BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads

Shintaro Iwasaki
The University of Tokyo
Tokyo, Japan
iwasaki@eidos.ic.i.u-tokyo.ac.jp

Abdelhalim Amer
Argonne National Laboratory
Lemont, Illinois, USA
aamer@anl.gov

Kenjiro Taura
The University of Tokyo
Tokyo, Japan
tau@eidos.ic.i.u-tokyo.ac.jp

Sangmin Seo
Argonne National Laboratory
Lemont, Illinois, USA
sseo@anl.gov

Pavan Balaji
Argonne National Laboratory
Lemont, Illinois, USA
balaji@anl.gov

***Abstract*—OpenMP is widely used by a number of applications, computational libraries, and runtime systems. As a result, multiple levels of the software stack use OpenMP independently of one another, often leading to nested parallel regions. Although exploiting such nested parallelism is a potential opportunity for performance improvement, it often causes destructive performance with leading OpenMP runtimes because of their reliance on heavyweight OS-level threads. User-level threads (ULTs) are more lightweight alternatives but existing ULT-based runtimes suffer from several shortcomings: 1) thread management costs remain significant and outweigh the benefits from additional parallelism; 2) the shift to ULTs often hurts the more common flat parallelism case; and 3) absence of user control over thread-to-CPU binding, a critical feature on modern systems.**

This paper presents BOLT, a practical ULT-based OpenMP runtime system that efficiently supports both flat and nested parallelism. This is accomplished on three fronts: 1) advanced data reuse and thread synchronization strategies; 2) thread coordination that adapts to the level of oversubscription; and 3) an implementation of the modern OpenMP thread-to-CPU binding interface tailored to ULT-based runtimes. The result is a highly optimized runtime that transparently achieves similar performance compared with leading state-of-the-art widely used OpenMP runtimes under flat parallelism, while outperforming all existing runtimes under nested parallelism.

***Keywords*—Multithreading; Runtime Systems; OpenMP; User-Level Threads**

I. Introduction

OpenMP is considered the most popular intranode parallel programming interface in high-performance computing (HPC). Numerous applications, computational libraries, and runtime systems have been successfully parallelized with OpenMP. Combining multiple OpenMP-parallelized codes, however, is trickier than one might imagine. Consider contemporary applications developed on top of deep software stacks. Because the parallelization at each software layer is more or less independent of the other layers, this often leads to nested parallelism where one OpenMP parallel region embeds another. Although an abundance of parallelism is potentially an opportunity for further performance improvement, it often results in catastrophic performance degradation by oversubscription of threads. Because most production runtimes use OS-level threads[1] to represent OpenMP threads, a naive implementation that blindly spawns additional threads at each nested parallel region can hardly tolerate the oversubscription cost.

To address this challenge, two orthogonal directions have been explored: avoidance of oversubscription and reduction of oversubscription overheads. The first direction is adopted by widely used commercial and open source OpenMP implementations such as GCC OpenMP [1], Intel OpenMP [2], and LLVM OpenMP [3]. Their default settings turn off nested parallel regions and consequently are tuned for flat parallelism (i.e., a single-level parallel region). This solution avoids the oversubscription issue without hurting the performance of flat parallelism, but such an aggressive serialization misses any parallelism opportunity exposed by nested parallel regions. This situation is especially true when the top-level parallel region does not have sufficient parallelism or the amount of computation across loop iterations is irregular.

The second direction aims at reducing oversubscription overheads. Leading OpenMP runtimes accomplish this by reusing nested teams (threads and data associated with parallel regions) across parallel regions [4] and by avoiding busy-waiting [5]. Reusing, and thus keeping alive a large number of OS-level threads, is more efficient than recreating and destroying them but requires nevertheless expensive suspension and reactivation operations. Furthermore, taking away busy-waiting implies the involvement of the kernel when synchronizing threads at the entry and exit of a parallel region, which hurts the performance of flat parallelism.

Mapping OpenMP threads to user-level threads (ULTs) has also the benefit of reducing oversubscription overheads thanks to lower fork-join costs. Numerous OpenMP implementations have followed this approach [6]–[11] but fall short for several reasons. First, they ignored other costs outside the native thread fork-join overheads.[2] An OpenMP runtime has to manage other OpenMP-specific data and descriptors that are

[1]Most of these runtimes rely on POSIX threads (Pthreads), which follow in most implementations a one-to-one mapping between a user thread and a kernel thread in OS.

[2]An OpenMP thread has its own descriptor in addition to encapsulating a native thread (be it OS- or user-level).

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00011

orthogonal to the native threading layer. Second, they overly relied on ULT-based fork-join operations that poorly handle flat parallelism, which is more efficiently implemented with busy-waiting methods. Third, they offer virtually no control to the user over thread-to-CPU binding, which is important on modern systems to improve data locality. As a result, existing ULT-based runtimes perform overall worse than finely configured production OpenMP systems, leaving open questions regarding their suitability as all-purpose OpenMP runtimes.

This paper presents BOLT, a practical ULT-based OpenMP runtime that attains unprecedented performance for nested parallelism while also transparently supporting efficient execution of flat parallelism. Through our in-depth investigation of the ULT-based OpenMP runtime optimization space by exploring both generic and OpenMP specification-driven optimizations, we found necessity of optimizations beyond the naive mapping of ULTs and OpenMP threads; our solutions are: 1) team-level data reuse and thread synchronization strategies to minimize overheads in the OpenMP runtime; 2) a novel thread coordination algorithm that transparently achieves high performance for both flat and nested parallelism by adapting to the level of oversubscription; and 3) an implementation of the modern OpenMP thread-to-CPU binding interface tailored specifically to ULT-based runtimes. Our evaluation with several microbenchmarks and N-body and quantum chemistry codes demonstrates that BOLT significantly outperforms existing OpenMP runtimes when parallel regions are nested, and it suffers the least performance loss under flat parallelism.

We note that BOLT has briefly appeared in the past literature [12]–[14], although none of the literature presents the details of its design and performance evaluation. The goal of this work is to identify and address the issues in efficiently utilizing ULTs in OpenMP.

II. Background

Compared with flat parallelism (i.e., a single-level parallel region), efficient handling of nested parallelism in OpenMP has been considered challenging and thus the subject of studies from the early years of OpenMP [6], [7]. In this paper we do not advocate the use of nested parallel regions in a standalone OpenMP program (e.g., [15]); after all, several alternatives, such as task and taskloop constructs, are offered by the OpenMP specification to leverage massive, deeply nested, or recursive parallelism efficiently. The primary focus of this paper is nested parallel regions that the user has limited control over; for example, nested parallelism that takes place across multiple layers of the contemporary software stack. We illustrate this situation with the example in Fig. 1. We observe that two layers of the software stack (the application layer and the external library) depend on the same OpenMP runtime. The user application code calls an external function (e.g., `dgemm()`) in a parallel region, which is also parallelized by a parallel region. An OpenMP runtime that blindly creates OpenMP threads at each parallel region would result in an exponential growth of the number of threads in the system and serious performance degradation.

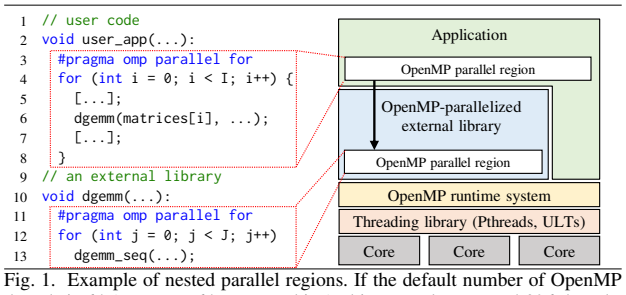


Fig. 1. Example of nested parallel regions. If the default number of OpenMP threads is 64 (e.g., on a 64-core machine), this example creates 4,096 threads.

As shown in Fig. 1, OpenMP runtimes relies on a lower-level threading layer, that we refer to as *native*, to implement OpenMP threads and could be classified into two broad categories: heavyweight OS-level threads and lightweight ULTs. The remainder of this section surveys the current landscape in supporting nested parallelism with respect to the native threading layer.

A. Current State in OS-Level Thread-Based Runtimes

Oversubscribing OS-level threads to hardware cores severely degrades performance in HPC environments because of expensive context switching between threads and preemptive scheduling. Out of fear of oversubscription, OS-level thread-based runtimes focus mostly on avoiding nesting altogether through various methods. In the following, we enumerate the most common practices. In this paper, we assume OpenMP 4.5 [16].

Disabling of nested parallelism. This is the prevailing method in practice. The OpenMP standard specifies an internal variable called `nest-var` that can be controlled via the environment variable `OMP_NESTED` or a function `omp_set_nested()`. The standard defines the default value of `nest-var` as `false`, pessimistically assuming negative side effects from setting it to `true`.[3] This workaround clearly wastes parallelism opportunities and might lead to CPU underutilization.

Manual concurrency control. Users can explicitly control the numbers of threads at each parallel region with the `nthreads-var` control variable. This approach gives more fine-grained concurrency control to the user, which is better than serializing all inner parallel regions. It remains cumbersome, however, to coordinate multiple independent libraries and applications on the degree of concurrency at each parallel region. The optimal configuration is nontrivial to find because it depends on various factors (target hardware, input problem, application characteristicetc.) [18], rendering tuning concurrency at each parallel region impractical.

Collapsing of nested loops. With compiler support, the `collapse` clause partitions iterations across nested loops, where chunks of the nested loops are uniformly distributed while keeping the number of OpenMP threads constant. This method is impractical for parallel regions situated in dependent soft-

[3]OpenMP 5.0 [17] marks `nest-var` as deprecated and sets its default value to implementation defined. We think that most production implementations will continue to support it and disable nested parallelism by default.

ware libraries because it requires code changes and is applicable only to consecutive nested loops.

Dynamic concurrency control. The specification defines the `thread-limit-var` and `dyn-var` control variables that allow users to cap the number of threads in a contention group and to dynamically adjust the number of threads in a parallel region, respectively. While these hints allow the runtime to avoid an exponential concurrency growth, their effectiveness is limited. The control variable `thread-limit-var` suffers from the same shortcomings as the manual concurrency control since users have to tune the upper bound on concurrency. Dynamic concurrency control is implementation defined; for instance, LLVM OpenMP 7.0 calculates the number of currently running OpenMP threads in the process in order to avoid oversubscription. As we demonstrate in Section IV-A, the number of threads is hardly a reliable metric to infer resource utilization because it ignores factors such as load imbalance and thread binding.

Use of OpenMP tasks. OpenMP tasks are designed as lightweight parallel units of execution [19]. For example, LLVM OpenMP implements user-space context switching between tasks. Furthermore, `taskloop` has recently emerged as a task-based substitute for `parallel for` [20]. Unfortunately, because of semantic differences, OpenMP threads are not always replaceable by tasks, as explored in [21]. For instance, OpenMP tasks do not support thread-local storage, some synchronization primitives (e.g., barrier), and CPU binding. In addition, this approach requires rewriting inner parallel loops in every external library, clearly making it impractical. Thus, improving support for nested parallelism remains the most practical direction given that `parallel for` is the predominant form of parallelization by applications and libraries.

B. Current State in ULT-Based Runtimes

A trade-off is possible between exposing parallelism through nested parallel regions and the corresponding thread management costs; however, the high penalizing OS-level thread management costs make searching for the best trade-off challenging. The workarounds introduced by the specification and implemented by leading OpenMP runtimes thus have limited effectiveness when OS-level native threads are adopted. In theory, with extremely lightweight native threads, overprovisioning of threads is significantly less penalizing and renders searching for the best trade-off much more practical. Mapping OpenMP threads to lightweight ULTs is thus a promising approach and has been investigated by numerous studies [6]–[11]

However, we found that existing ULT-based runtimes perform worse than do finely tuned OS-level thread-based runtimes from both flat and nested parallelism perspectives. The benefits of lightweight ULT-based native thread implementations are diminished by overheads of managing OpenMP-specific data and descriptors. The ULT-based runtimes rely on user-level context switching to fork and join threads before and after every parallel region, but this algorithm is inefficient under flat parallelism compared with busy-waiting-based

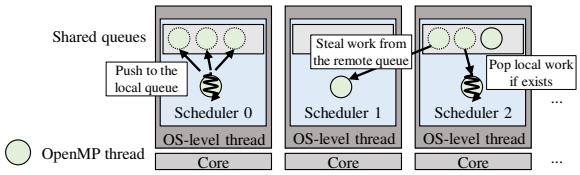


Fig. 2. Basic design of BOLT. Created threads by a scheduler are pushed to its local queue. A scheduler pulls a thread from its own queue; if it is empty, random work stealing takes place.

methods used by leading OS-level thread-based runtimes. In addition, because of absence of an interface to control thread-to-CPU binding, these runtimes miss the opportunity to exploit locality of hierarchical hardware and proximity among threads. As a result, effectiveness of ULT-based OpenMP runtimes remains questionable.

III. BOLT: Efficient ULT-Based OpenMP Runtime

Our goal is to develop a practical OpenMP runtime system that efficiently supports both flat and nested parallelism to exploit the ignored parallelism in contemporary applications built atop highly stacked software layers. However, developing a cutting-edge OpenMP runtime from the ground up is a daunting task. We therefore created a runtime system based on an existing leading OpenMP system, while, to boost the performance of nested parallelism, we adopted ULTs as OpenMP threads. Specifically, we chose the open-source LLVM OpenMP library [3] as a baseline OpenMP implementation in order to take advantage of its maturity in terms of performance and robustness. As for the native threading layer, we chose Argobots [12], a highly optimized user-level threading library.

The next section briefly describes the changes necessary to the upstream LLVM OpenMP in order to run over a user-level threading library. We follow with a deep dive into the various overheads and bottlenecks that occur under flat and nested parallelism and the corresponding solutions. We also describe how they relate to or leverage knowledge from the OpenMP standard, and we identify some shortcomings in the current specification that limit optimization opportunities.

A. Baseline ULT-Based Runtime

BOLT was derived from LLVM OpenMP 7.0 [3] to inherit its optimized and modern OpenMP support up to 4.5 [16] as well as its application binary interface (ABI) compatibility with other popular OpenMP runtimes. The first step in deriving BOLT from LLVM OpenMP is a straightforward replacement of any use of the Pthreads interface with corresponding Argobots function calls. This involves not only replacing fork-join calls and several synchronization mechanisms, but also inheriting configuration parameters (e.g., stack size) and adopting ULT-based thread-local storage (TLS). The critical aspect that motivated using Argobots is, in addition to its high performance, its functionalities similar to those provided by Pthreads, which eased the replacement.

Fig. 2 shows the basic components of BOLT and how OpenMP threads are managed. When the runtime is initialized,

BOLT spawns OS-level threads, typically as many as the number of CPU cores, and runs schedulers on top of each of the OS-level threads, which work as *processors* in the OpenMP standard. Each scheduler has a shared ULT queue that is accessed mainly by the owner but can be a target of work stealing. We adopt a simple random work-stealing algorithm [22]; the scheduler steals ULTs from another queue only when its own queue is empty. This model follows the same practice found in existing ULT-based OpenMP runtimes. OpenMP tasks are also mapped to ULTs, but careful mechanisms are in place to maintain a correct thread-task relationship in order to satisfy the OpenMP semantics (e.g., `tied` tasks).

At this stage, BOLT shows decent performance compared with several OS-level thread-based and ULT-based runtimes. Unfortunately, at this stage, which we refer to as *baseline*, BOLT underperforms in several cases as other ULT-based runtimes do; it relies on low threading overheads of ULTs but mostly ignores OpenMP-specific knowledge in the optimization space. In particular, massive thread parallelism created by nested parallel regions stresses the scalability of data management and thread synchronization operations, diminishing performance gain by the lightweight nature of the native ULTs. To tackle this scalability challenge under nested parallelism, we first focus on the OpenMP specification that enables efficient data reuse and two bottlenecks in thread synchronization that have been overlooked in LLVM OpenMP; these techniques are either known methods in the context of OS-level thread-based systems or general optimizations, while these performance issues are significant with lightweight threads. We then investigate efficient and transparent support for flat parallelism and ULT-based OpenMP thread-to-CPU binding and propose new techniques that are specific to ULT-based OpenMP runtimes.

B. Team-Aware Resource Management

Despite its abstract nature, the concept of *OpenMP team* has important performance implications for nested parallelism by promoting *reuse* and *isolation*. Let us first assume that only flat parallelism is supported. In this case, the concept of team is unnecessary since there is at most one active parallel region at a time. Consequently, it is sufficient to reuse the same set of threads to execute successive parallel regions. Reuse here applies to the OpenMP thread descriptors that contain native threads and the per-thread OpenMP-specific data. The set of threads can also dynamically expand to account for variable thread counts across parallel regions. In LLVM OpenMP and BOLT, an OpenMP thread descriptor embeds a native thread; it follows that reusing an OpenMP thread descriptor allows reusing the same native thread to avoid expensive fork-join calls. In order to support the OpenMP fork-join execution model, barrier synchronization is used in LLVM OpenMP instead. The same is achieved in BOLT by using the `join-revive` Argobots pattern; this is a unique trait of Argobots that allows joining threads without destroying them (similar to a barrier synchronization) and later revive them to execute a new parallel region.

This model is impractical for nested parallelism, however. It is unfeasible to execute all parallel regions by the same set of threads while maintaining parent-children dependencies between regions, independence among sibling regions, and region-local barrier synchronization. The notion of team satisfies these needs and allows for optimizations. The isolation of threads within the same team allows independent teams to run in parallel and limits the scope of barrier synchronization to team-local threads. The above global reuse model can also be adopted at the team level. When a team finishes executing a parallel region, its corresponding data, which includes OpenMP thread descriptors, can be reused for a subsequent parallel region. Exploiting team-level optimizations is not new but has been limited to leading OS-level thread-based runtimes. This reuse method, called *hot team* [4], has been adopted by Intel and LLVM OpenMP and saves not only on thread management operations but also on team-level data management and initialization (e.g., barrier-related data).

The original implementation of hot teams is limited to the outermost parallel region by default, because the OS-level thread is a precious system resource; since the number of OS-level threads grows exponentially, keeping them alive can rapidly reach the system limit. This is not an issue for ULT-based runtimes since threads are managed in the user space. Here the primary physical limit is memory but it can fit a massive number of ULTs since their memory footprint is relatively small (the largest object is stack, which is 4 MB by default). As a result, the hot team optimization has even more potential for ULT-based runtimes since caching ULTs only consumes memory without wasting a system resource.

C. Scalability Optimizations

Shifting the threading layer of the original LLVM OpenMP to ULTs reveals scalability bottlenecks that were not previously visible because the costs of managing OS-level threads dwarfed them. These bottlenecks are related to accessing shared data and the startup overhead of a parallel region that we address as follows.

1) Scalable Shared Data Management: The first bottleneck is related to how shared data within the runtime is protected. A coarse-grained critical section was protecting several runtime resources, including global thread and team descriptor pools, which are used if teams are not cached, and global thread IDs. Accessing this data is on the critical path of every construction of a parallel region. This critical section also serializes updates of thread counters that are used to adjust the number of threads (e.g., as hinted by `thread-limit-var`). This is a generic critical section contention issue, which we alleviated with established contention-avoidance practices. We divided the coarse-grained critical section into smaller ones that protect distinct resources. Since it belongs to the associated master thread, hot team data needs no protection, which gets manipulated in a lockless manner. Thread counters are kept consistent by using hardware atomic operations. These optimizations eliminate most serialization.

2) Scalable Thread Startup: The second bottleneck is related to the startup phase of a parallel region where the master thread distributes work to threads in the team. The baseline `join-revive` model employs an $O(N)$ distribution algorithm that limits scalability under nested parallelism. This is a known pattern that has been improved with $O(logN)$ divide-and-conquer algorithms such as done by Intel CilkPlus [23] and Intel Threading Building Blocks [24]. LLVM OpenMP adopts the same approach but only for `taskloop`, however. Thus, we applied this model to the `join-revive` pattern by distributing work in a binary tree manner until the number of revived threads is one.

Our two optimizations drastically improve the scalability of the baseline BOLT under nested parallelism. Nevertheless, it underperforms in the case of flat parallelism. Across successive parallel regions, native threads (i.e., ULTs) cached by the hot team optimization are coordinated by ULT-based synchronization that essentially relies on lightweight user-level context switching. This ULT-based coordination method, however, performs worse than busy-waiting when no oversubscription happens. The next section closely explores better thread coordination strategies across parallel regions, a direction that was completely overlooked by previous ULT-based runtimes.

D. Thread Coordination Across Successive Parallel Regions

With OS-level threads, even if the hot team eliminates thread creation costs with OpenMP threads as we presented in Section III-B, sleeping and awakening threads on every parallel region invocation are costly since they involve expensive OS-level context switching. Production OpenMP runtimes by default enable an aggressive synchronization that keeps finished OpenMP threads busy-waiting in order to save the cost of waking up threads on creating the succeeding parallel region. Nevertheless, busy-waiting is obviously harmful under oversubscription cases because it essentially wastes CPU resources. The OpenMP specification gives control to users via a run-time interface, `OMP_WAIT_POLICY`, that hints at the runtime the desired behavior of waiting threads with `active` and `passive` keywords; the specification explains that `active` implies busy-waiting, while `passive` sleep- or yield-based implementations. Although the specification defines the behavior as implementation defined, this wait policy is exploited by leading OpenMP runtimes to alleviate overheads of parallel region creation.[4] This setting is known to significantly affect the performance of flat and nested parallelism [18]; the active policy minimizes the latency for a repeated, short, and single-level parallel region, but it imposes immense overheads when oversubscription happens (e.g., parallelism is nested). On the other hand, with the passive policy, the latency is large under flat parallelism.

[4]For example, with `active`, LLVM OpenMP 7.0 infinitely busy-waits for the next parallel region; but if the total number of OpenMP threads is greater than the number of hardware threads, `sched_yield()` is called in the busy loop. Under the `passive` policy, threads immediately sleep after finishing their work; team reinvocation relies on a tree-based barrier using Pthreads condition variables.

```
1  const int YIELD_INTERVAL = 1e6;
2  void omp_thread_func(...):
3    START_THREAD:
4    [...]; // run an implicit task (= work of OpenMP thread)
5    switch (omp_wait_policy):
6    case ACTIVE:
7      while (1):
8        if (is_next_team_invoked()):
9          goto START_THREAD; // restart a thread
10       if (count++ % YIELD_INTERVAL == 0):
11         yield_to_sched(); // avoid hang
12   case PASSIVE:
13     return; // immediately returns to a scheduler
14   case HYBRID:
15     while (1):
16       if (is_next_team_invoked()):
17         goto START_THREAD; // restart a thread
18       if (thread = pop_from_one_of_pools()):
19         // finish, return to a scheduler, and execute thread
20         return_to_sched_with_thread(thread);
```

Fig. 3. Pseudo code of wait policy implementation in BOLT. We omit detailed flag management from this code. Here `yield_to_sched()` behaves as preemption, which is necessary to avoid a dead lock with the `active` policy.

1) Static Wait Policy: A common misconception is that the threading cost of ULTs is minuscule (e.g., as small as synchronization based on busy-waiting). Indeed, the previous ULT-based OpenMP runtimes have ignored the wait policy and merely implemented a passive behavior relying on user-level context switches. Nevertheless, we found that even with lightweight ULTs, thread coordination algorithms indicated by the wait policy have a large performance impact on flat and nested parallelism because under no oversubscription busy-waiting is more efficient than user-level context switches containing several memory accesses. BOLT is aware of the importance of flat parallelism and therefore employs both active and passive strategies, which allow efficient execution of flat parallelism if `active` is specified. Fig. 3 shows the pseudo implementations of the wait policies in BOLT; under the active policy a flag (`is_next_team_invoked()`) is checked in a busy loop (line 8), whereas with `passive` a thread finishes immediately after its work to possibly schedule another ready thread (line 13).

However, this static policy mechanism requires users to prioritize the performance of either flat or nested parallelism. The API proposal by Yan et al. [5], which allows the policy change at runtime with `omp_set_wait_policy()`, can alleviate the current one-time black-and-white setting. However, it imposes a burden on users to control optimal settings, which is impractical for real-world complicated applications that consist of both flat and nested parallelism.

2) Hybrid Wait Policy: To address this issue, we propose a *hybrid* policy, a new ULT technique that encompasses both strengths by executing the active and passive behaviors *alternately*. This optimization comes from the observation that both implementations are composed of busy loops; the active implementation obviously contains busy-waiting, whereas with `passive`, after the thread finishes, the scheduler enters a busy loop to pop and execute the next ULT. The hybrid implementation embeds the ULT pool operations in the thread coordination code and checks a flag and availability of ULTs alternately in a busy loop, as shown in Fig. 3. If a ULT is successfully taken in this loop (line 18), the thread exits and returns to the scheduler with the popped ULT, which will be executed next (line 20). This hybrid strategy consists

of both the active and passive strategies; it works as `active` with a pool-checking overhead (line 18) under flat parallelism, while `passive` with a flag-checking overhead (line 16) under oversubscription. Since any expensive operation is involved in a busy loop to check a pool and a flag alternately, it performs almost best in both flat and nested parallelism cases without the programmers' burden to choose the wait policy.

This hybrid technique is not applicable to OS-level thread-based implementations since a kernel does not expose a scheduling loop to users. Instead of a *hybrid* behavior, one might suggest an *adaptive* strategy that dynamically switches active and passive modes based on plausible metrics: for example, total numbers of threads and parallel regions, depth of nests, CPU loads, and real performance obtained by profiling. However, such an adaptive technique is potentially harmful because of the expensive wait policy change that requires suspension and reactivation of OS-level threads. We note that our hybrid technique has the least negative side effect under the assumption of ULT scheduling; especially the hybrid technique does not perform any context switching without acquiring the next work unlike `yield_to_sched()` (line 11) with `active`, which eliminates unnecessary context switches that increase the latency in the thread restart path.

Unfortunately, neither the OpenMP specification nor the proposal of extensions [5] contains a keyword that can be mapped to this hybrid behavior. This paper uses a keyword `hybrid`, while we simply suggest a keyword `runtime` or `auto` to the specification, which allows a runtime to choose the best algorithm (i.e., the hybrid algorithm in the case of BOLT).

E. Thread-to-Place Binding

In order to efficiently run a parallel program on modern hierarchical multi-core CPUs, in addition to efficient execution of parallel regions, exploitation of locality is essential. OpenMP introduced the concept of thread binding, which allows users to hint preferable thread affinity; this facilitates exploiting data locality by mapping threads to the hardware topology and by exploiting physical proximity among threads. Specifically, OpenMP allows fine-grained affinity control via *places*; users can define *places* that encapsulate sets of hardware threads, and OpenMP threads can be bound to specific places according to a given binding setting (`bind-var`). This thread-to-place binding interface is straightforward to use for flat parallelism, but it is not so trivial for nested parallelism. For instance, the user can use the binding interface to carefully map threads in a way that maximizes resource utilization. This solution, while already cumbersome for the user, becomes impractical the moment the thread count exceeds the number of processors or the per-thread workload is irregular. In this case, dynamically scheduling threads is more practical.

Dynamically moving threads can take place only within one place, however. This creates a multidimensional trade-off between data locality granularity (small places give more fine-grained control), load balancing (larger places allow for better utilization) and scheduling overheads (moving threads within one place). The quest for the best trade-off favors ULT-based

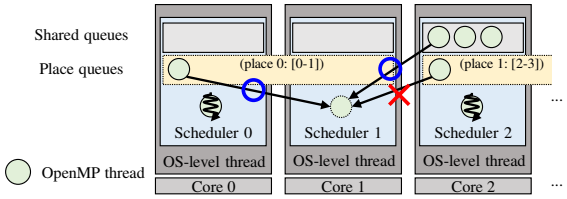


Fig. 4. Place queues in BOLT. In addition to shared queues, Scheduler 1 can access its own place queue (place 0), but not the other place queues.

runtimes, thanks to high-control and low-cost scheduling, but remains completely unexplored since existing ULT-based runtimes support only old OpenMP specifications. Whether an OpenMP runtime fulfills the thread-to-place binding specification is implementation defined; thus, the baseline BOLT remains standard-compliant but prohibits the user from the corresponding optimizations. We believe this should not be the case; that a ULT-based runtime can equip the user with the same optimization tools as OS-level thread-based runtimes do. In the following, we describe our binding support, which is fully compliant with the specification.

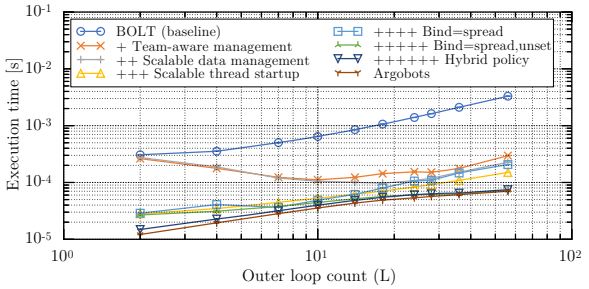
1) ULT-Based Thread-to-Place Binding: OS-level thread-based runtimes often rely on CPU masks to map threads to places; the OS schedules threads only on CPU sets that threads are allowed to run onto. This approach is not practical for ULT scheduling because they are scheduled by using decentralized thread queues. For a close mapping between places and thread queues, we created the concept of *place queue* associated with a place (i.e., a user-defined set of *processors*, or schedulers in BOLT) as shown in Fig. 4. Since only schedulers associated with a place have access to a corresponding place queue, ULTs bound to the place queue can be executed among these limited schedulers. By pinning schedulers in BOLT to hardware threads, ULTs are virtually bound to specific core sets. Places are immutable once defined, so BOLT does not need to create and destroy place queues dynamically; thus, the additional overhead to support places is negligible.

2) The Problem with Binding Policy Inheritance: The previous step allows BOLT users to control thread mapping as they would do with widely used OpenMP runtimes. Unfortunately, the deterministic nature of this binding interface constrains thread scheduling; it ignores processor utilization at runtime and can lead to load imbalance. We believe that to approach the optimal trade-off, a promising strategy is to combine tight binding policies for the outermost parallel region (to promote data locality) with loose binding policies for inner regions to allow threads more freedom to move and exploit dynamic load balancing. This strategy maps cleanly to the BOLT internal thread queue and scheduling system, but there is an obstacle in the specification. If the binding policy for the outermost parallel region is set, the inner binding policy either inherits the parent region's policy if the user does not set it or takes the user-chosen policy. In both cases, binding is always enforced onto the inner parallel regions, which prohibits dynamic scheduling.

To address this issue, we suggest an extension to the

```
1 #pragma omp parallel for num_threads(L)
2 for (int i = 0; i < L; i++)
3   #pragma omp parallel for num_threads(N)
4   for (int j = 0; j < N; j++)
5     empty(i, j); // no computation
```

(a) Kernel of the microbenchmark. N is the number of cores.



(b) Performance of the microbenchmark on Skylake.

Fig. 5. Microbenchmark that evaluates overheads of nested parallel regions.

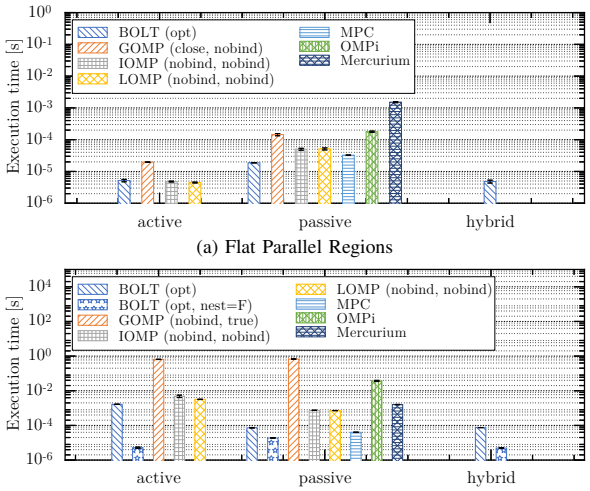
specification to support a new `bind-var` keyword, `unset`, which literally unsets `bind-var`; technically, it sets `bind-var` and `place-partition-var` to the default ones, while in BOLT the default `bind-var` is no thread binding. With this keyword, we can specify the strategy described above by, for instance, setting `OMP_PROC_BIND` to `spread,unset`. With this extension, BOLT is capable of dynamic scheduling through random work-stealing when binding is unset. We believe this keyword is also useful with OS-level thread-based implementation for dynamic load balancing.

F. Performance Breakdown and Analysis

The preceding sections described several implementation aspects that affect the performance of parallel regions in flat and nested parallelism regimes but did not quantify the individual contributions. This section provides a breakdown analysis using simple microbenchmarks that were run on a 56-core Intel Skylake server (detailed experimental setting is provided in Section IV).

1) Performance Breakdown Analysis: To evaluate BOLT under nested parallelism, we first used a microbenchmark that stresses the overheads of nested parallel regions, as shown in Fig. 5a. We set N to the number of cores (i.e., 56) and ran this microbenchmark with different values of L. Fig. 5b shows the results following an incremental bottleneck elimination approach; at each step, the optimization being applied is the one that eliminates the major bottleneck at that step (which does not necessarily follow the same order as the optimization descriptions above). We set `OMP_WAIT_POLICY` to `passive` by default, which is beneficial for nested parallelism.

We found that team construction and destruction take more than 92% of the total execution time on the critical path with $L = 56$. As a result, the hot team optimization (Section III-B) can significantly reduce the team management cost (**Team-aware management**) and improves the execution time by roughly 10x. The next most significant bottleneck is contention for the coarse-grained critical section, which consumes 13% of the execution time when L is 56. By adopting the **Scalable data management** optimization (Section III-C1),



(a) Flat Parallel Regions

(b) Nested Parallel Regions

Fig. 6. Performance of the microbenchmarks that evaluate the effect of wait policy on Skylake.

contention is significantly reduced and the benefits are proportional to the size of the outer parallel region L. Binary thread startup (Section III-C2) shortens the critical path by reducing the workload on the master thread (**Scalable thread startup**), which improves performance especially with smaller L. Merely setting affinity—set to `spread` in this example—does not improve performance (**Bind=spread**) because, as discussed in Section III-E1, the affinity setting is inherited by the inner parallel regions and incurs load imbalance because of the loss of scheduling flexibility. Giving scheduling freedom to the inner nested level by setting `spread,unset` (Section III-E2) improves CPU utilization by reducing load imbalance (**Bind=spread,unset**). The hybrid wait policy presented in Section III-D2 improves performance with smaller L because of its active behavior, while it shows the least performance degradation with larger L (**Hybrid policy**).

We also compared the optimized BOLT with the pure Argobots library, which we consider as the upper bound on performance. We created a microbenchmark that is directly parallelized with Argobots in the same way as done with the optimized BOLT; we mimicked BOLT's scheduling and thread management (e.g., affinity and team-aware resource management) but removed OpenMP function calls and omitted other unused OpenMP features for this microbenchmark (e.g., initialization of task queues and management of thread IDs). Fig. 5b indicates that the optimized BOLT incurs up to 23% overheads compared with Argobots (**Argobots** in the figure). We think any further performance improvement beyond this point must involve improving Argobots itself.

2) Wait Policy and Performance: To assess the trade-off of the wait policy, we evaluated the performance of flat and nested parallelism with both the active and passive policies. We used the optimized BOLT (**Bind=spread,unset** in Fig. 5b). In addition to the widely used OpenMP implementations (GCC, Intel, and LLVM OpenMP), we evaluated three ULT-based OpenMP runtimes: MPC [9], OMPi [8], and Mer-

curium [25]. Section IV includes the details of the OpenMP runtimes we evaluated. We note that these ULT-based runtimes employ only the passive strategy (i.e., no busy-waiting). We tuned the affinity settings of GCC, Intel, and LLVM OpenMP as done in Section IV-A.

Fig. 6a shows the overheads of a single parallel region creating 56 threads doing no computation on the Intel Skylake processor. We show the best affinity settings for GCC, Intel, and LLVM OpenMP (**GOMP**, **IOMP**, and **LOMP**) in the figures; the first affinity was set for `active` and the other for `passive`. Fig. 6a shows that **BOLT** with `passive` is faster than **GOMP**, **IOMP**, and **LOMP** with `passive`, thanks to lightweight ULTs, while it is slower than **IOMP** and **LOMP** with `active` because their coordination algorithms based on busy-waiting are more efficient than is the passive algorithm relying on user-level context switching. **BOLT** with `active` performs as good as do **IOMP** and **LOMP**. The previous ULT-based OpenMP runtimes (**MPC**, **OMPi**, and **Mercurium**) show higher overheads than do the OpenMP implementations with OS-level threads with `active`, indicating the importance of the active wait policy for efficient support of flat parallelism even with lightweight ULTs.

Fig. 6b shows the performance of nested parallel regions creating 56 threads at each level. **GOMP** suffers from immense thread management overheads, diminishing the performance difference between `active` and `passive`. **IOMP** and **LOMP** in this case significantly degrade performance with `active` because busy-waiting delays execution of other threads that have real work. **BOLT** shows the same performance tendency, but is faster than the other OpenMP runtimes except MPC; in Fig. 6b, MPC shows the best performance because the implementation of MPC does not allow oversubscription and completely serializes inner parallel regions. BOLT can achieve better performance by disabling nested parallelism (**BOLT (nest=F)**). In Section IV-A we discuss the performance penalty of aggressive serialization.

These results demonstrate that the performance of flat and nested parallelism is sensitive to the wait policy. BOLT with `hybrid` in both cases shows almost the best performance, proving the efficacy of the `hybrid` algorithm which eliminates a burden to manually tune the wait policy but maintains high performance under both flat and nested parallelism.

IV. Evaluation

In this section, we compare the performance of BOLT with six other OpenMP runtimes using carefully crafted microbenchmarks and two real-world N-body and chemistry applications that exhibit nested parallelism. We ran experiments on the heavily threaded Intel Skylake and KNL systems described in Table I. Since not every OpenMP runtime found in literature has a publicly available or usable implementation, we present results only with runtimes that we could collect and run. The OS-level thread-based category contains the GCC OpenMP [1], Intel OpenMP [2], and LLVM OpenMP [3] runtimes that ship with GCC 8.1, Intel 17.2.174 (17.0.4 on KNL), and Clang/LLVM 7.0 [26], respectively. The ULT-based category consists of MPC 3.3.0 [9], OMPi 1.2.3 [8] with psthreads 1.0.4 [27], and Mercurium 2.1.0 [25] with Nanos++ 0.14.1.[5] BOLT does not have its own compiler, so we compiled programs with the Intel compiler and replaced the OpenMP library with BOLT by modifying `LD_LIBRARY_PATH`.

TABLE I
Experimental environment

Name	Skylake	KNL
Processor	Intel Xeon Platinum 8180M	Intel Xeon Phi 7230
Architecture	Skylake	Knights Landing
Frequency	2.5 GHz	1.3 GHz
# of sockets	2	1
# of cores	56 (28 × 2)	64
# of HWTs	112 (56 × 2)	128 (64 × 2)
Memory	396 GB	128 GB
OS	Red Hat 7.4	Red Hat 7.4

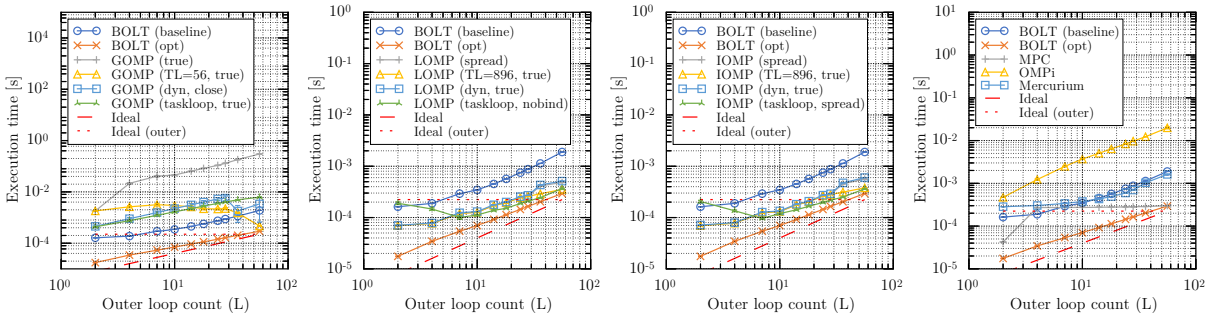
All programs were compiled with `-O3`. To evaluate nested parallelism, we set `OMP_NESTED` to `true`. For a fair comparison, we enabled nested hot teams for LLVM and Intel OpenMP for efficient resource management. In our evaluation, the hybrid policy was enabled for BOLT. For other runtimes, we followed the common practice; we set `OMP_WAIT_POLICY` to `active` if nested parallelism is not used while disabling the busy-wait configuration under any nested parallelism, an approach that is overall beneficial as discussed in Section III-F2. The optimized BOLT includes all the optimizations of Section III with the affinity setting of `OMP_PROC_BIND=spread,unset`, which performed best in the following experiments. We present the results as the arithmetic mean of ten runs with a 95% confidence interval shown as error bars. We note that some bars are hardly visible because of small error values.

A. Microbenchmarks

We first evaluated two microbenchmarks that reflect cases where the efficiency of nested parallel regions impacts performance. Unlike what we used in Section III-F, the computation is added in order not to let merely aggressive serialization be the best optimization. The first case is the microbenchmark shown in Fig. 9a. When L is less than the number of cores (N), parallelizing only the inner or the outer parallel loop cannot utilize all the available cores. We changed the outer loop count L and evaluated the performance.

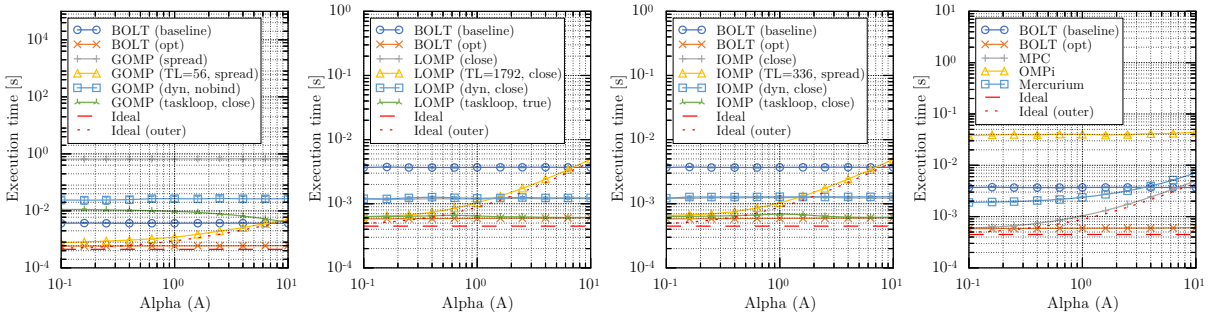
The second case is a program that has unbalanced inner parallel loops, as shown in Fig. 9b. Since the amount of work of the inner loop is uneven, load imbalance occurs if only the outer loop is parallelized. In theory, parallelizing only the inner loop achieves performance as good as that of nested parallelism, although disabling outermost parallelism is difficult in practice because outer parallel loops often contain other computations. This benchmark has a parameter α (`A` in Fig. 9b) to control the degree of imbalance. Let N be a number

[5]We could not find the source code of Omni/ST [7] and NanosCompiler [6]. The source code of ForestGOMP [11] is available, but we did not include it because we could not compile and run it in our environment. The latest libKOMP [10] (https://gitlab.inria.fr/openmp/libkomp) focuses on tasking and no longer maps OpenMP threads to ULTs, so we exclude it in our evaluation.



(a) GCC OpenMP (GOMP) (b) LLVM OpenMP (LOMP) (c) Intel OpenMP (IOMP) (d) ULT-based OpenMP

Fig. 7. Performance of the microbenchmark that has insufficient parallelism on Skylake.



(a) GCC OpenMP (GOMP) (b) LLVM OpenMP (LOMP) (c) Intel OpenMP (IOMP) (d) ULT-based OpenMP

Fig. 8. Performance of the microbenchmark that has unbalanced inner loop parallelism on Skylake.

```
1 #pragma omp parallel for num_threads(L)
2 for (int i = 0; i < L; i++)
3   #pragma omp parallel for num_threads(N / 2)
4   for (int j = 0; j < N / 2; j++)
5     computation(i, j, 20000 /* cycles */);
```

(a) Insufficient parallelism

```
1 #pragma omp parallel for num_threads(N)
2 for (int i = 0; i < N; i++) {
3   int c = 20000 * N * pow(i + 1, A) / (pow(1, A) + ... + pow(N, A));
4   #pragma omp parallel for num_threads(N)
5   for (int j = 0; j < N; j++)
6     computation(i, j, c /* cycles */);
7 }
```

(b) Unbalanced inner loop parallelism

Fig. 9. Kernels of the microbenchmarks that evaluate nested parallel regions.

of cores. The computation size of the ith outer loop iteration is set to W_i cycles, where W_i is calculated as follows:

$$W_i = 20000 \cdot N \cdot \frac{(i+1)^\alpha}{\sum_{j=1}^{N} j^\alpha}.$$

W_i is $20,000$ when α is 0 and gets unbalanced with larger α. By definition, the total amount of work is always $20000 \cdot N$ regardless of α. We changed α from 0.1 to 10.

Each measurement calculates the average execution time of the kernel repeated for three seconds after a one-second warm-up. To evaluate the common workarounds in OpenMP, in addition to the default setting, we measured the performance of the dynamic adjustment of thread counts (i.e., `dynamic-var` (**dyn**) and `thread-limit-var` (**TL**)). For `thread-limit-var`, we tried several numbers (N, $2N$, $4N$, $6N$, $8N$, $12N$, $16N$, and $32N$). Thread affinity impacts performance, so we evaluated four settings—**true**, **close**, and **spread** set `OMP_PROC_BIND` to `true`, `close`, and `spread`, respectively, and setting `OMP_PLACES` to `cores`, and **nobind**, which unsets those variables. Because of space limits, although we tried all the combinations, we show the performance of the fastest series[6].

Fig. 7 and Fig. 8 show the performance of BOLT, GCC, Intel, and LLVM OpenMP with several settings and three ULT-based OpenMP systems. We split charts for better readability, so the results of BOLT are identical among the four charts. **BOLT (baseline)** denotes the baseline BOLT, and **BOLT (opt)** includes all the optimizations. **Ideal** and **Ideal (outer)** show the theoretical maximum performance if all the parallelism is exploited and only the outer loop is parallelized, respectively. The results indicate that **BOLT (opt)** overall performs better than **BOLT (baseline)** and the other OpenMP systems. We note that MPC does not allow oversubscription, so it serializes inner parallelism except $L = 2$ in Fig. 7. This result shows that such an aggressive serialization adopted by MPC fails to exploit parallelism and, at maximum, achieves performance as good as that of **Ideal (outer)**. We note that OpenMP threads in BOLT is as efficient as or even faster than OpenMP `tasks` in GCC, Intel, and LLVM OpenMP; **taskloop** shows the performance of the microbenchmarks in which we replaced the inner parallel loop with `taskloop`.

To illustrate the benefits of nested parallelism in real-world cases, we chose two applications, KIFMM [28] and Qbox [29], for evaluation. They are good examples of nested parallel regions in real-world code; outer parallel loops appear in application codes, and inner parallel loops are in external math libraries. Our evaluation used Intel OpenMP for comparison

[6]Specifically, since one series contains multiple results at different X values, we chose one that has the smallest geometric mean of execution time.

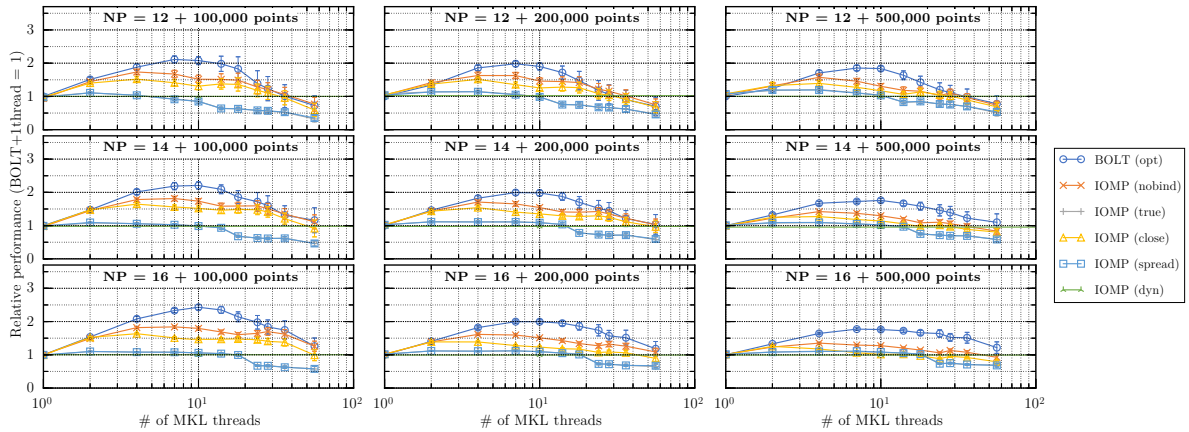


Fig. 10. Performance of the traversal in the upward phase of KIFMM on Skylake.

```
1 for (int i = 0; i < max_levels; i++)
2   #pragma omp parallel for
3   for (int j = 0; j < nodecounts[i]; j++) {
4     [...];
5     dgemv(...); // dgemv() creates a parallel region.
6   }
```

Fig. 11. Kernel of the upward phase in KIFMM; `dgemv()` is parallelized with OpenMP's `parallel for` in MKL.

because 1) it performs best among the existing OpenMP runtimes, and 2) its runtime and compiler are expected to perform best on Intel machines with Intel MKL.

B. KIFMM

KIFMM is a kernel-independent fast multipole method that is an efficient solver of N-body problems [30]. Our evaluation used its highly optimized implementation [28]. The phases *upward* and *downward* are time-consuming phases that have node traversals consisting of OpenMP-parallelized loops calling BLAS routine `dgemv()`, as presented in Fig. 11. Major BLAS implementations including Intel MKL and OpenBLAS provide OpenMP-parallelized implementations, so applications developers might want to exploit nested parallelism especially when loop counts of outer parallel loops are small.

We changed two factors in KIFMM to evaluate the performance of nested parallelism. The first is the number of points (i.e., N in N-body), which affects `nodecounts[i]` in Fig. 11; larger N creates more nodes at each level. When the input contains more points, inner loop parallelism becomes less important because outer parallelism is adequate. The second factor is `NP`, a parameter determining the accuracy of outputs. It affects the input matrix size of `dgemv()`; larger `NP` requires larger matrix-vector multiplication. Parallelizing small `dgemv()` does not perform well because of threading overheads and bad locality, so nested parallelism can be more efficiently exploited with larger `NP`. We artificially changed these input parameters and evaluated the effectiveness of nested parallelism. We used the input following the Plummer model. We also manually revectorized code with AVX-512, where the outdated SSE vectorization was embedded. We used OpenMP-parallelized Intel MKL for the BLAS library. Since it assumes Pthreads as a native thread, a few functions in Intel MKL create, use, and destroy TLS via the OS-level thread API (e.g., `pthread_specific` instead of `omp threadprivate`) or implement their own synchronization algorithms with non-OpenMP functions (e.g., a hand-written barrier instead of `omp barrier`); we overrode these functions to correctly run `dgemv()` on BOLT. We executed KIFMM ten times and calculated speedups of the upward's traversal with different `MKL_NUM_THREADS` while setting `OMP_NUM_THREADS` to 56. In addition to different affinity settings, we measured the performance of **dyn** by setting `OMP_DYNAMIC` to `true` and letting the runtime decide the number of OpenMP threads.

Fig. 10 shows the relative performance of the traversal in the upward phase, where the baseline is the performance of BOLT with one MKL thread; the performance obtained by parallelizing only the outer loop is the result with a single MKL thread. Fig. 10 indicates that nested parallelism contributes to overall performance improvement, and in all cases BOLT achieves the best performance with a certain number of MKL threads, while the excessive increase of inner threads enlarges threading overheads and degrades performance. As we increase the number of points (from left to right in Fig. 10), the performance improvement gets small because the outer loop parallelism becomes sufficiently large. On the other hand, larger `NP` increases the granularity of `dgemv()`, emphasizing larger benefits of nested parallelism. Importantly, if the number of MKL threads is 1, the performance of BOLT is the same as that of Intel OpenMP, showing that BOLT has no performance penalty for flat parallelism compared with Intel OpenMP.

C. Qbox

Qbox is a first-principles molecular dynamics code [29] supporting both intranode parallelism with OpenMP and internode parallelism with MPI. We chose the implementation in the CORAL benchmark with the Gold benchmark, which computes the electronic structure of gold atoms. We focus on a fast Fourier transform (FFT) phase in Qbox that creates nested parallel regions. Qbox contains several FFT operations in the kernel; in the Gold benchmark, 3D FFT is the most time-consuming among them. With OpenMP-parallelized

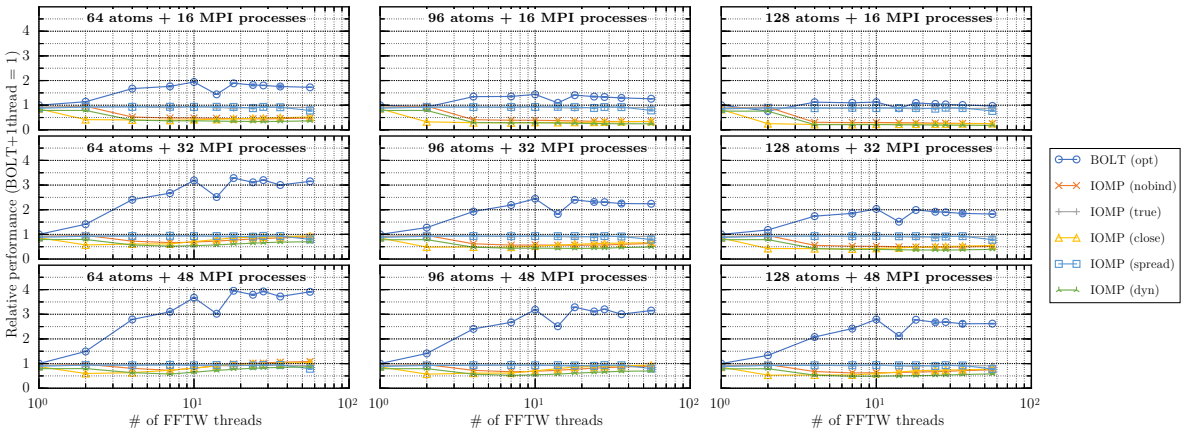


Fig. 12. Performance of the 3D FFT in the FFT backward phase on KNL. The range of single-threaded execution time per iteration is from 30 to 200 milliseconds.

```
1 #pragma omp parallel for
2 for (int i = 0; i < num / nprocs; i++)
3   // fftw_execute() internally creates a parallel region.
4   fftw_execute(plan_2d, ...);
```

Fig. 13. Kernel of the 3D FFT in Qbox with a 2D FFTW plan. The single-threaded execution time per iteration is about 0.8 milliseconds.

FFTW 3.3.8 [31], Qbox offers two ways to parallelize 3D FFT: one is executing a single plan that performs 3D FFT in FFTW3, and the other is running a parallel loop invoking 2D FFTs. We found that FFTW3 internally creates parallel regions for each dimension, so parallel regions are nested in both of the 2D and and 3D FFT cases. Since FFTW3 provides only one variable to control the number of threads, we cannot flexibly control parallelism in the 3D FFT case. We therefore evaluated the 2D FFT case.

Fig. 13 presents the kernel of the 3D FFT in Qbox. The inner OpenMP parallel loop calls `fftw_execute()`, which executes a 2D FFT. We changed two parameters to evaluate the nested parallelism. The first one is the number of MPI processes. As implied in Fig. 13, the outermost dimension of the 3D FFT is distributed over MPI processes, so the outer loop parallelism is reduced by increasing MPI processes, making the inner loop parallelism more important for utilizing all cores. The second parameter is the number of atoms, which changes `num` in Fig. 13; larger input having more atoms increases `num`, rendering nested parallelism less significant. We note that the size of the 2D FFT is independent of these parameters.

For evaluation, we extracted the FFT kernel from the Qbox code and simulated its performance by changing the outer loop count according to values we obtained by running Qbox. We created optimized wisdom files on KNL with `FFTW_PATIENT` for all combinations of numbers of FFTW threads and OpenMP settings and used the obtained highly optimized plans. The 2D FFT internally creates two-level nested parallelism in FFTW3. To avoid an excessively fine-grained decomposition, we disabled the third-level parallelism by setting `OMP_MAX_ACTIVE_LEVEL` to 2. The Intel OpenMP settings are the same as for KIFMM. We set `OMP_NUM_THREADS` to 64 and changed the number of FFTW threads.

Fig. 12 shows the relative performance of the 3D FFT in the backward phase. The baseline is the performance of BOLT with one FFTW thread. Fig. 12 demonstrates that only BOLT can exploit nested parallelism and improve performance. This improvement is more significant with fewer atoms and more MPI processes, indicating that exploiting nested parallelism is important for strong scaling, which is increasingly demanded for massively parallel hardware. The result also shows that BOLT outperforms the others; the autotuning process generates efficient plans for the lightweight OpenMP runtime.

V. Related Work

ULTs in OpenMP. Before OpenMP 3.0, OpenMP supports only threads as parallel units, so lightweight ULTs for OpenMP threads have been intensively explored to parallelize multilevel, recursive, or dynamic parallel programs with OpenMP. NanosCompiler [6] and Omni/ST [7] are early studies proposing ULT-mapping of OpenMP threads; however, they lack implementation details beyond the initial concept. As OpenMP evolved, several ULT-based OpenMP runtimes sought for efficient scheduling over ULTs for nested parallelism. The OMPi system [8] adopted a hierarchical scheduling algorithm to execute innermost threads on close cores, in order to improve locality [32]. ForestGOMP [11], an OpenMP runtime library over a lightweight threading library called Marcel [33], adopted a BubbleSched scheduler, which is a NUMA-aware hierarchical scheduling policy based on hardware locality information [34]. Our evaluation shows that OpenMP's thread affinity supported in the latest OpenMP standard with our proposal `unset` can realize thread affinity over ULTs in an efficient, transparent, and flexible manner.

Recent OpenMP studies using ULTs have focused on issues other than nested parallel regions. libKOMP [10], OpenMP over Qthreads [35], and OmpSs [36] utilized ULTs for efficient task parallelism, although their mapping of OpenMP threads are different; libKOMP mapped OpenMP threads to ULTs, and Qthreads' work mapped them to OS-level threads. OmpSs radically stops supporting parallel regions to focus on its own task-oriented parallel programming model, while their

compiler, Mercurium [25], keeps an OpenMP-compatible option. Several researchers tackled the interoperability issue. The developers of MPC [9] integrated their ULT-based OpenMP runtime with their MPI implementation. Some work mapped OpenMP threads to ULTs for distributed programming models, for example, OpenMP over Charm++ [37], [38], although it failed to strictly follow the OpenMP semantics since OpenMP is thoroughly designed for a shared-memory architecture.

Overall, these OpenMP parallel systems might secondarily achieve good performance of nested parallel regions, while to the best of our knowledge they lacked further improvements to efficiently process nested parallel regions, leaving the performance suboptimal and often worse than the leading OS-level thread-based runtimes (e.g., Intel and LLVM OpenMP). BOLT employs several optimizations to fully exploit lightweight ULTs and presents the lowest overheads of flat and nested parallel regions. We believe that our techniques are helpful for existing ULT-based systems to improve the performance of most OpenMP applications since large portion of real-world OpenMP programs are parallelized by parallel regions.

ULTs in Parallel Programming Models. Several high-level parallel programming models, such as Cilk [39], Cilk-Plus [23], Charm++ [40], and X10 [41], have only an abstract *task* as a parallel work unit, in order to leave room for implementing it as a ULT. These parallel systems do not critically suffer from the oversubscription issue because the number of OS-level threads is constant or easily adjustable during execution. OpenMP, however, exposes two types of parallel units: thread and task. As their names imply, thread and task are typically implemented with an OS-level thread and a ULT in many OpenMP implementations. This work shows that, as high-level programming models do, mapping parallel units to ULTs can exploit nested parallelism, while BOLT coexists with OpenMP-parallelized software resources.

Interoperability with Parallel Libraries. In a broader sense, our work tries to address the interoperability issue of multiple parallelized components. A number of studies, such as Lithe [42] and DoPE [43], have proposed low-level abstract systems designed to encapsulate several parallel runtimes and supervise them uniformly. These abstracted runtime layers essentially lose semantics in the original parallel programming models, and hence achieving optimal performance is difficult. Some studies have focused on the interoperability of OpenMP and other threads [5], [44]. Our work focuses solely on the interoperability issue within OpenMP.

VI. Concluding Remarks

We presented in this paper a ULT-based OpenMP runtime, BOLT, that is aimed at production use by offering modern OpenMP support, unprecedented efficiency for nested parallelism support, and flat parallelism support that competes with leading production runtimes. The design of BOLT resulted from an in-depth investigation of implementation aspects as well as OpenMP specification key concepts to drive performance optimizations. We also discovered some specification aspects that limit optimization opportunities in ULT-based runtimes. We showed that the previous ULT-based OpenMP implementations lack efficient support for nested parallel regions while providing outdated OpenMP specification support. The microbenchmarks show that BOLT achieves better performance of nested parallel regions than do both the widely used OS-level thread-based OpenMP runtimes and the state-of-the-art ULT-based runtimes.

This study heavily focused on traditional parallel region-type of execution but left other aspects, such as support for tasks and accelerators, in the context of ULTs as future work. This work solely focuses on OpenMP and thus does not integrate our techniques into non-OpenMP systems although some of our techniques are generally applicable; for example, our hybrid-wait policy should be beneficial for ULT-based parallel systems that have parallel loop abstractions. Hence, another direction of our future work is to evaluate the efficacy of these techniques on other ULT-based parallel systems.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular its subproject on Scaling OpenMP with LLVm for Exascale performance and portability (SOLLVE). We gratefully acknowledge the computing resources provided and operated by LCRC and JLSE at Argonne National Laboratory. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] D. Novillo, "OpenMP and automatic parallelization in GCC," in *Proceedings of the GCC Developers' Summit*, June 2006.

[2] Intel OpenMP Runtime Library. https://www.openmprtl.org/.

[3] OpenMP®: Support for the OpenMP Language. https://openmp.llvm.org/.

[4] X. Tian, J. P. Hoeflinger, G. Haab, Y.-K. Chen, M. Girkar, and S. Shah, "A compiler for exploiting nested parallelism in OpenMP programs," *Parallel Computing*, vol. 31, no. 10-12, pp. 960–983, Oct. 2005.

[5] Y. Yan, J. R. Hammond, C. Liao, and A. E. Eichenberger, "A proposal to OpenMP for addressing the CPU oversubscription challenge," in *Proceedings of the 12th International Workshop on OpenMP*, ser. IWOMP '16, Oct. 2016, pp. 187–202.

[6] G. Marc, A. Eduard, M. Xavier, L. Jesús, N. Nacho, and O. José, "NanosCompiler: Supporting flexible multilevel parallelism exploitation in OpenMP," *Concurrency: Practice and Experience*, vol. 12, no. 12, pp. 1205–1218, Oct. 2000.

[7] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa, "Performance evaluation of OpenMP applications with nested parallelism," in *Proceedings of the Fifth International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, ser. LCR '00, May 2000, pp. 100–112.

[8] P. E. Hadjidoukas and V. V. Dimakopoulos, "Support and efficiency of nested parallelism in OpenMP implementations," *Concurrent and Parallel Computing: Theory, Implementation and Applications*, pp. 185–204, 2008.

[9] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *Proceedings of the 14th International European Conference on Parallel Processing*, ser. Euro-Par 08, Aug. 2008, pp. 78–88.

[10] F. Broquedis, T. Gautier, and V. Danjean, "libKOMP, an efficient OpenMP runtime system for both fork-join and data flow paradigms," in *Proceedings of the Eighth International Conference on OpenMP*, ser. IWOMP '12, June 2012, pp. 102–115.

[11] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "ForestGOMP: An efficient OpenMP environment for NUMA architectures," *International Journal of Parallel Programming*, vol. 38, no. 5, pp. 418–439, Oct. 2010.

[12] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, Oct. 2017.

[13] A. Castelló, R. Mayo, K. Sala, V. Beltran, P. Balaji, and A. J. Peña, "On the adequacy of lightweight thread approaches for high-level parallel programming models," *Future Generation Computer Systems*, vol. 84, pp. 22–31, July 2018.

[14] A. Castelló, S. Seo, R. Mayo, P. Balaji, E. S. Quintana-Ortí, and A. J. Peña, "GLTO: On the adequacy of lightweight thread approaches for OpenMP implementations," in *Proceedings of the 46th International Conference on Parallel Processing*, ser. ICPP '17, Aug. 2017, pp. 60–69.

[15] H. M. Bücker, A. Rasch, and A. Wolf, "A class of OpenMP applications involving nested parallelism," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, ser. SAC '04, Mar. 2004, pp. 220–224.

[16] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.5," Nov. 2015.

[17] ——, "OpenMP Application Program Interface Version 5.0," Nov. 2018.

[18] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng, "Oversubscription on multicore processors," in *Proceedings of the 24th IEEE International Symposium on Parallel Distributed Processing*, ser. IPDPS '10, April 2010, pp. 958–968.

[19] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang, "A proposal for task parallelism in OpenMP," in *Proceedings of the Third International Workshop on OpenMP*, ser. IWOMP '07, June 2007, pp. 1–12.

[20] X. Teruel, M. Klemm, K. Li, X. Martorell, S. L. Olivier, and C. Terboven, "A proposal for task-generating loops in OpenMP," in *Proceedings of the Eighth International Workshop on OpenMP*, ser. IWOMP '13, Sept. 2013, pp. 1–14.

[21] S. N. Agathos, P. E. Hadjidoukas, and V. V. Dimakopoulos, "Task-based execution of nested OpenMP loops," in *Proceedings of the Eighth International Conference on OpenMP*, ser. IWOMP '12, June 2012, pp. 210–222.

[22] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, Sept. 1999.

[23] C. E. Leiserson, "The Cilk++ concurrency platform," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09, July 2009, pp. 522–527.

[24] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[25] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: A research compiler for OpenMP," in *Proceedings of the Sixth European Workshop on OpenMP*, ser. EWOMP '04, Oct. 2004, pp. 103–109.

[26] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the Second International Symposium on Code Generation and Optimization*, ser. CGO '04, San Jose, California, USA, Mar. 2004, pp. 75–86.

[27] P. E. Hadjidoukas and V. V. Dimakopoulos, "Nested parallelism in the OMPi OpenMP/C compiler," in *Proceedings of the 13th International European Conference on Parallel Processing*, ser. EuroPar '07, Aug. 2007, pp. 662–671.

[28] A. Chandramowlishwaran, J. Choi, K. Madduri, and R. Vuduc, "Brief announcement: Towards a communication optimal fast multipole method and its implications at exascale," in *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12, June 2012, pp. 182–184.

[29] F. Gygi, "Architecture of Qbox: A scalable first-principles molecular dynamics code," *IBM Journal of Research and Development*, vol. 52, no. 1.2, pp. 137–144, Jan. 2008.

[30] L. Ying, G. Biros, and D. Zorin, "A kernel-independent adaptive fast multipole algorithm in two and three dimensions," *Journal of Computational Physics*, vol. 196, no. 2, pp. 591–626, May 2004.

[31] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[32] P. E. Hadjidoukas, G. C. Philos, and V. V. Dimakopoulos, "Exploiting fine-grain thread parallelism on multicore architectures," *Scientific Programming*, vol. 17, no. 4, pp. 309–323, Dec. 2009.

[33] S. Thibault, R. Namyst, and P.-A. Wacrenier, "Building portable thread schedulers for hierarchical multiprocessors: The BubbleSched framework," in *Proceedings of the 13th International European Conference on Parallel Processing*, ser. EuroPar '07, Aug. 2007, pp. 42–51.

[34] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, ser. PDP '10, Feb. 2010, pp. 180–186.

[35] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 2, pp. 110–124, May 2012.

[36] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures." *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[37] S. Bak, H. Menon, S. White, M. Diener, and L. Kale, "Integrating OpenMP into the Charm++ programming model," in *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2'17, Nov. 2017, pp. 4:1–4:7.

[38] ——, "Multi-level load balancing with an integrated runtime approach," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '18, May 2018, pp. 31–40.

[39] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98, June 1998, pp. 212–223.

[40] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93, Sept. 1993, pp. 91–108.

[41] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05, Oct. 2005, pp. 519–538.

[42] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with Lithe," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10, June 2010, pp. 376–387.

[43] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August, "Parallelism orchestration using DoPE: The degree of parallelism executive," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, June 2011, pp. 26–37.

[44] X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen, "Compiler and runtime support for running OpenMP programs on Pentium- and Itanium-architectures," in *Proceedings of the Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, ser. HIPS '03, April 2003, pp. 47–55.

VII. Artifact Appendix

A. Abstract

Our artifact provides the software packages (including some links to the open-source projects) and the applications we used to perform our evaluation. All processes are automated by several scripts, so users can easily build the packages, run our benchmarks, and plot the data.

Our evaluation highly depends on an x86/64 machine that supports AVX-512F (e.g., Intel Skylake 8180) and Intel C/C++ Compilers 2017.2.174.

B. Artifact check-list (meta-information)

- **Algorithm:** BOLT: OpenMP runtime system over ULT.
- **Program:** Microbenchmarks that evaluate nested OpenMP parallel regions, KIFMM, and FFTW3.
- **Compilation:** Intel C/C++ Compilers 2017.2.174, Clang 7.0, LLVM OpenMP 7.0, GCC 8.1, MPC 3.3.0, OMPi 1.2.3, and OmpSs 17.12.1.
- **Transformations:** None.
- **Binary:** None.
- **Data set:** All the input is given by arguments embedded in the benchmarking scripts.
- **Run-time environment:** Linux. We used Red Hat 7.6, but our experiments should work on any (recent) Linux. The root permission is not required to build and run our benchmarks.
- **Hardware:** Our experiments should work on any x86/64 machine that supports AVX-512F, but we strongly recommend Intel Skylake 8180 for the KIFMM evaluation.
- **Run-time state:** None.
- **Execution:** The turbo-boost feature should be turned off to reproduce the performance.
- **Metrics:** Execution time.
- **Output:** All the plotting scripts interpret the output properly.
- **Experiments:** Users run benchmarking scripts.
- **How much disk space required (approximately)?:** Disk space enough to build and install compilers locally.
- **How much time is needed to prepare workflow (approximately)?:** Several hours to build and install compilers locally.
- **How much time is needed to complete experiments (approximately)?:** A few days to complete all the experiments.
- **Publicly available?:** We will make it publicly available if our artifact is accepted.

C. Description

1) How delivered: Our artifact is public and anyone can download from the following link:

```
https://zenodo.org/record/3372716
(DOI: 10.5281/zenodo.3372716)
```

2) Hardware dependencies: Our experiments need an x86/64 machine that supports AVX-512F (e.g., Intel Skylake 8180 and Intel Knights Landing). For the evaluation of KIFMM, we strongly recommend to run experiments on Intel Skylake 8180; otherwise, the deadlock or significant performance degradation might happen in Intel MKL.

3) Software dependencies: Our experiments require recent Intel C/C++ Compilers and Intel MKL. For the evaluation of KIFMM, we recommend to use the version of 2017.2.174; otherwise, the deadlock or significant performance degradation might happen in Intel MKL.

D. Installation

`download.sh` will download all the dependent packages except the Intel products. Root privilege is not required.

```
# On a node which is connected to the network.
$ sh download.sh
```

`build.sh` builds and installs all the packages. This process does not require network connection. All the packages are installed in the local artifact directory, so you can uninstall them just by removing the whole directory.

```
# On a node where you want to run the evaluation.
$ sh build.sh
```

E. Experiment workflow

For example, type the following to run our benchmarks:

```
# On a node where you want to run the evaluation.
$ sh run.sh nested_loop
$ sh run.sh kifmm
$ sh run.sh qbox_fftw
# All the outputs (raw log files) are placed in logs/.
$ ls logs
nested_loop qbox_fftw kifmm
$ ls logs/nested_loop
x.log
```

F. Evaluation and expected result

All the performance results are placed in the `log` directory. You can create figures we presented in the paper with our Python scripts. The obtained results should be similar to what we described in the paper.

```
$ python plot/plot_nested_loop.py log/nested_loop/x.log
$ python plot/plot_qbox_fftw.py log/qbox_fftw/y.log
$ python plot/plot_kifmm.py log/kifmm/z.log
# All outputs (pdfs) are placed in pdfs/.
$ ls pdfs
balanced_gcc.pdf balanced_icc.pdf balanced_lcc.pdf ...
```

G. Experiment customization

All the scripts are in the `scripts` directory. Please see the scripts to customize the experiments.

H. Notes

`README.md` in the root directory of the artifact contains all the details including minor software requirements (e.g., CMake and Python) and some tips to build and run our test suite.