Future Generation Computer Systems 159 (2024) 444–458

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

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

Poornima Nookala [a,*], Kyle Chard [b], Ioan Raicu [c]

[a] Intel Corporation, 2501 NE Century Blvd, Hillsboro, OR, USA
[b] Department of Computer Science, University of Chicago, 5801 S Ellis Ave, Chicago, IL, USA
[c] Department of Computer Science, Illinois Institute of Technology, 10 W 31st St, Chicago, IL, USA

ARTICLE INFO

Keywords:
Runtime
Parallel
Tasking
Task
Openmp
Lock-less
Lockfree
Locks
Atomics
Parallel computing
Workstealing
Open cilk
Onetbb

ABSTRACT

Processors with 100s of threads of execution 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. However, implementing efficient parallel runtimes that can scale up to high concurrency levels with extremely fine-grained tasks remains a challenge. Existing techniques do not scale to a large number of threads due to the high cost of synchronization in concurrent data structures. We present a thorough analysis of various synchronization mechanisms including mutex, semaphore, spinlock and atomic fetch-and-add that are typically used to build concurrent data structures in task-parallel runtime systems. To overcome these limitations, in a recent work we proposed XQueue, a novel lock-less concurrent queuing system with relaxed ordering semantics that is geared towards realizing scalability up to hundreds of concurrent threads. In this work, we extend XQueue and present X-OpenMP, a library for enabling extremely fine-grained parallelism on modern many-core systems with hundreds of cores. Work stealing is a popular choice for load balancing in task-based runtime systems as it efficiently distributes the load across worker threads; however, traditional approaches rely on synchronization primitives and thus work stealing can incur overheads. Here we implement a lock-less algorithm for work stealing for total-store order (TSO) memory architectures and evaluate the performance using micro and macro benchmarks. We compare the performance of X-OpenMP with native LLVM OpenMP, GNU OpenMP, OpenCilk and oneTBB implementations using task-based linear algebra routines from PLASMA numerical library, Strassen's matrix multiplication from the BOTS Benchmark Suite, and the Unbalanced Tree Search benchmark. Applications parallelized using OpenMP can run without modification by simply linking against the X-OpenMP library. X-OpenMP achieves up to 40X speedup compared to GNU OpenMP, up to 2X speedup compared to the native LLVM OpenMP, up to 6X speedup compared to OpenCilk and up to 5X speedup compared to oneTBB implementations. The tasking overheads in X-OpenMP are reduced by 50% compared to the native LLVM OpenMP.

1. Introduction

Modern many-core computing systems offer massive concurrency levels on the order of hundreds on CPUs to thousands on GPUs. Extreme on-node concurrency on the order of billions of concurrent tasks is required to achieve exascale performance levels according to a recent computing survey on the landscape of exascale research [1,2]. The report further states that it is possible to achieve this performance improvement by using lightweight tasking. However, it is crucial that such capabilities are delivered via productive abstractions as scientific productivity has been identified as one of the challenges today.

Task parallelism is an important type of parallelism in which computation is broken down into a set of inter-dependent tasks which can be executed concurrently on various cores. Data dependencies between tasks are used to control the execution of tasks in the runtime system. Many parallel languages use task parallelism, eg. HPX, oneTBB, Legion, Charm++, OpenCilk to name a few [3–9]. OpenMP [10] 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. 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.

* Corresponding author.
E-mail address: nookala.poornima@gmail.com (P. Nookala).

https://doi.org/10.1016/j.future.2024.05.019
Received 6 November 2023; Received in revised form 9 May 2024; Accepted 14 May 2024
Available online 17 May 2024

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.

Another method for concurrent queues is to integrate race-avoidance directly into the data structure, eliminating the need for separate synchronization. This approach offers the advantage of preventing common concurrency issues like deadlocks caused by misuse 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 [11–13], but the literature has shown that it is difficult to write correct lock-free code [14]. Even more compelling are ***lock-less*** data structures [15], 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 our recent work [16], we have introduced a lock-less concurrent framework for task parallel runtime systems. This framework enables extremely fine-grained task parallelism by minimizing the overhead of concurrent data structures used in runtime systems. Through benchmarks on modern architectures with hundreds of cores, we have shown significant performance enhancements. However, our framework uses a static round-robin load balancing strategy to distribute work across processors. While this approach can somewhat balance the load, it lacks dynamic load balancing, which can greatly impact 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 [17]. We extend their work on load balancing, integrating it with our prior work to construct a lock-less concurrent parallel framework [16] and propose a dynamic lock-less load balancing mechanism that can provide notable performance improvements using real application workloads.

The tasking model in OpenMP [18,19] enables efficient parallelization of dynamic task graphs and recursive algorithms. Several implementations of OpenMP exist: GNU OpenMP [20] is an implementation for GCC as part of the GNU project, LLVM OpenMP [21] is a subproject of LLVM, and Intel OpenMP is built using LLVM OpenMP. We implement lock-less work stealing in LLVM OpenMP [21] thereby enabling execution of unmodified OpenMP programs using our runtime library.

The main contributions of this paper are:

1. Provide a detailed performance evaluation of synchronization primitives, including mutexes, semaphores, spinlocks, and atomic fetch-and-add operations, on today's largest shared-memory systems from Intel, AMD, IBM, and ARM. Systems from 1 socket to 8 sockets, 1 to 8 NUMA zones, 8 cores to 192 cores, 16 hardware threads to 384 hardware threads, and frequencies from 1.5 GHz to 3.8 GHz are included.
2. We introduce X-OpenMP by extending our prior work XQueue and propose a work stealing algorithm that enables lightweight tasking and dynamic load balancing using lock-less techniques.
3. 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.
4. We evaluate X-OpenMP using micro benchmarks, numerical kernels and unbalanced trees and demonstrate significant performance improvements using our approach.

Table 1
Testbed for evaluation from the mystic system.

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

2. 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. Many researchers have proposed better performing lock-free data structures using atomic instructions supported by hardware. Lock-free approaches using atomic operations are believed to be highly efficient, but are hard to implement and maintain [22]. Over the years, lock-free implementations of data structures like linked lists, queues and stacks have been published, however their use is limited in production software due to its complexity and difficulty to reason about correctness. Architectures with relaxed memory consistency like ARM and Power9 also typically require the use of memory fences to ensure correctness in the implementation. Many libraries [11–13] and programming languages like Java and C++ implement lock-free data structures that take advantage of the hardware support for achieving high-performance. However, as we move towards many-core architectures with hundreds of cores, lock-free techniques do not scale well due to mutual exclusion and high contention on the memory bus.

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

2.1. Tesbed, software stack, and timing mechanisms

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

total of 192-cores and 384 hardware threads. The average system scale is about 50-cores and 100 hardware threads.

Software stack: All experiments in this paper are performed on Ubuntu 18.04.3 and compiled using LLVM Clang version 11.0.0 with O3 optimization level and $-march = native$.

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. Similar functionality exists in the Power9 architecture. 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. Performance of synchronization mechanisms

In order to program for shared memory systems using multi-threading, 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. Atomic operations are instructions supported by hardware and they lock the memory bus to access the shared resource. 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 methods utilizing atomic operations are considered to be highly efficient, yet they pose challenges in implementation and maintenance. Lock-free algorithms can be implemented by using special hardware primitives such as CAS (compare and swap) and LL/SC (load-link/store conditional). Most implementations of mutexes are built on top of atomic instructions supported by hardware.

The primary focus here is to analyze the cost of low-level thread synchronization mechanisms and for this purpose, we benchmarked `pthread_mutex_lock/ pthread_mutex_unlock`, `sem_wait/ sem_post`, `fetch-and-add` and `spin_lock/spin_unlock` to measure latency. The implementation of `spin_lock` and `spin_unlock` uses `compare_and_swap` atomic primitive. Fetch-and-add is supported by x86 architectures using 'lock xadd' instruction. The Power9 variant for fetch-and-add instruction is 'lwarx/stwcx' and ARMv8 provides 'ldxr/stxr' which are load exclusive and store exclusive instructions used for implementing atomic read, modify, write operations. These benchmarks are obtained by running a tight loop of 1 billion operations and collecting the aggregate of the results. Each iteration acquires the lock, increments a shared integer and releases the lock, excluding fetch-and-add which performs an increment operation atomically.

Figs. 1(a), 1(b), 1(c) and 1(d) show that all synchronization mechanisms exhibit higher latencies due to contention at higher levels of concurrency. The latencies increase as the systems are over-provisioned by increasing the number of threads. There are many factors that impact the cycle counts like cache coherence, communication latency between cores on 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 33 592 cycles whereas on Intel Xeon Phi Knights Landing with 64-cores and 256 hardware threads, latency reaches 3868 cycles. Similar behavior is observed on other architectures with latencies reaching up to thousands of CPU cycles solely for acquiring the lock, incrementing a variable and releasing the lock.

Although AMD, Intel, ARM and IBM have distinctly different architectures, it is interesting to note that the increase in latency of synchronization mechanisms on all the architectures as concurrency increases is close to linear. 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 semaphore, 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 linearly 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 [23].

It is clear that having a single lock across all threads is not scalable and severely limits parallelism across many threads. Fig. 2 shows the comparison between latency of mutex, semaphore, spinlock and atomic fetch-and-add on skylake-192 system. Latency of atomic fetch-and-add is higher as compared to other mechanisms up to 12 threads and shows slightly lower latencies at higher concurrency levels. If a mutex lock or semaphore cannot be acquired, the thread is put to sleep. With a spin lock, thread is running a busy loop continuously checking if the lock is available. In case of atomic fetch-and-add, although the memory bus is locked for the atomic operation, the thread is not put to sleep and hence achieves lower latencies with more threads compared to the other mechanisms.

Concurrent queues are commonly implemented using mutex locks or other forms of synchronization to protect the queue operations. We implement a single producer single consumer (SPSC) queue which does not require synchronization for queue operations and a multiple producer multiple consumer (MPMC) queue using mutexes. We analyze the performance of these implementations on various architectures by measuring latency and throughput of queue operations at varying concurrency levels. For an SPSC queue, 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/s on

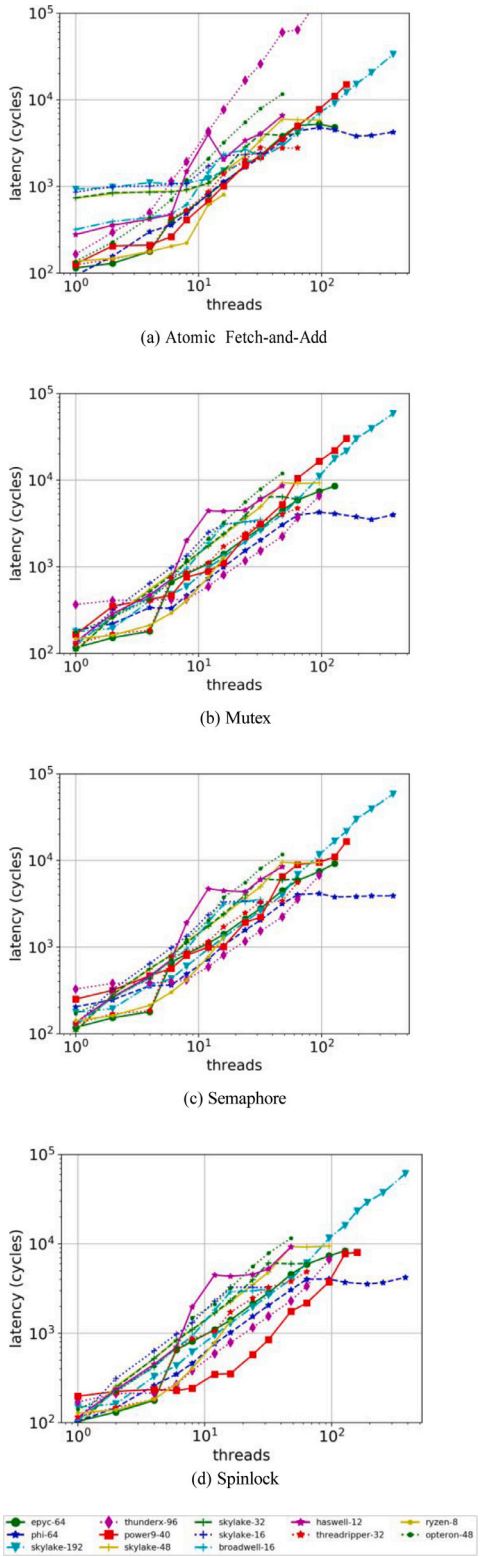


Fig. 1. Average latency of different synchronization mechanisms for incrementing an integer.

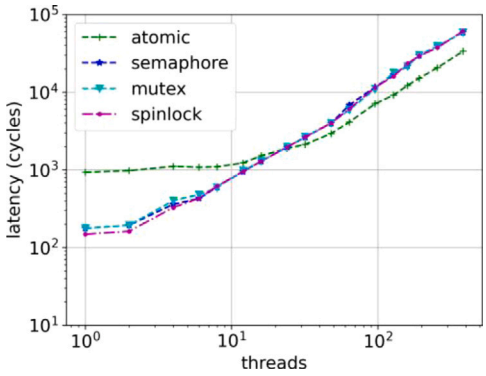


Fig. 2. Average latency of mutex, semaphore, spinlock and fetch-and-add on skylake-192. This graph shows that all the synchronization mechanisms are expensive at higher concurrency levels, atomic fetch-and-add exhibit lower latency as compared to other mechanisms after 24 threads.

skylake-192 machine. Although these results are significant, showing excellent single threaded performance, an SPSC queue is limited in parallel runtime systems because it cannot alone be used to implement parallelism and concurrency. On the other hand, our results for MPMC queues indicate that latency can reach up to millions of cycles under high contention, and throughput can drop down to as low as 300,000 operations per second. For the skylake-192 system, which had the best single core performance at 270 million operations/s, 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/s.

We were not surprised by these findings, as the fundamental problem stems from the cost of synchronization such as mutex, semaphores, spinlock, and atomics. 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.

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 [24,25]. Dynamic load balancing enables better distribution of work across 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 executes 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. Workers can steal a random amount of work from the victim's task pool.

Fig. 3 shows the timeline plot of the Unbalanced Tree Search (UTS) benchmark [26] executed using GNU's implementation of OpenMP. The green dots indicate effective CPU time and the black dots indicate idle time. The plot shows a significant load imbalance for this application where several workers (bottom of the figure) are idle for most of the application run, and other workers are idle for a significant amount of the time. The load imbalance results in a major slowdown in the execution time of the application. The UTS benchmark is designed to understand the efficiency of dynamic load balancing in parallel runtime systems and this plot clearly highlights the imbalance in existing task-based runtime systems. Processors are heavily under-utilized resulting in poor overall performance.

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. Upon initialization of a runtime and the creation of workers, they begin searching for tasks to execute. If there is no work available in the local task pool, the mechanism of work stealing is triggered. Studies have shown that several steal requests

are generated at the beginning and tail end of the execution [27]. Work stealing implemented using traditional synchronization-based mechanisms tend to have huge overheads. Hence, work stealing should be triggered sparingly and only when necessary to minimize overheads.

Work stealing and work sharing are commonly used scheduling paradigms for redistributing work across processes for the purposes of load balancing. In work sharing, workers attempt to migrate jobs to other workers whereas in work stealing, idle workers try to steal work from other workers. Both approaches have significant communication overheads. Traditional work stealing approaches use pull-based approach for stealing work where the work is redistributed to other workers by the process a worker stealing jobs. In this work, we explore a push-based approach for distributing work across processes. The main motivation behind this idea of pushing tasks comes from the behavior of recursive applications like Fibonacci where the application starts with a single task and can generate several millions tasks as the task graph evolves. In a traditional runtime where work is locally produced and locally consumed, work stealing has a significant overhead to redistribute the tasks across workers. In our prior work, the task-pushing proved to be effective in several benchmarks that were evaluated. This work makes a trade-off between task locality and migration costs that proves to be performant for a number of use-cases.

3. XQueue— Lock-less queuing mechanism for task-parallel runtime systems

In a recent work, we introduced XQueue [16], a novel lock-less MPMC, out-of-order queuing mechanism that can scale up to hundreds of threads. XQueue uses B-queue [28] 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 [29–31] and we aim to demonstrate that XQueue can be built with any fast and scalable SPSC queue. We describe XQueue design in detail in this section which lays the foundation for the next section.

Fig. 4 shows the architectural diagram 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 Fig. 4. A worker then tries to dequeue an item from its auxiliary queues and dequeued items are processed immediately.

A simplified version of pseudocode for worker logic is outlined in Algorithm 1. Since all queues in XQueue are concurrent SPSC queues, producer and consumer threads can act concurrently processing items in the queues. The strategy of distributing work across queues (as shown in Fig. 4) ensures that there is a only a single producer and single consumer for every queue at any point in time. Due to this design, locks can be completely avoided thereby reducing the latencies of queue operations and improving overall performance.

Algorithm 1 Worker logic

```
id ← 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 if
  end if
  ret ← dequeueFromAuxiliary(id, item);
  if ret = SUCCESS then
    retItem ← processItem(item);
    if retItem ≠ NULL then
      enqueueToAuxiliary(next, retItem);
    end if
  end if
  next ← (next + 1)%numCores;
  if next == id then
    next++;
  end if
end while
```

3.1. Static load balancing in XQueue

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 [24,32]. 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 modeled 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 Fig. 4, 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.

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

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

3.2. XQueue integration with the OpenMP runtime

In order to extend our research to real systems, we integrated XQueue into OpenMP [21] 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. Broadly speaking, there are two implementations of OpenMP: GNU OpenMP (for GCC) [20] and LLVM OpenMP [21]. We chose to integrate XQueue into the LLVM OpenMP since it is open source and has superior performance as compared to GNU OpenMP with fine-grained tasks [33].

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 its 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. Fibonacci benchmark performed 5X times faster using 192 threads on skylake-192 server due to the reduced overhead of checking queues for every dequeue. 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.

Our prior study showed that XQueue-enabled OpenMP is able to achieve up to 6× speedup compared to native LLVM OpenMP and up to 4× speedup compared to GNU OpenMP in most cases. While the results are promising, static load-balancing strategies fall short in real application workloads with varying task execution times and workload patterns. The adaptability of a runtime to dynamically adjust task assignments based on workload characteristics becomes crucial for maximizing performance and resource utilization.

4. X-OpenMP — eXtreme fine-grained tasking runtime

We extend XQueue and implement dynamic load balancing to overcome the limitations of static round-robin load balancing. We introduce X-OpenMP with the goal of enabling extreme fine-grained parallelism for task-parallel applications. 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 [7,34,35] use load balancing mechanisms like work stealing and work sharing 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.

4.1. Lock-less work stealing using wait

A mechanism that does not use synchronization is required for implementing work stealing using XQueue. Intel's x86 architecture has a memory model that supports Total Store Ordering (TSO) [36]. 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. Muller et al. proposed an algorithm that does not require atomic read–modify–write operations for shared memory work stealing [17] that works on TSO memory architectures like Intel's x86. The details of the original implementation can be found in the technical report [17]. We employed a modified version of this algorithm for work stealing in X-OpenMP to implement dynamic load balancing.

Our 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) randomly picks a victim. As shown in Algorithm 3, the stealer first checks if the victim is accepting requests. This is shown in line 3 where the 40-bit round number is extracted 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 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 does not match the victim's round or a stolen task is not received. In lines 7–9, the stealer writes the steal request to the victim again if needed since the stealer is still waiting, and it does so by comparing the local round number with the victim's copy of the round number in the steal request memory cell. While waiting, it also writes a steal request to its own memory cell and leaves it unserved. This self query ensures that no other steal requests come to this thread since it is idle (lines 12–15). 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.

Algorithm 2 Work Stealing With Wait — Victim's Logic

```
1: local_steal_req ← thread− > steal_req;
2: round ← local_steal_req &((1 << 40) − 1);
3: if round == thread− > round then
4:    ret ← dequeue(thread_id, item);
5:    if ret == SUCCESS then
6:       stealer_id ← local_steal_req >> 40;
7:       threads[stealer_id]− > stolen_task ← item;
8:    end ifthread− > round + +;
9: end if
```

Fig. 3. Load Imbalance in Unbalanced Tree Search using GNU OpenMP and 192 threads (green shows useful work and black shows idle time).

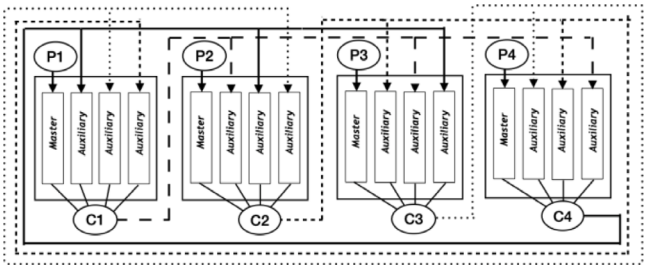


Fig. 4. Architecture of XQueue on a 4-core machine with 4 queues per consumer.

Algorithm 3 Work Stealing With Wait — Stealer's Logic

```
1:  thread->num_tries = MIN_TRIES;
2:  num_tries = 0;
3:  if (victim->steal_req&((1 << 40) - 1) < victim->round) then
4:      round = victim->round;
5:      victim->steal_req = round + (thread_id << 40);
6:      while round == victim->round||thread->stolen_task ≠ NULL
        do
7:          if (victim->steal_req&((1 << 40) - 1)) < round then
8:              victim->steal_req = round + (thread_id << 40);
9:          end if
10:         self_r = thread->round;
11:         self_query = self_r + thread_id << 40;
12:         if (thread->steal_req_id ≠ self_query) then
13:             thread->steal_req_id = self_query + 1;
14:             thread->round++;
15:         end if
16:         if (++num_tries > thread->num_tries) then
17:             if (thread->stolen_task ≠ NULL) then
18:                 if (thread->num_tries > MIN_TRIES) then
19:                     thread->num_tries-= MIN_TRIES;
20:                 end if
21:                 return thread->stolen_task;
22:                 break;
23:             end if
24:         end if
25:     end while
26:     if (thread->num_tries < MAX_TRIES) then
27:         thread->num_tries *= 2;
28:     else
29:         thread->num_tries = MAX_TRIES;
30:     end if
31:     return NULL;
32: end if
```

On the other hand, a busy victim looks at its 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 its round number to invalidate any incoming steal requests. 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 omitted details are specific to the LLVM's implementation for handling termination of the application and handling of stolen tasks so they are correctly executed. The algorithms presented here cover the bulk of the work stealing functionality. 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 [17] 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 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 [37]. In our model, the wait time is controlled by the number of loop iterations, starting with a very small number MIN_TRIES and doubling every time a steal request fails until it reaches a certain limit set by MAX_TRIES (lines 16–23 and 26–31 in Algorithm 3). 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. Please note that line 1 in Algorithm 3 is only executed at the beginning of the program run and the value is tuned based on the success or failure of steals as the execution progresses. 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. Lock-less work stealing without wait (no-wait)

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 its 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 No-Wait Work Stealing — Victim's Logic

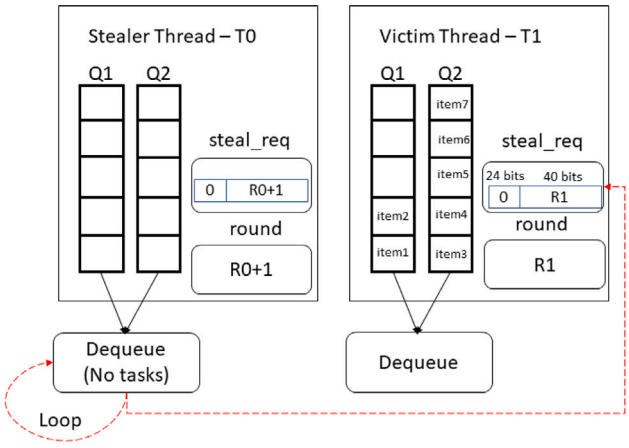
```
1: local_steal_req ← thread− > steal_req;
2: round ← local_steal_req&((1ULL << 40) − 1);
3: if round == thread− > round then
4:     ret ← dequeue(thread_id, item);
5:     if ret == SUCCESS then
6:         stealer_id ← local_steal_req >> 40;
7:         threads[stealer_id]− > enqueue(item);  thread− > round + +;
8:     end if
9: end if
```

Algorithm 5 No-Wait Work Stealing — Stealer's Logic

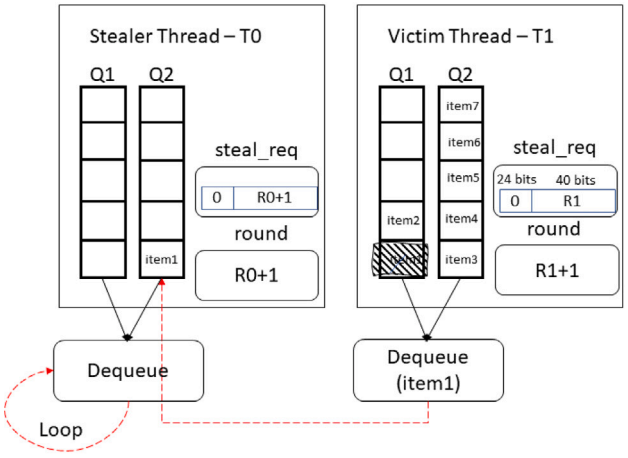
```
1: if (victim− > steal_req&((1 << 40) − 1) < victim− > round) then
2:     round = victim− > round;
3:     victim− > steal_req = round + (thread_id << 40);
4:     return NULL;
5: end if
```

It is worth noting that the no-wait work stealing algorithm results in many more steal requests being submitted than the wait-based approach, thereby resulting in more successful steals and better load balancing in terms of the number of tasks. A significant difference between the traditional work stealing approach and the lock-less approaches described above is that in the traditional approach, an idle worker is doing all the work for stealing a task. However, in the case of the lock-less approach, a busy worker is facilitating work stealing by checking its 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 no-wait lock-less work stealing algorithm is shown in Fig. 5. 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 Fig. 5-A, the stealer thread T0 checks its own queues for tasks during dequeue operation. If no tasks are found, T0 writes a steal request into T1's memory cell as shown by the dotted red line. After putting a steal request, T0 checks if the termination condition for the runtime is satisfied and if not, returns back to the dequeue operation which is shown by the dotted red loop for dequeue. Victim thread T1 checks for incoming steal requests during a dequeue operation. If a request is received, thread T1 checks its queues for two tasks, one for executing itself and the other for fulfilling the steal request. Only the stealing part is shown in the figure. Thread T1 dequeues an item and enqueues it to queue Q2 of thread T0. It then increments its round value to allow other incoming steal requests. Also, thread T0 writes a self query using its own round number incremented by one into its own steal request memory cell. This tells the other workers that steal requests are not currently being accepted by this worker (as shown in Algorithm 3).



(a) Stealer putting a steal request to the victim



(b) Victim serving the steal request

Fig. 5. No-wait work stealing in action.

4.3. 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 [27]. The no-wait 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.

5. Evaluation

We evaluate X-OpenMP using a set of synthetic benchmarks and real-world applications. The microbenchmarks are specifically designed to explore the performance of lock-less techniques described in this

paper for tasking and load balancing. We evaluate four 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-NOWAIT/ XOMP-DYNAMIC-NOWAIT-STEALONE - uses static load balancing and dynamic no-wait work stealing, stealing one task at a time;
4. XOMP-DYNAMIC-NOWAIT-STEALTWO - uses static load balancing and dynamic no-wait work stealing, stealing two tasks at a time.

We compare the performance of X-OpenMP (XOMP) with native LLVM OpenMP (OMP), GNU OpenMP (GOMP), OpenCilk (CILK) and Intel oneAPI Thread Building Blocks (TBB). OpenCilk [8] recently emerged as an open source paradigm for task parallelism. OneTBB [38] is a library part of the oneAPI toolkit which simplifies the work of adding parallelism to applications and provides tasking constructs. To quantify the performance improvements in real application workloads, we evaluate strassen's matrix multiplication from the BOTS benchmark suite [39], cholesky factorization and symmetric rank-k update routines from the PLASMA linear algebra library [40] and the Unbalanced Tree Search benchmark [26]. PLASMA library is written in C and oneTBB is a library for C++ applications, hence we have omitted TBB from the benchmarks that use the PLASMA library. All experiments in this paper are conducted on an Intel Skylake Server with 192 cores (384 hardware threads) at 2.1 GHz with 8 sockets and 8 NUMA zones. We compiled all the benchmarks using LLVM Clang version 11.0 and O3 optimization level and ran experiments on Ubuntu 20.04.4.

5.1. 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 [41]. 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 paper. 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 based on the deviation of task distribution from the ideal case when only the master worker receives all the tasks; (4) Memory footprint — measures the execution time using various queue sizes to understand the impact of queue sizes on the overall performance.

Fig. 6 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 duration between 1 ns (ns) and 1 (µs). The experiment is repeated 20 times and the plot shows the average overhead time. To get the overhead, we calculate the ideal time for the benchmark based on the number of tasks and the task duration and subtract it from the overall execution time. The tasking overhead measured for X-OpenMP with static round-robin load balancing is about 110 ns. The overhead of X-OpenMP with workstealing is about 150 to 200 ns. In native LLVM OpenMP, the tasking overhead is about 400 ns. GNU OpenMP exhibits significantly higher overhead for extremely fine-grained tasks at about 20.32 µs for 1 ns tasks, with the overhead going down up to 1 µs for 1 µs tasks. OpenCilk has 30 ns overhead for executing 1 ns tasks and 110 ns

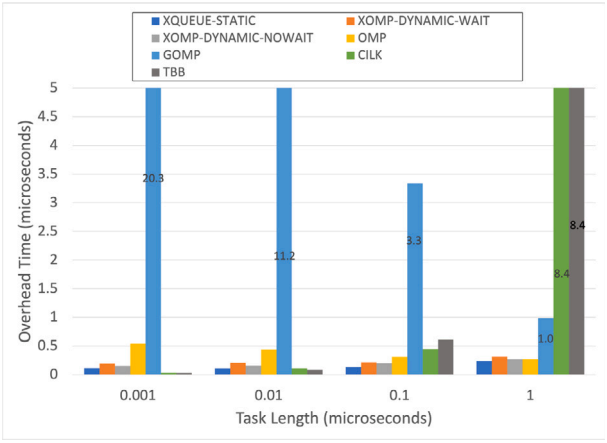


Fig. 6. Parallel Tasking Overhead on skylake-192 using 192 threads (lower is better).

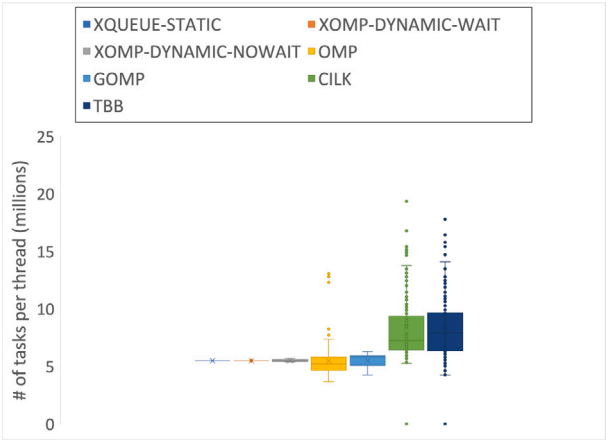


Fig. 7. Average Task Distribution on skylake-192 using 192 threads.

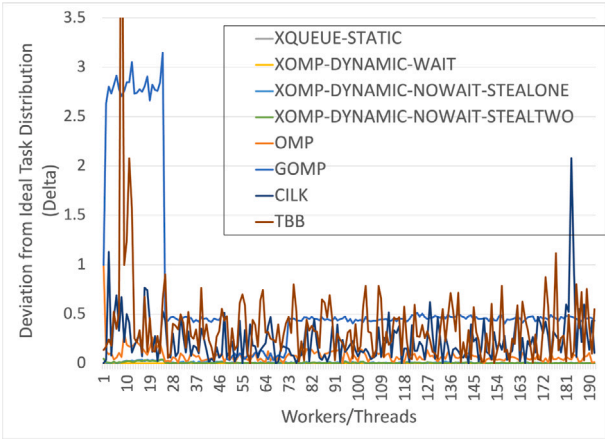


Fig. 8. Delta of Task Distribution using Workstealing on skylake-192 using 192 threads (lower is better).

overhead for 10 ns tasks. OneTBB shows 30 ns overhead for 1 ns tasks and 85 ns overhead for 10 ns tasks. However, the overhead increases with increasing task granularity. Profiling showed that as task granularity increases, the time spent in *spinlock* increases which results in higher overhead. Spinlocks are mainly used for the scheduling

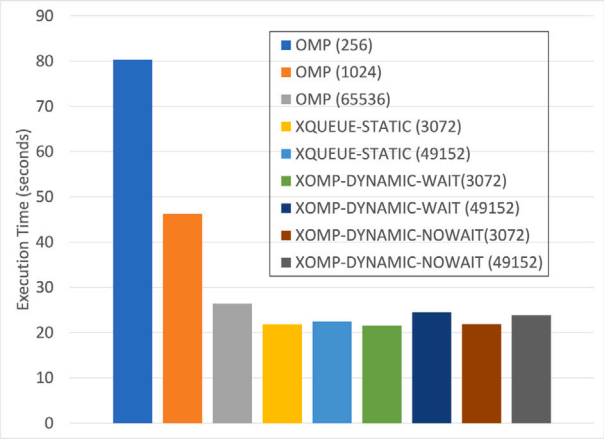


Fig. 9. Execution time of a synthetic benchmark with varying queue sizes.

and join operations. These results clearly illustrate that the overheads of tasking can be significantly reduced by using lock-less concurrent queuing mechanisms.

Fig. 7 shows a box plot of task distribution across workers for 20 runs of 8 million fixed-length 0.1 µs 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. OpenCilk and oneTBB show significant variability in the number of tasks executed by each worker with some workers executing over 10x the tasks as compared to other workers. 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 other runtimes evaluated. The execution time is directly correlated with the number of tasks executed by each worker. Compared to 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. X-OpenMP is about 30% faster compared to OpenCilk and oneTBB in this benchmark.

Fig. 8 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 µs 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. For OpenCilk, the delta ranges between 0 and 2.08 and for oneTBB, the delta is between 0.001 and 5.84. Both oneTBB and OpenCilk dynamically control task granularity. OpenCilk uses fast clone for spawned tasks and slow clone for stolen tasks. If no stealing happens, fast clones are run sequentially and slow clone induces parallelism. On the other hand, oneTBB uses mailboxing concept for workstealing instead of completely randomized scheduling which results in better task affinity. Task granularity is controlled by using RDTSC to put a ban of work sharing to coarsen the grain-size during this interval. These optimizations result in significant load imbalance, particularly with extremely fine-grained tasks. The main takeaway from this benchmark is that lock-less implementations of work stealing perform similar to traditional work stealing implementations.

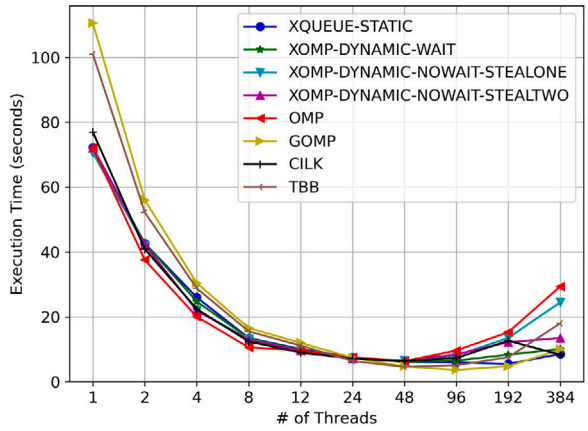
The memory footprint benchmark is designed to study the amount of memory required by the runtime to efficiently execute applications. X-OpenMP uses multiple queues per worker as opposed to single queue per worker in the other runtimes, which raises a question about the amount of memory required for the runtime itself. In this benchmark, the master worker receives $N \times 10$ tasks (where N is the number of workers) which run for 1 s each. The ideal time to execute these tasks with perfect load balancing is 10 s. Also, every worker receives 8 million delay tasks of length 0.1 µs, which should ideally execute within a second. The idea behind this synthetic benchmark is to simulate load imbalance and to understand how queue sizes can impact the overall execution time. Fig. 9 shows the results obtained by running this benchmark. Legend shows number of queue slots per worker. Please note that the queue sizes could not be matched for X-OpenMP and LLVM implementations since the latter requires powers of two for queue length. The number of queues for X-OpenMP = 192 threads * 16 (per queue size) = 3072 and 192 * 256 (per queue size) = 49 152. 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. GNU OpenMP, OpenCilk and oneTBB are excluded from this plot since we used pre-built libraries for the evaluation. The execution time of this synthetic benchmark for LLVM OpenMP shows that the performance of the runtime is sensitive to queue size. Default queue size in LLVM OpenMP is 256 which results in 8X slowdown compared to the ideal execution time. Increasing the number of queue slots per worker results in better load balancing and better performance. 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.

5.2. Macrobenchmarks

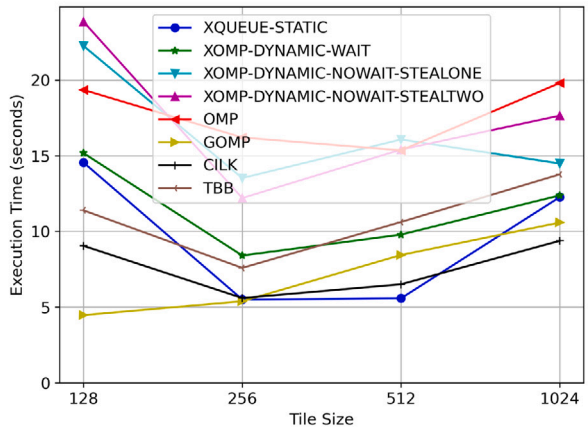
To demonstrate the behavior of X-OpenMP in real application scenarios, we chose benchmarks which are commonly studied and relevant to real-world HPC applications: a matrix multiplication benchmark, two linear algebra routines, and an unbalanced tree search benchmark. We evaluate these applications on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

Strassen's Matrix Multiplication [39,42] 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 sequential divide and conquer approach. The depth based cutoff value for the sequential divide and conquer algorithm is set to 3.

Fig. 10(a) shows the scalability plot for Strassen's matrix multiplication algorithm. The experiment multiplies square matrices of size 8192×8192 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 s using 96 threads, followed by oneTBB which runs in 4.5 s using 48 threads, followed by XOMP-STATIC which runs in about 5.9 s using 96 threads. The native LLVM version runs about 5%



(a) Scaling of Strassen's Algorithm using 8K matrix on skylake-192 (lower is better)



(b) Performance of Strassen's Algorithm using 8K matrix on skylake-192 with varying base sizes (lower is better)

Fig. 10. Strassen's matrix multiplication using 8K matrix on skylake-192.

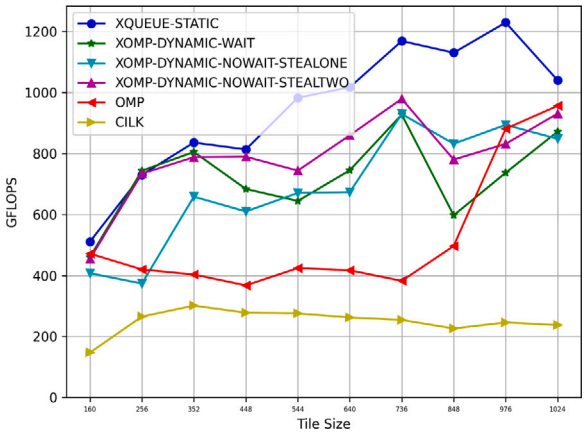


Fig. 11. Symmetric rank update using 12K matrix on skylake-192 using 96 threads (higher is better).

slower than X-OpenMP using 96 threads and OpenCilk is about 21% slower. It is interesting to note that while GNU OpenMP scales well beyond 96 threads, LLVM OpenMP quickly degrades in performance. GNU OpenMP employs centralized greedy scheduler where a shared queue is protected by a global lock. It has hierarchical queues where parent tasks have a queue and child tasks have their own queues. The hierarchical queuing along with centralized scheduling inherently results in good performance for this algorithm. The best running time of strassen's using OpenCilk is 6.37 s using 48 threads, and for oneTBB, the best running time is 4.5 s with 48 threads, which is 15% faster than the best running time of X-OpenMP. At 96 thread scale, the performance of X-OpenMP and oneTBB are comparable at 5.9 s.

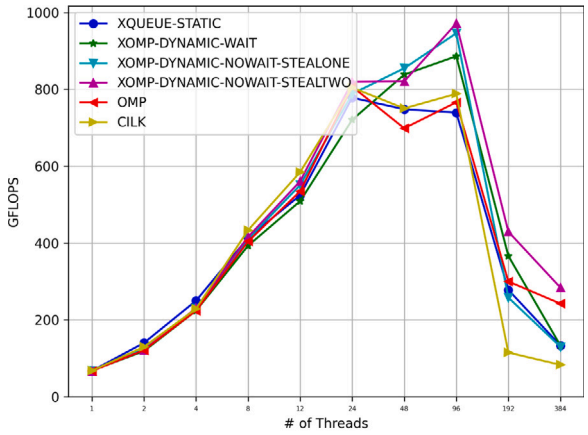
Fig. 10(b) shows the results obtained by running Strassen's algorithm on an 8192 × 8192 matrix using 192 threads and varying base sizes for the matrix from 128 to 1024. The plot shows the average of three runs. The best performance is achieved using base sizes of 256 and 512 in case of X-OpenMP. It is worth noting that X-OpenMP using static load balancing is sufficient to achieve good performance for this algorithm. Dynamic work stealing induces additional overhead increasing the overall running time for this particular application.

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. OpenCilk achieves a runtime of 5.5 s similar to X-OpenMP with 256 tile size, whereas oneTBB is about 36% slower than X-OpenMP.

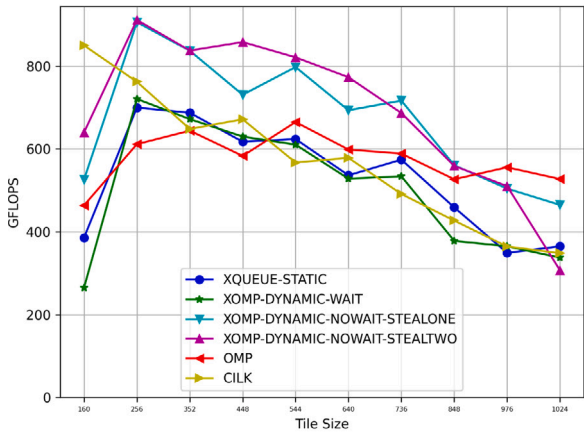
Symmetric Rank-k Update (SYRK) [43] is an important building block of many linear algebra algorithms and included in the Basic Linear Algebra Subprograms (BLAS) specification [44]. The SYRK algorithm computes the upper or lower part of the result of a matrix product where the given matrix is a symmetric matrix. Parallel Linear Algebra Software for Multicore Architectures (PLASMA) numerical library [40] 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.

Fig. 11 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 no-wait approach and stealing two tasks at a time achieves 979 GFLOPS using 736 tile size. Larger tiles provide cache efficiency for DSYRK algorithm by reusing the data within the cache. The round-robin distribution of tasks within the X-OpenMP framework confers an advantage in L3 cache locality when contrasted with the randomized stealing of tasks. This approach yields superior performance outcomes using X-OpenMP for this benchmark. The native LLVM version achieves 956 GFLOPS using 1024 tile size, however it is about 50% slower with smaller block sizes. With fine-granular tasks, the native LLVM version has a long tail at the end of the computation where a single or only a few workers are busy executing tasks which results in significant loss in performance. The overheads of synchronization and wait times in continuation stealing significantly degrade the performance of OpenCilk across all tile sizes.

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).

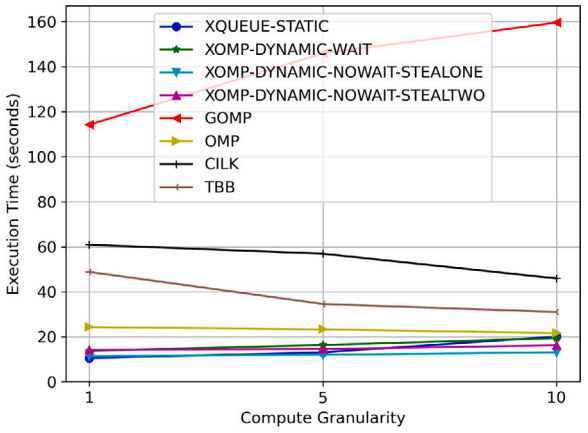


(a) Scaling using 256 tile size

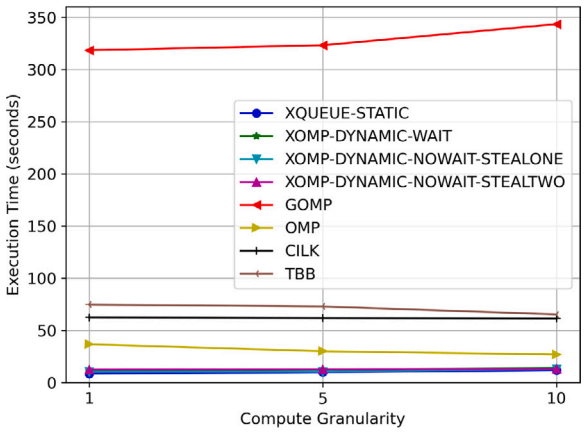


(b) Using 96 threads and varying tile sizes (higher is better)

Fig. 12. Cholesky factorization on skylake-192.



(a) Using 96 threads



(b) Using 192 threads

Fig. 13. Unbalanced Tree Search on skylake-192 (lower is better).

Fig. 12(a) shows the scalability plot of Cholesky factorization using X-OpenMP, LLVM, GNU and OpenCilk versions. The algorithm scales up to 96 threads for all the runtimes evaluated with peak performance achieved using no-wait based X-OpenMP implementation. The performance of all the implementations drops significantly with 192 and 384 threads which clearly indicates there is room for improvement in existing algorithms to leverage the parallelism available on modern hardware.

Fig. 12(b) 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 no-wait work stealing performs best for this algorithm. XQueue with static round-robin load balancing has performance similar to LLVM overall and the dynamic work stealing highly improves the performance compared to native LLVM OpenMP. OpenCilk achieves highest performance with 160 tile size and as task granularity increases, performance drops. This behavior is consistent with the microbenchmarks where the overheads of tasking keep increasing with task granularity. OpenCilk runtime also spends increasing amount of time in synchronization with increased task granularity thereby resulting in lower performance. It is notable that the native LLVM version achieves its peak performance with a tile size of 352, whereas all versions of X-OpenMP reach their peak performance with a tile size of 256. This observation underscores the effectiveness of lightweight tasking and reduced synchronization overheads, enabling acceleration of applications utilizing tasks with much finer granularity than typically supported by contemporary runtime systems. Additionally, it underscores the potential for exploring over-decomposition of task-based applications to unlock maximum speedup 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.

Unbalanced Tree Search (UTS) [26] 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 17 844 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.

Figs. 13(a) and 13(b) 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. 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

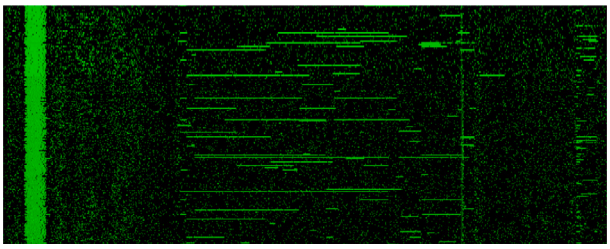


Fig. 14. Unbalanced Tree Search using X-OpenMP and 192 threads on skylake-192.

round robin load balancing achieves the best execution time of 8.6 s, 9.9 s, and 11.8 s respectively using 192 threads. GNU OpenMP incurs significant overheads with this workload with about 40X slowdown across all task granularities. In general, binomial trees are the best adversary for load balancing because they create highly unbalanced trees [26]. Despite the hierarchical queuing in GNU OpenMP favoring Strassen's matrix multiplication benchmark, the runtime suffers from severe load imbalance in this benchmark due to the nature of the benchmark, in which some child nodes in the binomial tree result in a large number of tasks and no tasks on the other child. This results in a few threads having a huge number of tasks in comparison to others and the lack of dynamic load balancing results in significant performance degradation due to the lack of a dynamic load balancing mechanism in GNU OpenMP. OpenCilk is 6x slower than X-OpenMP and oneTBB is 5x slower. Both OpenCilk and oneTBB use a traditional lock-based randomized work stealing as a load balancing mechanism which results in overheads due to the synchronization required for achieving load balancing for this workload. X-OpenMP achieves the highest performance for this benchmarks with 40X speedup over GNU OpenMP, 6x speedup over OpenCilk and 5x speedup over oneTBB. The observed speedups can be credited to the innovative integration of a static round-robin load balancing strategy with dynamic work stealing. This combination ensures effective task distribution among all workers, resulting in enhanced performance. Fig. 14 shows a part of the timeline plot of one execution of UTS using T3L graph and X-OpenMP. 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 and light-weight load balancing techniques 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 no-wait load balancing and stealing one task at a time performs the best at 11.9 s and 13.1 s. At 96 thread scale, X-OpenMP is 10X faster than GNU OpenMP and 2X faster than LLVM OpenMP.

5.3. Discussion and summary

This evaluation showed that static load balancing mechanisms are suitable for some applications, while others require more dynamic approaches. Configuring how many tasks to steal at a time is dependent on the application and the computational complexity of the tasks. If tasks are of similar lengths in terms of execution time, static round-robin load balancing along with stealing one task at a time works well. For highly imbalanced applications, traditional work stealing approaches can incur extremely high overheads due to synchronization at higher concurrency levels. Such applications can benefit from lock-less approaches presented in this paper. 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.

Our results clearly demonstrate the performance improvements that can be achieved using lightweight tasking and reduced synchronization overheads. The techniques presented in this paper can be used to enhance existing parallel runtime systems to improve the efficiency of fine-grained parallelism on many-core architectures.

6. Related work

Task-based Parallel Runtimes: Most parallel runtime systems and execution models (e.g., OpenMP [21], Charm++ [6], and Swift/T [45]) use concurrent queues for sharing data between threads or processes. Charm++'s run-time [6] uses message-driven execution to hide the latency of communication between tasks and remote data; it demonstrates about 10–20 percent improvement in performance by using optimization techniques like lock-free queues, CPU affinity, and memory management [46]. Researchers have investigated contention management in thread-safe MPI libraries [47] and the use of abort locking [48]. Recently, Cpp-taskflow [35] emerged as an alternative to OpenMP for C++ with support for template instantiation to compose dependency graphs. Cilk [7] is an influential programming language based on C in which programmers are responsible for exposing parallelism via tasks and dependencies. OpenCilk [8] has emerged as a new parallel runtime after Intel CilkPlus has been deprecated in 2018. StarPU [49] integrates tasking across CPUs and accelerators. XKAAPI [50] is a run-time for scheduling irregular fine-grained tasks with dataflow dependencies. PARSEC [51] provides a highly-efficient distributed low-level task-based runtime supporting a varied set of Domain Specific Languages (DSL) programming languages and APIs. In all these examples the programmer specifies what shared data each task will access and how. Intel's TBB [4] provides task-based parallelism using C++ templates in which dependencies are handled by explicitly waiting for spawned tasks. Legion [5,52] queues tasks eagerly, but defers their execution until it is safe to do so. Wool [53] is a user-level task management library which is aimed at efficient load balancing with very low overheads for task creation. In each of these systems, the developer needs to be aware of limitations of task granularity and decomposition to achieve efficient performance. In our work, we aim for a task-based parallel programming model in which applications can achieve improved performance at finer granularity and decomposition.

Load Balancing: Several researchers have proposed various load balancing mechanisms [24,32]. Blumofe and Leiserson et al. introduced work stealing and proved that it is superior to work sharing [54]. Quintin et al. proposed hierarchical work stealing for exploiting data locality to achieve speed up compared to classical work stealing algorithms [55]. 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 [56]. Several applications implement their own load balancing mechanisms in order to achieve ideal performance on various architectures. Recently Shiina et al. introduced "Almost Deterministic Work Stealing" which addresses the issue of data locality by making scheduling almost deterministic [57]. 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. Conclusion and future work

We propose X-OpenMP as a framework to enable extremely fine-grained task parallelism on modern shared memory architectures with hundreds of cores. The work stealing algorithm in X-OpenMP does not require any atomicity for read, write, and modify operations on TSO architectures, and achieves performance comparable with state-of-the-art implementations. Existing OpenMP applications can use X-OpenMP without modification by simply 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, up to 2X compared to the native LLVM OpenMP implementation, up to 6X over OpenCilk and up to 5X over oneTBB.

Integration of our techniques with LLVM OpenMP creates the opportunity to transparently accelerate many applications with fine-grained parallelism. Our work in this area of task-based parallel runtime systems creates new avenues for exploring lock-less techniques in the HPC space. We plan to evaluate real-world scientific applications in Computational Biology, Materials Science, Computational Chemistry, and Astrophysics using X-OpenMP to demonstrate the performance improvements achievable in parallel runtimes as a step towards exascale goals. By enabling efficient support of fine-grained parallelism across the growing range of scales seen in modern and future hardware, we believe this work will enhance the productivity of parallel programmers.

CRediT authorship contribution statement

Poornima Nookala: Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Kyle Chard:** Writing – review & editing, Supervision, Resources, Project administration, Funding acquisition. **Ioan Raicu:** Writing – review & editing, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported in part by National Science Foundation (NSF) CNS-1730689 CRI and the OAC-2107548/2107283 Core awards.

References

[1] S. Heldens, P. Hijma, B.V. Werkhoven, J. Maassen, A.S.Z. Belloum, R.V. Van Nieuwpoort, The landscape of exascale research: A data-driven literature analysis, ACM Comput. Surv. 53 (2) (2020) http://dx.doi.org/10.1145/3372390.

[2] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges, Technical Report, USDOE Office of Science (SC)(United States), 2014.

[3] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, Hpx: A task based programming model in a global address space, in: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, 2014, pp. 1–11.

[4] A. Kukanov, M.J. Voss, The foundations for scalable multi-core software in intel threading building blocks., Intel Technol. J. 11 (4) (2007).

[5] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing locality and independence with logical regions, in: Proceedings of Supercomputing, SC 2012, 2012.

[6] L. Kalé, S. Krishnan, CHARM++: A portable concurrent object oriented system based on C++, in: OOPSLA'93, 1993.

[7] M. Frigo, C.E. Leiserson, K.H. Randall, The implementation of the cilk-5 multithreaded language, in: PLDI'98, 1998, pp. 212–223, http://dx.doi.org/10.1145/277650.277725.

[8] T.B. Schardl, I.-T.A. Lee, OpenCilk: A modular and extensible software infrastructure for fast task-parallel code, in: Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 2023, pp. 189–203.

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

[10] L. Dagum, R. Menon, OpenMP: An industry-standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998).

[11] X. Meng, X. Zeng, X. Chen, X. Ye, A cache-friendly concurrent lock-free queue for efficient inter-core communication, in: ICCSN'17, 2017.

[12] I. Rickards, J. Donner, S. Vigna, W. Brown, C. via the C Programming Forum, LIBLFDS, 2009, URL http://www.liblfds.org/.

[13] S.A. Bahra, Concurrency kit, 2011, URL http://concurrencykit.org/.

[14] H. Sutter, The trouble with locks, 2005, URL http://www.drdobbs.com/cpp/the-trouble-with-locks/184401930.

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

[16] P. Nookala, P. Dinda, K.C. Hale, K. Chard, 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, IEEE, 2021, pp. 1–8.

[17] U.A. Acar, A. Charguéraud, S. Muller, M. Rainey, Atomic Read-Modify-Write Operations are Unnecessary for Shared-Memory Work Stealing, Research Report, 2013, URL https://hal.inria.fr/hal-00910130/file/main.pdf.

[18] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.

[19] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, G. Zhang, The design of OpenMP tasks, in: TPDS'09, Vol. 20, IEEE, 2009.

[20] GCC team, GOMP: An OpenMP implementation for GCC, 2023, URL https://gcc.gnu.org/projects/gomp/.

[21] OpenMP Architecture Review Board, OpenMP®: Support for the OpenMP language, 2023, URL https://openmp.llvm.org/.

[22] H. Sutter, Lock-free code: A false sense of security, 2008, URL https://drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279.

[23] TUG, Intel® 64 and IA-32 architectures software developer's manual, 2018, URL https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[24] J. Dinan, D.B. Larkins, P. Sadayappan, S. Krishnamoorthy, J. Nieplocha, Scalable work stealing, in: Proc. ACM Conf. on High Performance Computing Networking, Storage and Analysis, SC, 2009.

[25] N.S. Arora, R.D. Blumofe, C.G. Plaxton, Thread scheduling for multiprogrammed multiprocessors, Theory Comput. Syst. 34 (2) (2001) 115–144.

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

[27] K. Wang, A. Kulkarni, M. Lang, D. Arnold, I. Raicu, Exploring the design tradeoffs for extreme-scale high-performance computing system software, IEEE Trans. Parallel Distrib. Syst. 27 (4) (2015) 1070–1084.

[28] J. Wang, K. Zhang, X. Tang, B. Hua, B-queue: Efficient and practical queuing for fast core-to-core communication, IJPP 41 (1) (2013) 137–159, http://dx.doi.org/10.1007/s10766-012-0213-x.

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

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

[31] S. Arnautov, P. Felber, C. Fetzer, B. Trach, FFQ: A Fast Single-Producer/Multiple-Consumer Concurrent FIFO Queue, IEEE, 2017.

[32] Y. Guo, R. Barik, R. Raman, 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, IPDPS, 2009.

[33] A. Podobas, M. Brorsson, V. Vlassov, Scheduling for improved data-driven task performance with fast dependency resolution, in: IWOMP'14, Springer, Salvador, Brazil, 2014, pp. 45–57, http://dx.doi.org/10.1007/978-3-319-11454-5_4.

[34] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, K. Warren, Introduction to UPC and Language Specification, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[35] G.G. Tsung-Wei Huang, M. Wong, Cpp-Taskflow: Fast task-based parallel programming using modern C++, IPDPS'19 (2019) 974–983.

[36] P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli, M.O. Myreen, X86-TSO: a rigorous and usable programmer's model for x86 multiprocessors, Commun. ACM 53 (7) (2010) 89–97.

[37] B.-J. Kwak, N.-O. Song, L.E. Miller, Performance analysis of exponential backoff, IEEE/ACM Trans. Netw. 13 (2) (2005) 343–355.

[38] I. Corporation, Intel® oneAPI threading building blocks, 2023, URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html.

[39] A. Duran, X. Teruel, R. Ferrer, X. Martorell, 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.

[40] 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 Trans. Math. Softw. 45 (2) (2019) 1–35.

[41] J.M. Bull, F. Reid, N. McDonnell, A microbenchmark suite for OpenMP tasks, in: International Workshop on OpenMP, Springer, 2012, pp. 271–274.

[42] S. Huss-Lederman, E.M. Jacobson, J.R. Johnson, A. Tsao, T. Turnbull, Implementation of strassen's algorithm for matrix multiplication, in: Supercomputing'96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, IEEE, 1996, p. 32.
[43] J.A. Calvin, C.A. Lewis, 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, 2015, pp. 1–8.
[44] I.S. Duff, M.A. Heroux, R. Pozo, An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum, ACM Trans. Math. Softw. 28 (2) (2002) 239–267.
[45] J.M. Wozniak, T.G. Armstrong, M. Wilde, D.S. Katz, E.L. Lusk, I.T. Foster, Swift/t: scalable data flow programming for many-task applications, in: PPOPP'13, 2013.
[46] C. Mei, G. Zheng, F. Gioachin, L.V. Kalé, Optimizing a parallel runtime system for multicore clusters: A case study, in: Proceedings of the 2010 TeraGrid Conference, 2010.
[47] A. Amer, H. Lu, P. Balaji, M. Chabbi, Y. Wei, J. Hammond, S. Matsuoka, Lock contention management in multithreaded MPI, ACM Trans. Parallel Comput. (TOPC) 5 (3) (2018) http://dx.doi.org/10.1145/3275443.
[48] M. Chabbi, A. Amer, S. Wen, X. Liu, An efficient abortable-locking protocol for multi- level NUMA systems, in: PPOPP'17, ACM, New York, NY, USA, 2017, pp. 61–74.
[49] C. Augonnet, S. Thibault, R. Namyst, P. Wacrenier, StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput. Pract. Exp. 23 (2011) 187–198.
[50] T. Gautier, J.V.F. Lima, N. Maillard, B. Raffin, XKaapi: A runtime system for data-flow task programming on heterogeneous architectures, in: Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013, 2013, pp. 1299–1308, http://dx.doi.org/10.1109/IPDPS.2013.66.
[51] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, J.J. Dongarra, Parsec: Exploiting heterogeneity to enhance scalability, Comput. Sci. Eng. 15 (6) (2013) 36–45.
[52] S. Treichler, M. Bauer, A. Aiken, Language support for dynamic, hierarchical data partitioning, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2013, 2013, pp. 495–514.
[53] SIGARCH Comput. Archit. News 36 (5) (2008).
[54] R.D. Blumofe, C.E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (5) (1999) 720–748.
[55] J.-N. Quintin, F. Wagner, Hierarchical work-stealing, in: European Conference on Parallel Processing, Springer, 2010, pp. 217–229.
[56] M. Mitzenmacher, The power of two choices in randomized load balancing, IEEE Trans. Parallel Distrib. Syst. 12 (10) (2001) 1094–1104.
[57] S. Shiina, K. Taura, Almost deterministic work stealing, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–16.

Poornima Nookala is a Research Scientist at Intel. She received her Ph.D. in Computer Science in 2022 from Illinois Institute of Technology. Her research interests include parallel programming models and runtime systems for extreme-scale supercomputing systems, computer architecture, cloud computing systems, and big-data computing. She is particularly interested in bridging the gap between software and hardware layers for enabling both functionality and performance, as well as questioning assumptions made by the software stacks we use today in a rapidly evolving hardware landscape.

Kyle Chard is a Research Associate Professor in the Department of Computer Science at the University of Chicago and a researcher at Argonne National Laboratory. He received his Ph.D. in computer science from Victoria University of Wellington, New Zealand. His research focuses on developing new systems to address various computational and data-intensive problems.

Ioan Raicu is an associate professor in Computer Science at Illinois Institute of Technology (IIT), as well as a guest research faculty in the Math and Computer Science Division at Argonne National Laboratory. He is also the founder and director of the Data-Intensive Distributed Systems Laboratory at IIT. He obtained his Ph.D. in Computer Science from University of Chicago under the guidance of Dr. Ian Foster in 2009. His research work and interests are in the general area of distributed systems. His work has focused on defining and exploring both the theory and practical aspects of realizing many-task computing across a wide range of large-scale distributed systems. He is particularly interested in resource management in large scale distributed systems with a focus on many-task computing, data intensive computing, cloud computing, grid computing, and many-core computing.