core architectures. We run 1 billion enqueue operations followed by a sequence of dequeues. We measure the latency of each operation and calculate the average time per enqueue/dequeue pair. For throughput experiments, we measure the total time taken for 1 billion enqueue/dequeue operations and calculate the throughput. Results in Table II show the average latency and throughput of both enqueue and dequeue operations. We see that the latency of any operation on queues takes between 29 and 68 cycles depending on the architecture and clock frequency. Average throughput reaches 270 million ops/sec on skylake-192 machine. Although these results are significant, an SPSC queue is limited in parallel runtime systems because it cannot alone be used to implement parallelism and concurrency.

TABLE II: Average latency and throughput of enqueue/dequeue operations on SPSC queue

Machine	Latency (cycles)	Throughput (ops/sec)
skylake-192	41	270M
epyc-64	47	155M
phi-64	68	26M
thunderx-96	36	58M
power9-40	29	36M

We measure the latency and throughput of an MPMC queue by implementing the queue using a semaphore which tracks free spaces in the queue and uses `pthread_mutex_lock` to lock the queue during enqueue and dequeue operations. This experiment aims to quantify the poor scalability of MPMC queues using mutex locks. Each experiment enqueues and dequeues 1 billion items using an equal number of producer and consumer threads. In all experiments, we use a round-robin pinning of threads, with producer and consumer threads being on the same core (but distinct hyperthreads) which

can result in better cache utilization, thereby reducing costly memory accesses.



(a) Latency (b) Throughput

Fig. 1: Performance of MPMC queue operations

Figures 1a and 1b 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.

We were not surprised by these findings, as the fundamental problem stems from the cost of synchronization. In prior work, we studied the cost of synchronization mechanisms from low to high levels of concurrency, and found that none of these mechanisms offered scalable solutions beyond single digit concurrent threads [13]. Use of such concurrent data structures in modern parallel runtimes have significant overheads for managing extremely fine-grained tasks. For example, when computing the 44th Fibonacci number recursively using LLVM OpenMP, the runtime overhead dominates the overall execution time by consuming over 90% of CPU time for synchronization and scheduling. These findings motivated our investigation into methods to eliminate synchronization mechanisms in order to unleash the full performance of many-core architectures under high concurrency.

III. XQueue- Scalable concurrent queues on modern many-core architectures

XQueue is motivated in large part by the significant latency gap observed with SPSC and MPMC models (Section II).

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

We introduce XQueue, a novel lock-less MPMC, out-of-order queuing mechanism that can scale up to hundreds of threads. XQueue uses B-queue [14] 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 [15]–[17] and we aim to demonstrate that XQueue can be built with any fast and scalable SPSC queue.

Figure 2 shows the architectural of XQueue on a 4-core system. The key idea here is to have N SPSC concurrent queues per worker if there are N workers. There is one master queue and $N-1$ auxiliary queues per worker, with N (equal to number of workers) producers adding items into master queues. Every item is a void pointer that represents a task where a task could be a function pointer or data pointer. One worker exists for dequeueing tasks from the master queue as well as the auxiliary queues. A worker first tries to dequeue a task from the master queue. If a task is dequeued successfully, it is processed immediately. The item when processed can generate one or more items to be enqueued into the auxiliary queues of the other CPU cores. Every worker distributes work to auxiliary queues in a round-robin fashion as shown in Figure 2. A worker then tries to dequeue an item from its auxiliary queues and dequeued items are processed immediately.

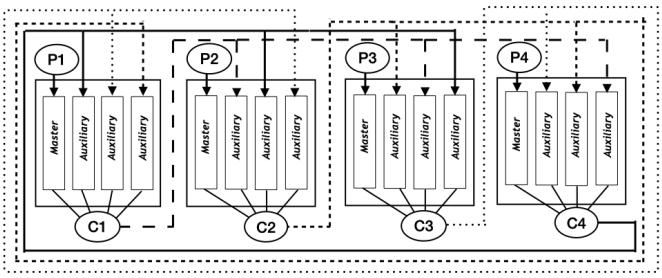


Fig. 2: Architecture of XQueue on a 4-core machine with 4 queues per consumer.

A simplified version of pseudocode for worker logic is outlined in Algorithm 1. Since all queues in XQueue are concurrent SPSC queues, producer and consumer threads can act concurrently processing items in the queues. The strategy of distributing work across queues (as shown in Figure 2) 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.

A. 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 [18], [19]. While XQueue also uses multiple queues, it balances load by the virtue of its design with N

```
Data: id ← coreId; next ← nextCoreId;
while 1 do
    ret ← dequeueFromMaster(id, item);
    if ret = SUCCESS then
        retItem ← processItem(item);
        if retItem ≠ NULL then
            enqueueToAuxiliary(next, retItem);
        end
    end
    ret ← dequeueFromAuxiliary(id, item);
    if ret = SUCCESS then
        retItem ← processItem(item);
        if retItem ≠ NULL then
            enqueueToAuxiliary(next, retItem);
        end
    end
    next ← (next + 1)%numCores;
    if next == id then
        next++;                                  /* do not enqueue to self */
    end
end
```

Algorithm 1: Worker logic.

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 2, 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 IV.

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.

B. XQueue Integration with the OpenMP Runtime

In order to extend our research to real systems, we integrated XQueue into OpenMP [20] 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) [21], LLVM OpenMP [20], 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 [22].

Implementation: In the LLVM OpenMP tasking implementation, every thread owns a queue and the enqueue/dequeue operations are protected by locks implemented using Lamport's bakery algorithm. We replaced the task queues in OpenMP with multiple SPSC queues per worker to model XQueue. OpenMP implements a work-stealing scheduler. Every thread first checks it's own queue for tasks. If no tasks are found, a thread is randomly chosen to steal a single task. We replaced the work stealing scheduler with the scheduler for XQueue as shown in Algorithm 1. In our XQueue-based OpenMP implementation, every thread checks it's own queue for tasks. If no tasks are found, the scheduler checks all auxiliary queues. This process of checking the master queue and auxiliary queues is repeated until a termination condition is satisfied.

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

IV. Performance Evaluation

We evaluate the performance of XQueue using synthetic and real workloads. For the purposes of evaluating XQueue inde-

pendently, 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 I); 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.

A. 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 [5] with a separate queue per worker.

Cilk deque is implemented as part of Cilk 5 multi-threaded language [5] and uses a shared-memory, mutual-exclusion protocol called the THE protocol [23] 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.

B. 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 3a 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 3b compares the latency of XQueue (SPSC) with Cilk Deque and MPMC queues on skylake-192. Here, Cilk Deque/MPMC is a single queue shared across all the workers. With 192 producers/consumers, latency of MPMC queue is 13× 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 3c is a log-log plot showing the throughput of XQueue using lock-based and lock-less queues on the skylake-192 system. The throughput achieved on this system with XQueue with lock-less queue is 1 billion operations per second with all hyper threads being utilized. For XQueue using lock-based queue, the average throughput achieved is 200 million operations per second and 397 million for the Cilk deque. In the case of MPMC queue, each mutex lock is held for short intervals and contention is low, but acquiring the lock has a cost which explains the 5× 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× gap), although the cost is lower compared to mutex-based locks. As noted in Section II for MPMC queue, with high contention on the mutex lock with more than 8 threads, throughput drops to about 300K operations per second on skylake-192 with 384 threads. In case of Cilk deque, the throughput drops to 4 million operations per second. This clearly shows a 3300X gap in throughput between XQueue with lock-less queue and single lock-based queue with hundreds of threads.

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

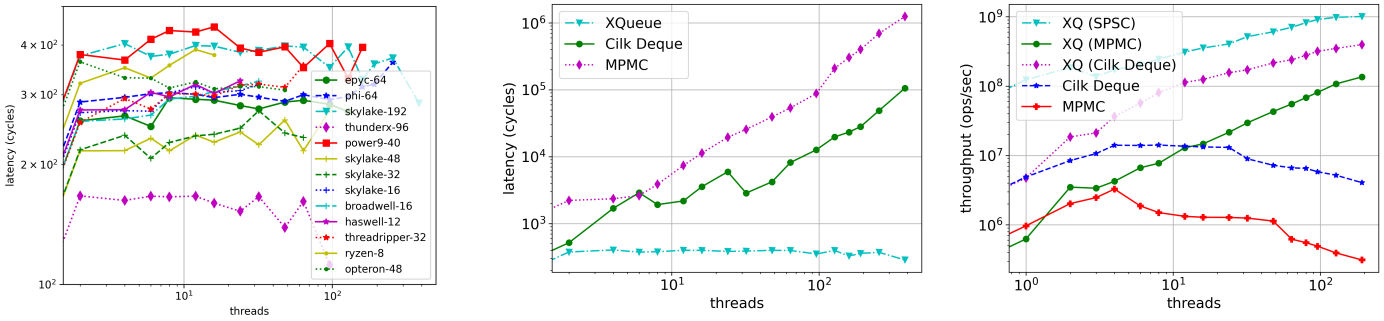
C. Macro-benchmark Performance Results

To quantify the improvements in real application workloads, we evaluate the speedup achieved using XQueue-enabled LLVM OpenMP as compared to the native LLVM OpenMP and GNU OpenMP libraries. We evaluate five out of nine applications from the BOTS benchmark suite [24]: Fibonacci, FFT (Fast Fourier Transform), Multisort, NQueens and Health. Results are shown in Figure 4. We also evaluate the breadth first search application from the GAP benchmark suite [25] with real-world social network graphs such as those from Friendster and Twitter. Results are shown in Figure 6. The application workloads are summarized in Table III.

TABLE III: Application - number of tasks

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

Fibonacci (Fib) computes the Nth Fibonacci number using recursive parallelism. While Fib is hardly a critical parallel application, it *does* have extremely fine-grained tasks (e.g., addition of two numbers) with extremely large number of tasks, and thus exposes the limits of a tasking runtime in



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

(b) Latency Comparison

(c) Throughput Comparison

Fig. 3: XQueue Performance on Skylake-192

terms of granularity. Figure 4 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 4, 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 [26]. 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 5 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.

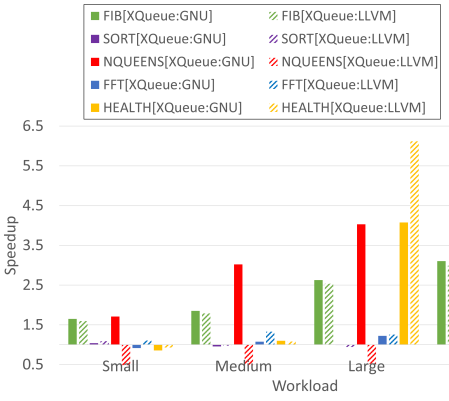


Fig. 4: Speedup of XQueue over standard GNU and LLVM OpenMP implementations on the BOTS benchmarks on skylake-192 using 192 threads.

NQueens computes all the solutions for placing N queens

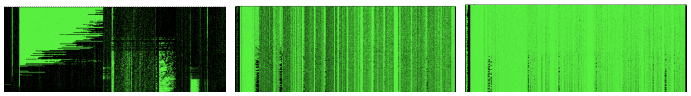


Fig. 5: Load balance of FFT on skylake-192 - LLVM+XQueue (left), Native LLVM (middle), GNU(right)

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 4 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.

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 [25] 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 6 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.

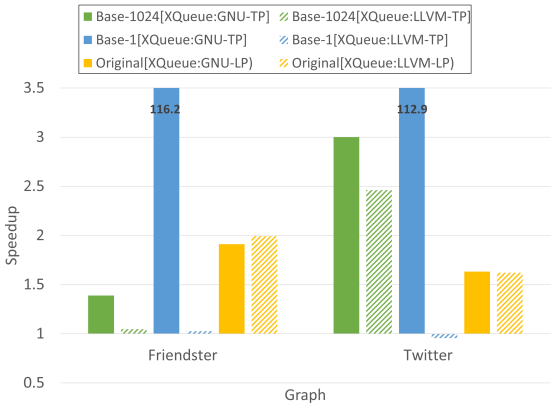


Fig. 6: Speedup of XQueue over standard GNU and LLVM OpenMP implementations when applied to Breadth First Search from GAP Benchmark Suite on skylake-192 using 192 threads.

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

V. Related Work

XQueue is most closely related to work in concurrent queues and parallel runtime systems.

Concurrent queues: Several researchers have proposed concurrent queue implementations. Scogland et al. [27] 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. [28] 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. [29] proposed a lock-free queue algorithm for machines that provide atomic primitives. Cache-friendly concurrent lock-free queue (CFCLF) [7] 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 [30] is a lock-free queue that exploits batching to gain better performance. Morrison et al. [31] proposed a concurrent nonblocking linearizable FIFO queue using atomic FAA that outperforms CAS based implementations by up to 2×.

Parallel runtime systems: Most parallel runtime systems and execution models, such as OpenMP [20], Charm++ [32], and Swift/T [3], use concurrent queues for sharing data between threads or processes. OpenMP's task construct [33] 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 [34]. Recently, Cpp-taskflow [6] emerged as an alternative to OpenMP task parallelism for C++.

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.

VI. Conclusion and Future Work

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 [35]. In the future we will investigate lock-less work stealing [36] 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. We also plan to explore integration with GNU OpenMP [21], the Swift/T workflow system [3], as well as the Parsl parallel programming library [2] in order to further broaden the applications that could take advantage of the proposed techniques.

References

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

[2] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] I. Rickards, J. Donner, S. Vigna, W. Brown, and C. via the C Programming Forum. (2009) Liblfds. [Online]. Available: http://www.liblfds.org/

[9] S. A. Bahra. (2011) Concurrency kit. [Online]. Available: http://concurrencykit.org/

[10] H. Sutter. (2005) The trouble with locks. [Online]. Available: http://www.drdobbs.com/cpp/the-trouble-with-locks/184401930

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

[12] 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.

[13] 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.

[14] J. Wang, K. Zhang, X. Tang, and B. Hua, "B-queue: Efficient and practical queuing for fast core-to-core communication," *IJPP*, vol. 41, no. 1, pp. 137–159, Feb 2013.

[15] 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.

[16] 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.

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

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

[19] Y. Guo, R. Barik, R. Raman, , and V. Sarkar., "Work-first and help-first scheduling policies for terminally strict parallel programs." in *IPDPS'09*, 2009.

[20] O. A. R. Board. Openmp®: Support for the openmp language. [Online]. Available: https://openmp.llvm.org/

[21] G. team. Gomp: An openmp implementation for gcc. [Online]. Available: https://gcc.gnu.org/projects/gomp/

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

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

[24] 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*, 2009, pp. 124–131.

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

[26] S. R. Das and R. M. Fujimoto, "A performance study of the cancelback protocol for time warp," in *SIGSIM Simul.*, vol. 23, no. 1, 1993, pp. 135–142.

[27] Scogland, T. R. W, , and W. chun Feng., "Design and evaluation of scalable concurrent queues for many-core architectures." in *ICPE'15*, 2015.

[28] H. Schweizer, M. Besta, , and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *PACT'15*. San Francisco, CA, USA: IEEE, October 2015.

[29] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *PODC'96*, 1996.

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

[31] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *PPOPP'13*. New York, NY, USA: PPoPP, 2013, pp. 103–112. [Online]. Available: https://dl.acm.org/citation.cfm?id=2442527

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

[33] 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, no. 23. IEEE, 2009.

[34] 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.

[35] C. Lehman, P. Nookala, and I. Raicu, "Scalable load-balancing concurrent queues in modern many-core architectures," in *SC19*. Denver, CO: ACM, 2019.

[36] U. A. Acar, A. Charguéraud, S. Muller, and M. Rainey, "Atomic Read-Modify-Write Operations are Unnecessary for Shared-Memory Work Stealing," Research Report, Sep. 2013.