博士論文

Doctoral Dissertation

Exploring Lightweight User-level Threading Frameworks for Massive Fine-Grained Parallelism

(細粒度並列プログラムを効率的に実行する軽量なスレッド処理系)

2019-12-06 提出

東京大学大学院 情報理工学系研究科 電子情報学専攻

Department of Information and Communication Engineering
Graduate School of Information Science and Technology
The University of Tokyo

岩崎 慎太郎

Shintaro Iwasaki

学籍番号: 48-177402

指導教員	田浦 健次朗 教授
Advisor	Prof. Kenjiro Taura

Abstract

As the growth of CPU clock speed has been no longer sustained, modern processors adopt a multicore architecture to provide more computing power to users. Since multithreading becomes essential to unleashing the power of modern multicore machines, more and more applications, libraries, and runtime systems are parallelized by threads. The key to achieving high performance on massively parallel processors is keeping all cores busy. Fine-grained multithreading that decomposes a problem into many small pieces of work is considered to be a promising approach to exploiting parallelism in irregular and complex parallel programs. A threading overhead becomes more and more important since it dictates the finest grain size of threads.

Multithreading programming with a traditional *OS-level thread*, however, imposes significant threading overheads, leading to poor scalability of fine-grained and irregular multithreaded programs on highly parallel systems. To overcome the heavyweight nature of OS-level threads, another thread implementation has gained popularity. A *stackless thread* is a lightweight threading method that trims down overheads by removing context switches on fork and join operations. A stackless thread has a few hundred times lower fork-join overheads than an OS-level thread does. However, omitting context switches lacks several scheduling capabilities including suspension and intermediate termination, limiting its programmability.

A *user-level thread* (ULT) is an alternative threading technique that implements a context switch in user space. A ULT is positioned between the two opposite threading techniques; thanks to a lightweight user-level context switch, it is more efficient than an OS-level thread while more capable than a stackless thread. Nonetheless, because of its limited functionality than that of an OS-level thread and its higher threading overhead than that of a stackless thread, parallel unit abstractions in major parallel systems are based on either OS-level threads out of fear of missing capabilities or lightweight stackless threads for performance. This prevailing practice imposes high threading overheads or excessively limits the functionality, making high-performance multithreading unnecessarily challenging.

This dissertation presents that a ULT can be a solid replacement of an OS-level thread and a stackless thread as a means of multithreading. To investigate a highly optimized implementation of ULTs, we design a user-level threading library that accommodate several user-level threading techniques with different trade-offs of performance and functionality. We also develop a ULT-based runtime implementa-

tion for OpenMP, which is one of the most popular high-level multithreading programming models. Our OpenMP library demonstrates that mapping lightweight ULTs to OpenMP threads and tasks remarkably enhances the performance of real-world applications without violating the OpenMP standard.

We first show the design of a highly customizable user-level threading library that offers optimization opportunities to knowledgeable developers. Our proposed framework, Argobots, is a highly optimized user-level threading library that balances generality and specialization by providing a low-level interface to define and control the runtime behavior. The default policies and parameters in Argobots are well optimized for generic use, while Argobots provides flexibility to manage memory resources and customize scheduling algorithms via several functions. Argobots exposes three different parallel execution units associated with an OS-level thread, a stackless thread, and a ULT, each of which provides different capabilities so that users can control the concurrency with necessary features.

Minimizing threading overheads is indispensable to exploiting fine-grained parallelism. We perform an in-depth analysis of performance vs. functionality trade-offs of user-level threading techniques. We identify a point during the execution of a thread that triggers a context switch as one of the highest sources of overheads. This point, which we refer to as a *deviation*, disrupts an otherwise low-overhead run-to-completion execution. We conduct a comprehensive investigation of a wide spectrum of user-level threading methods with respect to how they handle deviations while covering both parent- and child-first scheduling policies.

We implement Argobots with these user-level threading techniques and evaluate their performance on various CPU architectures including Intel Skylake, Intel Xeon Phi, IBM POWER8, and 64-bit ARM. Our evaluation involves an instruction- and cache-level analysis of all methods. Our experiments present that threading methods that assume the absence of deviation and dynamically provide context-switching support on deviation offer the best trade-off between performance and functionality when the likelihood of deviation is low.

To showcase that a ULT can be substituted for a thread abstraction in a high-level multithreading programming model, we developed BOLT, a highly optimized ULT-based OpenMP library derived from LLVM OpenMP. BOLT significantly improves performance over existing OpenMP implementations using OS-level threads thanks to lightweight threading operations. We find that such performance improvement is hardly obtained by a simple adaption of lightweight ULTs. Specifically, it is accomplished on three fronts: (1) advanced data reuse and thread synchronization strategies; (2) a novel thread coordination algorithm 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.

Our evaluation focuses on OpenMP nested parallel regions, which are often unintentionally introduced in real-world applications as a result of independent OpenMP parallelization in multiple software layers. Nested parallel regions have been known to cause the destructive performance with leading OpenMP runtimes because of their reliance on heavyweight OS-level threads. Our BOLT runtime system can successfully exploit such nested parallelism; our experimental results show that BOLT outperforms all existing runtimes under nested parallelism while transparently achieving similar performance compared with leading state-of-the-art OpenMP runtimes under flat parallelism.

These lightweight threading frameworks with our extensions and optimizations are publicly available. Thanks to numerous collaborators and contributors, these libraries are maintained high quality and used by several research and industrial projects. Our developed lightweight threading libraries, Argobots and BOLT, prove that a ULT is a scalable and practical tool for multithreading, which elevates the performance of massive and fine-grained parallel programs in the era of multicore and many-core processors.

Acknowledgment

This thesis contains texts and experiments in publications I wrote while I was studying at the University of Tokyo and Argonne National Laboratory [1–4]. I would like to thank all the coauthors for helping me form this thesis.

First and foremost, I would like to thank Taura-sensei for his tremendous support for as long as six years at Taura Laboratory. He is undoubtedly one of the most passionate and most knowledgeable advisors I have ever met. As a mentor, a teacher, and an advisor, he taught me all skills, techniques, knowledge, and a mindset to be a good researcher. I would not choose a research career if I would have not met him at the University of Tokyo. He also gave me a precious chance to study at Argonne National Laboratory, where I conducted most of my studies in this dissertation.

At Argonne National Laboratory I could work with fantastic researchers. Abdelhalim Amer was my supervisor in the first two years of my internship at Argonne National Laboratory. We had many research discussions, all of which were extremely valuable to shape this dissertation. His careful and in-depth analysis and truly extraordinary writing skill helped me publish papers. Pavan Balaji is my supervisor for the rest of the years at Argonne National Laboratory. He always amazed me by showing his exceptional research and project management skills. I would like to specially thank him for letting me lead the Argobots and BOLT projects. He gave me countless opportunities to have presentations, seminars, and tutorials. They were held at not only Argonne National Laboratory but also other national laboratories such as Sandia and Oak Ridge National Laboratories and academic conferences including PACT, PPoPP, and SC. All of them are precious experiences for my future research career. I deeply thank Sangmin Seo, who devised the designs of Argobots and BOLT and implemented their initial versions, which became the base of my research.

I could not be happier than having prominent researchers as members of my Ph.D. program committee: Kitsuregawa-sensei, Aizawa-sensei, Nakajima-sensei, Sato-sensei, and Suda-sensei in addition to Pavan Balaji and Taura-sensei. I truly appreciate their constructive and critical feedback, which significantly improved my thesis. Irie-sensei and Toyoda-sensei also deserve many thanks for deep and insightful discussions we had during the advisor meetings, helping me develop my work.

It was my great pleasure to spend time with young brilliant students as friends at Taura Laboratory. I would especially thank Byambajav-kun, Christian-san (Helm-san), Endo-kun, Huynh-san, Ito-kun, Inoue-san, Nakazawa-kun, Qiao-san, Sato-san, Shiina-kun, Shimazu-san, and Shoetsu-kun (Sato-kun) for great discussions on coding practice, programming languages, paper writing, and latest studies in the high-performance computing field.

I also thank all the colleagues at Argonne National Laboratory for helping my research and offering me invaluable help to stay in the United States. The staff members including Giuseppe Congiu, Rob Latham, Min Si, and Hui Zhou helped me develop and improve the Argobots interoperability layer in MPICH. Yanfei Guo deserves many thanks for driving the ECP SOLLVE project, which was the primary funding source of my internship at Argonne National Laboratory. Many thanks to Ken Raffenetti for setting up CI environments for Argobots and BOLT. They are essential to maintaining and improving the quality of these threading libraries. All of the staff members kindly helped me involve in the MPICH project and contribute to the MPICH implementation. I would also like to thank Gail Pieper for her revising and improving my papers.

I was inspired by many talented and hard-working internship students I have met at Argonne National Laboratory. I would specially thank Seonmyeong Bak, Yanhao Chen, Amal Fethi, Clément Mommessin, Poornima Nookala, Srinivasan Ramesh, Hengjie Wang, and Xi Wang. As a long-term visiting student, I particularly spent a long time with Kaiming Ouyang, Kwang (Sarunya Pumma), Shu-Mei Tseng, and Rohit Zambre. Without them, I could not have spent a fruitful and enjoyable time at Argonne. Special thanks go to Rohit Zambre for teaching me how to survive in the USA from A to Z. I truly appreciate his support.

The ongoing project on MPI+ULT is collaborative work with Qthreads and Open MPI developers at Sandia and Los Alamos National Laboratories. I would like to thank Stephen Olivier, Noah Evans, and Howard Pritchard for discussions on the MPI+ULT interoperability issues. Feedback in the MPI+ULT hackathon at Albuquerque helped me deeply think about this problem.

Last but not least, I would like to thank my family. I would never have gone through my career without their love, support, and encouragement.

This thesis includes work partially supported by the Exascale Computing Project, 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). I gratefully acknowledge the computing resources provided and operated by the Laboratory Computing Resource Center (LCRC) and the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. I would like to acknowledge the Doctoral Student Special Incentives Program (IST-RA) of Graduate School of Information Science and Technology, the University of Tokyo, for financial support while I was in Japan.

Contents

List of Figures

List of Tables

1 Introduction

The exponential growth of computing power, which is known as Moore's Law, has driven scientific and social innovation in the world. The twentieth century witnessed this unprecedented growth achieved by shrinking the process of semiconductors, which increased clock speed without enlarging power consumption. Dennard's scaling was over, however, after entering the twenty-first century. Since the exponential growth of the single-core performance has become no longer sustainable, the semiconductor industry instead has increased hardware parallelism to enhance the performance of chips as a whole. Applications must be written to utilize hardware parallelism to truly exploit compute power in such modern highly parallel hardware.

One of the major sources of hardware parallelism is core-level parallelism. Multiple cores that run in parallel are embedded into a single processor. Multithreading is the dominant form of parallelization to execute a single program on multiple cores. In a multithreading programming model, multiple independent sequences of work are packaged as *threads* and distributed to cores so that they can be executed in parallel. Numerous research- and production-level parallel programming systems provide an implementation of threads and have successfully exploited thread-level parallelism in countless applications.

Despite the abundance of potential thread-level parallelism in programs, an overhead in threading operations such as thread creation and destruction (often referred to as fork and join), a yield operation, and synchronization dictates how small granularity of each thread in programs can be. For example, consider that a fork-join overhead is 10 microseconds on a certain processor. To keep the ratio of the parallelization overhead less than 5%, the average execution time of threads needs to be at least larger than 200 microseconds. As the numbers of cores in a single chip and compute nodes in a whole system are increasing, achieving strong scaling—shortening execution time while keeping the problem size—becomes increasingly challenging, which demands more lightweight thread implementation.

Moreover, in a highly parallel environment, dynamic load balancing with fine-grained parallelism becomes critical to achieve efficient utilization of CPUs. Such dynamic load balancing is required for not only irregular parallelism that often appears in recursive divide-and-conquer algorithms and graph algorithms but also apparently regular parallelism that becomes irregular because of nondeterminism of hardware performance. For example, even if each thread computes the same amount of data and thus the computation size looks uniform across threads, several speculative behaviors in modern pro-

Table 1.1: Difference of functionalities between an OS-level thread, a stackless thread, and a ULT. We assume Pthreads [95], Argobots [1] tasklets, and Argobots ULTs as implementations of each type of thread.

	OS-level threads	Stackless threads	User-level threads (ULTs)
Fork-join	✓	✓	✓
Synchronization other than fork-join	✓		✓
Suspension	✓		✓
Intermediate termination	✓		✓
Per-thread signal mask	✓		
Non-cooperative preemption	✓		

cessors including out-of-order execution, branch prediction, hierarchical memory caches, and memory prefetching vary execution time of each thread. To keep all cores busy, fine-grained decomposition of a problem and dynamically balance computational load across cores are known to be effective, rendering lightweight threading more important.

Nonetheless, not all the popular thread implementations are suitable for fine-grained lightweight multithreading. A thread implementation provided by an operating system (OS), which is called an *OS-level thread*, [1] has been the most commonly used implementation of threads. An OS-level thread supports all the threading functionalities that nowadays people naturally expect for "threads": fork and join, suspension, various synchronization, thread-local storage, and non-cooperative preemption. However, an OS-level thread has been criticized because of its heavyweight nature associated with kernel operations.

Another thread implementation has been proposed to minimize the threading cost by simply implementing minimum thread functionalities with a function pointer and its argument. Such a thread does not have its stack and can be invoked by a mere function call. Its stackless implementation, however, lacks several threading capabilities including suspension and several types of synchronizations because it is unable to freely save and resume the execution context during execution. In this work, we refer to this thread implementation as a *stackless thread*[2].

An OS-level thread and a stackless thread have extremely opposite trade-offs; an OS-level thread is more capable but heavyweight while a stackless thread is lightweight but lacks several threading features. A *user-level thread* (ULT) is a thread implementation positioned between these two threading techniques. Unlike a stackless thread, each ULT manages its call stack so that every thread has an independent execution context, enabling several threading operations including suspension and thread-thread

[1]An OS-level thread is sometimes referred to as a kernel thread, or a kernel-level thread. In some literature, it is simply called a POSIX thread (Pthreads [95]) since, in most Linux systems, Pthreads follows a one-to-one mapping between a user thread and a kernel thread.

[2]Some literature calls this a run-to-completion thread since such a thread cannot suspend while running. Other studies call this type of thread a task, a microtask, or a tasklet to differentiate it from an OS-level thread.

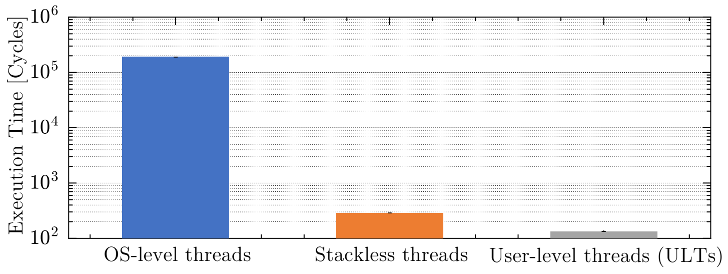


Figure 1.1: Fork-join overheads on an Intel Knights Landing processor using a microbenchmark presented in Figure 3.9 ($(N, n) = (256, 0)$). This experiment used Pthreads, Argobots [1] stackless threads, and Argobots ULTs as implementations of OS-level threads, stackless threads, and ULTs, respectively.

synchronization. ULTs are more lightweight than OS-level threads because ULT's context switch implemented in the user space (which is often called *user-level contest switch*) does not incur heavyweight kernel operations that are the performance bottlenecks of OS-level threads.

Table 1.1 overviews the difference of functionalities between three thread implementations: an OS-level thread, a stackless thread, and a ULT. The table shows that an OS-level thread provides the widest variety of threading functionalities while a stackless thread only supports a basic fork-join operation. A ULT supports more threading operations than a stackless thread but fewer than an OS-level thread because some features require the involvement of an OS kernel.

A stackless thread, in contrast, achieves the best fork-join performance among three thread types. Figure 1.1 presents the fork-join overheads of three thread types on Intel Knights Landing 7210[3]. Notably, OS-level threads incur more than 600 times larger fork-join overheads compared with stackless threads. ULTs are not as slow as OS-level threads, although they have 2.2x larger overheads than stackless threads because of the cost associated with user-level context switching and stack manipulations.

Such an opposite trade-off of performance and functionalities leads the community to settle on a black-and-white perspective. Nowadays, major multithreading systems avoid using ULTs because of its limited threading features than those of OS-level threads or its higher threading cost than that of stackless threads. For example, thread implementations natively supported by major compilers such as C++ standard threads [104] and OpenMP multithreading extensions [102, 126, 132] map their "threads" to OS-level threads for their rich threading capabilities. On the other hand, numerous lightweight multithreading frameworks including Intel Thread Building Blocks (Intel TBB) [143] and OpenMP tasks [102, 126, 132] adopt a lightweight stackless thread for performance. This prevailing standpoint of the runtime developers often imposes excessive threading overheads when underlying thread implementations are OS-level threads, or unnecessarily limits the threading functionalities when runtimes

[3]We used a microbenchmark shown in Figure 3.9 while setting n and N to 0 and 256. In Section 3.7 we explain the detailed experimental environment.

employ stackless threads, making the high-performance multithreading unnecessarily challenging from the viewpoint of performance or functionalities.

This thesis argues that a ULT is placed at the best position in this trade-off relationship and can be a solid replacement of an OS-level thread and a stackless thread as a means of high-performance multithreading. This work makes the following statements:

1. A ULT can be as efficient as a stackless thread. With proper designs and implementations, the threading overhead of ULTs can be as low as that of stackless threads without losing critical threading capabilities (Chapter 3).

2. A ULT can be a substitute for an OS-level thread as a thread abstraction in parallel programming languages. A ULT can be used for a thread abstraction in one of the most popular multithreading programming models, OpenMP [129]. Such a replacement significantly reduces threading overheads thanks to lightweight threading operations (Chapter 4).

We first show a highly optimized user-level threading library, Argobots [1], and describe numerous thread implementations in Argobots that have different trade-offs between performance and functionalities, proving that some user-level threading techniques have low threading overheads close to that of a stackless thread [2]. Next, we present our highly optimized OpenMP implementation over Argobots, BOLT [3]. BOLT shows that replacing OS-level threads with ULTs in an OpenMP threading layer can significantly improve the performance of fine-grained parallel programs without breaking the OpenMP specification. These artifacts demonstrate that a ULT is a scalable thread implementation with rich threading capabilities and suitable for multithreading in the era of multicore and many-core processors.

1.1 Contributions

The primary contributions of this dissertation are twofold. First, a highly optimized user-level threading library with numerous lightweight user-level threading methods, some of which are as fast as a stackless thread. Second, design and implementation of a practical and scalable ULT-based OpenMP library. We develop both libraries with in-depth performance analysis and confirm the performance improvement by evaluating them with several applications. Specifically, our detailed contributions are as follows.

Lightweight User-Level Threading Library

1. We first show design and implementation of a highly optimized user-level threading library, Argobots [1]. Argobots exposes a low-level interface to control scheduling and synchronization

methods so that application and runtime developers can keenly optimize programs running on top of Argobots.

2. We analyze the difference of performance and functionalities between two thread types in Argobots: a ULT and a stackless thread. Our highly optimized implementations of a ULT and a stackless thread reveal that a deviation [151, 152] is the fundamental cause of context switching and thus the associated fork-join cost when implementing ULTs.

3. We explore the full spectrum of threading techniques with an in-depth analysis at the instruction and cache levels. We characterize performance vs. functionalities trade-offs of these threading techniques regarding the probability of deviation while covering all feasible methods for building a generic threading library, including a few methods missing from the past literature.

4. Our detailed analysis demonstrates that some of our threading techniques have as low fork-join overheads as that of stackless threads. This result poses a question to the prevailing use of stackless threads for performance at the cost of several threading features that are useful to write parallel programs with ease.

5. We provide highly optimized implementations of all the methods within the same threading library for a fair comparison of all the threading techniques. Our implementation covers major hardware architectures in the high-performance computing community—Intel Skylake, Intel Knights Landing, ARM 64, and IBM POWER8 processors—and highlights the importance of lightweight user-level threading techniques for architectures that employ less powerful cores and have larger thread contexts.

6. We discuss Argobots from aspects of software maturity and composability. Thanks to its customizability, Argobots can be easily adopted by other runtime systems and libraries such as OpenMP [129] (i.e., BOLT) and MPI [121]. Argobots is used in several research and production runtimes, which enables these software components to interoperate via the Argobots interface.

ULT-based OpenMP Library

1. We present that a ULT is suitable for thread implementation of multithreading programming models that traditionally use OS-level threads. Specifically, we target one of the most popular multithreading programming models in high-performance computing, OpenMP [130]. We describe the detailed design of a ULT-based OpenMP library, called BOLT, which integrates ULTs into LLVM OpenMP [132] without violating the OpenMP specification.

2. We showcase that naive use of lightweight ULTs by replacing call sites of OS-level threads in a runtime system originally designed for OS-level threads cannot fully exploit lightweight ULTs. In particular, our performance analysis finds several performance barriers of an OpenMP parallel region, which is the most popular parallelization method in OpenMP.

3. We focus on nested OpenMP parallel regions in LLVM OpenMP and devise several solutions to address the scalability issue: 1) several optimizations to remove bottlenecks in LLVM OpenMP (e.g., thread resource management and thread creation algorithms) which are negligible in the original LLVM OpenMP implementation relying on heavyweight OS-level threads, 2) an algorithm to implement OpenMP's thread-to-CPU binding interface tailored specifically to ULT-based runtimes, and 3) a novel thread coordination algorithm that transparently achieves high performance for both flat and nested parallelism, which is required by an OpenMP multithreading model since OpenMP is used for both coarse- and fine-grained parallelism.

4. Our evaluation with several microbenchmarks demonstrates that BOLT significantly outperforms existing OpenMP runtimes when parallel regions are nested without hurting performance under flat parallelism.

5. We assess the practicality of BOLT for mainstream use. Thanks to LLVM OpenMP, BOLT works with existing OpenMP-parallelized applications and libraries compiled with GCC, LLVM, and Intel OpenMP compilers without recompilation, which significantly improves practicality. We also discuss limitations in using ULTs for OpenMP threads that stem from the fact that 1) some well-known computational libraries assume an OS-level thread as an OpenMP thread and 2) the specification has been designed primarily for OS-level threads-based OpenMP runtimes.

Evaluation with Applications

1. We evaluate all the user-level threading methods in Argobots with highly optimized FMM implementation and distributed graph analytics code. We argue that ULTs in applications that require low fork-join overheads incur fewer deviations. Our evaluation indicates that, in such applications, user-level threading techniques that defer context management until a deviation happens show the best performance vs. functionality trade-off.

2. We evaluate BOLT with OpenMP-parallelized N-body and quantum chemistry code which have parallel regions unintentionally nested in multiple OpenMP-parallelized software layers. Our experiments show that our ULT-based lightweight OpenMP runtime, BOLT, can exploit such nested parallelism efficiently compared with existing OS-level thread-based and ULT-based OpenMP runtimes.

3. Our experiments evaluate the effect of choosing optimal user-level threading methods in BOLT. Our evaluation with KMeans code shows that adopting the optimal ULT type in the underlying Argobots layer enhances the performance of fine-grained multitasking of OpenMP when the deviation probability is low.

1.2 Outline

The organization of this thesis is as follows. In Chapter 2 we first describe the background of three types of thread implementation and then explain OpenMP as one of the most widely used multithreading programming models. Chapter 3 discusses the design and implementation of Argobots and explores the various lightweight user-level threading techniques implemented in Argobots with in-depth analysis from the viewpoints of performance and functionalities of threads. Chapter 4 presents the design and implementation of BOLT, an OpenMP library over Argobots. In Chapter 5 we evaluate the performance of Argobots and BOLT with several applications. Chapter 6 discusses the practicality of our proposed frameworks. Chapter 7 covers the related work and Chapter 8 gives conclusions of this thesis.

2 Background

This chapter overviews the background knowledge about lightweight threading frameworks. The first section describes three major thread implementations–OS-level threads, stackless threads, and ULTs–and looks into the difference of threading features and associated overheads between these three techniques. The next section explains one of the most widely used multithreading programming models, OpenMP [130], and its representative implementations.

2.1 Thread Implementations

Modern compute resources expose several levels of parallelism: for example, parallelism in a single instruction (e.g., SIMD instructions), parallelism across multiple instructions (e.g., out-of-order execution and pipelining), core-level parallelism (e.g., multicore execution), and node-level parallelism (e.g., distributed computing). Multithreading is referred to as a means to exploit core-level parallelism by running *threads* in parallel on multiple cores. In comparison to multiprocessing (e.g., MPI [121], which runs *processes* that have isolated virtual memory spaces[1] in parallel), most *thread* implementations by default share the memory space among threads.[2] In multithreading, therefore, all data are visible to every thread without explicit memory read and write operations although correct multithreaded execution often requires explicit memory synchronizations such as a memory barrier and atomic operations. In this work, we define a thread as a schedulable work unit that executes a sequence of work in a shared memory environment. We note that we use the term *thread* instead of *task*, which is also a widely used term to refer to such a parallel work unit; literature on a *parallel programming model* tends to refer to such a lightweight parallel unit as a *task*, while their *underlying implementations* are often discussed as *threads*. This paper focuses on implementations and thus uses *threads*.

Since multithreading is a dominant form to exploit core-level parallelism, there exist several thread implementations that have different characteristics in terms of performance and capabilities. In this section, we overview three representative implementations of threads and discuss their performance and functionalities.

[1]Some process implementations share memory space [89].

[2]Some thread implementations do not transparently share memory and require explicit operations [14].

2.1.1 OS-Level Threads

OS-level threads are thread implementations that are provided by and integrated into OS. The most notable implementation is POSIX threads (Pthreads), which implements threads specified in the POSIX standard [95]. Although the POSIX specification defines only an interface and functionalities of threads, threads in most Pthreads packages are mapped to kernel threads managed by OS. Solaris threads [154] and Windows threads are other examples of OS-level threads. In the following, we assume Pthreads as an OS-level thread.

OS-level threads provide the widest range of functionalities. In addition to the most basic fork and join operations (`pthreads_create()` and `pthreads_join()`), most OS-level thread implementations support several synchronization objects such as a barrier (`pthreads_barrier_t`), a spinlock (`pthreads_-spinlock_t`), a mutex (`pthreads_mutex_t`), and a condition variable (`pthreads_cond_t`). A yield operation (`pthreads_yield()`) and an intermediate termination operation (`pthreads_exit()`) provide scheduling flexibility. Furthermore, OS-level threads support thread-local storage (TLS, which is supported by `pthread_specific_t`), thread-core binding (`cpu_set_t`), and per-thread signal masks (`signal_set()`). One of the unique functionalities of OS-level threads is non-cooperative preemption, which interrupts the currently running thread by a kernel timer, saves its execution context, and schedules other threads for fair scheduling. Because an OS-level thread is integrated into a kernel and thus considered as a first-level thread implementation, major compilers assume OS-level threads as "threads" by default. These compilers provide an efficient implementation of TLS accesses, which can be optimized by compilers. For example, most C++ compilers support TLS for OS-level threads by default as `std::thread` introduced in C++11 [104].

OS-level threads, however, suffer from high threading overheads incurred by kernel involvement. Because schedulers of OS-level threads are integrated into OS, any scheduling operation requires heavyweight system calls, which contributes to the enormous overheads of OS-level threads. For example, on an Intel Skylake machine, system calls to change the CPU affinity or update the register pointing to TLS cost more than a few hundred cycles. Another heavy operation associated with scheduling is context switching, which exchanges a stack and saves and restores register states. Context switching requires stack changes and pollutes caches. These overheads inflate the fork-join cost of OS-level threads, inhibiting scalability under fine-grained multithreading. Such a heavyweight but fully capable thread implementation is sufficient for coarse-grained multithreading, whereas more lightweight threads are indispensable to fine-grained multithreading.

2.1.2 Stackless Threads

A stackless thread is regarded as an implementation of a lightweight thread that is suitable for fine-grained parallelism. A stackless thread achieves least fork-join overheads by eliminating extra resource management and associated cost that is necessary for threading capabilities other than fork and join operations, both of which are considered to be the minimum requirements for threads. As the essence of threads is a schedulable function that can be detached from the current execution context and later invoked, understanding a mechanism of function calls is important to know fundamental overheads of fork and join operations. We first briefly overview how a function is called.

```
1   void callee(void *arg) {
2       // prologue.
3       [grow stack];
4       [save callee-saved registers];
5       // function body.
6       ...
7       // epilogue.
8       [restore callee-saved registers];
9       [restore a stack pointer];
10      [pop the parent instruction address];
11      // return to a caller.
12      [jump to the original instruction address];
13  }
14  void caller(...) {
15      ...
16      [save caller-saved registers];
17      [set an argument in a register/stack];
18      [push the current instruction address];
19      [set a stack pointer];
20      jump callee;
21      [restore caller-saved registers];
22      ...
23  }
```

Figure 2.1: Flow of a function call. In reality, compiler optimizations might reorder and eliminate some of these instructions.

A function call can be executed only in sequential order, but it is the simplest way to perform a unit of work. Figure 2.1 roughly presents an assembly-level flow of a function call. In the following discussion, we use a `void(*)(void*)` function, whereby arguments can be packed and passed via a `void` pointer and return values can be stored in one of the arguments. Values stored in caller-saved registers are not kept after a function call, so a caller function needs to saves these values in callee-saved registers or stack if necessary (line 16). The first argument is assigned to a predetermined place, usually a specific register. After saving an instruction address and updating a stack register to point to the top of the stack, the control jumps to a target function. The callee grows a stack by updating a stack pointer. The callee is responsible for keeping values in callee-saved registers before and after the function. In this case, a callee saves and restores callee-saved registers that are modified in the function body at lines 4 and 8. After running its body, it loads a parent stack pointer and returns to a callee using the parent instruction address. The caller restores the evicted values originally in the caller-saved registers as needed at line 21 and resumes.

A standard function call is lightweight, but its execution order is predetermined because the call site is embedded in a program. Threads need a mechanism to delay and synchronize execution. In addition to overheads of function call, minimal additional operations to implement threads are twofold: (1) a thread descriptor that stores completion status, a function pointer, and its argument and (2) a scheduling mechanism that keeps thread descriptors and runs a ready ULT, both of which are fundamental for detaching and deferring the execution of the function. The simplest and most lightweight threading method satisfies only these requirements. This technique, however, abandons all threading features that require context switching. Since this thread does not have its own stack, we call it a stackless thread. We first present the implementation of a stackless thread and describe why it does not allow context switching during execution.

```
struct thd_desc_t {
    void (*f)(void *arg);
    void *arg;
    int state;
    ...
};
void schedule_stackless(thd_desc_t *thd) {
    thd->state = STARTED;
    thd->f(thd->arg);
    thd->state = COMPLETED;
}
```

Figure 2.2: Pseudocode of invocation of a stackless thread.

We present an algorithm to invoke a stackless thread in Figure 2.2. A stackless thread requires only a thread descriptor that stores thread information and scheduling mechanism that is necessary to choose and invoke a stackless thread. On thread creation, a thread descriptor that holds a function pointer and its argument is allocated. A stackless thread can be invoked by simply *calling* it on top of the stack of the current thread. Compared with an immediate function call natively supported by programming languages, a stackless thread incurs overheads of thread descriptor management and scheduling mechanism, both of which are indispensable costs to detach the execution. A stackless thread has the least fork-join overheads and thus is used in several implementations such as Filaments [116] and Intel TBB [143]. A stackless thread, however, lacks threading capabilities that require context switching because an invoked child function is called on top of a parent invoking thread and thus we cannot resume the parent thread until the child thread is completed. Consider a yield operation that returns control from a currently running stackless thread to an invoker. To restore the context of the invoking thread, values of hardware registers (including a stack pointer and an instruction address) must be reinstated. Nevertheless, the stackless thread saves none of them explicitly on invocation; thus, a threading library cannot retrieve these values although they are possibly stored somewhere in the call stack of the stackless thread as instructed by a compiler. Even if registers could be restored, because both the child and the parent threads share the same stack region, any stack growth such as a function call and an allocation of new variables in a stack would overwrite the call stack of the child stackless thread. This parent-child welding deprives a stackless thread of threading features that require an independent invoker's context. This limitation is the largest drawback of a stackless thread.

2.1.3 User-Level Threads (ULTs)

A stackless thread lacks the context-switching capability because it bonds contexts of a parent thread and a child thread together. If their contexts are maintained independently, however, an invoked thread can return to schedulers at any point. A user-level thread (ULT) creates and maintains a thread context in order to support various threading capabilities that allow efficient scheduling. However, a ULT suffers from context management overheads. In implementing ULTs, lightweight context switching plays an important role to address the issue of context welding between an invoker and an invoked thread. Hence, we first explain a lightweight context-switching implementation entirely performed in the user space, which is often referred to as user-level context switch. Our implementation of user-level context switch follows that of Boost C++ Libraries [35]; similar codes are found in major threading packages, for example, in Qthreads [173], Nanos++ [120], Converse [108], and MassiveThreads [124] as well as Argobots [1].

Figure 2.3 presents the pseudocode of user-level context switch. This implementation represents a context as a single pointer to the call stack (`ctx_t*` in the figure) since all the other data are saved at the top of the stack. Since a caller of `switch_ctx()` is responsible for saving and restoring *caller-saved* registers before and after calling `switch_ctx()`, `switch_ctx()` itself needs to manage only *callee-saved* registers (lines 3 and 9). We note that threading libraries must save and restore all callee-saved registers specified by application binary interfaces (ABIs) because, without a special compiler help, libraries are unable to obtain information about which callee-saved registers are read after calling `switch_ctx()`. This routine first saves all the callee-saved registers including an instruction address on top of the stack (lines 3 and 4) and stores the current stack pointer in `self_ctx` (line 5). Then, `switch_ctx()` updates the stack

```
1   void switch_ctx(ctx_t **self_ctx, ctx_t *target_ctx) {
2       // save the current context.
3       [push callee-saved registers];
4       [push the parent instruction address];
5       *self_ctx = stack_pointer;
6       // restore the target context.
7       stack_pointer = target_ctx;
8       [pop the target instruction address to regA]; // regA is caller-saved.
9       [pop callee-saved registers];
10      // jump to the target context.
11      jump *regA;
12  }
```

Figure 2.3: Pseudo assembly code of user-level context switch.

```
1   void start_ctx(ctx_t **self_ctx, void *stack, void (*f)(void *), void *arg) {
2       // save the current context.
3       [push callee-saved registers];
4       [push the parent instruction address];
5       *self_ctx = stack_pointer;
6       stack_pointer = stack; // start f on top of stack.
7       // call the target function.
8       f(arg);
9   }
10  void end_ctx(ctx_t *target_ctx) {
11      // restore the target context.
12      stack_pointer = target_ctx;
13      [pop the target instruction address to regA]; // regA is caller-saved.
14      [pop callee-saved registers];
15      // jump to the target context.
16      jump *regA;
17  }
```

Figure 2.4: Pseudo assembly code to start and finish thread contexts.

pointer to the stack address pointed to by `target_ctx` (line 7) and restores the instruction address and the callee-saved register values from the stack of the target in reverse order (lines 8 and 9). The target, which is suspended in `switch_ctx()`, is resumed by jumping to the target instruction address (line 11). We note that all of these operations are executed in the user space.

If a scheduler's context has been saved properly, `switch_ctx()` enables a ULT to save its context and resume a scheduler whenever it needs to return to a scheduler. This method, however, is inappropriate for initiating a ULT because `switch_ctx()` takes `target_ctx` that must have been already initialized. This routine always saves the context of the current ULT, but this action is unnecessary when a ULT finishes because that ULT will never be resumed again. To efficiently handle these cases, we split the functionality of `switch_ctx()` and create two methods, `start_ctx()` and `end_ctx()`, to start and finish contexts, respectively. Figure 2.4 shows the pseudocodes of these functions. Their implementations come from the first and the latter parts of `switch_ctx()`. `start_ctx()` saves the context of the current thread (lines 3–5) but freshly executes a function `f()` on top of `stack` (lines 6 and 8), while `end_ctx()` restores and resumes the target context (lines 12–14, 16) without saving the current context.

ULTs with the three context-switching functions described above address the parent-child bonding issue and enable unconstrained scheduling among threads. We explain an invocation algorithm of ULTs with `switch_ctx()`, `start_ctx()`, and `end_ctx()`. We note that the following description assumes a typical parent-first ULT (which we call **Full** in Chapter 3) as the representative ULT implementation.

```
1   struct thd_desc_t {
2       void (*f)(void *arg);
3       void *arg;
4       int state;
5       void *stack;
6       ctx_t *ctx; // context of this thread.
7       ...
8   };
9   ctx_t *g_caller_ctx; // a global variable
10  void schedule_ult(thd_desc_t *thd) {
11      if (thd->state != STARTED) {
12          start_ctx(&g_caller_ctx, thd->stack, ult_wrapper, thd);
13      } else {
14          switch_ctx(&g_caller_ctx, thd->ctx);
15      }
16  }
17  void ult_wrapper(thd_desc_t *thd) {
18      thd->state = STARTED;
19      thd->f(thd->arg)
20      thd->state = COMPLETED;
21      end_ctx(g_caller_ctx);
22  }
```

Figure 2.5: Pseudocode of invocation of a ULT (**Full** in Chapter 3).

Figure 2.5 shows the pseudocode of a function that invokes a ULT. A given ULT is started by `start_ctx()` (line 12) if `thd` has not been executed previously; otherwise, it resumes `thd` by `switch_ctx()` (line 14) since its context has already been initialized. A user-given thread function is called in a wrapper function `ult_wrapper()` (line 17) so that `end_ctx()` is executed on completion (line 21) because a ULT invoked by `start_ctx()` cannot return to the parent thread just by a standard `return` procedure. Since both `start_ctx()` and `switch_ctx()` save the caller's context in `g_caller_ctx`, the caller can be resumed by `switch_ctx()` or `end_ctx()` at any time.

Independent management of contexts of parent and child threads allows yielding, intermediate termination, and efficient synchronization as explained in Section 3.2. Because of rich capabilities, several threading libraries adopt ULTs; for example, Qthreads [173], X10 [50], Task Parallel Library [114], Nanos [120], Converse [108], and MassiveThreads [124] are well-known user-level threading libraries, while some programming languages such as Python [165] and Go [145] natively support ULTs.

However, ULTs have larger fork-join overheads because of `start_ctx()` and `switch_ctx()` are heavier than fork and join operations of a stackless thread. The stack change increases memory footprint and pollutes caches, which lowers performance. Chapter 3 will discuss a high-performance user-level threading library, called Argobots, and explore the implementations of lightweight ULTs that have as low overheads as that of a stackless thread but are implemented without losing functionalities of ULTs.

Although ULTs have rich capabilities and low overheads, most applications do not directly use ULTs; they rather use ULTs as an implementation of an abstracted parallel work unit. Numerous high-level parallel runtimes use ULTs for this purpose. Representative runtimes that use ULTs include Charm++ [107], Chapel [47], XcalableMP [113], Legion [28], ParalleX [75], PaRSEC [36], Realm [13], Kokkos [43], RAJA [90], and Adaptive MPI [91]. To deeply study a use case of ULTs in a real high-level parallel runtime system, we focus on OpenMP [130], the most popular multithreading programming languages in the high-performance computing field [29]. In the next section, we discuss existing work that tackled fine-grained parallelism in OpenMP.

2.2 OpenMP for Multithreading

OpenMP [130] is a widely used multithreading application programming interface (API) that supports shared memory parallel programming in C, C++, and Fortran. The OpenMP community continues to extend its standard to cover several types of parallelism. The latest OpenMP specification (OpenMP 5.0 [130]) includes SIMD-level parallelism [110], offloading for accelerators such as GPU [30] and FPGA [149], and multitasking that supports deep recursion [22], fine-grained loops [159], and dependency-based synchronization [78]. Nonetheless, multithreading has been initially supported since OpenMP 1.0 [131] and is widely used to exploit the computing power of multi- and manycore CPUs. Because of simplicity, expressiveness, and multi-platform support, countless applications and libraries use OpenMP for multithreading. For example, the survey of the US Exascale Computing Project reported that OpenMP is the most popular programming model for multithreading [29]. Major C, C++, and Fortran compilers including the GNU Compiler Collection (gcc, g++, and gfortran) [77], Intel C/C++ Compilers (ICC) [98], Intel Fortran Compiler (ifort) [55], Clang [54], flang [133], and F18 [67] support OpenMP by default.

```
void foo() {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        compute(data[i]);
    }
}
```

Figure 2.6: Loop parallelization by the OpenMP's `parallel for` construct.

2.2.1 Programming Model of OpenMP

We briefly describe the basic programming model of OpenMP from the viewpoint of multithreading. Figure 2.6 shows how a loop can be parallelized with OpenMP. In OpenMP, parallelism is expressed by directives (in C and C++) or special comments (in Fortran). In the case of C and C++, `#pragma omp` is an OpenMP directive and the following `parallel for` directs an OpenMP compiler to parallelize the loop next to this directive. When the control encounters a statement annotated with a `parallel` construct, the statement is regarded as an OpenMP parallel region and an OpenMP *team* is created for the parallel region. The team consists of one or more OpenMP *threads*,[3] where the OpenMP thread that reaches and creates a parallel region, is called a master thread. Loop iterations annotated by the OpenMP's `for` keyword are distributed to OpenMP threads of the associated team and executed in parallel. There is an implicit barrier after a parallel region, so the master thread exits the parallel region after completing all the loop iterations. We note that, since it is multithreading, all data except loop induction variables (i.e., `i` in Figure 2.6) and local variables in the loop iteration are implicitly shared among OpenMP threads.

As shown in the example, despite the simple syntax, `parallel for` is a powerful multithreading construct that can easily exploit loop parallelism, which is the most common parallel form in applications. Knowledgeable users can use advanced OpenMP features to precisely control thread-level parallelism by controlling a thread count (`num_threads` clause), data sharing policies (several data-sharing attribute clauses such as `private` and `firstprivate`), loop distribution policies (`schedule` clause), granularity (`chunk` clause), how many loops in nested parallel loops should be collapsed (`collapse` clause), affinity (`proc_bind` clause), and reduction operations (`reduction` clause). OpenMP also provides various synchronization mechanisms including atomic operations (`atomic` construct), critical sections (`master` and `single` constructs), a barrier (`barrier` construct), and locks (`omp_set_lock()` and `omp_unset_lock()` functions). OpenMP maintains both simplicity and high expressiveness and successfully meets demands for both basic and advanced parallelization.

The latest OpenMP is more powerful since OpenMP extended its specification to support other types

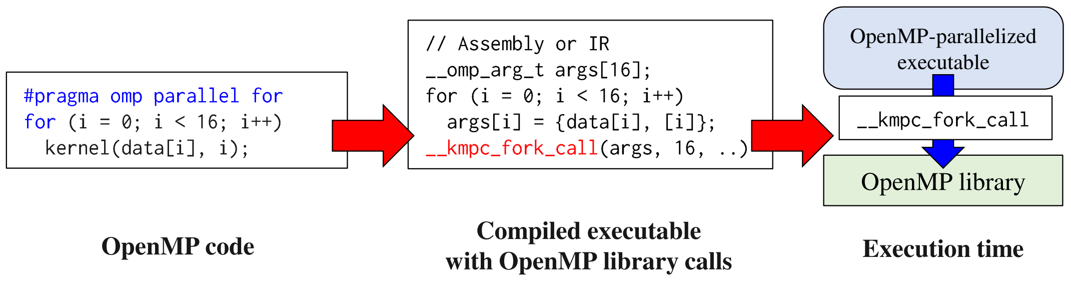
[3]we note that this "thread" is a term in OpenMP. In this dissertation, we use an OpenMP thread to refer to it if the term "thread" is confusing with ULTs or any thread implementation.

of parallelism; the latest OpenMP 5.0 specification can direct compilers and runtimes to exploit SIMD-level parallelism and data-level parallelism. In addition, offloading features for accelerators are actively discussed and developed, which significantly expands the applicable range of OpenMP. As a result, numerous applications, computational libraries, and runtime systems have been successfully parallelized with OpenMP.

The multithreading constructs including `parallel` and `parallel for` are the most basic and most widely used parallelization methods in OpenMP since multithreading has been and remains the center of OpenMP and thus is well maintained and highly optimized on most platforms. Since the number of cores in a processor is growing, fine-grained parallelism for strong scaling becomes increasingly important. Moreover, multiple software layers that are independently parallelized by OpenMP often introduce nested parallel regions, creating further fine-grained threads. Therefore, highly optimized multithreading implementation in OpenMP has drawn attention.

OpenMP itself is a specification and does not indicate any specific implementation. To understand the performance issue of multithreading programs in OpenMP, the following section describes implementations of multithreading mechanisms that are found in the major research and production OpenMP systems.

2.2.2 Implementations of OpenMP



```
#pragma omp parallel for
for (i = 0; i < 16; i++)
  kernel(data[i], i);
```

```
// Assembly or IR
__omp_arg_t args[16];
for (i = 0; i < 16; i++)
  args[i] = {data[i], [i]};
__kmpc_fork_call(args, 16, ..)
```

OpenMP-parallelized executable
__kmpc_fork_call
OpenMP library
OpenMP code
Compiled executable with OpenMP library calls
Execution time

Figure 2.7: Compilation and execution flow of OpenMP programs.

OpenMP is not an implementation but an international standard of APIs for C, C++, and Fortran. Compilation of OpenMP-parallelized codes thus requires an OpenMP compiler that accepts the OpenMP API and parallelizes programs as specified by the OpenMP standard. One possible implementation is developing an OpenMP compiler that embeds thread management mechanism with direct thread primitives (e.g., the Pthreads API) in the executable. However, since the OpenMP standard includes more and more complicated functionalities, most OpenMP compilers choose to leave resource management to OpenMP runtime libraries. The typical compilation and execution flow of OpenMP programs is illustrated in Figure 2.7. Major OpenMP compilers do not inline OpenMP's resource management logic in a program but simply emit call sites of OpenMP internal runtime functions that are implemented in OpenMP runtime libraries bundled with compilers. At execution time, these OpenMP libraries handle OpenMP's complicated resource management for multithreading.

```
// automatically created by a compiler.
void omp_outlined(int gtid) {
    int incr = 1;
    int chunk = 1;
    int lower, upper, stride;
    __kmpc_for_static_init(gtid, &lower, &upper, &stride, incr, chunk);
    for (int i = lower; i < upper; i += stride) {
        compute(data[i]);
    }
    __kmpc_for_static_fini(gtid);
}
void foo() {
    __kmpc_fork_call(N, omp_outlined);
}
```

Figure 2.8: Pseudo assembly code of loop parallelization by the OpenMP's `parallel for` construct presented in Figure 2.6. This pseudo code assumes an OpenMP ABI used by LLVM OpenMP [132]. We note that several internal parameters and implementation details are omitted for the sake of brevity.

For example, consider LLVM OpenMP [132], which is an OpenMP system developed in the LLVM project [111] and integrated into its C and C++ frontend compilers (Clang [54]) and its Fortran compilers (Flang [133] and F18 [67]). Figure 2.8 presents the pseudo assembly code after compiling a program illustrated in Figure 2.6. As shown in Figure 2.8, the OpenMP compiler does not inline resource management and parallelization logic in the executable (e.g., dividing the loop iterations into several chunks and assigning them to newly created OpenMP threads based on the number of the available threads in the associated OpenMP team). Instead, it calls OpenMP runtime functions starting from `__kmpc` and leaves all the complicated operations to the OpenMP runtime library. In this example, `__kmpc_fork_call()` initializes the whole parallel region and assigns several newly created OpenMP threads to the corresponding OpenMP team. Each OpenMP thread runs `omp_outlined()` automatically generated by a compiler and executes loop bodies associated with a certain range calculated by `__kmpc_for_static_init()`. Compared to inlining the OpenMP logic, relying on an external OpenMP library for runtime management increases function calling overheads. However, this separation significantly improves the modularity of the OpenMP runtime component and thus is preferred by many compiler projects. Despite different ABIs, all the major production OpenMP compilers including GNU C/C++ Compilers (GCC) [126], Intel C/C++ Compilers (ICC) [102], and Clang [132] have their own OpenMP runtime libraries.

At present, all the production OpenMP implementations we mentioned above rely on OS-level threads (e.g., Pthreads) to implement OpenMP threads; that is, `__kmpc_fork_call()` creates OS-level threads to execute `omp_outlined()`. Such an OS-level thread-based implementation has been known to suffer from large threading overheads if the granularity of OpenMP threads is small. Nested parallel regions exacerbate the situation by incurring oversubscription of OS-level threads, which significantly degrades performance. In practice, a large threading overhead compels programmers to manually control thread granularity. Such a manual optimization does not only increase programmers' burdens but also often turns out to be infeasible since optimal thread granularity depends on underlying processors, the number of available threads, and input. Several workarounds existing in the specification are rather ad-hoc because they do not reduce large overheads of OS-level threads but reduce parallelism to alleviate threading overheads, often adversely reducing the amount of parallelism and results in underutilization of cores.

Our solution is fixing the root cause of this issue by replacing the heavyweight threading layer by our lightweight ULTs explained in Chapter 3. In Chapter 4, we propose our ULT-based LLVM OpenMP runtime, called BOLT, which employs several optimizations and resource management techniques to unleash ULTs in an OpenMP runtime. Our evaluation shows that our lightweight OpenMP library can efficiently exploit fine-grained nested parallelism in OpenMP parallel codes.

3 Argobots and Lightweight User-Level Threading Techniques

This chapter describes our highly optimized user-level threading library, Argobots [1], and the work of user-level context-switching methods that have different trade-offs of performance and functionalities [2]. Argobots has been and is being developed by many collaborators and thus the article on the Argobots framework lists many authors [1]. A paper on the investigation of efficient ULT implementations was written with Abdelhalim Amer, Kenjiro Taura, and Pavan Balaji.

3.1 Introduction

Multithreading is the predominant form of parallelization to exploit modern highly parallel multicore and many-core processors. On self-bootable systems, such as traditional servers or the second generation Intel MIC accelerators, the majority of programming systems map their threading abstractions to OS-level threads (e.g., most systems target the thread specification of POSIX, which itself maps Pthreads to OS-level threads). This approach is known to be too heavyweight to exploit dynamic, irregular, and massive parallelism because of its expensive thread management costs involving OS kernel operations. Therefore, several threading libraries that adopt a stackless thread or a ULT have been proposed thanks to their ability to bypass OS and relying mostly on user-space operations. Numerous production and research parallel systems including Cilk [32, 71], Intel CilkPlus [115], major production OpenMP runtimes (as tasks) [102, 126, 132], Qthreads [173], Converse [108], Nanos++ [120], Filaments [116], and MassiveThreads [124] have adopted lightweight threads as implementations of their abstract parallel units.

To fully exploit fine-grained parallelism on modern massively parallel processors, however, a lightweight and flexible multithreading library is required. Nevertheless, the existing user-level threading libraries are either too general [124, 173] so that programmers cannot utilize domain-specific knowledge to optimize applications and runtimes, or too specific [108, 120] for specific usages so that it loses generality. Too general runtime systems often degrade performance by general but inefficient designs of thread pool operations and scheduling policies, masking all the benefits of lightweight ULTs. On the other hand, a runtime that is too specific to certain applications or domains is inapplicable to other applications and domains. Our first goal is to design a highly optimized low-level runtime system so that programmers can flexibly customize its behavior and utilize highly optimized threading operations.

Specifically, the major challenges in designing and implementing a low-level runtime system are as follows:

- Flexible resource management.
- User-definable thread schedulers.
- Low-overhead thread pools.
- Rich and efficient synchronization objects.
- Highly optimized implementation of threads.

Argobots [1] is a lightweight threading library to address these challenges. Argobots has a low-level API to optimize resource usage in Argobots, modify a scheduler's behavior, and choose an optimal thread pool implementations. Argobots supports several synchronization objects including a barrier, a mutex, a condition variable, a future, an eventual, and a readers-writer lock. Furthermore, Argobots implements numerous types of threads that have different trade-offs between performance and functionalities so that application developers can adopt the most lightweight thread type with minimum threading functionalities required for certain usage. We evaluate the overheads of various user-level threading techniques in Argobots with several microbenchmarks.

The next section discusses the design and the implementation of Argobots, a lightweight threading library that exposes low-level APIs for high customizability. Section 3.3 focuses on implementations of ULTs and looks into the details of commonly seen ULT implementations and their overheads. Section 3.4 analyzes user-level threading techniques to investigate ULT implementations that keep functionalities but are as efficient as a stackless thread. We find that threading techniques that dynamically promote threads on demand during execution show the best trade-off, which we call *dynamic promotion* techniques. Section 3.7 explains the results of microbenchmarks that measure the overheads of these threading techniques. We conclude this chapter in Section 3.9.

3.2 Design of Argobots

This section explains the basic design of our user-level threading library, Argobots. The most notable feature of Argobots is an exposition of a low-level interface for customizability. For further explanation, we first clarify the terminology used in the Argobots project and the corresponding terms we use in this thesis.

3.2.1 Argobots Terminology

Multithreading libraries have a long history of research and development. Nonetheless, there exists no consensus of terminology for multithreading and thus all systems use different terms for their components although some of them have similar to or the same as others' components. For consistency, this dissertation uses terms that are different from those in Argobots. We first clarify the terminology in Argobots and that in this work.

- **Execution Stream**: A user-level threading library requires an executor that is assigned to a core so that it can run threads on top of it. This is called an execution stream in Argobots, while some work calls it a worker, a processor, or a virtual processor. We call it a worker since we believe this term is most widely used.
- **Scheduler**: Unlike OS-level threads, threads need to be scheduled by a library, so Argobots needs to have a policy to decide when and which threads should be scheduled. This scheduling policy is often embedded into a worker, but Argobots exposes an object called a *scheduler* that explicitly manages scheduling policies of threads. This scheduler is associated with a worker, while Argobots can change the scheduling policy by replacing a scheduler. In this thesis, we also call this scheduling entity a *scheduler*.
- **Thread Pool**: To delay the execution of threads, the runtime system has to store ready threads in a certain data structure so that a scheduler can take threads from it later. Since many threading systems use a queue implementation to store threads, this data structure is often referred to as a queue, a task queue, or a thread queue. In Argobots, this data structure is customizable and thus might be implemented differently (e.g., LIFO instead of FIFO). Hence, Argobots as well as this work calls this storage a *thread pool* or simply a *pool*.
- **ULT, Thread, and Tasklet**: Most threading libraries have only one implementation of an abstracted parallel work unit. From an aspect of parallel programming model, such a work unit is called a fiber, a task, a microtask, a thread, or a user-level thread . The Argobots API, nevertheless, distinguishes two types of fine-grained parallel work units from the viewpoint of functionalities. The official Argobots terminology uses the term *ULTs* or *threads* for a work unit that has the same functionalities as a fully fledged ULT in this work, while a *tasklet* is a work unit which is run-to-completion and does not allow any threading operations that require user-level context switch. In this dissertation, ULTs have the same meaning, but we use the term *stackless threads* for tasklets to clarify that such a work unit is also classified as a thread. In this work, *threads* contain ULTs, stackless threads, and OS-level threads unlike Argobots terminology. We use a special term (e.g., **Full**) for a specific ULT implementation.

With these terms, we will explain the basic work flow of Argobots.

3.2.2 Execution Model of Argobots

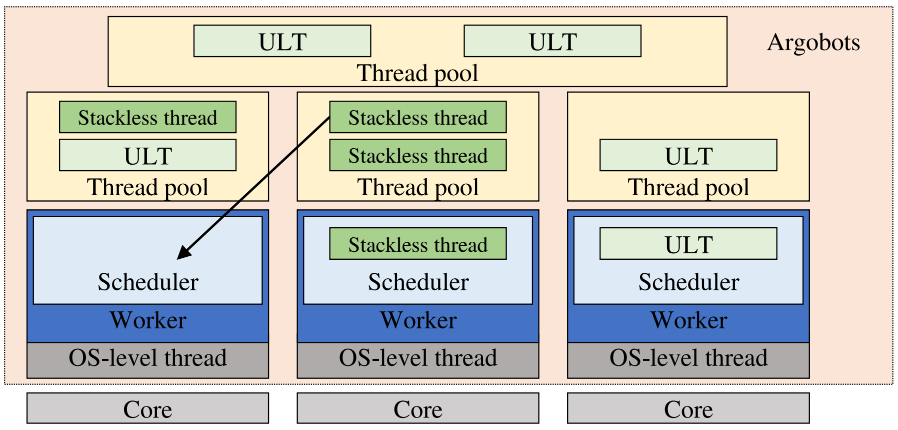


Figure 3.1: Execution model of Argobots.

Figure 3.1 illustrates an execution model of Argobots. Argobots exposes two levels of parallelism to users: workers and threads. A worker is a sequential instruction stream that can run ULTs and stackless threads. From the viewpoint of implementation, a worker is mapped to an OS-level thread, so the number of workers implies the number of OS-level threads used in Argobots. Since the number of workers is independent of how many ULTs and stackless threads an application creates, programmers can easily control the core utilization and avoid oversubscription of OS-level threads in Argobots. Each worker runs its associated scheduler that picks up a thread from one of the thread pools based on a scheduling policy and executes it. The scheduler needs to check a flag periodically so that a worker can exit from the busy scheduling loop if asked by a program (e.g., finalization of Argobots).

The basic flow of execution is as follows. The program first initializes the Argobots library by calling an initialization function and creates several workers as many as necessary, each of which runs a scheduler. The scheduler pops a stackless thread or a ULT from one of the associated thread pools based on their scheduling strategies specified by users. A stackless thread is just "called" by a scheduler and executed on top of a scheduler's stack while a ULT is invoked by a user-level context switch. If a ULT yields during execution, the control goes back to the parent scheduler and a suspended ULT is pushed back to the belonging thread pool. A finalization function joins all workers except the first worker and frees all resources associated with Argobots.

3.2.3 Towards a Lightweight and Low-Level Threading Framework

To meet a demand for a highly scalable and customizable threading library, Argobots is carefully designed to expose several components without hurting performance. First, Argobots provides an interface to efficiently use available compute resources including OS-level threads and memory. Schedulers and thread pools in Argobots can be defined by users so that programmers who have application- or domain-specific knowledge can write efficient and effective algorithms. To reduce the programmers' burden, Argobots implements various highly optimized synchronization objects. For performance, Argobots exposes several types of threads to let programmers choose the optimal one.

Flexible Resource Management

Argobots exposes an interface to control the resource usage in Argobots. Since a CPU core is one of the most important hardware resources, Argobots provides several functions to control core utilization. Specifically, Argobots supports functions to fork and join a worker so that users can dynamically change the number of workers. Users can finely and dynamically change affinity of workers (i.e., core binding) to improve locality and avoid contentions with OS-level threads belonging to other processes and libraries. Argobots also has an interface to control memory usage; stacks for ULTs are dominant memory consumption in a threading library, so Argobots has an interface to control stack sizes. In addition to global settings of stack size for schedulers and ULTs, a ULT creation function in Argobots takes a stack size argument to set a specific stack size per ULT, which allows users to finely control the memory usage and minimize the memory consumption.

User-Definable Thread Schedulers

Scheduling is known to highly impact the efficiency of multithreaded programs. Most threading libraries only provide a few predefined scheduling policies, whereas Argobots not only provides several predefined scheduling policies but also exposes a user-defined scheduling interface for advanced users who want to specialize scheduling algorithms for their applications and runtimes.

Figure 3.2 illustrates an example code that defines a user-defined scheduler in Argobots. Programmers need to provide a scheduler function (`scheduler()`) as a function pointer, which will be run by a worker. Based on their scheduling strategy, the scheduler obtains a thread from a thread pool and executes it. For example, Figure 3.2 shows a template of a user-defined scheduler that pops a ready thread from one of the thread pools associated with a scheduler and executes it. This flexible implementation can accommodate several scheduling algorithms including priority-aware scheduling [96] and locality-aware scheduling [8].

```
void scheduler() {
    const int FLAG_CHECK_INTERVAL = 1024;
    int work_count = 0;
    while (true) {
        // pop a thread from pools. pop_pool() can be any function
        // that obtains a thread from pools.
        thd_desc_t *thd = pop_pool();
        if (thd) {
            schedule_thread(thd);
        }
        // the flag is checked every FLAG_CHECK_INTERVAL iterations
        if (++work_count % FLAG_CHECK_INTERVAL == 0 && get_worker_flag() == STOP) {
            break;
        }
    }
}
```

Figure 3.2: Design of a user-defined scheduler in Argobots.

An implementation restriction of the scheduler design in Argobots is that the scheduler must periodically call an event check function (`get_worker_flag()` in Figure 3.2) to exit from the scheduler if necessary. Since a scheduler implementation often becomes a busy loop that repeats checking the availability of threads in thread pools, such a periodic flag-check mechanism is necessary to finish a scheduler and join an OS-level thread safely. We note that this restriction does not impose significant overheads since the frequency of flag checking can be easily adjusted to amortize the associated overheads.

Low-Overhead Thread Pools

The efficiency of thread pools highly affects the performance of a user-level threading library since most scheduling operations access thread pools. Argobots provides several types of pools so that the runtime can internally use optimal implementations. Specifically, Argobots distinguishes the following five types of pools regarding which workers can access a pool.

- Private (`PRIV`): a `PRIV` pool is only accessible by a single worker.
- Single-producer single-consumer (`SPSC`): an `SPSC` pool can be only pushed by a specific worker and also popped by a specific worker. Unlike a `PRIV` pool, however, a worker that pushes a thread and one that pops a thread can be different.
- Single-producer multiple-consumer (`SPMC`): an `SPMC` pool can be only pushed by a specific worker but popped by any worker.

- Multiple-producer single-consumer (`MPSC`): an `MPSC` pool can be only popped by a specific worker but be pushed by any worker.
- Multiple-producer multiple-consumer (`MPMC`): an `MPMC` pool is accessible by any worker.

An `MPMC` pool is usable in any case while the implementation is less efficient. On the other hand, a `PRIV` pool limits the applicability but has the least overheads; especially a `PRIV` pool does not require any atomic operation to ensure thread safety, so it shows the best performance. Programmers are responsible for using them correctly.

Fine-Grained Scheduling Primitives

Argobots provides several scheduling primitives to control scheduling not only from schedulers but also on top of ULTs. `yield()` is a function that suspends the current execution of a thread and returns the control to the parent scheduler. The resumed scheduler will push the suspended thread back to its belonging pool and try to schedule another ready thread. Typically, this yield functionality is useful to progress other threads while a certain ULT is waiting for completion of memory and communication operations or synchronization of other threads. Specifically, just adding `yield()` to a busy wait including a polling operation can remove waste of CPU resources in a busy loop and improve core utilization.

Another important function is `exit()`, which immediately finishes an execution of a running ULT and returns a control to a parent scheduler. Although its functionality is similar to `yield()`, in the case of `exit()`, the parent scheduler will not push the finished ULT back to a pool since the terminated ULT has never been resumed. In addition, Argobots optimizes a context-switching function for `exit()` by omitting the context saving because the context of the exiting thread will not be used after `exit()`.

More fine-grained scheduling controls are feasible via `suspend()` and `resume()`. `suspend()` suspends an execution of a current ULT and returns a control to a parent by user-level context switching. Unlike `yield()`, the suspended ULT will not be pushed back to the pool immediately after returning to a parent; the ULT will be pushed to the associated pool when another thread calls `resume()` with the suspended ULT's descriptor. Compared to `yield()` in a busy-loop, `suspend()` and `resume()` can avoid unnecessary ULT scheduling by resuming suspended ULTs and making them schedulable only when ULTs can make progress.

These scheduling primitives cannot be used on stackless threads since these operations require user-level context switching; using these scheduling functions on stackless threads will crash the program. This restriction critically limits applicable cases of stackless threads, motivating us to investigate lightweight user-level threading techniques that can perform as good as stackless threads but can perform context switching.

Rich and Efficient Synchronization Objects

Various scheduling primitives provided by Argobots are generic enough to build most scheduling operations, but correct and scalable implementations of widely used synchronization patterns that use synchronization objects such as mutex and barrier are challenging. Argobots by default provides rich and efficient synchronization objects to ease multithreaded programming. Argobots not only supports the basic synchronization objects provided by Pthreads, including mutex, barrier, and condition variable, but also some advanced ones including future, eventual, and readers-writer lock. Argobots avoids naive implementations that use `yield()` in a busy-wait loop and instead adopt implementations optimized with `suspend()` and `resume()` so that waiters are scheduled only when they can progress.

We emphasize that these operations are forbidden on stackless threads since these synchronization objects might perform user-level context switching internally. This restriction limits the use cases of a stackless thread although it has smaller fork-join overheads.

Highly Optimized Implementation of Threads

Fork-join cost of threads highly impacts the performance of fine-grained parallel programs, so Argobots highly optimizes the implementation of threads. As we have explained, Argobots exposes two types of threads. A ULT is a thread that has its execution context and thus can perform user-level context switching and return to a parent thread during its execution, while its large context-switching overheads inflate the fork-join cost. The other type of thread is a stackless thread, which has the smallest fork-join overheads but cannot perform context switching during its execution. These threads have different creation and join functions, so users can use both thread types in the same program.

As described in this section, Argobots has rich functionalities for scheduling and synchronization, high customizability, and exposes several tuning knobs. The implementation of threads, however, has not been fully investigated and thus remains suboptimal. In the next section, we take a close look at the difference between a stackless thread and a ULT to explore the performance improvement opportunity.

3.3 Performance Comparison Between Stackless Threads and ULTs

Our explanation in this chapter is based on a threading library with a simplified API sketched in Figure 3.3, which can be found in most threading libraries. Among functions listed in Figure 3.3a, fork and join are the most basic operations; a fork function (`create_thd()`) creates a ULT and a join function (`join_thd()`) waits for the completion of a given ULT and frees its resource. Here we do not impose fully strict computation [32] and allow arbitrary synchronization operations (including a barrier and a

```
thd_desc_t *create_thd(void (*f)(void *), void *arg);
void join_thd(thd_desc_t *thd);
void yield_thd(void);
```

(a) Threading operations that appear in the discussion of lightweight user-level threading techniques. A stackless thread does not support a yield operation (`yield_thd()`) because it requires a context-switching capability. Other operations such as intermediate termination and synchronization objects (e.g., barrier and mutex) are omitted since they do not appear in our example codes.

```
void comp(void *arg) { [...]; }
// A parallel version of the following loop:
// for (int i = 0; i < n; i++) comp(args[i]);
void parallel_loop(void **args, int n) {
    thd_desc_t *thds[n];
    for (int i = 0; i < n; i++) // fork ULTs.
        thds[i] = create_thd(comp, args[i]);
    for (int i = 0; i < n; i++) // join ULTs.
        join_thd(thds[i]);
}
```

(b) Example code using `create_thd()` and `join_thd()`.

Figure 3.3: Basic threading API of a user-level threading library and example code with this API. We use this API to discuss implementations of lightweight ULTs.

mutex) between threads in order to maintain flexibility and generality. A *thread pool* is a data structure to keep ready ULTs. A ready ULT is popped from a thread pool and executed by a *scheduler* that runs with its own stack on the corresponding OS-level thread (*worker*). Our following explanation assumes a work-stealing model [33] for load balancing; each worker has its own thread pool and attempts to steal a ready ULT from another worker's pool if needed (e.g., when its local pool is empty). We note that this model does not assume a specific implementation of thread pools and work-stealing algorithms. This fork-join mechanism is powerful enough to parallelize several parallel patterns including a parallel loop presented in Figure 3.3b.

Consider the simplest thread implementation that supports only `create_thd()` and `join_thd()`. The essence of fork and join operations is a schedulable function that can be detached from the current execution context and later invoked. Compared with a function call, minimal additional operations to implement such ULTs are twofold: (1) a thread descriptor that stores completion status, a function pointer, and its argument and (2) a scheduling mechanism that keeps thread descriptors and runs a ready ULT, both of which are fundamental for detaching and deferring the execution of the function. A threading method that satisfies only these requirements is a stackless thread, which is the simplest and most lightweight technique. A stackless thread, however, abandons all threading features that require a context switch; that is, once scheduled, such a ULT does not stop until completion.

We first revisit the implementation of a stackless thread in a user-level threading library and show why a stackless thread can only run to completion. We then describe the ULT implementations by showing how to overcome the limitation of stackless threads. Our description follows the scheduler implementation in Argobots, which has its own stack and runs on the corresponding worker.

3.3.1 Stackless Threads in a User-Level Threading Library

```
void scheduler() {
    [...];
    while (true) {
        thd_desc_t *thd = pop_pool();
        if (thd) {
            schedule_stackless(thd);
        }
        [flag check];
    }
}
```

Figure 3.4: Pseudocode of a stackless thread. Real schedulers might sleep when no ULTs are ready in the pool and have a branch to check a flag in order to terminate a scheduler.

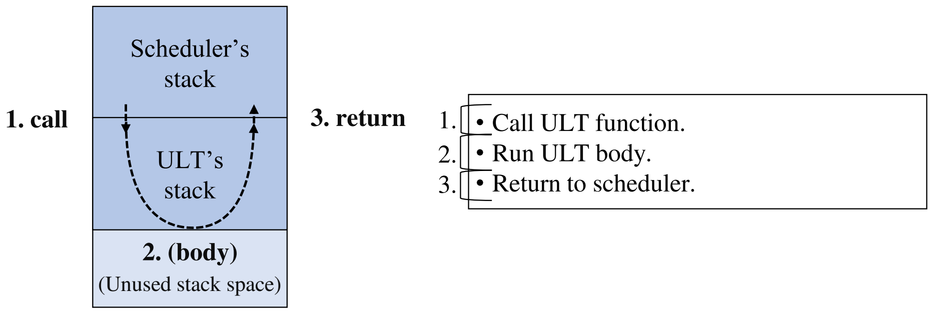


Figure 3.5: Flow of fork-join (stackless threads).

We present the pseudocode of a stackless thread in Figure 3.4 and its execution flow in Figure 3.5. In a scheduling a loop, the scheduler tries to pop a thread in one of the pools. A thread in a pool is popped by a scheduler running on a worker and simply called on top of the scheduler in `schedule_stackless()` presented in Figure 2.2. As shown in Figure 3.5, a stackless thread only imposes overheads associated with scheduling, which is necessary for a threading library.

Although a stackless thread has the least fork-join overheads, it lacks threading capabilities that require a context switch because a simple function call welds together a scheduler and an invoked thread; a scheduler that spawns a stackless thread thread cannot be resumed while the invoked thread is running. Consider a yield operation (`yield_thd()` in Figure 3.3a) that returns the control from a ULT to a scheduler. In order to restore the context of the scheduler, values of hardware registers (including a stack pointer and an instruction address) must be reinstated. Nevertheless, a stackless thread saves none of them explicitly on invocation; thus, although these values are possibly stored somewhere in the call stack of the child thread as instructed by a compiler, a threading library cannot retrieve these values. Even if registers could be restored, because the invoked stackless thread and the scheduler share the same stack region, any stack growth caused by a function call or an invocation of another stackless thread would overwrite the call stack of the previous stackless thread. This scheduler-thread welding deprives a stackless thread of threading features that require an independent invoker's context; unsupported features are not only yielding but also intermediate termination, efficient synchronization, and child-first scheduling. This limitation critically lowers the applicabilities of a stackless thread.

3.3.2 Threads with Full Threading Capabilities

Stackless threads lack the context-switching capability because it bonds contexts of a scheduler and a thread together. If their contexts are maintained independently, however, ULTs can return to schedulers at any point. A *fully fledged* threading technique creates and maintains a thread context in order to support full threading capabilities. Such a thread allows efficient scheduling, but it suffers from context management overheads. To understand the difference in performance and capabilities between these two opposite threading techniques, we first explain user-level context switch, an essential operation to implement fully fledged threads. We note that most ULT implementations do not maintain signal masks and compiler-level thread-local storage for every ULT, so they are shared among ULTs running on the same worker.

Fully fledged threads that support the full threading capabilities are implemented with the three context switch functions described above. In the following, we explain the implementation of fully fledged threads for parent- and child-first scheduling.

Fully fledged threads that support the full threading capabilities are implemented with the three context-switching functions described in Section 2.1. In reality, we can find two implementations of fully fledged threads that have been developed to support different scheduling policies; one is for *parent-first* scheduling, and the other is for *child-first* scheduling.[1] We first explain parent-first fully fledged threads and then child-first threads.

[1]Parent-first scheduling is sometimes called *help-first* scheduling while child-first is called *work-first*.

3.3.3 Parent-First Fully Fledged Threads (Full)

```
1   thread_local ctx_t *g_sched_ctx; // worker-local variable.
2   void scheduler() {
3       [...];
4       while (true) {
5           thd_desc_t *thd = pop_pool();
6           if (thd) {
7               if (thd->state != STARTED) {
8                   start_ctx(&g_sched_ctx, thd->stack, p_ult_wrapper, thd);
9               } else {
10                  switch_ctx(&g_sched_ctx, thd->ctx);
11              }
12              if (thd->state != COMPLETED) {
13                  // return thd to a pool.
14                  push_pool(thd);
15              }
16          }
17          [flag check];
18      }
19  }
20  void p_ult_wrapper(thd_desc_t *thd) {
21      thd->state = STARTED;
22      thd->f(thd->arg);
23      thd->state = COMPLETED;
24      end_ctx(g_sched_ctx);
25  }
```

Figure 3.6: Pseudocode of **Full**.

A parent-first scheduling policy is the same as the scheduling order of a stackless thread; on `create_thd()`, a parent (i.e., a caller) pushes a child thread to a thread pool and resumes the execution of the parent itself, and later a scheduler executes the child stored in the thread pool. For example, in Figure 3.3b, a parent thread that runs `parallel_loop()` first creates all child threads and pushes them to a thread pool in the loop (lines 6–7). On `join_thd()` (line 9), the parent thread checks the completion of each child thread. If the child thread is not completed (e.g., by this worker or other workers), the parent cannot make progress and thus context-switches to a scheduler and runs a ready ULT.

In this work, we refer to the implementation of a fully fledged threading technique with parent-first scheduling as **Full**. Figure 3.6 shows the pseudocode of **Full**. The scheduling mechanism is similar to Figure 3.4; the scheduler first pops a ULT (`thd`) from a pool (line 5) and starts it. A ULT is invoked by either `start_ctx()` or `switch_ctx()` as well as Figure 2.5.

Since both `start_ctx()` and `switch_ctx()` save the scheduler's context in `g_sched_ctx`, the scheduler can be resumed by `switch_ctx()` or `end_ctx()` at any time, thus allowing suspension, intermediate termination, and efficient synchronization. Child-first scheduling also requires user-level context switch, as we describe in the following section.

3.3.4 Child-First Fully Fledged Threads (C-Full)

Child-first scheduling [122] is a different scheduling policy from that of a stackless thread and **Full**; under the child-first scheduling policy, on thread creation, a parent thread yields to a child thread, and the child pushes the parent into a thread pool so that another scheduler can steal the continuation of the parent ULT. After the child completes, it preferably jumps back to the parent thread if the parent is still in the thread pool. For instance, in Figure 3.3b, a caller of `parallel_loop()` (i.e., a parent) pushes its continuation to a thread pool and executes a child thread first on `create_thd()` (line 7). Parallelization is achieved by exposing a continuation of a parent thread to other workers. If no work stealing happens, the child returns to the parent context on completion and the parent creates a next child thread in the loop (lines 6–7). Since this child-first scheduling naturally executes threads in sequential order (or depth-first order) if no work stealing happens, it is often adopted to parallelize divide-and-conquer recursive algorithms for better locality [32, 71]. Such a child-first fully fledged thread, which we call **C-Full** in this paper, can also be implemented with the context-switching functions shown in Figure 2.3 and Figure 2.4.

The pseudocode of **C-Full** is presented in Figure 3.7. **C-Full** performs a context switch in a thread creation function (`create_thd()`); after allocating and initializing a thread descriptor, a parent thread saves its context and jumps to a child thread by `start_ctx()` (line 15). The child pushes the parent to a local thread pool (line 19) before running a user-given thread function (line 21) to expose concurrency. On completion, the child thread checks the next thread in the pool, which is ideally the parent thread so that execution order is depth-first. However, the child does not always succeed in taking the parent because it might have been either stolen by another scheduler or resumed by threading operations (e.g., `yield_thd()`). If this is the case, the child thread jumps to another thread if it exists; otherwise, the child thread returns to the scheduler (`scheduler()` in Figure 3.6). We note that **C-Full** has also the full threading capabilities that are not restricted by fully strict computation [32] since all parent and child threads and schedulers maintain their contexts independently, allowing arbitrary synchronization operations (including suspension) between threads in order to maintain flexibility and generality.

3.3.5 Performance Comparison

In both the parent- and child-first cases, the management of call stacks and callee-saved registers plays a key role in supporting full threading capabilities and child-first scheduling. In real applications, however, the ULT is often executed as if it were just *called* by following a normal function-call procedure; **Full** threads can finish without any context switch during execution, and **C-Full** threads can just run to

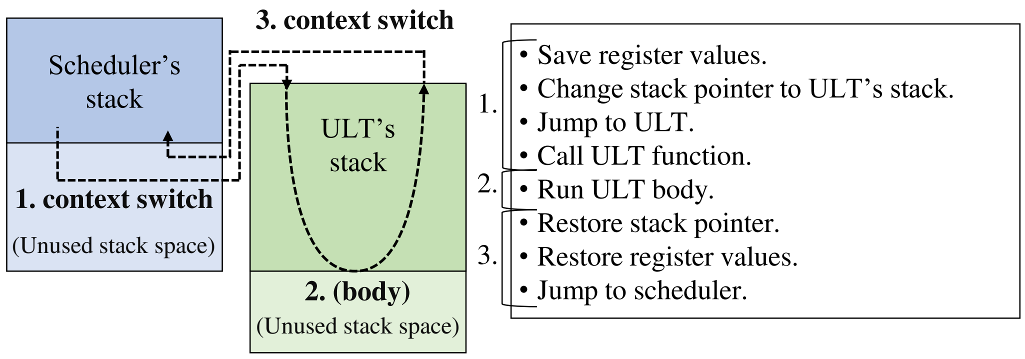
```
struct thd_desc_t {
    void (*f)(void *arg);
    void *arg;
    int state;
    void *stack;
    ctx_t *ctx; // context of this thread.
    thd_desc_t *parent;
    ...
};
thread_local thd_desc_t *g_current_thread; // worker-local variable.
thd_desc_t *create_thd(...) {
    thd_desc_t *thd = allocate_thd_desc();
    init_thd_desc(thd, ...);
    thd->parent = g_current_thread;
    start_ctx(&g_current_thread, thd->stack, c_thd_wrapper, thd);
    return thd;
}
void c_thd_wrapper(thd_desc_t *thd) {
    push_local_pool(thd->parent);
    thd->state = STARTED;
    thd->f(thd->arg);
    thd->state = FINISHED;
    // child-first scheduling expects next_ctx == thd->parent->ctx.
    ctx_t *next_ctx = pop_local_pool_or_get_sched_ctx();
    end_ctx(next_ctx);
}
```

Figure 3.7: Pseudocode of **C-Full**.

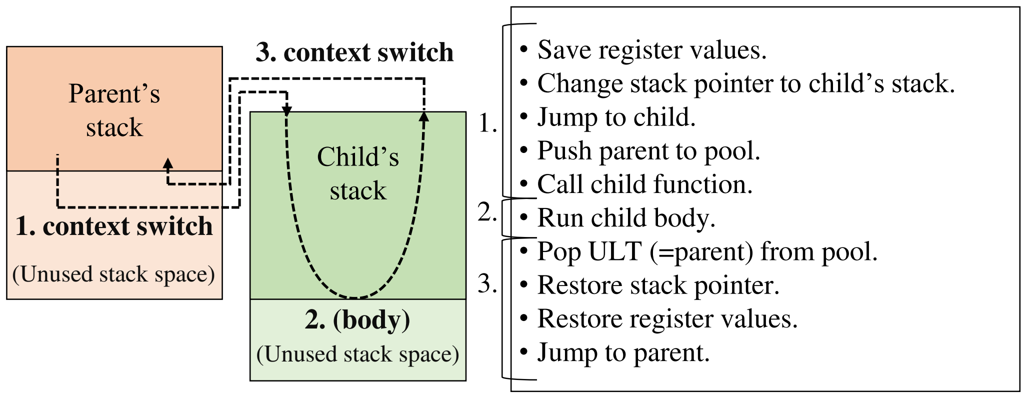
completion and return to the parent thread.

To analyze the performance difference, we use a notion of *deviation* appearing in [151, 152][2]. A deviation in the work by Spoonhower et al. [152] is defined as an event that prevents a ULT from sequential execution (i.e., execution order where all thread creations are inlined). Since sequential execution is often most efficient in terms of memory locality, the number of deviations has been used as a metric that represents how far the resulting parallel execution differs from sequential execution. This idea works well for child-first scheduling; the number of deviations can be zero if neither work stealing nor yielding happens. If we follow the original definition, however, all fork and join operations of parent-first threads incur deviations because, unlike child-first order, parent-first order is different from sequential execution order even when a parent-first ULT is executed in the run-to-completion manner. This paper, therefore, extends the idea of deviation by defining it as an event causing an execution that is different from sequential execution *except* such an event on forking and joining a parent-first thread. With this definition, the number of deviations under parent-first scheduling can be zero in a case where no context

[2]We note that this is called differently in other literature; for example, such an event is called as "drifted" in [8].



(a) Flow of **Full**.



(b) Flow of **C-Full**.

Figure 3.8: Flow of fork-join without deviation (**Full** and **C-Full**).

switch happens during the execution of thread. Deviations include any threading operations requiring a context switch during execution (e.g., yielding, intermediate termination, and synchronization) and, in a child-first case, an event where a parent thread is stolen by another scheduler. We note that no deviation is allowed with a stackless thread.

Figure 3.8 illustrates the execution paths of **Full** and **C-Full** without deviation. Comparison of Figure 3.8 with Figure 3.5 shows that both **Full** and **C-Full** incur the following additional overheads compared with a stackless thread, lowering the performance of **Full** and **C-Full** even when no deviation happens.

1. Save callee-saved registers on ULT invocation (`start_ctx()`).

2. Restore callee-saved registers on ULT completion (`end_ctx()`).

3. Manage call stacks for `thd->stack`.

To quantify the performance difference, we created a microbenchmark that controls the chances of deviation by adding a yield operation. Specifically, we ran a microbenchmark that creates and joins

```
void kernel(void *yield_flag) {
    if (yield_flag != NULL) {
        yield_thd();
    }
}
void microbenchmark(int n) {
    void *yield_flags[N];
    thd_desc_t *thds[N];
    // n yield_flags are set to non-NULL while 0 <= n <= N.
    set_yield_flags(yield_flags, n);
    // fork ULTs.
    for (i = 0; i < N; i++) {
        thds[i] = create_thd(kernel, yield_flags[i]);
    }
    // join and free ULTs.
    for (i = 0; i < N; i++) {
        join_thd(thds[i]);
    }
}
```

Figure 3.9: Microbenchmark that forks and joins N ULTs while random n out of N ULTs encounter deviation events invoked by suspension.

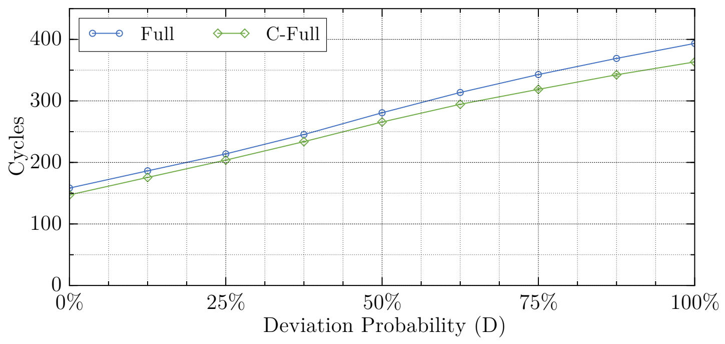


Figure 3.10: Fork-join overheads on an Intel Skylake machine using a microbenchmark presented in Figure 3.9 (N = 4,096). **Stackless** shows the performance at $D = 0\%$ because a stackless thread does not allow any deviation.

N empty ULTs as shown in Figure 3.9. In this benchmark, n randomly chosen ULTs yield once, so $n/N\%$ of ULTs encounter deviations. We define a deviation possibility D as n/N and changed D by controlling n while fixing N to 4,096. We ran this microbenchmark on a single core of an Intel Xeon Platinum 8180M processor (see Section 3.7 for details).

Figure 3.10 shows the fork-join overheads regarding the deviation probability. Our result is the arithmetic mean of all iterations. Because a stackless thread does not allow deviation, we draw a horizontal line that has the value at $D = 0\%$ (i.e., no deviation). The result shows that even when no deviation takes place, the overhead of **Full** is 1.7x higher than that of a stackless thread because of context management. We note that although their scheduling policies are different, **Full** and **C-Full** perform similarly because the expensive operations including register and stack management are common, implying that **C-Full** also suffers from unnecessary context management overheads.

Ideally, **Full** would perform as well as a stackless thread when no deviation occurs, and so would **C-Full**. However, it has been an open question whether this performance gap is inevitable when employing full threading capabilities or whether other threading techniques offer different performance and capability trade-offs. In the next section we analyze the performance discrepancy and investigate threading techniques that exist between these two opposite directions, that are more efficient than **Full** and **C-Full** when no deviation happens, and that keep full threading capabilities.

3.4 Lightweight Parent-First ULTs

In the following sections, we analyze the performance gap between fully fledged techniques (**Full** and **C-Full**) and a stackless thread method and explore intermediate threading techniques. The analysis uses the same microbenchmark presented in the preceding section (Figure 3.9) and progressively cuts down the overheads of **Full** and **C-Full** at $D = 0\%$. Each step of the analysis finds a lightweight threading technique that has a different trade-off between performance with and without deviation and programming constraints.

We note that this analysis does not attempt to revive the old *threads* vs. *event* debate [168]. Although running to completion is the mode of operation in event-driven programming models, our target is traditional threading models. That is, on encountering a blocking operation, we do not require the programmer to rip the code [11] and register event handlers as done in event-driven programming. Furthermore, the trade-off between **Full** and a stackless thread can also be encountered in the OS context, such as the distinction between work queues and tasklets in the Linux kernel [174]. Our analysis targets user-space applications and leaves the kernel space out of the scope. We do not explore a granularity control technique that serializes threads [7, 61, 106]; our approach tackles the granularity issue from a different aspect by minimizing threading overheads, which can coexist with granularity control strategies proposed in the previous studies. All the techniques described in this thesis are suitable for building generic threading libraries because they do not rely on compiler modifications, special compiler extensions, kernel modifications, or source-to-source translations.

We first look at parent-first threading techniques. Our analysis reduces the overhead of **Full** toward that of a stackless thread while keeping the capabilities of **Full**. Instruction breakdowns and performance data of all the parent-first methods are summarized in Figure 3.17 and Figure 3.18, respectively.

3.4.1 Removing Context Switch on Completion (RoC)

```
1   void start_ctx_RoC(ctx_t **self_ctx, void *stack, void (*f)(void *), void *arg) {
2       [push callee-saved registers];
3       [push an instruction address];
4       *self_ctx = stack_pointer;
5       stack_pointer = stack;
6       f(arg); // a user function is directly called.
7       return;
8   }
9   void end_ctx_to_sched() {
10      end_ctx(g_sched_ctx);
11  }
```

Figure 3.11: Pseudo assembly code of a context switch in **RoC**.

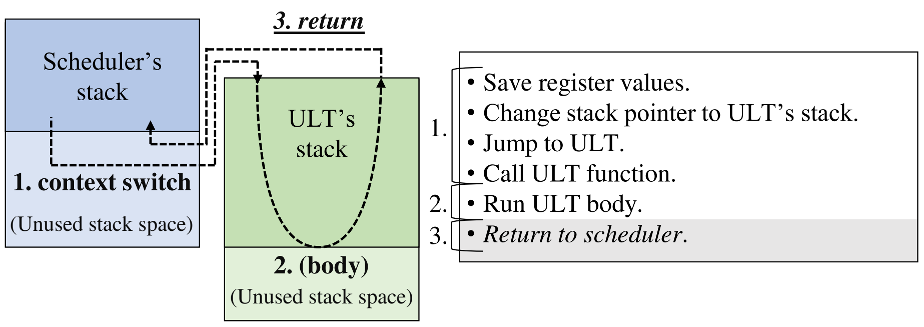


Figure 3.12: Flow of **RoC** when no deviation happens. The difference from **Full** (Figure 3.8) is written in italic.

Our instruction analysis shows a large difference in instruction counts between **Full** and a stackless thread at $D = 0\%$; even if no deviation occurs, **Full** performs context switch twice, imposing as many as 50 instructions. The first context switch from a scheduler to a ULT is necessary in order to make a scheduler resumable at any point. If no deviation occurs, however, the second manipulation of callee-saved registers is unnecessary since the register values of the scheduler are restored by a user-given thread function (`thd->f()`). A *return-on-completion* technique (**RoC**) exploits the fact that the first context switch is inevitable but the last one can be omitted if no deviation takes place during execution; **RoC** replaces the second context switch by a standard return procedure to reduce the context-switching cost on completion.

Figure 3.11 presents the pseudocode of a function that invokes **RoC**, and Figure 3.12 illustrates its execution flow without deviation. `start_ctx_RoC()` first saves callee-saved registers (lines 2 and 3), changes a stack (lines 4 and 5) as `start_ctx()` does (Figure 2.4), and directly calls a thread function `f()` (line 6). If a created thread has not encountered a deviation, the parent scheduler has never been resumed, so the **RoC** thread can simply return to a scheduler without restoring callee-saved registers saved at lines 2 and 3 because they were restored by `f()`. Thus, `start_ctx_RoC()` can return to a scheduler by a `return` instruction. We note that our explanation assumes a return mechanism similar to that of the x86/64 ABI [92]; a `return` instruction pops an instruction address from the call stack and jumps to that address. Unlike the x86/64 instruction set [97], however, several architectures including ARM [21] and POWER [138] do not have such a multifunctional return instruction. Nevertheless, their ABIs [5, 140] adopt similar calling conventions, which save an instruction address at a predefined location at a stack frame boundary. Thus we can implement the same algorithm on these architectures by combining multiple instructions as most compiler-generated codes do in a function epilogue.

However, the scheduler cannot simply be resumed by a return procedure if a deviation has happened because a deviation staled the callee-saved registers saved in `f()`. In order to address this issue without extra overheads, the return address stored in the call stack of `start_ctx_RoC()` is updated to `end_ctx_-to_sched()` when a first deviation happens; if the **RoC** thread has confronted deviation, `start_ctx_RoC()` does not directly return to the scheduler but jumps to `end_ctx_to_sched()` by `return` so that the context of the scheduler can be properly restored by `end_ctx()` (line 10). We note that only the first deviation needs to modify the return address, so succeeding deviation events do not incur any overhead.

When D is 0%, **RoC** omits one context switch per fork-join and successfully saves 24 instructions compared with **Full**, achieving 16% less overheads than does **Full**. However, **RoC** degrades performance when D is large (4% worse at $D = 100\%$) because when a deviation event happens, **RoC** performs the same number of context switches but complicates the control flow.

3.4.2 Removing Context Switch on Invocation (SS)

```
void start_ctx_SS(ctx_t **self_ctx, void *stack, void (*f)(void *), void *arg) {
    *self_ctx = stack_pointer;
    stack_pointer = stack;
    f(arg); // a user function is directly called.
    return;
}
void end_ctx_invoke_sched() {
    stack_pointer = [scheduler's stack_top];
    scheduler();
}
```

Figure 3.13: Pseudo assembly code of a context switch in **SS**.

Although **RoC** successfully skips register manipulations on completion at $D = 0\%$, saving a context on invocation makes **RoC** slower than a stackless thread. We save the scheduler's context in order to resume it later, but if the scheduler does not need to preserve its state including local variables and its

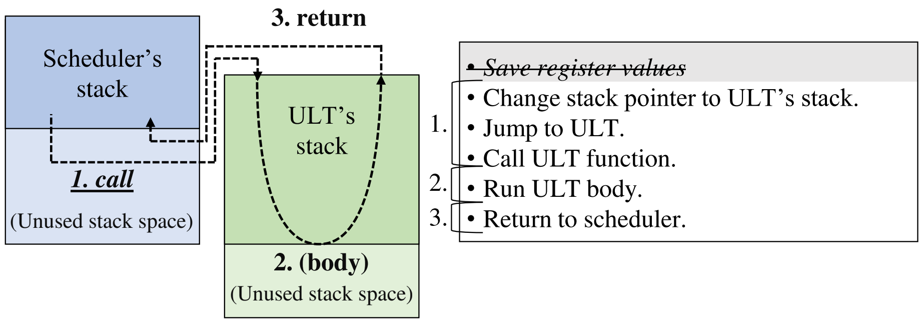


Figure 3.14: Flow of **SS** when no deviation happens. The difference from **RoC** (Figure 3.12) is written in italic.

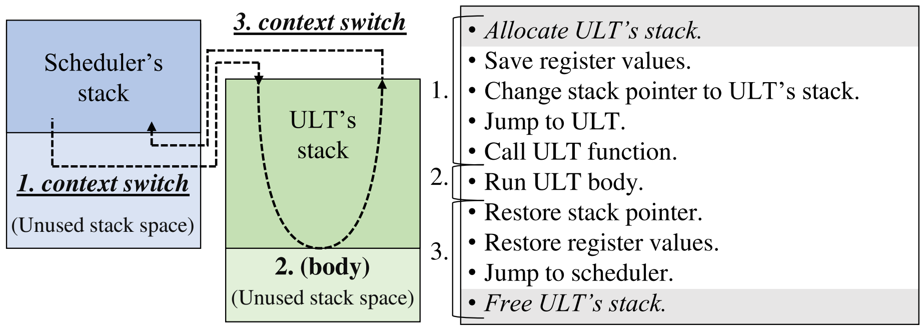
progress, we can freshly start a new one. We call this property of a scheduler *statelessness*. We propose a new threading technique *stack separation* (**SS**) that separates stacks but does not save a context of the scheduler on invocation, while this technique requires a stateless scheduler.

Figure 3.13 shows the pseudocode of **SS**. After changing a stack pointer (lines 2 and 3), **SS** directly calls a thread function (line 4). As illustrated in Figure 3.14, if no deviation happens, it returns to the scheduler with a standard `return` (line 5) as **RoC** does. If a deviation occurs, the return address in the call stack of the **SS** thread is updated so that **SS** jumps to a function without restoring the outdated scheduler's context. However, **SS** cannot resume the scheduler by `end_ctx_to_sched()` in Figure 3.11 because **SS** does not save the scheduler's callee-saved registers. Instead, **SS** calls `scheduler()` on the stack of the original scheduler, which flushes all the local variables in the call stack and the progress stored in an instruction address.

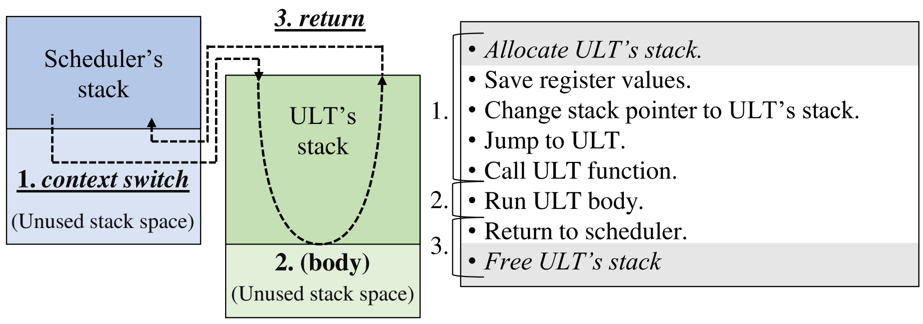
SS further reduces 14 instructions compared with **RoC** when D is 0%, achieving 14% higher performance than **RoC** does. However, **SS** lowers performance if a deviation happens (7% slower than **Full** at $D = 100\%$) because **SS** essentially needs to rerun a scheduler from the beginning of the function, which is unnecessary if the scheduler context is properly saved.

Although **SS** performs better than **Full** and **RoC** at $D = 0\%$, **SS** imposes a programming constraint that requires a stateless scheduler, narrowing the applicability of **SS**. A random work-stealing scheduler [33] can be implemented as stateless, but we note that not all schedulers are trivially stateless; for example, a scheduler is not stateless if it saves counters in local variables to select a victim of work stealing or if it sleeps when work stealing fails continuously.

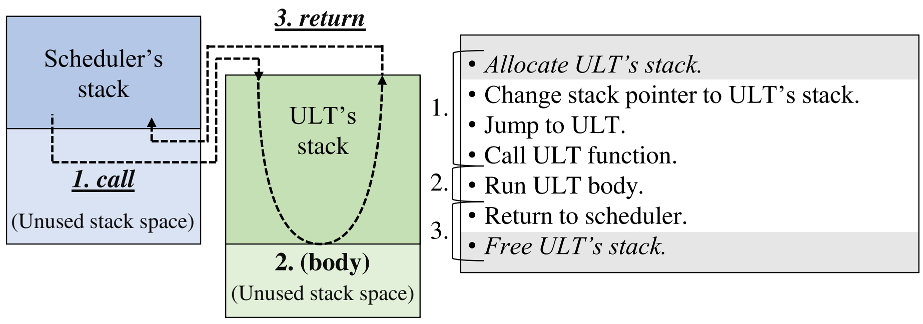
3.4.3 Lazy Stack Allocation (Full-L, RoC-L, and SS-L)



(a) Flow of **Full-L**.



(b) Flow of **RoC-L**.



(c) Flow of **SS-L**.

Figure 3.15: Flow of threading techniques with lazy stack allocation (**Full-L**, **RoC-L**, and **SS-L**) when no deviation takes place. The differences from **Full** (Figure 3.8), **RoC** (Figure 3.12), and **SS** (Figure 3.14) are highlighted by italicizing them.

SS remains slower than a stackless thread. We observe that a stackless thread incurs fewer L1 and L2 cache misses than do the other techniques at $D = 0\%$ because a stackless thread accesses only the scheduler's stack while each invocation of **Full**, **RoC**, and **SS** touches an independent call stack that is preallocated on creation. Such an eager stack allocation strategy is common in practice to facilitate management of a thread descriptor and a stack; it allows a runtime to reduce memory management operations by allocating together thread descriptors and their corresponding stacks (i.e., use part of a stack region as a descriptor). Nevertheless, this practice increases the memory accesses since each ULT invocation accesses a different stack area that is unlikely in caches. As a result, **Full**, **RoC**, and **SS** increase L1 and L2 cache misses at $D = 0\%$.

However, not all the ready ULTs need to have independent stacks; only simultaneously active ULTs require independent stacks. To reduce the memory footprint, we introduce a *lazy stack allocation* method (LSA) that decouples the management of thread descriptors and stacks and assigns a stack at invocation time. Since most ULTs are forked and joined sequentially when D is small, a call stack can be reused across thread invocation. **Full**, **RoC**, and **SS** can adopt LSA without changing their context-switching algorithms. We refer to these techniques by adding a suffix **-L** to their names. We show the flow of threading techniques with LSA in Figure 3.15. As presented in the figure, these techniques allocate stacks on creation and release them on completion, which promotes stack reuse.

We observe that **Full-L**, **RoC-L**, and **SS-L** achieve slightly higher performance than do the original techniques by successfully reducing L1 and L2 cache misses at $D = 0\%$; their numbers of L1 and L2 cache misses are almost the same as those of stackless threads. However, LSA adds 11 instructions to manage a stack and a thread descriptor independently, so LSA possibly degrades performance on different machines that have different instruction, memory, and cache costs. The results also show that the advantage of LSA becomes negligible as D gets higher since more ULTs need independent stacks; as a result, the additional allocation operations incurred by LSA lower the performance.

3.4.4 Removing Stack Change (SC)

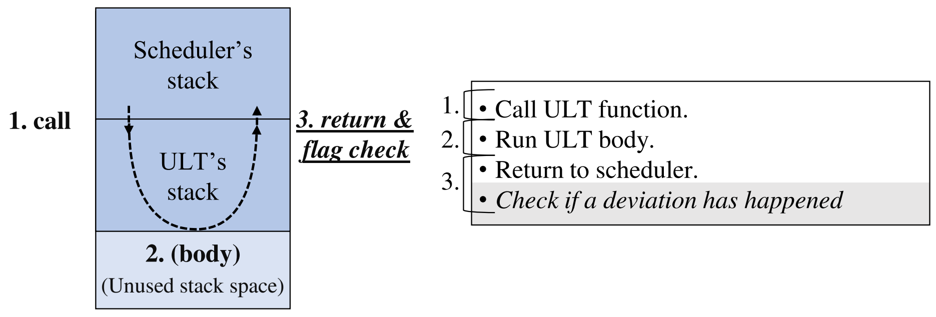
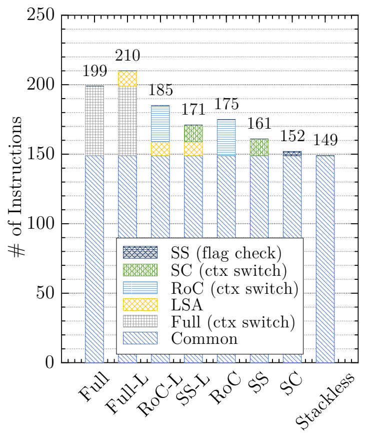
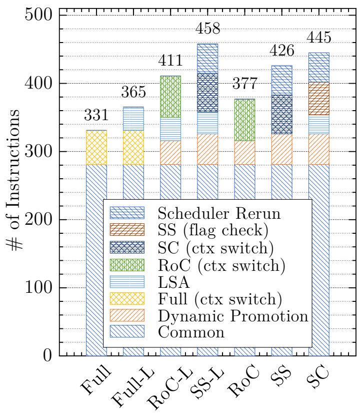


Figure 3.16: Flow of **SC** when no deviation occurs. The difference from **SS** (Figure 3.14) is written in italic.

Threading overheads still exist in the stack management, which fundamentally makes **SS-L** slower than a stackless thread. If a scheduler is stateless, however, a scheduler can be spawned on top of the newly allocated stack, which eliminates management of both callee-saved registers and stacks. The technique that newly creates a scheduler has been adopted by some runtimes [62, 68, 180]. We refer to



(a) $D = 0\%$



(b) $D = 100\%$

Figure 3.17: Instruction breakdown of fork and join operations on Skylake (parent-first methods). We used Intel SDE [103] to obtain series of instructions.

this technique as *scheduler creation* (**SC**). When an **SC** thread encounters a deviation for the first time, it spawns a ULT with a new stack and starts a scheduler on top of it. At the same time, the original scheduler currently running the **SC** thread is invalidated by updating a flag in order to keep the number of active schedulers. On completion, a scheduler checks the invalidation flag; if invalidated, it jumps to an active scheduler using g_sched_ctx.

In addition to the requirement of a stateless scheduler as **SS** and **SS-L** have, **SC** imposes a new constraint on the stack size; because stacks are shared with **SC** threads and schedulers, the stack size of all **SC** threads must be the same as that of the scheduler, forcing users to adopt the largest stack size that fits all threads in a program. This constraint is significant when one application contains multiple types of threads each of which requires a different stack size.

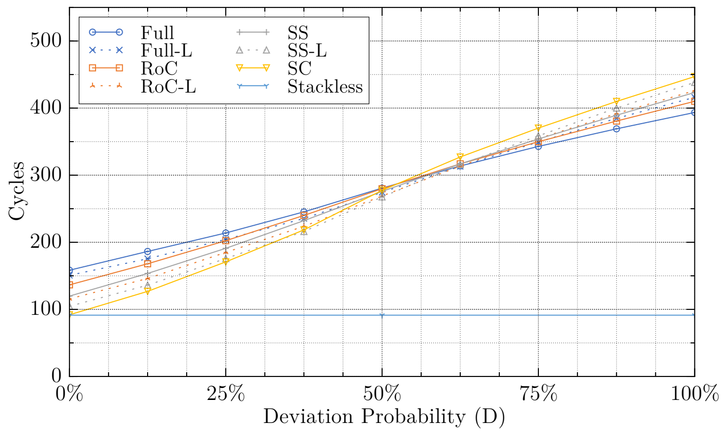
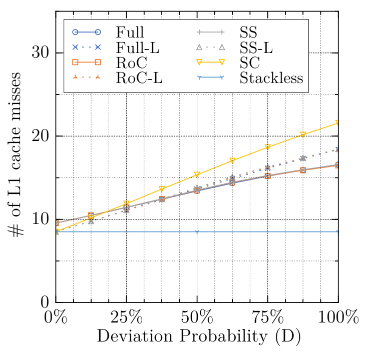
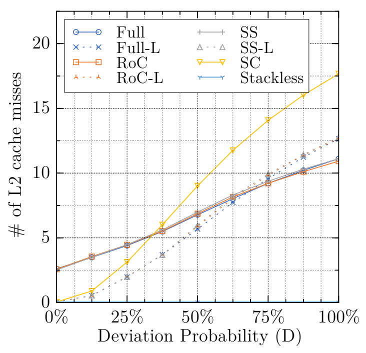


Figure 3.18: Fork-join overheads of the parent-first threading methods on Skylake.



(a) Number of L1 cache misses.



(b) Number of L2 cache misses.

Figure 3.19: Cache misses of the parent-first threading methods on Skylake. These values are obtained by PAPI [41]. Almost no L3 cache miss happens in this experiment because each ULT accesses a small portion of a call stack; we therefore omit the data.

Figure 3.17 summarizes the instruction breakdowns with and without deviation and Figure 3.18 shows the performance and cache misses of parent-first threading techniques. Figure 3.17a shows that at $D = 0\%$ **SC** adds only three instructions to check the invalidation flag. As a result, the overhead of **SC** is as small as that of stackless threads at $D = 0\%$ while **SC** supports all the threading capabilities that may cause deviations. However, restarting a scheduler on a new stack is expensive in terms of the number of instructions and memory accesses; hence, **SC** shows the worst performance among the seven methods at $D = 100\%$.

3.5 Lightweight Child-First ULTs

In this section, we apply the same analysis methodology to child-first techniques. We use the same microbenchmark to evaluate their overheads. Their instruction breakdowns and performance are summarized in Figure 3.25 and Figure 3.26.

3.5.1 Eager Stack Release (C-Full-E)

C-Full suffers from large L1 and L2 cache misses because each ULT has an independent stack. LSA, which allocates stack on invocation, seems promising to reduce cache misses incurred by stack accesses. However, LSA itself is not applicable to child-first techniques because creation and invocation are done in the same function (i.e., `create_thd()`). To promote stack reuse, we devise an *eager stack release* method (ESR) that frees stacks not when threads are joined (i.e., `join_thread()`) but on completion of ULTs. ESR allows consecutively spawned ULTs to reuse the same stack region if no deviation happens; however, ESR needs to decouple the management of thread descriptors and stacks, adding extra

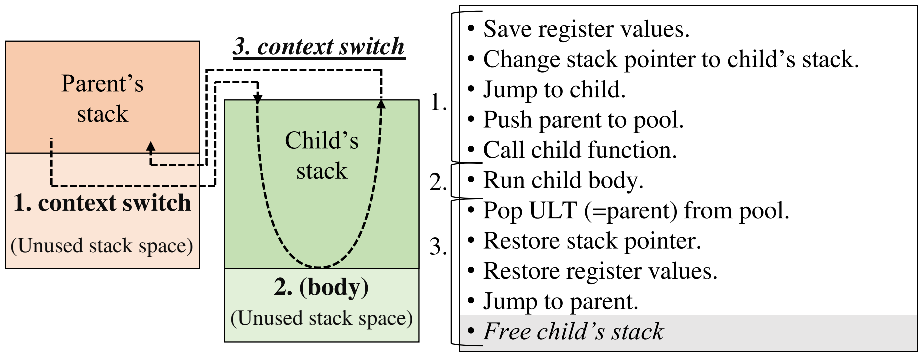


Figure 3.20: Flow of **C-Full-E** without deviation. The difference from **C-Full** (Figure 3.8) is written in italic.

overheads to handle them separately. We call this technique **C-Full-E**.

Although ESR imposes 11 instructions for independent management of thread descriptors and stacks, **C-Full-E** successfully eliminates L2 cache misses and reduces L1 cache misses, achieving an overall performance improvement of 13% when no deviation takes place. However, ESR fails to effectively reuse stacks and degrades performance by additional stack and thread descriptor management as the deviation probability increases.

3.5.2 Removing Context Switch on Completion (C-RoC and C-RoC-E)

As **Full-L** does, **C-Full-E** manipulates callee-saved registers on both invocation and completion. Unlike parent-first scheduling, child-first scheduling must preserve the context of the parent ULT since it is controlled by the users. Hence, child-first scheduling needs to maintain an independent stack and manage callee-saved registers on invocation. If no deviation happens, however, the parent thread can be resumed by a return function using a *return-on-completion* technique. This technique is applicable to both **C-Full** and **C-Full-E**; we call them **C-RoC** and **C-RoC-E**, respectively. Child-first return-on-completion techniques, however, need to deal with a deviation caused by work stealing to the parent thread, which does not happen with parent-first scheduling since a scheduler, which corresponding to a parent in child-first scheduling, is never stolen by another worker. Therefore, **C-RoC** and **C-RoC-E** need an algorithm that allows a thief to update the return address in the call stack of the child ULT.

We first look at the algorithm of **C-RoC**. The pseudocode of **C-RoC** is shown in Figure 3.21. The first context switch uses `start_ctx_RoC()` presented in Figure 3.11; if no deviation happens, the child ULT can return to the parent (line 19). As **RoC** does, the first occurrence of any threading operation that causes a deviation updates the return address stored in the stack. In the case of **C-RoC**, however, deviations can be caused by work stealing; even if no context-switching operation is performed by the child ULT, the child may not simply return to the parent ULT if the parent has been stolen. To handle this case, the thief

```
1   thd_desc_t *create_thd(...) {
2       thd_desc_t *thd = allocate_thd_desc();
3       init_thd_desc(thd, ...);
4       thd->parent = g_current_thread;
5       start_ctx_RoC(&g_current_thread, thd->stack, C_RoC_thd_wrapper, thd);
6       if ([true if stolen by another worker]) {
7           *(thd->stack + RETURN_ADDRESS_OFFSET) = end_ctx_to_sched;
8       }
9       return thd;
10  }
11  void C_RoC_thd_wrapper(thd_desc_t *thd) {
12      push_local_pool(thd->parent);
13      thd->state = STARTED;
14      thd->f(thd->arg);
15      thd->state = FINISHED;
16      // child-first scheduling expects next_ctx == thd->parent->ctx.
17      ctx_t *next_ctx = pop_local_pool_or_get_sched_ctx();
18      if (next_ctx == thd->parent->ctx) {
19          return;
20      } else {
21          end_ctx(next_ctx);
22      }
23  }
```

Figure 3.21: Pseudo assembly code of context switch in **C-RoC**.

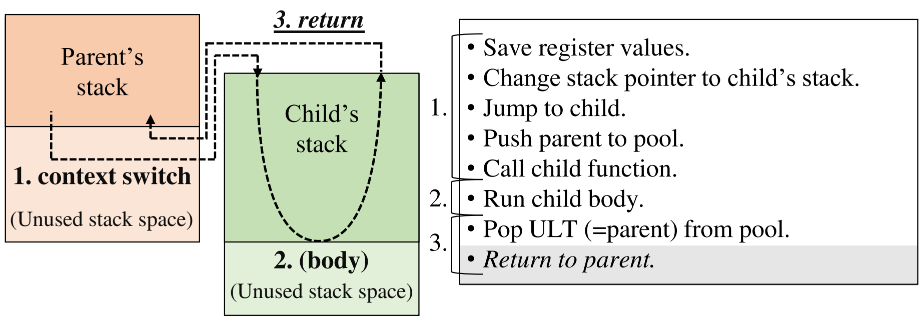


Figure 3.22: Flow of **C-RoC** when no deviation occurs. We italicize the difference from **C-Full** (Figure 3.8).

updates the return address of the child (lines 6–7), which does not exist in **RoC**. We note that there is no data race between an update by a thief (line 7) and reading a return address by a child (line 19) because the child ULT performs `return` only after taking the parent ULT (line 12). As illustrated by Figure 3.22, **C-RoC** successfully removes callee-saved register management on completion when no deviation takes place.

In addition to changing the stack management, **C-RoC-E** requires a small modification to **C-RoC** as presented in Figure 3.23 because a parent may update a child stack that has been already freed under the ESR policy (i.e., at line 7 in Figure 3.21). A thread descriptor of a child is, however, always available

```
void *C_RoC_E_thd_wrapper(thd_desc_t *thd) {
    push_local_pool(thd->parent);
    thd->state = STARTED;
    thd->f(thd->arg);
    thd->state = FINISHED;
    // child-first scheduling expects next_ctx == thd->parent->ctx.
    ctx_t *next_ctx = pop_local_pool_or_get_sched_ctx();
    if (next_ctx == thd->parent->ctx) {
        *(thd->stack + RETURN_ADDRESS_OFFSET) = [thd's return address];
        return;
    } else {
        end_ctx(next_ctx);
    }
}
```

Figure 3.23: Pseudo assembly code of context switch in **C-RoC-E**.

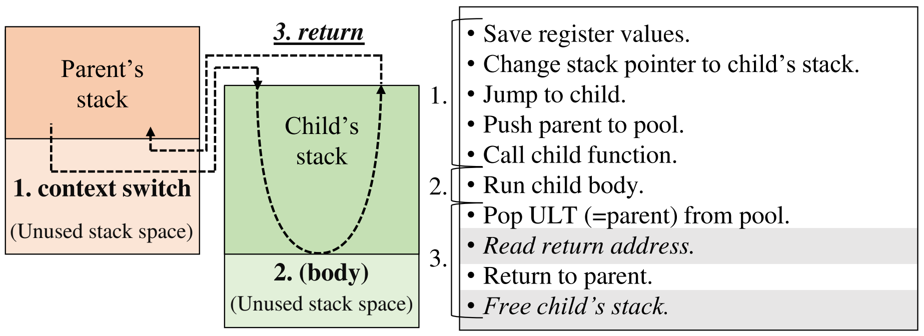


Figure 3.24: Flow of **C-RoC-E** when no deviation happens. The difference from **C-RoC** (Figure 3.22) is emphasized by italicizing it.

in `create_thd()` since the descriptor has not yet been returned to the caller of `create_thd()`. **C-RoC-E**, therefore, does not access a return address in the call stack but reads a member variable in a thread descriptor. This change makes an update by a thief safe but imposes additional overheads for reference in comparison with directly manipulating values in a call stack at $D = 0\%$.

The instruction breakdowns and performance of the four child-first threading methods are summarized in Figure 3.25 and Figure 3.26. Figure 3.25 shows that at $D = 0\%$ **C-RoC** and **C-RoC-E** in total reduce 5 and 4 instructions compared with **C-Full** and **C-Full-E**, respectively. However, **C-RoC** increases the memory footprint because the complicated operations in `thd_wrapper()` require larger stack space, which increases L1 and L2 cache misses. Overall, **C-RoC** is 10% slower than **C-Full** even if no deviation happens. **C-RoC-E** overcomes this issue of memory footprint by reusing stacks, which successfully trims down the overhead by 29% compared with **C-Full** at $D = 0\%$ while worsening performance by 7% at $D = 100\%$.

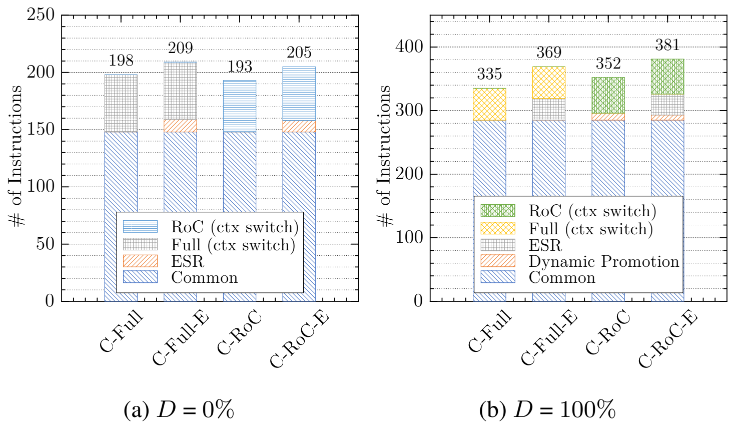


Figure 3.25: Instruction breakdown for fork and join of the child-first methods on Skylake. We used Intel SDE [103] to obtain series of instructions.

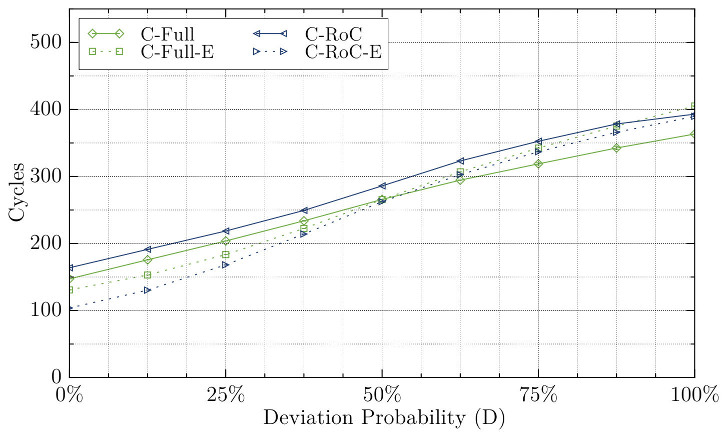


Figure 3.26: Fork-join overheads of the child-first threading methods on Skylake.

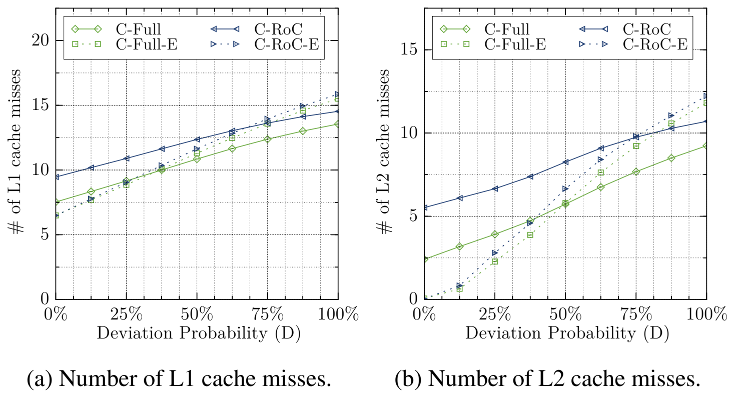


Figure 3.27: Performance of the child-first methods on Skylake. Cache misses are obtained by PAPI [41]. Higher levels of caches do not suffer from cache misses in this experiment because each ULT accesses a small portion of a call stack; we therefore omit the data.

3.6 Trade-off of Performance and Functionalities

Table 3.1: Summary of the threading techniques.

			$D = 0\%$			$D = 100\%$	
		LSA or ESR?	Change stack?	# of Register Manage-ments	Overheads	Overheads	Constraints
Parent-First	*Fully Fledged Thread* (Full)	No	Yes	2	Highest	Lowest	No
	Fully Fledged Thread (LSA) (Full-L)	Yes	Yes	2			No
	Return on Completion (RoC)	No	Yes	1			No
	Return on Completion (LSA) (RoC-L)	Yes	Yes	1			No
	Stack Separation (SS)	No	Yes	0			Scheduler must be stateless.
	Stack Separation (LSA) (SS-L)	Yes	Yes	0			Scheduler must be stateless.
	Scheduler Creation (SC)	Yes	No	0		Highest	Scheduler must be stateless. Stack size is shared.
	(Stackless Thread) (Stackless)	-	No	0	Lowest	-	No deviation is allowed.
Child-First	*Fully Fledged Thread* (C-Full)	No	Yes	2	Highest	Lowest	No
	Fully Fledged Thread (ESR) (C-Full-E)	Yes	Yes	2			No
	Return on Completion (C-RoC)	No	Yes	1			No
	Return on Completion (ESR) (C-RoC-E)	Yes	Yes	1	Lowest	Highest	No

Table 3.1 shows the trade-off regarding performance and programmability. **Full**, **Full-L**, **RoC**, **RoC-L**, and all the child-first threading techniques have no programming constraints because they save a parent context, while **SS**, **SS-L**, and **SC** require stateless schedulers. **SC** has an additional constraint on stack size, which further narrows its applicability. A stackless thread supports no threading capability

that requires a context switch such as yielding, intermediate termination, and efficient synchronization. However, highly constrained threading techniques perform better if no deviation happens; in the case of parent-first scheduling, Stackless threads and **SC** perform better than the others. We also note that in both parent- and child-first cases, threading techniques that show better performance at low D tend to perform worse at large D because if deviation happens, dynamic promotion methods that lazily manage the stack and callee-saved registers incur extra overheads than do eager methods. We note that these techniques can co-exist in a single threading library without impacting other techniques. We will discuss how to choose the best technique in Section 3.8.

3.6.1 Coverage of Our Techniques

Table 3.2: Coverage of our analysis.

Change Stack?	# of Register Managements	LSA/ESR	Parent-First	Child-First
Yes	2	No	Full	C-Full
		Yes	Full-L	C-Full-E
	1	No	RoC	C-RoC
		Yes	RoC-L	C-RoC-E
	0	No	SS	(*2)
		Yes	SS-L	
No	2	No	(*1)	
		Yes		
	1	No		
		Yes		
	0	No	(*3)	
		Yes	SC	

Table 3.2 summarizes the coverage of our analysis. An area labeled with (*) in the table denotes an absence of practical techniques. In the following, we explain reasons why these techniques are infeasible for general threading libraries.

Saving Registers (*1)

Our analysis does not include threading techniques that do not change a stack but explicitly maintain callee-saved registers on invocation. Intuitively, if a stack is shared between a parent and a scheduler, resuming a scheduler is prohibitive since a scheduler can potentially overwrite the invoked stack. On the other hand, if we totally rerun a new scheduler as **SC** does, storing callee-saved registers is pointless. In the past, however, such techniques have been proposed for child-first scheduling [79, 157, 158, 176].

Cilk 1.0-3 over Tapir/LLVM [144] is an actively developed multitasking framework that adopts this idea.

We note that these techniques are not suitable for building threading libraries because compiler modifications are required. Their approaches assume the following premises:

1. All local variables in the stack are addressed by a frame pointer instead of a stack pointer.
2. The call stack is not dynamically grown after a function prologue.
3. All threads are joined in a function that creates them (i.e., fully strict computation [32]).

Under these premises, a parent can call a child function on top of the parent stack after saving (or clobbering) callee-saved registers. The algorithm works as follows. If no work stealing happens, the child just returns to the parent. When another worker steals the parent ULT, the thief worker restores the original registers while setting a newly allocated stack to a stack pointer and resumes the parent on top of the new stack. Premise 1 guarantees that spaces for all local variables have already been allocated or reserved before the child invocation and these locations are referenced by a frame pointer. Premise 2 assures no stack growth, so a parent thread will not erode the stack used by the child thread. We note that premise 2 allows function calls because the stack address of a new function is based on a stack pointer. Premise 3, which narrows the expressiveness of parallelization, is required in order to prevent the caller of the parent ULT from overwriting the stack of the child thread prior to the completion of a child.

However, premises 1 and 2 require compiler modifications, and therefore StackThreads/MP [157], LazyThreads [79], and Cilk [71] modified a compiler. Yang and Mellor-Crummey [176] tried to avoid compiler modifications by adding a GCC compiler flag, `-fno-omit-frame-pointer`, but it does not guarantee premise 1. Unfortunately, the current popular C compilers do not provide a flag that guarantees premises 1 and 2. These techniques are not evaluated in this paper because our work targets threading techniques without compiler modifications.

Restarting Parent ULTs (*2)

With parent-first scheduling, **SS**, **SS-L**, and **SC** restart a stateless scheduler on deviation. This technique is not applicable to child-first threads, however, since parent ULTs are in most cases not stateless; rerunning a parent ULT loses not only the result computed by the parent ULT but also a child thread descriptor if the parent is in the midst of the thread creation function. This is an impractical restriction as a thread, so we do not show child-first techniques that require stateless parent ULTs.

Eager Stack Management for SC (*3)

From the viewpoint of stack management, **SC** follows the LSA policy; **SC** allocates a stack for a scheduler not on creation but on deviation. One might suggest allocating a stack and a thread descriptor together on creation for **SC**, but such an eager stack allocation strategy does not work for **SC**. **Full**, **RoC**, and **SS** always keep the same pair of a stack and a thread descriptor, while **SC** needs to decouple the management of the stack and thread descriptor since a stack required on deviation is assigned to a new scheduler, not a thread associated with a thread descriptor.

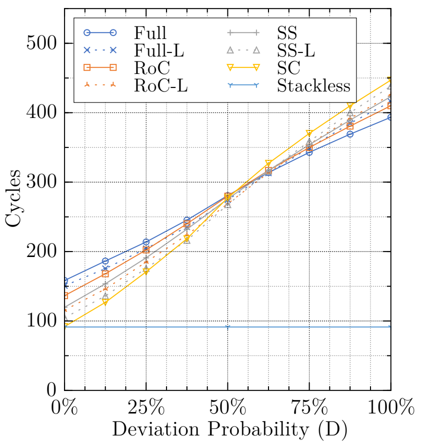
3.7 Evaluation with a Microbenchmark

Table 3.3: Experimental environments used for the evaluation in Section 3.7.

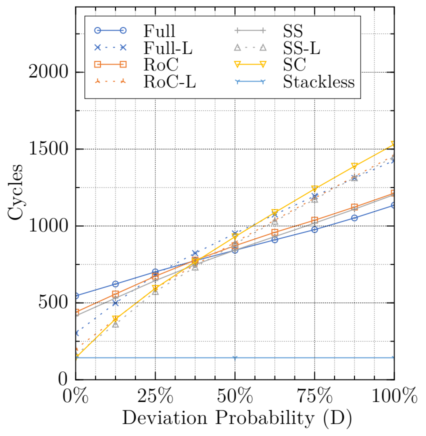
Name	Skylake	KNL	POWER8	ARM64
Processor	Intel Xeon Platinum 8180M	Intel Xeon Phi 7210	IBM S822LC (10 cores)	AMD Opteron A1120
Architecture	Skylake	Knights Landing	POWER8 LE	ARMv8-A
Frequency	2.5 GHz	1.3 GHz	2.9 GHz	1.7 GHz
# of sockets	2	1	2	1
# of cores	56	64	20	4
# of HWTs	112	256	160	4
Memory	396 GB	198 GB	130 GB	8 GB
OS	Red Hat 7.5	Red Hat 7.5	Red Hat 7.6	openSUSE 42.2
Compilers	Intel Compiler 17.2.174	Intel Compiler 17.2.174	IBM XL Compilers 16.1.1	GNU Compilers 4.8.5

In this section, we evaluate the performance of all the threading techniques presented in the preceding sections with a microbenchmark and three fine-grained parallel applications. All the parent- and child-first threading techniques were implemented in Argobots. Our experimental environments are described in Table 3.3. All the programs were compiled with `-O3`. We set the same stack size (16 KB) to both ULTs and schedulers to the advantage of **SC**. All results reported in this paper are the arithmetic mean. The error bars in the charts indicate the 95% confidence intervals.

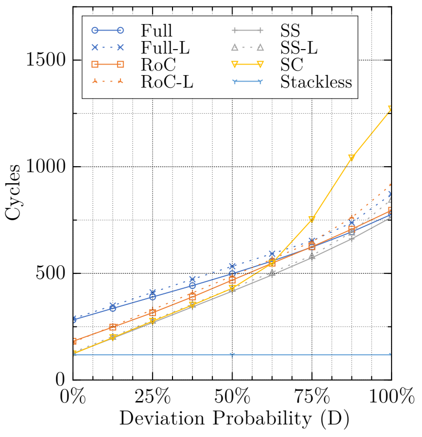
We evaluate the threading overheads with the fork-join microbenchmark used in the preceding sections; the code is presented in Figure 3.9. This microbenchmark repeats creating and joining N ULTs on a single worker. Deviations are artificially introduced by `yield_thd()`; the deviation probability D is calculated by n/N, where n ULTs uniformly selected out of N ULTs yield once. We used a lightweight private pool [1] to minimize overheads of pool operations. In the microbenchmark, the set of N fork-join operations was repeated $2^{19}/N$ times and obtained the average of the execution time. The result of stackless threads is at $n = 0$ (i.e., $D = 0\%$) because stackless threads cannot yield. We ran this microbenchmark 50 times on Skylake, KNL, POWER8, and ARM64.



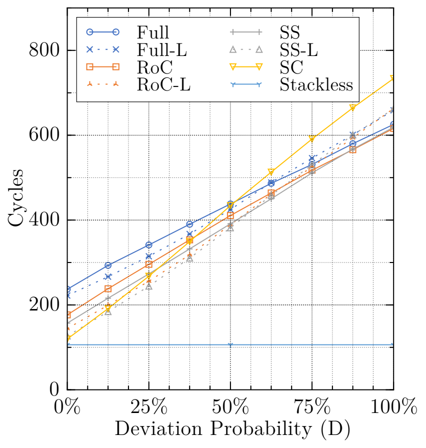
(a) Skylake



(b) KNL



(c) POWER8



(d) ARM64

Figure 3.28: Cycles per fork-join with various D values with parent-first threading techniques ($N = 4{,}096$).

3.7.1 Performance with Different Deviation Probabilities (D)

We first shows the results with various D values where N is fixed to 4,096 in order to see how the deviation probability affects the threading overheads. For better visibility, we separate results by scheduling type; Figure 3.28 shows the parent-first techniques while Figure 3.29 plots only the child-first ones. First, all the results indicate the same performance trend: **SC**, **SS**, and **RoC** outperformed **Full** at $D = 0\%$ because these dynamic promotion techniques alleviate the context management overheads when no deviation takes place. In the case of child-first scheduling, at $D = 0\%$ **C-RoC** outperformed **C-Full** on KNL, POWER8, and ARM64 by reducing context-switching overheads, while it degraded performance on Skylake because of complex control as we discussed in Section 3.5.2. LSA and ESR (**-L** and **-E**) mitigated cache misses at the cost of additional stack management overheads, with elevated performance

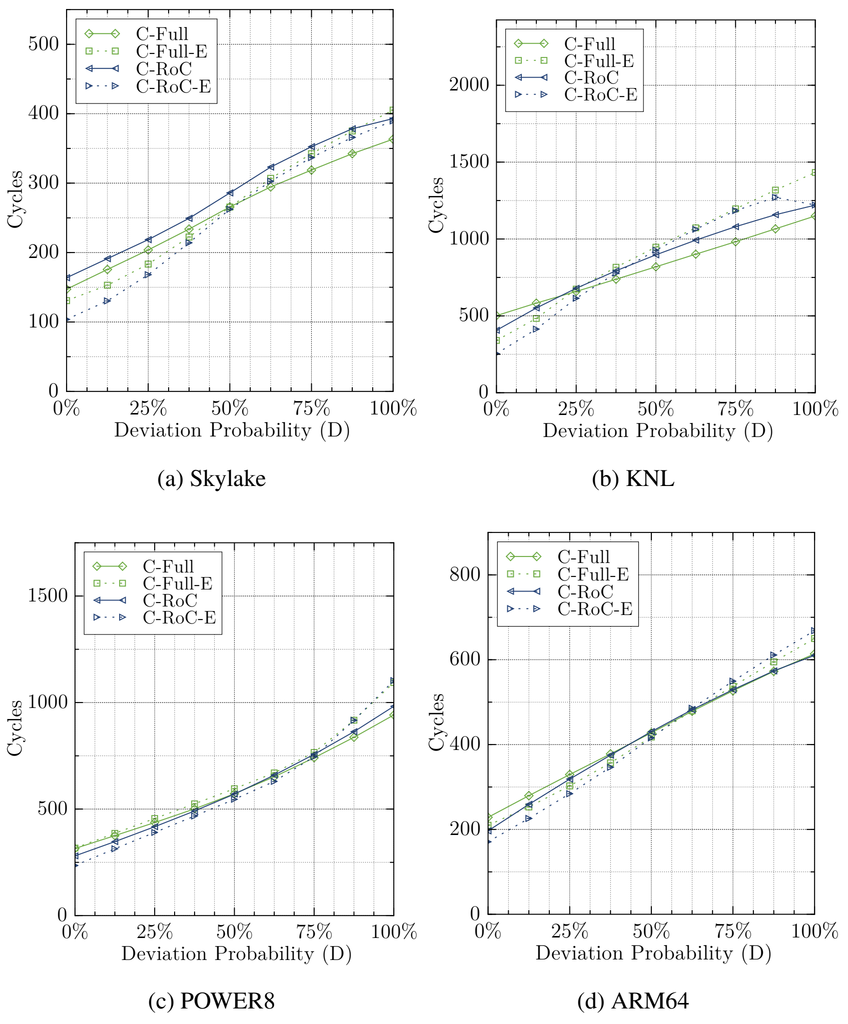


Figure 3.29: Cycles per fork-join with various D values with child-first threading techniques (N = 4,096).

overall. The performance of **SS-L** and **SC** was close to that of stackless threads but these threading techniques have programming constraints as discussed in Section 3.6.1.

On the other hand, at a larger D, **SC**, **SS**, **RoC**, and **C-RoC** were slower than the traditional fully fledged techniques (**Full** and **C-Full**) because the dynamic promotion techniques lose their advantages and become extra overheads. LSA and ESR (**-L** and **-E**) further degraded the performance since they no longer promote stack reuse and result in additional overheads to manage thread descriptors and stacks independently.

Although there is a fundamental scheduling difference between parent- and child-first techniques, the results of corresponding techniques (e.g., **RoC** and **C-RoC**) are similar because the expensive context-switching operations are common; in terms of context-switching overheads, there is no significant performance difference between parent- and child-first scheduling, while the applicability of dynamic pro-

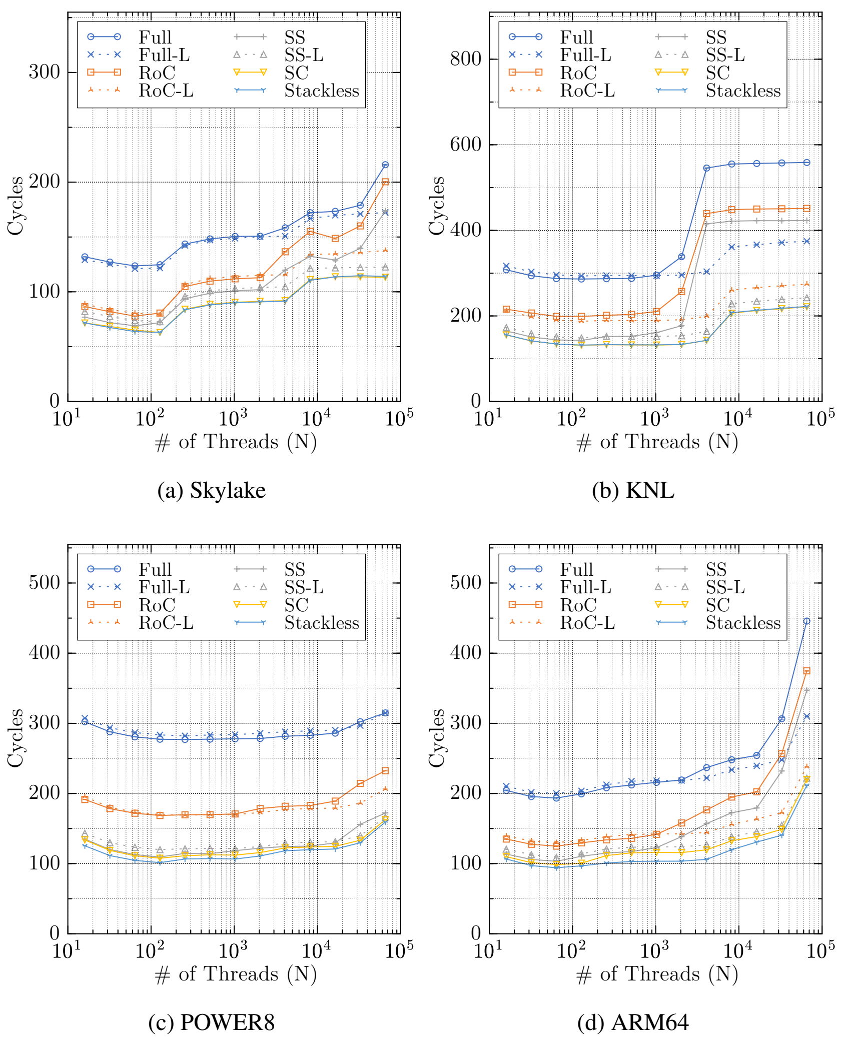
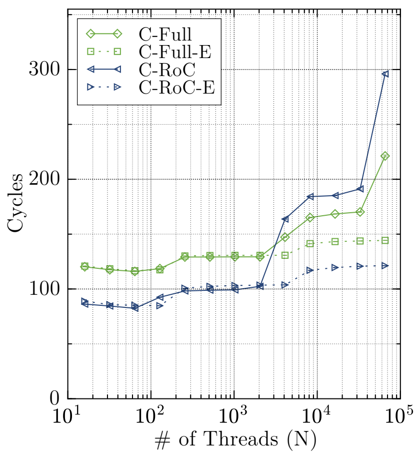


Figure 3.30: Cycles per fork-join with various numbers of threads with parent-first threading techniques (D is fixed to 0%).

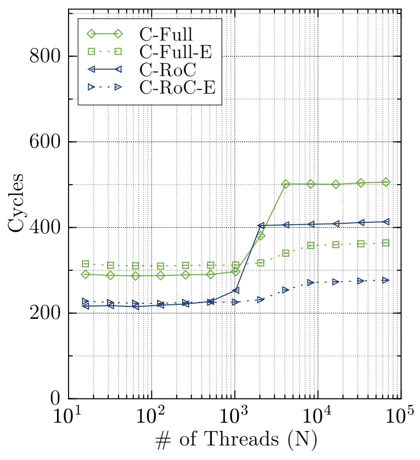
motion techniques is limited in the child-first cases. We note that, as pointed out by vast literature (e.g., [82] and [127]), the scheduling policies have been known to affect the application performance. KMeans and ExaFMM in our evaluation showcase the difference in application-level performance, while in both cases the dynamic promotion methods enhance performance by reducing threading overheads.

3.7.2 Performance with Different Numbers of Threads (N)

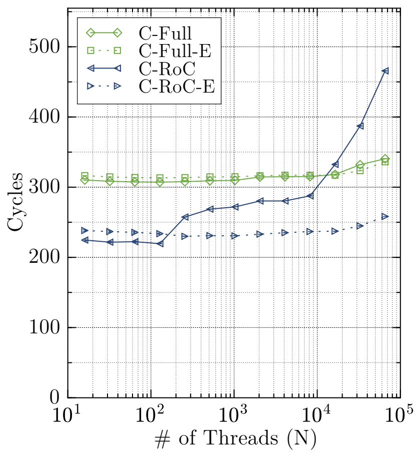
With the same microbenchmark, we examined the effect of the dynamic promotion techniques by varying ULT counts (N) while fixing D to 0% and 100%. Figure 3.30, Figure 3.31, Figure 3.32, and Figure 3.33 show fork-join overheads with different N. Overall the performance trend is the same; at



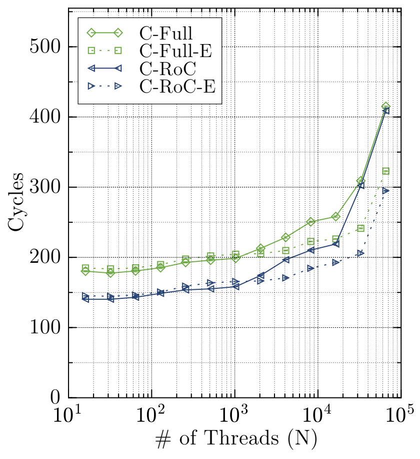
(a) Skylake



(b) KNL



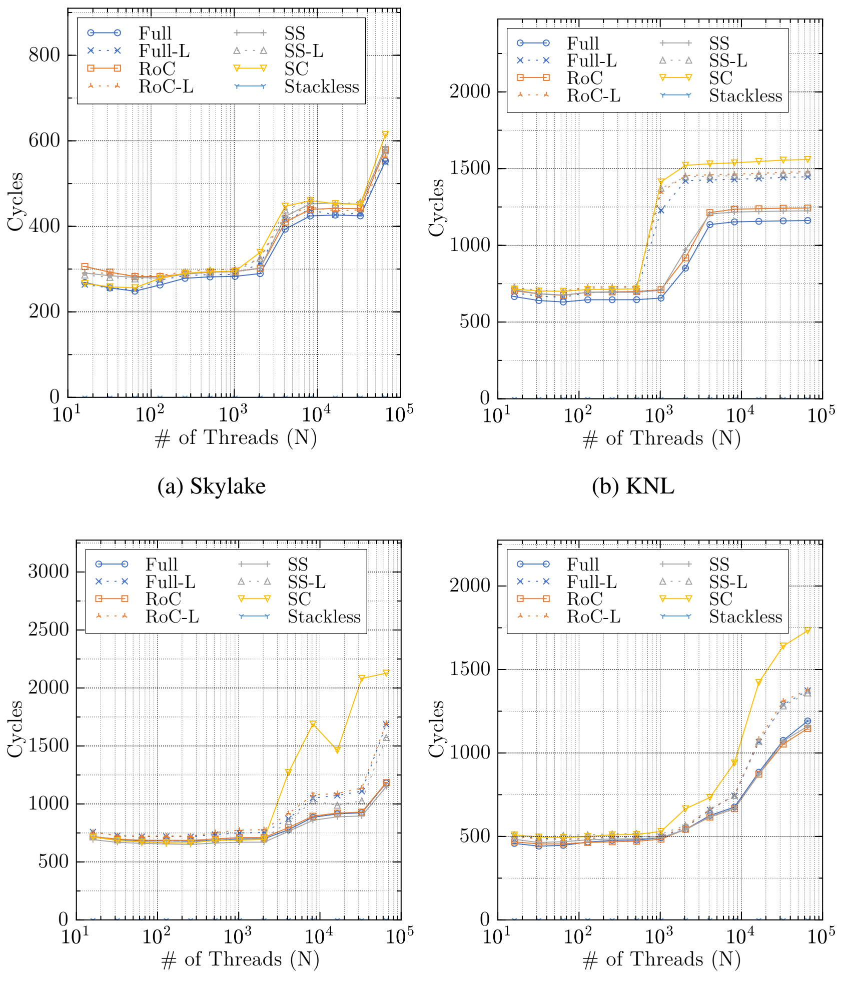
(c) POWER8



(d) ARM64

Figure 3.31: Cycles per fork-join with various numbers of threads with child-first threading techniques (D is fixed to 0%).

$D = 0\%$ the dynamic promotion techniques (**SC**, **SS**, **RoC**, and **C-RoC**) are faster than fully fledged techniques (**Full** and **C-Full**). With smaller N, however, LSA and ESR (**-L** and **-E**) are insignificant because stack accesses hit caches without LSA and ESR; on the contrary, decoupling the management of thread descriptors and stacks negatively affects the performance even at $D = 0\%$. The results indicate that if only few ULTs are used in the runtime, LSA and ESR do not contribute to performance improvement and possibly just impose additional overheads.



(a) Skylake

(b) KNL

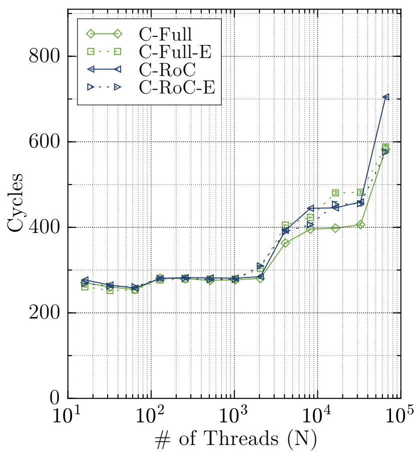
(c) POWER8

(d) ARM64

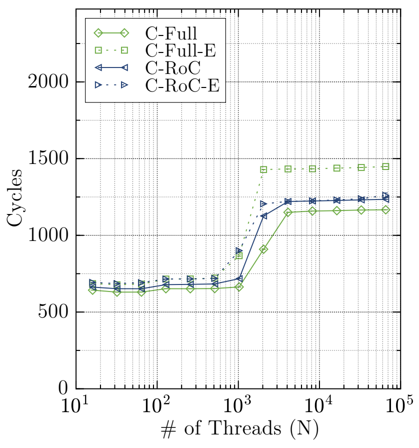
Figure 3.32: Cycles per fork-join with various numbers of threads with parent-first threading techniques (D is fixed to 100%).

Performance on Different Architectures

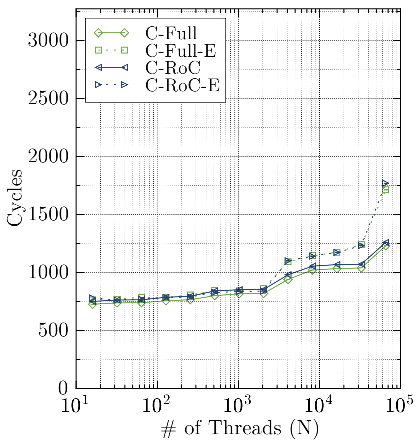
The effectiveness of the dynamic promotion techniques varies on different architectures. Figure 3.28 and Figure 3.29 indicate that KNL shows the largest performance difference at $D = 0\%$; the gaps between **Full** and stackless threads are 1.7x, 3.8x, 2.4x, and 2.2x while speedups of **C-RoC-E** over **C-Full** are 1.4x, 2.0x, 1.3x, and 1.3x on Skylake, KNL, POWER8, and ARM64, respectively. This difference comes from the design of KNL. In comparison with Skylake, which is a general-purpose Intel CPU, in spite of the identical calling convention, KNL showed a larger gap because of its throughput-oriented architecture; KNL poorly performs pointer-based operations with many branches and noncontiguous memory accesses, both of which highly impact the context-switching performance. The context-



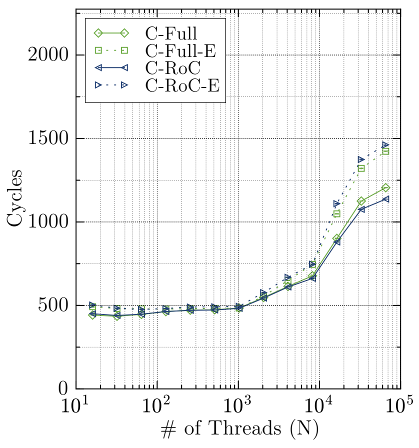
(a) Skylake



(b) KNL



(c) POWER8



(d) ARM64

Figure 3.33: Cycles per fork-join with various numbers of threads with child-first threading techniques (D is fixed to 100%).

switching overhead on POWER8 is also high. For example, at $D = 0\%$ and $N = 128$, the performance gap between **Full** and stackless threads is 2.7x while the gaps of Skylake, KNL, and ARM64 are between 1.9x and 2.2x. Context switching on POWER8 is costly because more instructions are required to save its larger context; the context size of POWER8 is as large as 528 bytes because its ABI marks more registers as callee-saved [5]. In contrast, the size of x86/64 and ARMv8-A is only 64 bytes and 176 bytes, respectively [92, 140]. We observe that the dynamic promotion techniques are more effective on architectures that are throughput oriented or have a large context.

3.8 How to Choose the Best Threading Technique

Our work investigate several user-level threading techniques that have different performance characteristics and programming constraints. As all the threading techniques can coexist in a single library, users and developers can choose the suitable techniques from Table 3.1 for their assuming workloads. Practically, the first decision should be a choice of either parent- or child-first scheduling. In general, parent-first scheduling performs better for shallow parallelism (e.g., loop parallelism in KMeans as presented in Section 5.2.1) while a child-first scheduling is preferred when the parallelism is deep and nested (e.g., divide-and-conquer parallelism in ExaFMM as shown in Section 5.1.1). Several papers investigated the performance differences between parent- and child-first scheduling policies and proposed mixed scheduling policies [51–53, 166], but this direction is out of scope of this work. The optimal technique under a specific scheduling policy should be chosen based on scenarios regarding the number of created ULTs, typical deviation probability, and required threading capabilities, all of which depend on their algorithms, machines, and inputs.

If the user has no idea about the application behavior, we recommend **RoC-L** for parent-first scheduling or **C-RoC-E** for child-first scheduling; they perform well at low deviation probability with minimum memory footprint while retaining all the threading capabilities. Nevertheless, among fully capable threads, these two do not always perform the best. When a deviation probability is high, they are slower than **Full** and **C-Full**. As we have seen in the evaluation, however, when deviations are frequent, threading overheads often become negligible because events that cause deviations (e.g., lock contention and blocking communication) hide the benefit of lightweight threads. **RoC** and **C-RoC** also outperform **RoC-L** and **C-RoC-E** when deviations rarely happen, and fewer ULTs are used because LSA and ESR are not effective when stack accesses hit caches. Although our microbenchmark uses an empty function for a thread function, real thread functions are likely to require larger function stacks for computation, rendering LSA and ESR more beneficial. We recommend **RoC-L** or **C-RoC-E** in general, but the most promising approach is the automatic selection of the best threading techniques, which is one direction of our future work.

3.9 Summary

This chapter explores user-level threading techniques that are suitable for building threading libraries from a viewpoint of threading overheads. Our in-depth instruction- and cache-level analysis of twelve methods revealed their performance characteristics and programming constraints. We found that a deviation—an event that inhibits a run-to-completion execution of thread—highly impacts fork-join overheads. We implemented all the techniques in the same runtime system and evaluated fork-join overheads on Skylake, KNL, POWER8, and ARM64 architectures. Our evaluation with a microbenchmark indicates that the dynamic promotion techniques that defer the context management show the best trade-off between fork-join overheads and programming constraints when the chances of deviation are low.

4 BOLT: OpenMP over Argobots

This chapter focuses on the OpenMP parallel programming model [130]. We explore the highly efficient OpenMP runtime library over lightweight ULTs we discussed in the previous chapter and develop a ULT-based OpenMP library, BOLT [3]. This work was conducted with Abdelhalim Amer, Kenjiro Taura, Sangmin Seo, and Pavan Balaji.

4.1 Introduction

Multithreading becomes essential to exploiting multicore processors, threading primitives provided by threading libraries are often too low-level for high-level application developers to describe parallel algorithms. Among several highly productive parallel programming models, OpenMP [130] is considered the most popular intranode parallel programming interface in the high-performance computing field [29]. OpenMP provides various features to exploit hardware parallelism methods including SIMD instructions, irregular task-level parallelism, data-flow parallelism, and utilization of accelerators, while multithreading is the most widely used form to exploit compute resource of multi- and many core CPUs. Numerous applications, computational libraries, and runtime systems have been successfully parallelized with OpenMP multithreading constructs.

Such multithreading often targets uniform and regular data-level parallelism such as a loop in which every loop iteration computes the same amount of data, so coarse-grained parallelism has been often considered to be sufficient for multithreading in OpenMP. However, as the current complex applications become composed of several OpenMP-parallelized components, fine-grained parallelism becomes commonly seen. 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 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 [126], Intel OpenMP [102], and LLVM OpenMP [132]. 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.

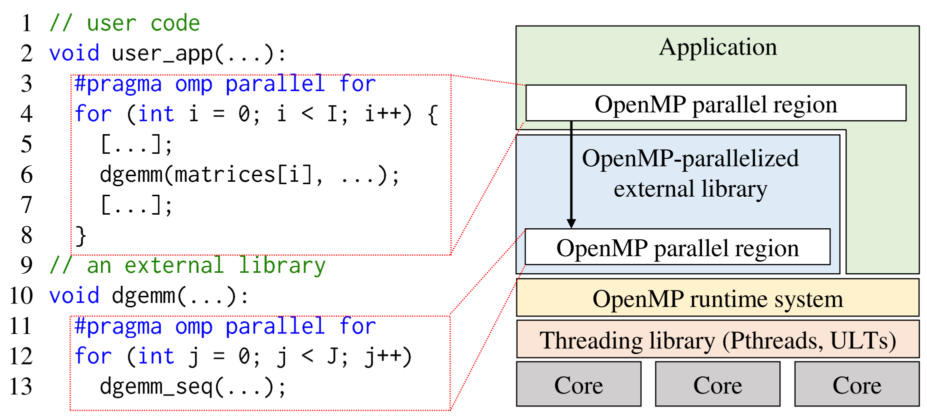
The second direction aims at reducing oversubscription overheads. Leading OpenMP runtimes accomplish this by reusing nested teams (threads and data associated with parallel regions) across parallel regions [164] and by avoiding busy waiting [175]. Reusing, and thus keeping alive a large number of OS-level 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 ULTs has also the benefit of reducing oversubscription overheads thanks to lower fork-join costs. Numerous OpenMP implementations have followed this approach [39, 40, 86, 118, 135, 155] but fall short for several reasons. First, although an OpenMP thread has its own descriptor in addition to encapsulating a native thread (be it OS- or user-level), these systems ignored other costs outside the native thread fork-join overheads. An OpenMP runtime has to manage other OpenMP-specific data and descriptors that are 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 in 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.

In this chapter, we explain BOLT [3], a practical ULT-based OpenMP runtime that attains unprecedented performance for nested parallelism while also transparently supporting the 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 our study in this chapter is different from the past literature that mentions this BOLT project [44, 45], since none of the literature presents the details of its design and performance evaluation. This work is to identify and address the issues in efficiently utilizing ULTs in OpenMP.

4.2 Fine-Grained Parallelism in Nested Parallel Regions



```
// user code
void user_app(...):
  #pragma omp parallel for
  for (int i = 0; i < I; i++) {
    [...];
    dgemm(matrices[i], ...);
    [...];
  }
// an external library
void dgemm(...):
  #pragma omp parallel for
  for (int j = 0; j < J; j++)
    dgemm_seq(...);
```

Figure 4.1: 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.

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 [118, 155]. In this work we do not advocate the use of nested parallel regions in a standalone OpenMP program (e.g., [42]); 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 our investigation 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 Figure 4.1. 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 the exponential growth of the number of threads in the system and serious performance degradation.

As shown in Figure 4.1, OpenMP runtimes rely 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. In the following, we classify the various runtimes with respect to the weight of the dependent threading library given its significant role: runtimes using (1) OS-level threads and (2) ULTs.

4.2.1 Current State in OS-Level Thread-Based Runtimes

Oversubscribing OS-level threads to hardware cores severely degrades performance 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 work, we assume OpenMP 4.5 [129].

Disabling 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`. This workaround wastes parallelism opportunities and might lead to CPU underutilization.

We note that OpenMP 5.0 [130] 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.

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 (for example, target hardware, input problem, and application characteristics) [94], 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 independent software 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 the 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 4.4.1, the number of threads is hardly a reliable metric to infer resource utilization because it ignores factors such as load imbalance and thread binding.

We note that the OpenMP specification explicitly states that the dynamic concurrency is unsafe when programs require a specific number of threads during execution (e.g., accessing data indexed by thread ID).

Use of OpenMP Tasks

OpenMP tasks are designed as lightweight parallel units of execution [22]. For example, LLVM OpenMP implements user-space context switching between tasks. Furthermore, `taskloop` has recently emerged as a task-based substitute for `parallel for` [159]. Unfortunately, because of semantic differences, OpenMP threads are not always replaceable by tasks, as explored in [12]. 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.

4.2.2 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 [39, 40, 86, 118, 135, 155]

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

4.3 BOLT: Lightweight 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 [132] 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, our 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.

4.3.1 Basic Design of BOLT

Developing a cutting-edge ULT-based OpenMP runtime from the ground up is a daunting task; it would require tremendous effort and time to bring it up to the level of leading production runtimes. Consequently, the practical direction is to reuse an existing runtime as a baseline. On the one hand, reusing one of the leading OS-thread-based runtimes benefits from its maturity but would require a nontrivial shift of its threading layer to a more lightweight ULT one. On the other hand, reusing one of the infrastructures of an existing ULT-based runtime is attractive but requires a significant upgrade to modern OpenMP and carries the risk of considerable effort to bring its performance and robustness to the level of leading OS-thread-based runtimes. Fortunately, recent years have seen the emergence and convergence of highly performing open-source implementations of OpenMP. In particular, the maturity, in terms of performance and robustness, of the recently released open-source Intel OpenMP runtime, which is now part of the LLVM effort, is the product of more than a decade of performance engineering. We seized this opportunity by taking the former route; we developed BOLT based on the LLVM OpenMP runtime by shifting its threading layer.

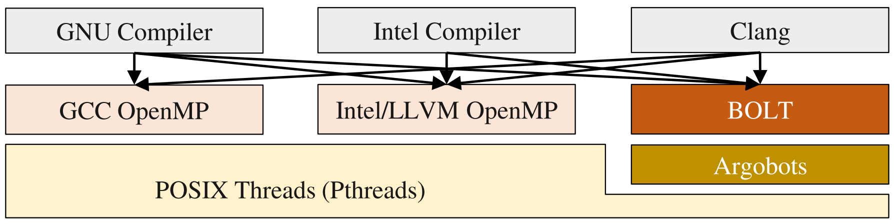


Figure 4.2: Compiler compatibility of BOLT. As well as Intel OpenMP and LLVM OpenMP, BOLT supports GNU Compiler, Intel Compiler, and Clang as an OpenMP frontend compiler.

BOLT was derived from LLVM OpenMP 7.0 [132] to inherit its optimized and modern OpenMP support as well as its application binary interface (ABI) compatibility with other leading OpenMP runtimes. Indeed, despite LLVM OpenMP being a recent project, it is a capable runtime because it builds on the accumulated experience of the Intel OpenMP runtime [102], from which it has been derived. Furthermore, as illustrated in Figure 4.2, the ABI compatibility with GCC OpenMP and Intel OpenMP allows BOLT to work with OpenMP-parallelized applications compiled with the GNU, Intel, and Clang compilers. Unlike existing ULT-based runtimes, this compatibility allows BOLT to evaluate real workloads without modification and recompilation; in particular, BOLT can be used even with commercial and closed source OpenMP-parallelized codes. BOLT can utilize cutting-edge compiler support, including the `omp simd` directive that promotes vectorization [110] and the `omp atomic` directive that forces a compiler and a runtime to perform an operation atomically.

Our main focus is optimizing threading and tasking in OpenMP, so BOLT directly uses most functionalities from LLVM OpenMP. Thanks to rich runtime features supported by LLVM OpenMP, BOLT supports most of OpenMP 4.5 features. It includes several functions to get and set runtime status and parameters, memory allocator, target offloading for accelerators [30, 149], and data-flow parallelism [78]. Since at present most of the applications and runtimes parallelized by OpenMP use OpenMP 4.5, BOLT can run most OpenMP parallelized applications in the real world without missing OpenMP directives specified by programmers.

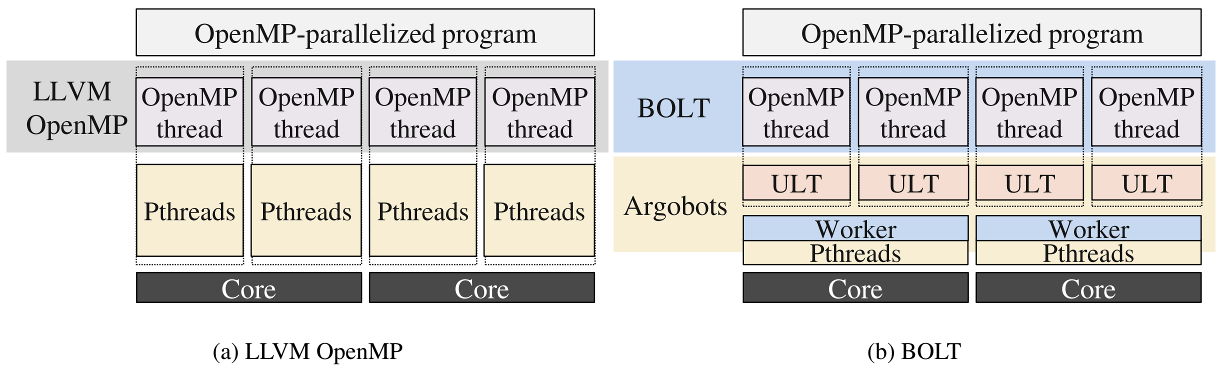


Figure 4.3: Difference of threading layers between LLVM OpenMP and BOLT.

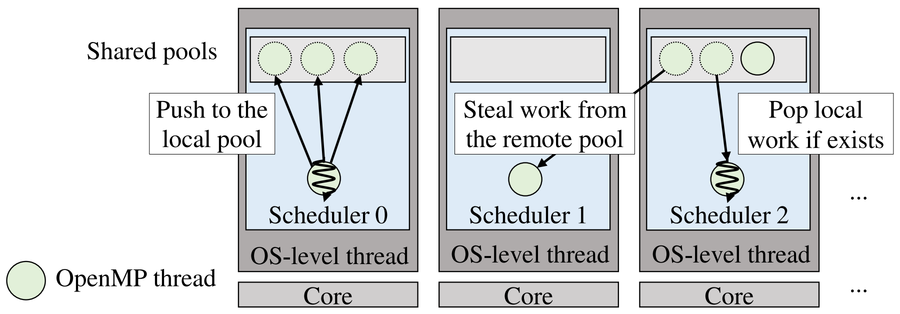


Figure 4.4: Execution Model of BOLT. Created threads by a scheduler are pushed to its local pool. A scheduler pulls a thread from its own pool; if it is empty, random work stealing takes place.

Optimizations of a threading layer are performed as follows. The first step in deriving BOLT from LLVM OpenMP is a straightforward replacement of any use of the Pthreads interface with the similar interface provided by Argobots. This involves not only replacing fork-join calls but also inheriting configuration parameters (e.g., stack size) and adopting ULT-based thread-local storage (TLS) and several synchronization mechanisms. The critical aspect that motivated using Argobots is, in addition to a heavily optimized ULT library, the ability to fully control scheduling and ULT pools, which allows BOLT fine-grained control over thread management. Figure 4.3 visualizes the threading layers in LLVM OpenMP and BOLT. As presented in Figure 4.3, BOLT can keep the number of OS-level threads regardless of the number of OpenMP threads, while LLVM OpenMP has oversubscription of threads since OpenMP threads are mapped to OS-level threads.

Figure 4.4 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 pool that is accessed mainly by the owner but can be a target of work stealing. We adopt a simple random work-stealing algorithm [33]; the scheduler steals ULTs from another pool only when its own pool 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.

4.3.2 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 the 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 the 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 a *hot team* [164], 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 resource 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.

4.3.3 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.

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.

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 [115] and Intel Threading Building Blocks [143]. 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.

4.3.4 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 4.3.2, 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. 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. This setting is known to significantly affect the performance of flat and nested parallelism [94]; the active policy minimizes the latency for a repeated, short, and single-level parallel region, but it imposes immense overheads when oversubscription hap-

```
void omp_thread_func(...) {
    const int YIELD_INTERVAL = 1e6;
START_THREAD:
    [run an implicit task (= work of an OpenMP thread)];
    int count = 0;
    switch (omp_wait_policy) {
    case ACTIVE:
        while (true) {
            if (is_next_team_invoked()) {
                goto START_THREAD; // restart a thread.
            }
            if (count++ % YIELD_INTERVAL == 0) {
                yield_to_sched(); // avoid hang.
            }
        }
    case PASSIVE:
        return; // immediately returns to a scheduler.
    case HYBRID:
        while (true) {
            if (is_next_team_invoked()) {
                goto START_THREAD; // restart a thread.
            }
            thd_desc_t *thd = pop_from_one_of_pools();
            if (thd) {
                // finish, return to a scheduler, and execute a thread.
                return_to_sched_with_thread(thread);
            }
        }
    }
}
```

Figure 4.5: 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.

pens (e.g., parallelism is nested). On the other hand, with the passive policy, the latency is large under flat parallelism.

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. Figure 4.5 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 9), whereas with `passive` a thread finishes immediately after its work to possibly schedule another ready thread (line 17).

However, this static policy mechanism requires users to prioritize the performance of either flat or nested parallelism. The API proposal by Yan et al. [175], 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.

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 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 Figure 4.5. If a ULT is successfully taken in this loop (line 23), the thread exits and returns to the scheduler with the popped ULT, which will be executed next (line 26). This hybrid strategy consists of both the active and passive strategies; it works as `active` with a pool-checking overhead (line 21) under flat parallelism, while `passive` with a flag-checking overhead (line 20) 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 13) 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 [175] contains a keyword that can be mapped to this hybrid behavior. BOLT 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).

4.3.5 Thread-to-Place Binding

To efficiently run a parallel program on modern hierarchical multi-core CPUs, in addition to the efficient execution of parallel regions, exploitation of locality is essential. Several studies tackle locality issues by scheduling them in a locality-aware manner [160, 167], but their proposals are neither portable nor implicit. As a specification, OpenMP 4.0 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 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.

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 threads are scheduled by using decentralized thread pools. For a close mapping

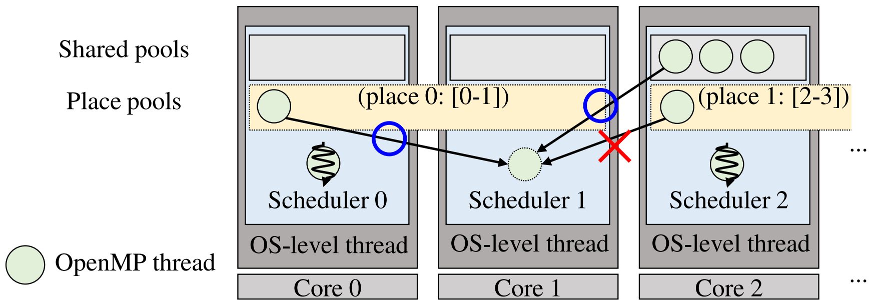


Figure 4.6: Place pools in BOLT. In addition to shared pools, Scheduler 1 can access its own place pool (place 0), but not the other place pools.

between places and thread pools, we created the concept of *place pool* associated with a place (i.e., a user-defined set of *processors*, or schedulers in BOLT) as shown in Figure 4.6. Since only schedulers associated with a place have access to a corresponding place pool, ULTs bound to the place pool 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 pools dynamically; thus, the additional overhead to support places is negligible.

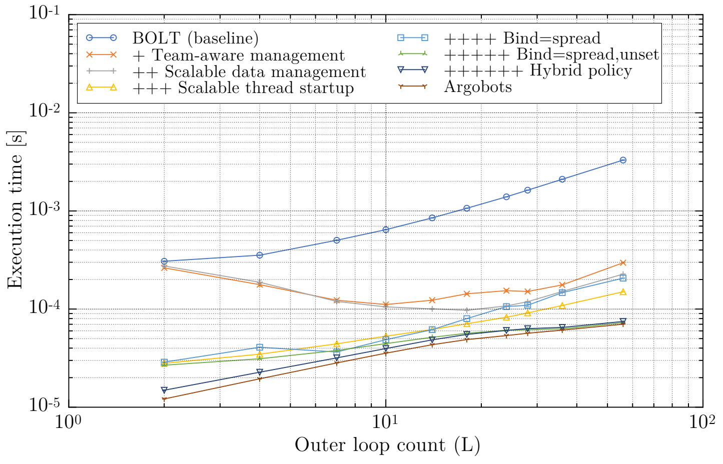
The Problem with Binding Policy Inheritance

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 pool 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 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.

```
#pragma omp parallel for num_threads(L)
for (int i = 0; i < L; i++) {
    #pragma omp parallel for num_threads(N)
    for (int j = 0; j < N; j++) {
        empty(i, j); // no computation
    }
}
```

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



(b) Performance of the microbenchmark on Skylake.

Figure 4.7: Microbenchmark that evaluates overheads of nested parallel regions.

4.3.6 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 4.4).

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 Figure 4.7a. We set N to the number of cores (i.e., 56) and ran this microbenchmark with different values of L. Figure 4.7b 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.

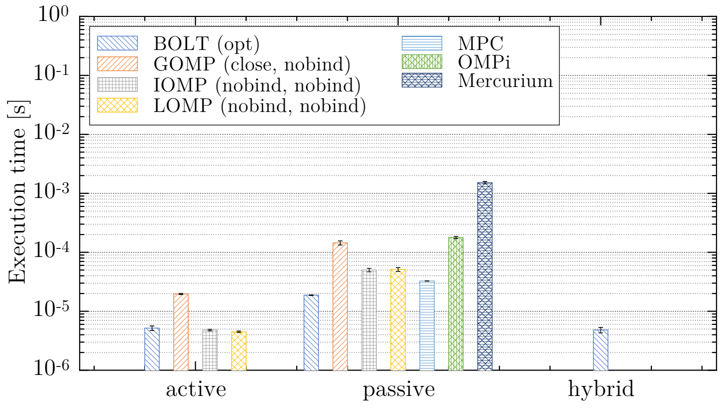
We found that team construction and destruction take more than 92% of the total execution time on the critical path with $L = 56$. As a result, the hot team optimization (Section 4.3.2) can significantly reduce the team management cost (**Team-aware management**) and improves the execution time by roughly 10x. The next most significant bottleneck is the 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 4.3.3), the contention is significantly reduced and the benefits are proportional to the size of the outer parallel region L. Binary thread startup (Section 4.3.3) 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 4.3.5, 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 4.3.5) improves CPU utilization by reducing load imbalance (**Bind=spread,unset**). The hybrid wait policy presented in Section 4.3.4 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 pools and management of thread IDs). Figure 4.7b 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.

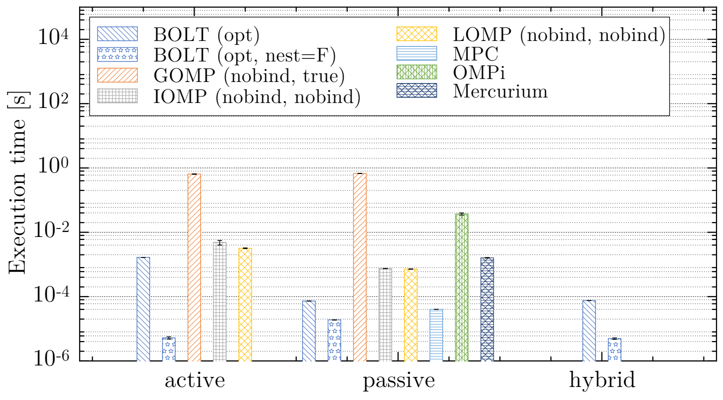
Wait Policy and Performance

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 Figure 4.7b). In addition to the widely used OpenMP implementations (GCC, Intel, and LLVM OpenMP), we evaluated three ULT-based OpenMP runtimes: MPC [135], OMPi [86], and Mercurium [27]. Section 4.4 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 4.4.1.

Figure 4.8a 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



(a) Flat Parallel Regions



(b) Nested Parallel Regions

Figure 4.8: Performance of the microbenchmarks that evaluate the effect of wait policy on Skylake.

`passive`. Figure 4.8a 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.

Figure 4.8b 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 for MPC; in Figure 4.8b, 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 4.4.1 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.

4.4 Evaluation with Microbenchmarks

Table 4.1: Experimental environment used for microbenchmarks of BOLT

Processor	Intel Xeon Platinum 8180M	Architecture	Skylake
Frequency	2.5 GHz	# of sockets	2
# of cores	56 (28 $\times$ 2)	# of HWTs	112 (56 $\times$ 2)
Memory	396 GB	OS	Red Hat 7.4
Compilers	Intel Compiler 17.2.174		

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 system described in Table 4.1. 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 [126], Intel OpenMP [102], and LLVM OpenMP [132] runtimes that ship with GCC 8.1, Intel 17.2.174, and Clang/LLVM 7.0 [111], respectively. The ULT-based category consists of MPC 3.3.0 [135], OMPi 1.2.3 [86] with psthreads 1.0.4 [85], and Mercurium 2.1.0 [27] with Nanos++ 0.14.1.[1] 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`.

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

[1]We could not find the source code of Omni/ST [155] and NanosCompiler [118]. The source code of ForestGOMP [39] is available, but we did not include it because we could not compile and run it in our environment. The latest libKOMP [40] (`https://gitlab.inria.fr/openmp/libkomp`) focuses on tasking and no longer maps OpenMP threads to ULTs, so we exclude it in our evaluation.

the busy-wait configuration under any nested parallelism, an approach that is overall beneficial as discussed in Section 4.3.6. We note that `OMP_WAIT_POLICY` was set to `active` for the **taskloop** evaluation to maximize the performance because it requires only a single-level parallel region. The optimized BOLT includes all the optimizations of Section 4.3 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.

4.4.1 Doubly Nested Loops

We first evaluated two microbenchmarks that reflect cases where the efficiency of nested parallel regions impacts performance. Unlike what we used in Section 4.3.6, the computation is added in order not to let merely aggressive serialization be the best optimization. The first case is the microbenchmark shown in Figure 4.9a. 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.

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

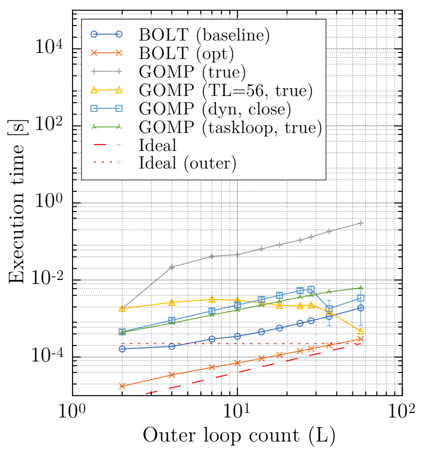
(a) Insufficient parallelism

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

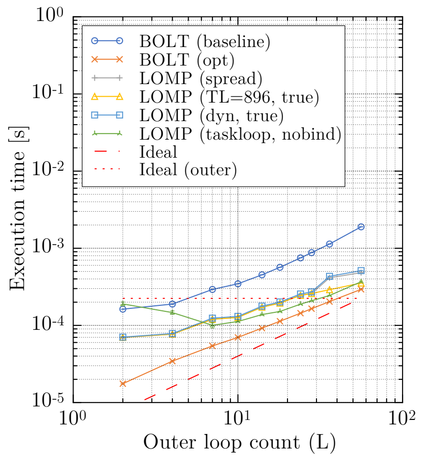
(b) Unbalanced inner loop parallelism

Figure 4.9: Kernels of the microbenchmarks that evaluate nested parallel regions.

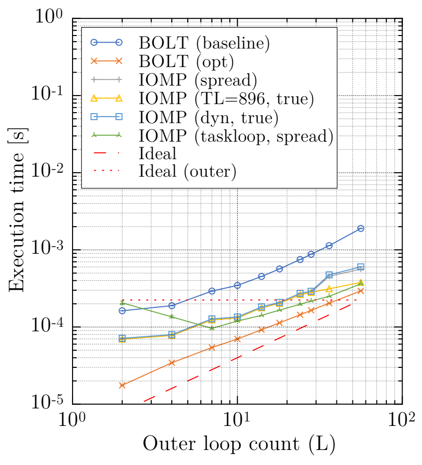
The second case is a program that has unbalanced inner parallel loops, as shown in Figure 4.9b. Since the amount of work of the inner loop is uneven, load imbalance occurs if only the outer loop is parallelized. In theory, parallelizing only the inner loop achieves performance as good as that of nested



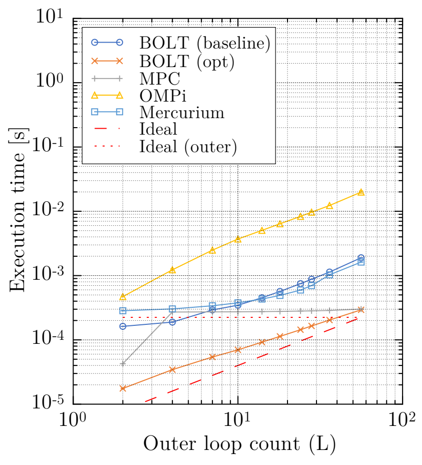
(a) GCC OpenMP (GOMP)



(b) LLVM OpenMP (LOMP)



(c) Intel OpenMP (IOMP)



(d) ULT-based OpenMP

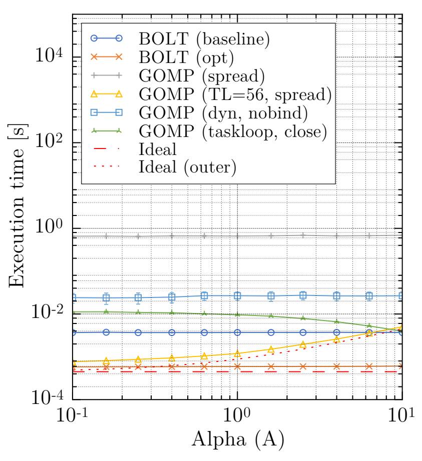
Figure 4.10: Performance of the microbenchmark that has insufficient parallelism on Skylake.

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 Figure 4.9b) to control the degree of imbalance. Let N be a number of cores. The computation size of the ith outer loop iteration is set to W_i cycles, where W_i is calculated as follows:

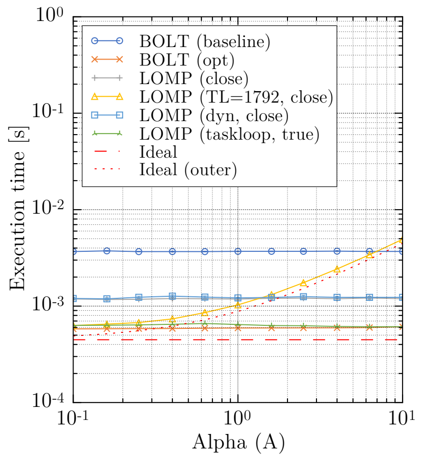
$$W_i = 20000 \cdot N \cdot \frac{(i+1)^\alpha}{\sum_{j=1}^{N} j^\alpha}.$$

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.

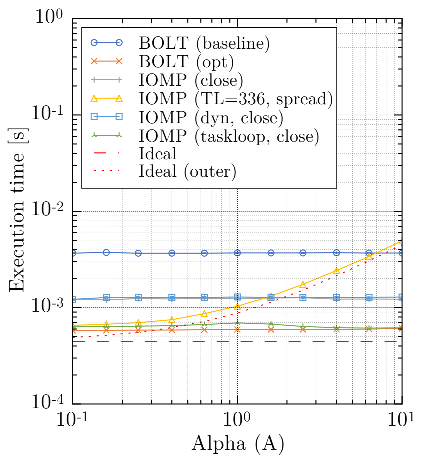
Each measurement calculates the average execution time of the kernel repeated for three seconds after



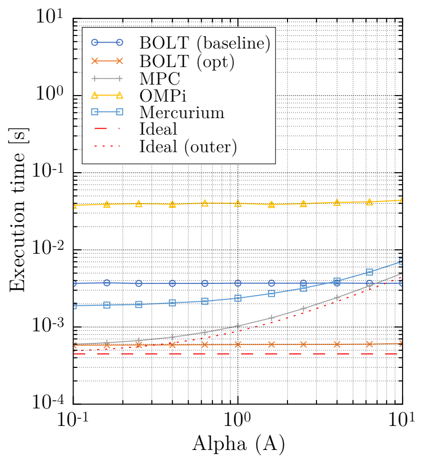
(a) GCC OpenMP (GOMP)



(b) LLVM OpenMP (LOMP)



(c) Intel OpenMP (IOMP)



(d) ULT-based OpenMP

Figure 4.11: Performance of the microbenchmark that has unbalanced inner loop parallelism on Skylake.

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; specifically, since one series contains multiple results at different X values, we chose one that has the smallest geometric mean of execution time.

Figure 4.10 and Figure 4.11 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 Figure 4.10. 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`.

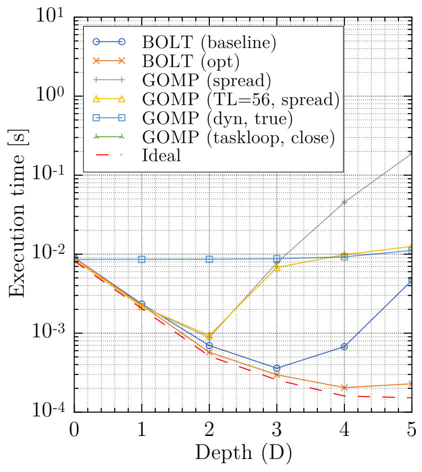
4.4.2 Deeply Nested Loops

We further investigate the overheads of OpenMP parallel regions by artificially introducing deeply nested parallel regions, which can be caused by highly stacked OpenMP-parallelized components. Figure 4.12 shows a microbenchmark to evaluate the overheads under deeply nested parallel regions. In this benchmark each thread recursively creates a parallel region that consists of four threads until the depth reaches D. The leaf thread consumes cycles $20,000 \cdot 4^{5-D}$ so that the total amount of work is fixed to $20,000 \cdot 4^5$ cycles. When D is 0, only the master thread executes the entire work. The experimental setting is the same as that of the experiments with doubly nested loops.

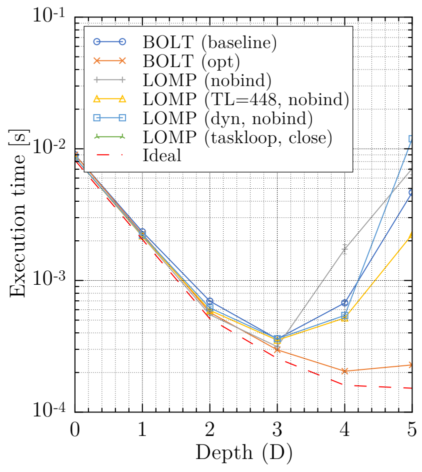
```c
void parallel_region(int depth) {
    if (depth == D) {
        int c = 20000 * pow(4, 5 - D);
        computation(i, j, c /* cycles */);
    } else {
        #pragma omp parallel for num_threads(4) firstprivate(depth)
        for (int i = 0; i < 4; i++) {
            parallel_region(depth + 1);
        }
    }
}

parallel_region(0);
```

Figure 4.12: Kernels of the microbenchmarks that evaluate deeply nested parallel regions.

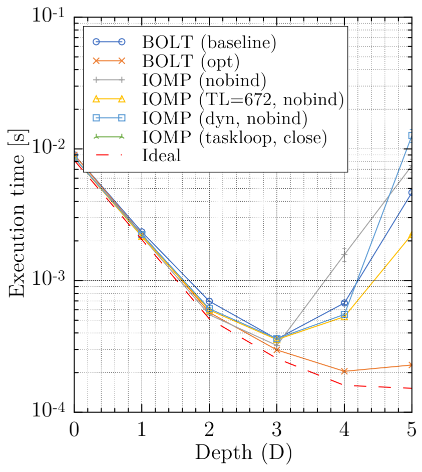
Figure 4.13 shows the performance. In the figure **Ideal** shows the theoretical maximum performance under available parallelism with a given D. **BOLT (baseline)** represents the baseline BOLT and **BOLT (opt)** denotes BOLT with all the optimizations we described. The result shows that with a smaller D



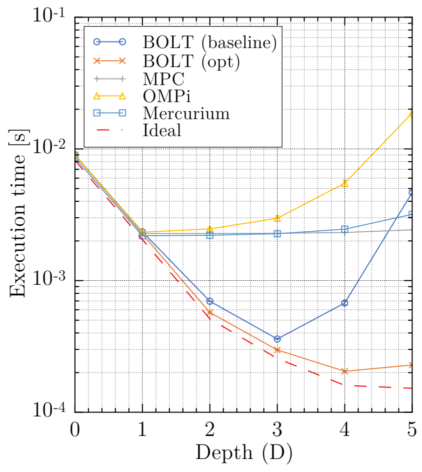
(a) GCC OpenMP (GOMP)



(b) LLVM OpenMP (LOMP)



(c) Intel OpenMP (IOMP)



(d) ULT-based OpenMP

Figure 4.13: Performance of the microbenchmark that has deep nested parallel regions on Skylake.

(e.g., $D < 2$) the performance difference is negligible since the parallelism is nonexistent or flat, while as D becomes larger only **BOLT (opt)** can exploit the nested parallelism.

We note that, although BOLT can minimize the parallelization cost of deeply nested parallelism, at present exploiting such deeply nested parallelism is not beneficial in most cases since it often exceedingly decomposes work and hurts locality. As far as we investigated, we could not find a program that creates more than doubly nested parallel regions at different software stacks. One possible use case would be self-recursive parallel algorithms (e.g., recursive divide-and-conquer parallelism), but OpenMP tasks are provided for such deeply nested parallelism. One can argue that, as our results in Figure 4.10 and Figure 4.11 imply, OpenMP parallel regions using efficient implementation of OpenMP threads can be as lightweight as OpenMP task and taskloop, while task and taskloop provide a better abstraction for self-recursive parallel algorithms. Deeply nested parallel regions could be useful in the future if programs get more and more highly stacked and complex and numbers of cores of a single processor keep increasing.

4.5 Summary

We presented a ULT-based OpenMP runtime called BOLT, which is aimed at production use by offering modern OpenMP support, unprecedented scalability for nested parallelism support, and high efficiency of flat parallelism support compared with leading production runtimes. We showed that the previous ULT-based OpenMP implementations lack efficient support for nested parallel regions while failing to provide latest OpenMP features. The design of BOLT came from an in-depth investigation of LLVM OpenMP implementations as well as OpenMP specification. We also discovered some limitations in specification that inhibit optimization for lightweight ULT-based runtimes. Our microbenchmarks show that BOLT achieves better performance under nested parallelism than do both the widely used OS-level thread-based OpenMP runtimes and the state-of-the-art ULT-based runtimes without hurting the performance of flat parallel regions.

The study of BOLT heavily focuses on traditional parallel region but leaves other aspects such as support for tasks and accelerators. Investigating their efficient supports in the context of ULTs is our future work. We solely focus on OpenMP and thus do 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 a parallel loop abstraction. Hence, another direction of extending this work is to evaluate the efficacy of our techniques on other ULT-based parallel systems.

5 Application Benchmarks

In this chapter we evaluate our lightweight threading libraries, Argobots and BOLT, with five real-world applications. The first section evaluates the user-level threading techniques that support parent- and child-first scheduling implemented in Argobots. Section 5.2 shows the performance of BOLT in two applications that have OpenMP nested parallel regions. Finally, Section 5.3 presents the result of an OpenMP-parallelized KMeans over BOLT to see the effect of choosing different user-level threading techniques in BOLT.

5.1 Evaluation of Lightweight Threading Techniques in Argobots

Section 3.7 uses a microbenchmark to measure the threading overheads of the various user-level threading techniques in Argobots regarding the deviation probability. We evaluate the benefits of the lightweight user-level threading techniques with two fine-grained parallel applications: ExaFMM and Graph500. These applications utilize context switch to efficiently schedule other ready work when currently running ULTs need to wait for the completion of other threads or communications. All the applications were compiled with `-O3`. The stack sizes of ULTs and schedulers were set to 64 KB for ExaFMM and 16 KB for Graph500. All the following results are the arithmetic mean. The error bars in the charts indicate the 95% confidence intervals.

5.1.1 ExaFMM

Table 5.1: Experimental environment of ExaFMM

Processor	Intel Xeon Phi 7210	Architecture	Knights Landing
Frequency	1.3 GHz	# of sockets	1
# of cores	64	# of HWTs	128
Memory	198 GB	OS	Red Hat 7.5
Compilers	Intel Compiler 17.2.174		

ExaFMM [179] is a highly optimized $O(N)$ N-Body solver using a fast multiple method. In the kernel, a tree is traversed in a divide-and-conquer manner, and leaf nodes calculate the actual forces.

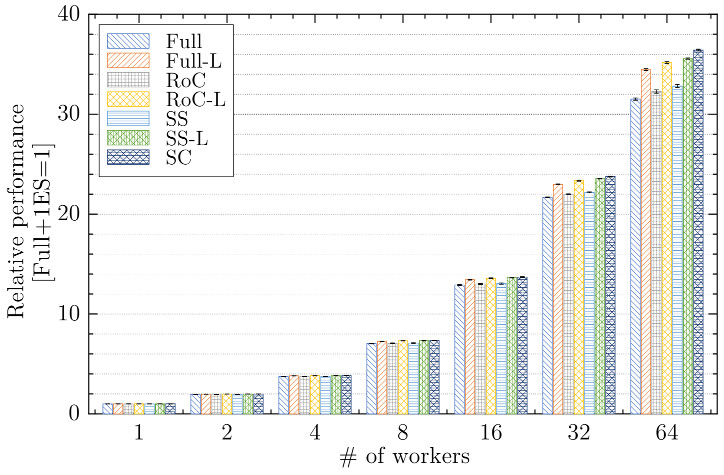
Recursive divide-and-conquer task parallelism has been known to efficiently parallelize the ExaFMM kernel [156]. Deviations can happen while waiting for completion of child ULTs, but they never occur in leaf nodes because they just perform computation without synchronization. The most efficient solution seems mapping leaf nodes to stackless threads and internal nodes to ULTs. However, this optimization requires identifying leaf ULTs on creation, which is not only cumbersome but also expensive if the leaf condition is complicated. The dynamic promotion techniques alleviate the programmers' burden without hurting performance.

We evaluated ExaFMM on KNL as described in Table 5.1. We ran ExaFMM ten times on KNL with `--ncrit 16 -t 0.15 -P 4 --dual -n 524288` as arguments. As the number of workers changed, we adjusted `--nspawn` to keep the number of created ULTs per worker constant (within 5% of error) while `--nspawn 256` was set with 64 workers. In addition to the original vectorization in ExaFMM [178], we further manually vectorized the compute kernels to efficiently utilize SIMD units in KNL. To reduce internal nodes, we applied the orthogonal decomposition; we changed the way of work decomposition and collapsed internal nodes in the traversal tree while keeping the computation of leaf nodes. We measured the performance of the tree traversal where the program spends more than 90% of the total execution time.

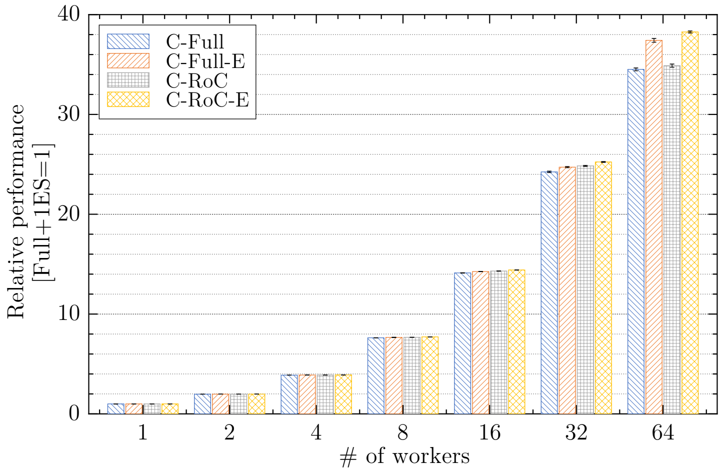
Figure 5.1 presents the performance of ExaFMM with different numbers of workers. Since the deviation probability is low (regardless of the number of workers, approximately 1.5% of ULTs are dynamically promoted), the dynamic promotion techniques achieved better performance. LSA and ESR (**-L** and **-E**) improved performance with 64 workers. Reduction of context-switching overheads contributes to a 9% speedup for parent-first (**SC** over **Full-L**) and 2% for child-first scheduling (**C-RoC-E** over **C-Full-E**). This ExaFMM showcases the merit of child-first scheduling; the child-first methods overall perform better than the parent-first ones because child-first scheduling can efficiently exploit locality when parallelism is deep and narrow [122].

We note that the dynamic promotion techniques are suitable for divide-and-conquer recursive parallelism; in a k-ary tree approximately $1 - \frac{1}{k}\%$ of ULTs are leaves;[1] and therefore if no deviation happens in leaf ULTs (e.g., no yielding), D is approximately $\frac{1}{k}\%$. The results in Figure 3.28 and Figure 3.29 show that the dynamic promotion techniques perform better than the fully fledged threads when D is less than 30 to 50%. Because D is at most 50% ($k = 2$), the dynamic promotion techniques are beneficial in most cases.

[1]Denote the number of internal nodes in a task tree N and the number of leaf nodes n. When $N = 1$, (N, n) is $(1, k)$. Because 1 leaf can be replaced with 1 internal node and k leaves, (N, n) becomes $(N, N(k-1)+1)$. Hence, the ratio of leaf ULTs is calculated as $\frac{n}{N+n} \approx 1 - \frac{1}{k}$ with a larger N.



(a) Parent-first scheduling

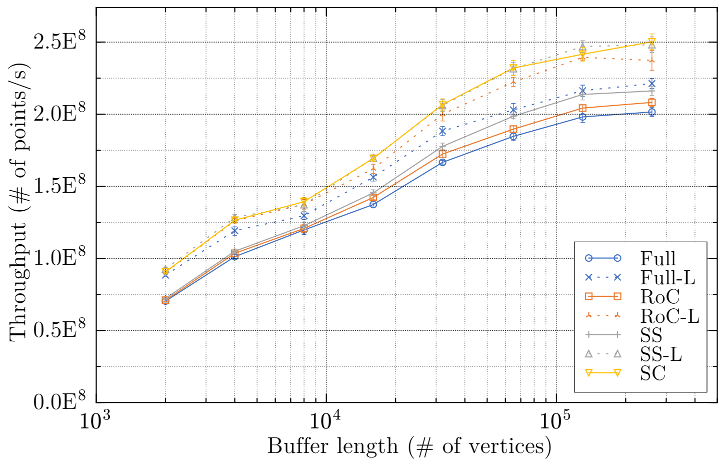


(b) Child-first scheduling

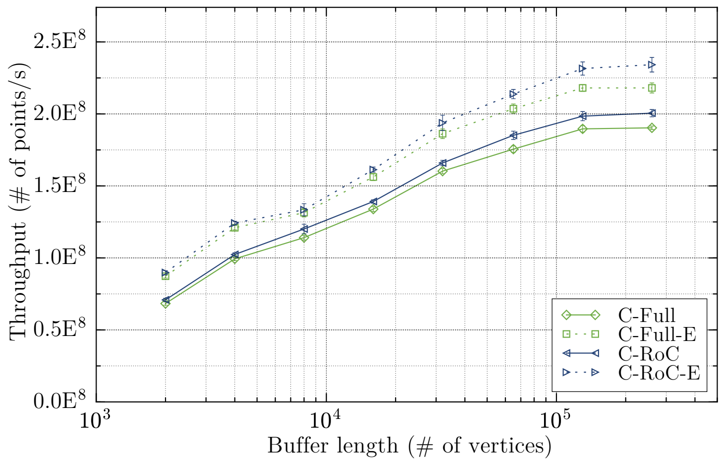
Figure 5.1: Relative performance of ExaFMM on KNL. The baseline is the performance of fully fledged threads (**Full** and **C-Full**) with a single worker.

5.1.2 Distributed Graph500

Graph500 [19] is a well-known benchmark that traverses a distributed graph in a breadth-first manner. Our Graph500 is based on the reference implementation of MPI+Thread found in [17], which achieves distribute memory parallelization with MPI [121] while threads are used for intra-node parallelism. Since each process has only a part of the whole graph, interprocess communication is necessary in order to visit vertices in remote subgraphs. Specifically, in each iteration, every process repeats visiting adjacent vertices. A process can update a vertex if locally owned, while it needs to send a message to another process if the vertex exists in a remote node. In order to avoid the finest communication, message aggregation is commonly adopted. Each process has buffers associated with all the target



(a) Parent-first scheduling



(b) Child-first scheduling

Figure 5.2: Traversed edges per second of Graph500 using 1,024 cores.

ranks to store visit messages; messages are sent only when a buffer gets full and thus needs to be flushed. Because the bottleneck of Graph500 is communication, hybrid parallelism is often used to reduce the intranode communication overheads. MPI+Thread, where ULT is used as Thread, has been studied to exploit fine-grained communication on a distributed system [72, 117], because a ULT can efficiently switch to another ULT when an MPI function blocks. We note that the current state-of-the-art MPI+Thread implementation [123] sometimes acquires a lock even in nonblocking MPI calls (e.g., `MPI_Isend()` and `MPI_Test()`) since MPI functions need to periodically handle global progress for active messages and nonblocking collectives, which is typically protected by a global lock. The dynamic promotion techniques are expected to reduce overheads in cases where ULTs do not call an MPI function or happen to avoid lock contentions in the MPI runtime.

Table 5.2: Experimental environment of Graph500

Processor	Intel Xeon Phi 7230	Architecture	Knights Landing
Frequency	1.3 GHz	# of sockets	1
# of cores	64	# of HWTs	128
Memory	99 GB	OS	CentOS 7.6
Compilers	Intel Compiler 17.0.4	Interconnect	Intel Omni-Path

We parallelized the Graph500 implementation in [17] with Argobots and evaluated the performance over Argobots-aware MPICH [123]. We associated one visit with a ULT to focus on threading overheads. A buffer length B is a parameter to control the communication granularity; with a larger B, more memory is consumed, but more messages are aggregated. We set a scale factor to 26, so the whole graph over nodes consists of 2^{26} vertices. We executed this benchmark on 16 KNLs described in Table 5.2. We spawned a single MPI process per node, each of which ran 64 workers, so 1,024 workers were used in total.

Figure 5.2 shows the averages of ten times execution. In this setting, the ratio of promoted ULTs is less than 1.0%, highlighting the efficacy of the dynamic promotion techniques. The fully fledged (**Full** and **C-Full**) techniques should perform better with extremely small B and higher deviation probability, while the communication overheads would mask their benefit.

5.2 Evaluation of BOLT

In this section, we evaluate the performance of BOLT. To illustrate the benefits of nested parallelism in real-world cases, we chose two applications for evaluation: KIFMM [48] and Qbox [84]. 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 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 [100].

5.2.1 KIFMM

Table 5.3: Experimental environment of KIFMM

Processor	Intel Xeon Platinum 8180M	Architecture	Skylake
Frequency	2.5 GHz	# of sockets	2
# of cores	56 (28×2)	# of HWTs	112 (56×2)
Memory	396 GB	OS	Red Hat 7.4
Compilers	Intel Compiler 17.2.174		

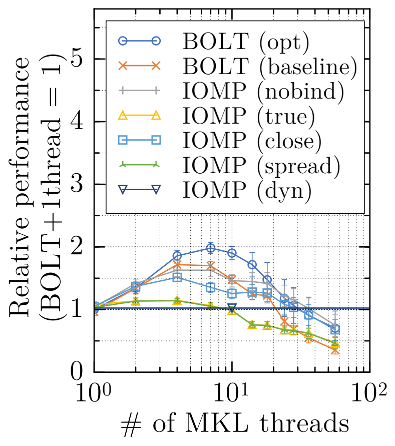
KIFMM is a kernel-independent fast multipole method that is another efficient solver of N-body problems [177]. Our evaluation used its highly optimized implementation [48]. The phases *upward* and *downward* are time-consuming phases that have node traversals consisting of OpenMP-parallelized loops calling BLAS routine `dgemv()`, as presented in Figure 5.3. 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.

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

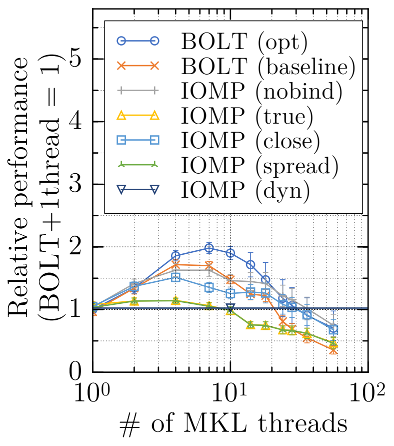
Figure 5.3: Kernel of the upward phase in KIFMM; `dgemv()` is parallelized with OpenMP's `parallel for` in MKL.

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 Figure 5.3; 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 vectorized 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.

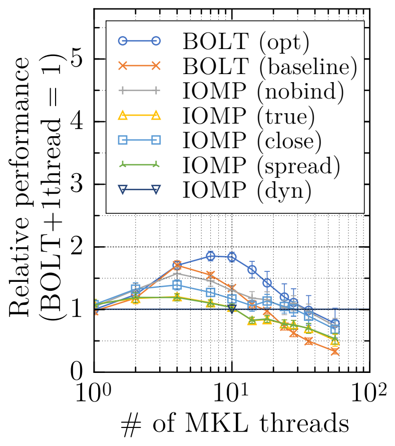
Figure 5.4 shows the relative performance of the traversal in the upward phase, where the baseline is the performance of BOLT with one MKL thread; the performance obtained by parallelizing only the outer loop is the result with a single MKL thread. We ran KIFMM on an Intel Skylake processor pre-



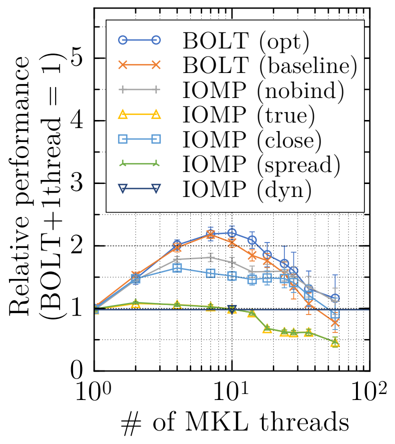
(a) NP=12 + 100,000 points



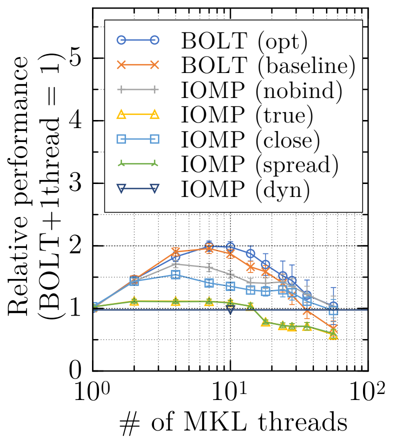
(b) NP=12 + 200,000 points



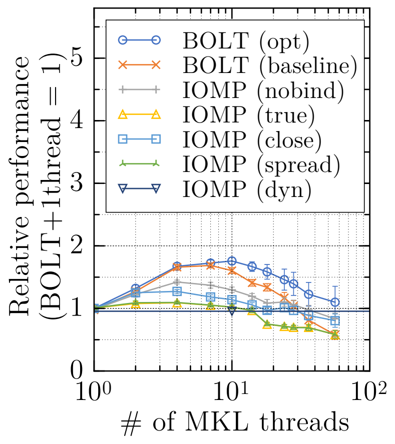
(c) NP=12 + 500,000 points



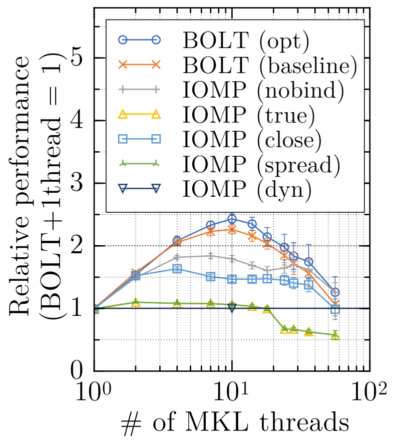
(d) NP=14 + 100,000 points



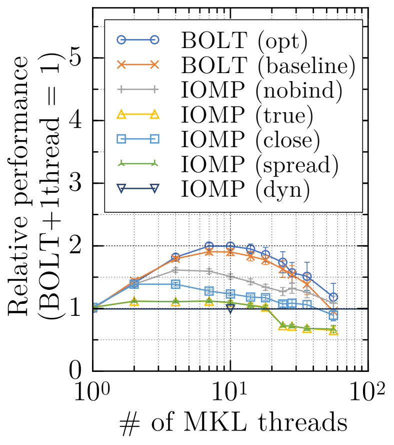
(e) NP=14 + 200,000 points



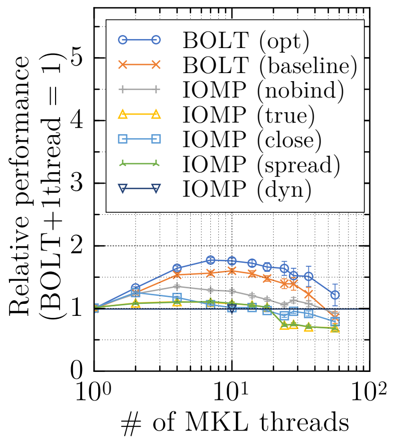
(f) NP=14 + 500,000 points



(g) NP=16 + 100,000 points



(h) NP=16 + 200,000 points



(i) NP=16 + 500,000 points

Figure 5.4: Performance of the traversal in the upward phase of KIFMM on Skylake.

sented in Table 5.3. Figure 5.4 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 Figure 5.4), 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.

5.2.2 Qbox

Table 5.4: Experimental environment of Qbox

Processor	Intel Xeon Phi 7230	Architecture	Knights Landing
Frequency	1.3 GHz	# of sockets	1
# of cores	64	# of HWTs	128
Memory	99 GB	OS	CentOS 7.6

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

Figure 5.5: Kernel of the 3D FFT in Qbox with a 2D FFTW plan.

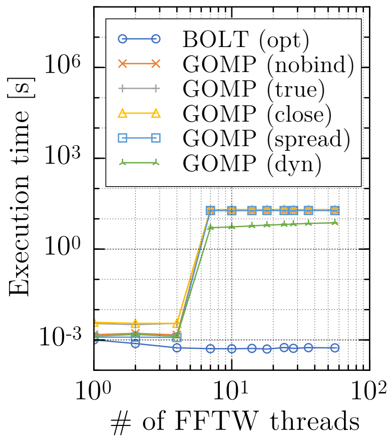
Qbox is a first-principles molecular dynamics code [84] supporting both intranode parallelism with OpenMP and internode parallelism with MPI. We chose the implementation in the CORAL benchmark with the Gold benchmark, which computes the electronic structure of gold atoms. We focus on a fast Fourier transform (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 FFTW 3.3.8 [69, 70], 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 3D FFT cases. We use a KNL machine described in Table 5.4 for both 2D and 3D cases.

Two-Dimensional FFT

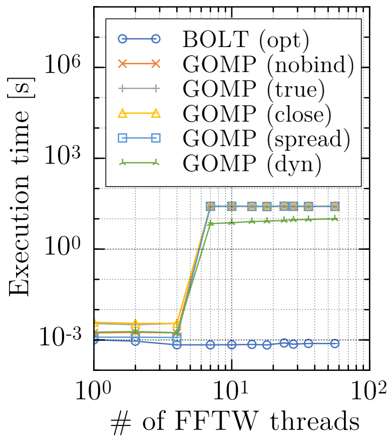
We first evaluated the 2D FFT case. Figure 5.5 presents the kernel of the FFT kernel in Qbox; the inner OpenMP parallel loop calls `fftw_execute()`, which executes a 2D FFT. We changed two parameters to evaluate the nested parallelism. The first one is the number of MPI processes. As implied in Figure 5.5, 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 Figure 5.5; larger input having more atoms increases `num`, rendering nested parallelism less significant. We note that the computational 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. Since heuristics-based FFTW plans obtained with `FFTW_ESTIMATE` were suboptimal, 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. We note that we are aware of nesting levels during the autotuning process; 2D FFT plans were autotuned in a parallel region with `single` because the created FFT plans are used in a parallel loop in our evaluation. In this evaluation, we used GCC 8.1 for GNU OpenMP, Clang/LLVM 7.0 for LLVM OpenMP, and Intel Compiler 17.0.4 for BOLT and Intel OpenMP and compile the FFTW3 library and the kernel with `-O3` and hardware-specific optimization flags to fully exploit KNL. We omit the performance data of MPC, OMPi, and Mercurium because they did not work correctly.

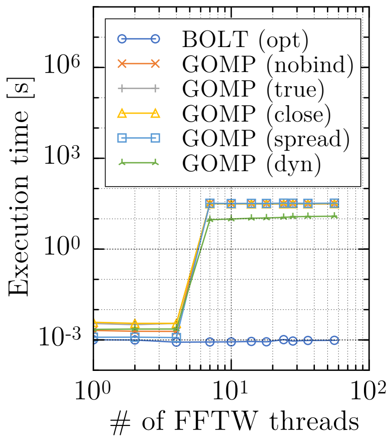
Figure 5.6, Figure 5.7, and Figure 5.8 show the relative performance of the 3D FFT in the backward phase compared with GNU, Intel, and LLVM OpenMP. The baseline is the performance of BOLT with one FFTW thread. These figures demonstrate that only BOLT can exploit nested parallelism and improve performance. 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 to exploit massively parallel hardware. The result also shows that BOLT outperforms the others; the autotuning process generates efficient plans for the lightweight OpenMP runtime.



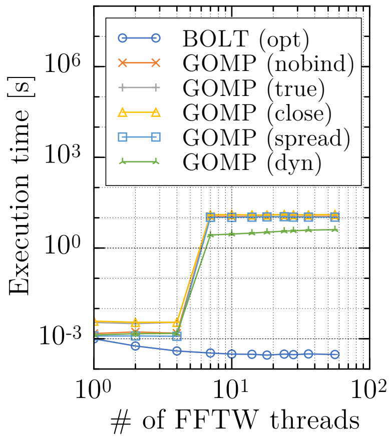
(a) 64 atoms + 16 MPI processes



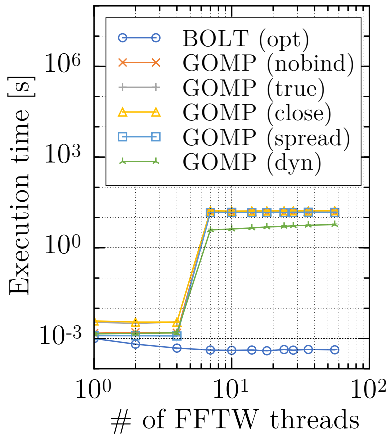
(b) 96 atoms + 16 MPI processes



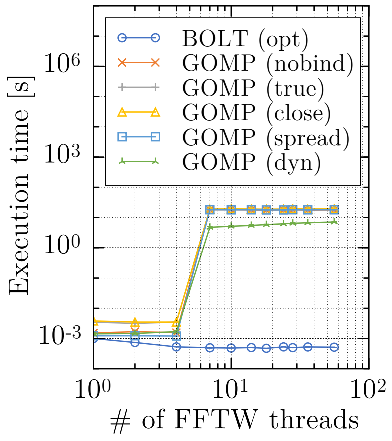
(c) 128 atoms + 16 MPI processes



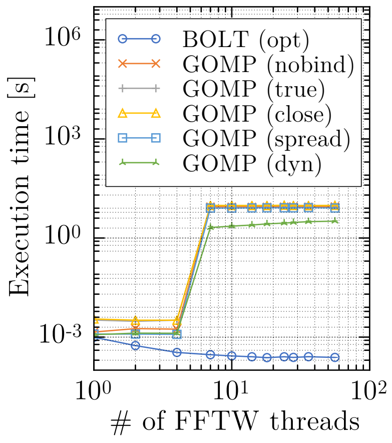
(d) 64 atoms + 32 MPI processes



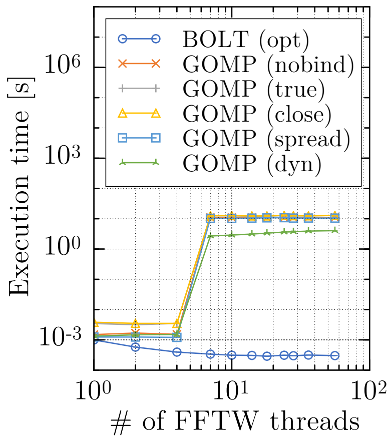
(e) 96 atoms + 32 MPI processes



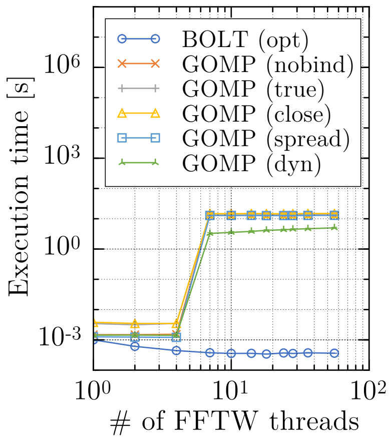
(f) 128 atoms + 32 MPI processes



(g) 64 atoms + 48 MPI processes

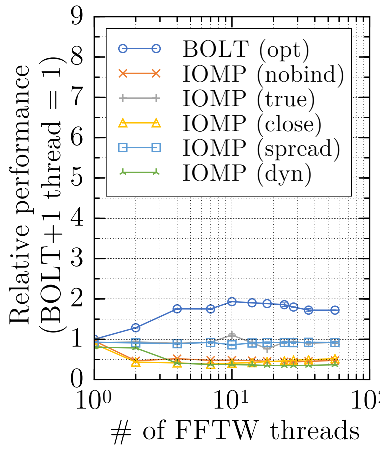


(h) 96 atoms + 48 MPI processes

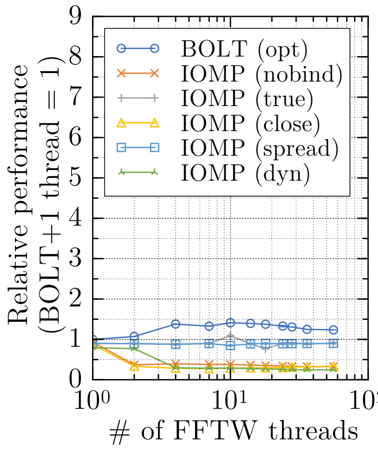


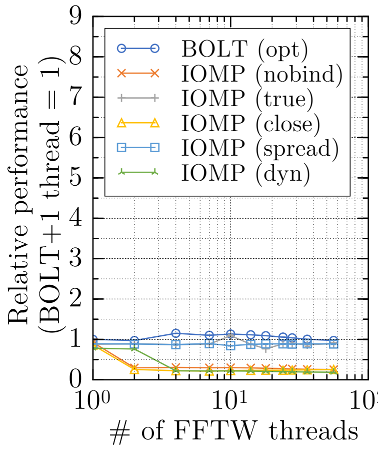
(i) 128 atoms + 48 MPI processes

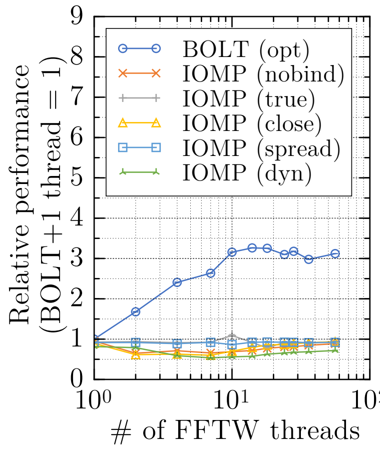
Figure 5.6: Performance of the 3D FFT kernel in Qbox with a 2D FFTW plan compiled with GNU C Compiler.

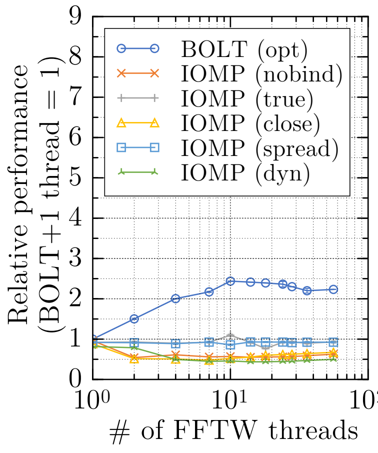


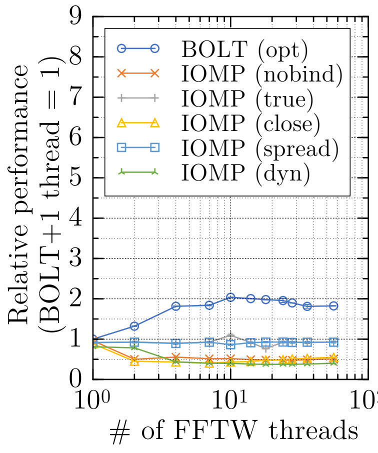
(a) 64 atoms + 16 MPI processes

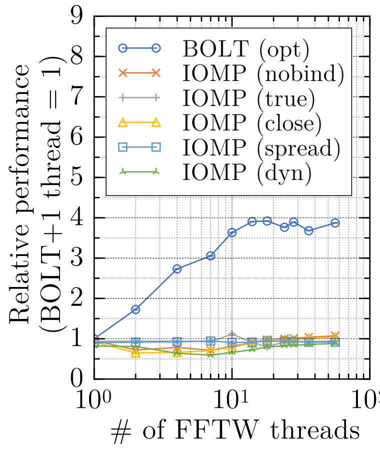

(b) 96 atoms + 16 MPI processes

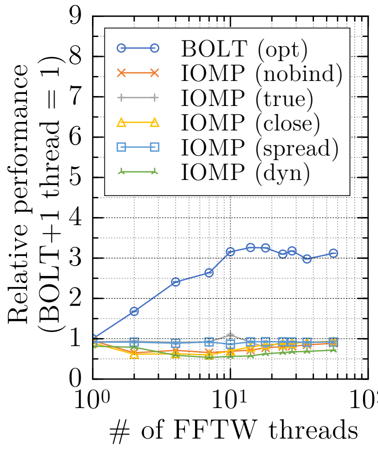

(c) 128 atoms + 16 MPI processes


(d) 64 atoms + 32 MPI processes


(e) 96 atoms + 32 MPI processes


(f) 128 atoms + 32 MPI processes


(g) 64 atoms + 48 MPI processes


(h) 96 atoms + 48 MPI processes

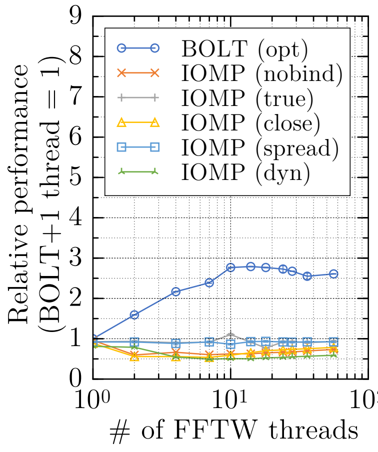
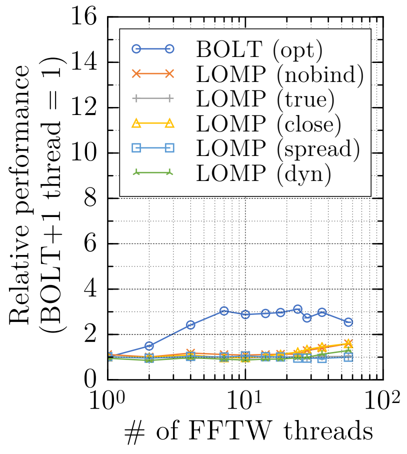
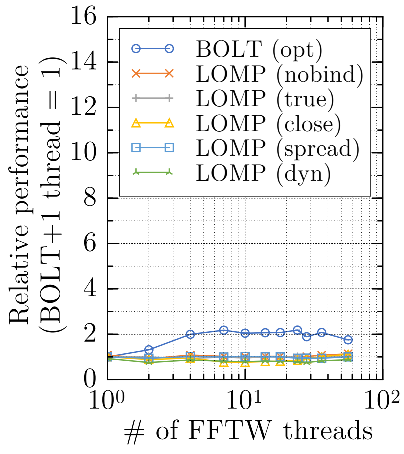

(i) 128 atoms + 48 MPI processes

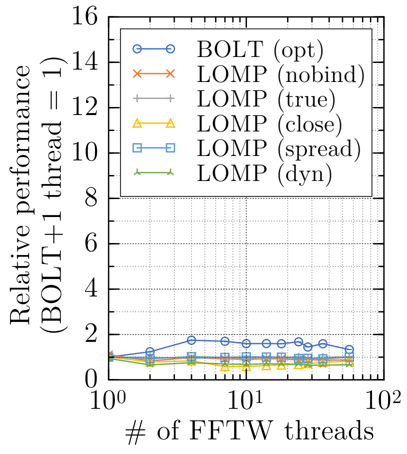
Figure 5.7: Performance of the 3D FFT kernel in Qbox with a 2D FFTW plan compiled with Intel C Compiler.



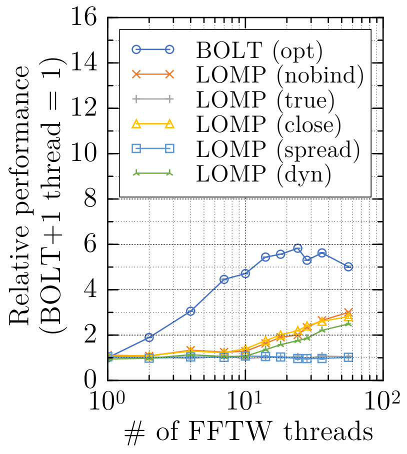
(a) 64 atoms + 16 MPI processes



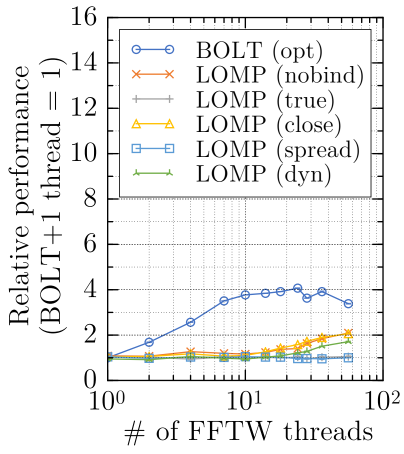
(b) 96 atoms + 16 MPI processes



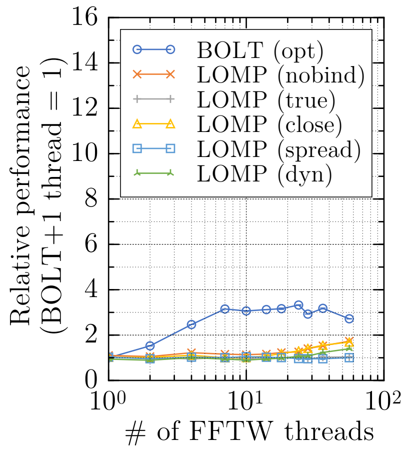
(c) 128 atoms + 16 MPI processes



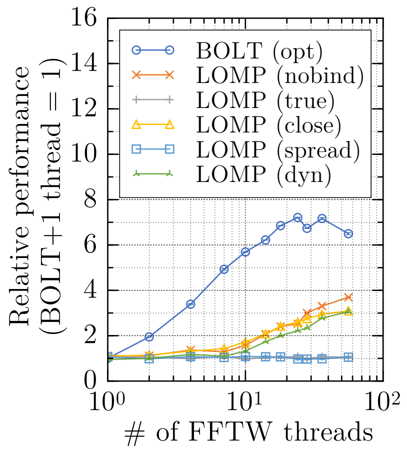
(d) 64 atoms + 32 MPI processes



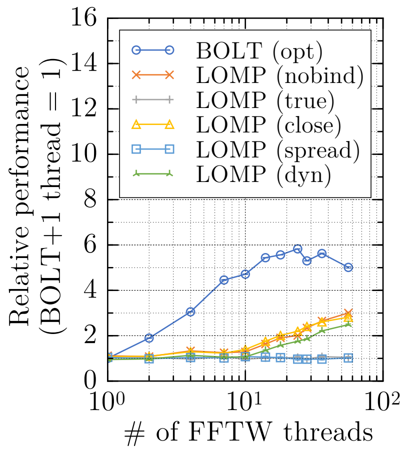
(e) 96 atoms + 32 MPI processes



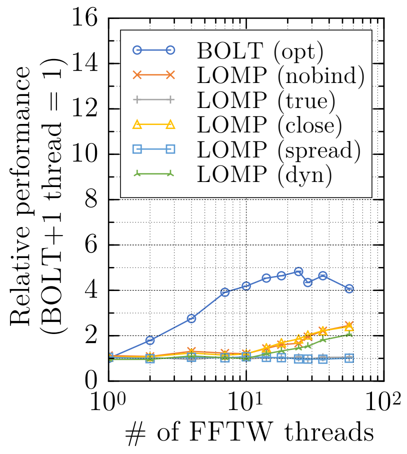
(f) 128 atoms + 32 MPI processes



(g) 64 atoms + 48 MPI processes



(h) 96 atoms + 48 MPI processes



(i) 128 atoms + 48 MPI processes

Figure 5.8: Performance of the 3D FFT kernel in Qbox with a 2D FFTW plan compiled with Clang/LLVM.

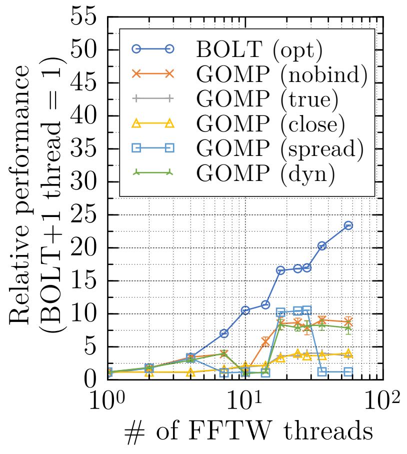
Three-Dimensional FFT

```
// fftw_execute() creates nested parallel regions.
fftw_execute(plan_3d, ...);
```

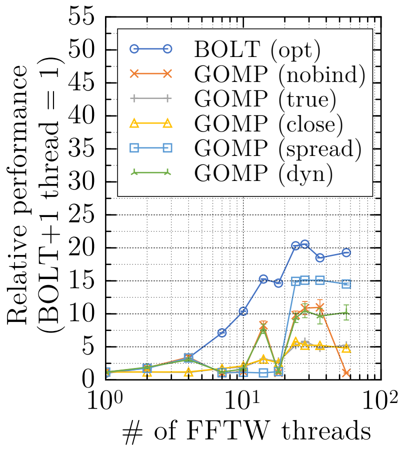
Figure 5.9: Kernel of the 3D FFT in Qbox with a 3D FFTW plan.

We also evaluate the performance of the 3D case that has the kernel presented in Figure 5.9. The 3D case is simple since users only need to run a single function. Compared with the 2D case, however, since FFTW3 provides only one variable to control the number of threads, users cannot flexibly control parallelism in the 3D FFT case. The experimental setting is the same as that of the 2D FFT case. We first created optimized wisdom with `FFTW_PATIENT` for all combinations of numbers of FFTW threads and OpenMP settings to use the optimal plans. The 3D FFT creates three-level nested parallelism in the FFTW3 library, which we found excessively fine grained. To avoid excessively fine-grained parallelism, we disabled the third-level parallelism by setting `OMP_MAX_ACTIVE_LEVEL` to 2. We changed the number of threads in FFTW3 and evaluated the performance. Unlike the 2D FFTW case, this kernel works with MPC 3.3.0 and Mercurium 2.1.0, so in addition to GNU, Intel, and LLVM OpenMP, we compared BOLT with them in this evaluation.

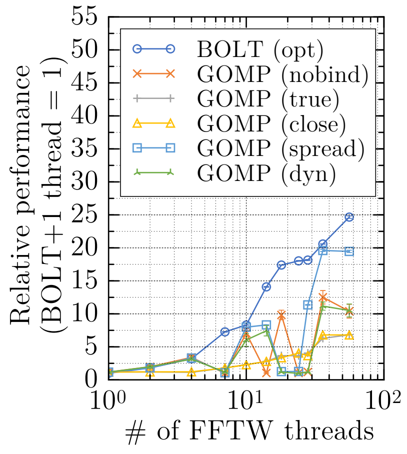
Figure 5.10, Figure 5.11, Figure 5.12, Figure 5.13, and Figure 5.14 present the relative performance of the 3D FFT cases compared with GNU OpenMP, Intel OpenMP, LLVM OpenMP, MPC, and Mercurium. The baseline is the performance of BOLT with one FFTW thread and thus it shows the single-threaded performance. These results show that BOLT achieves the best performance among all the OpenMP systems, proving that, as well as the 2D FFT case, optimized BOLT can efficiently exploit nested OpenMP parallel regions.



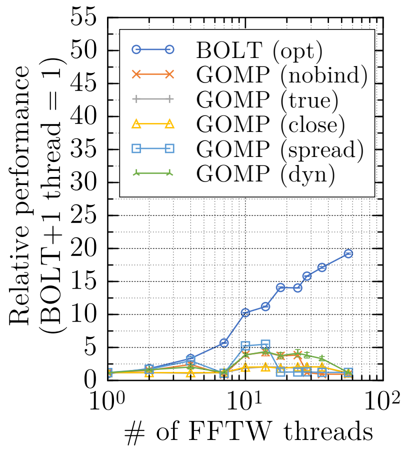
(a) 64 atoms + 16 MPI processes



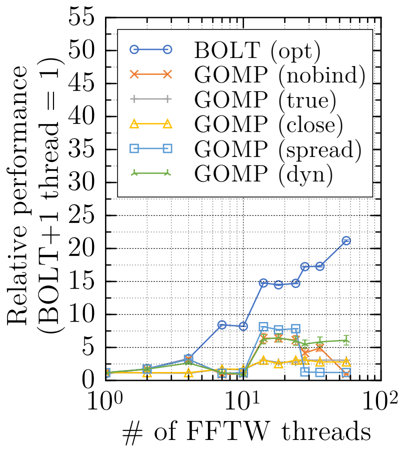
(b) 96 atoms + 16 MPI processes



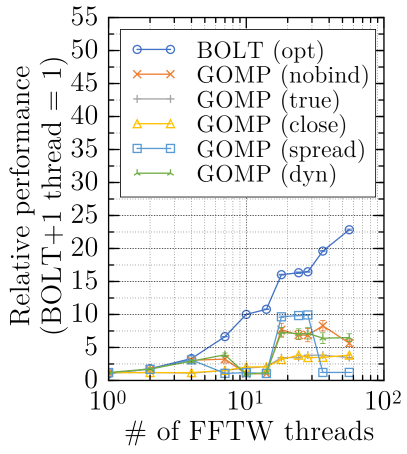
(c) 128 atoms + 16 MPI processes



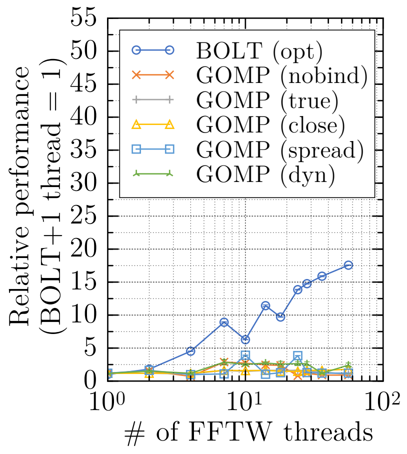
(d) 64 atoms + 32 MPI processes



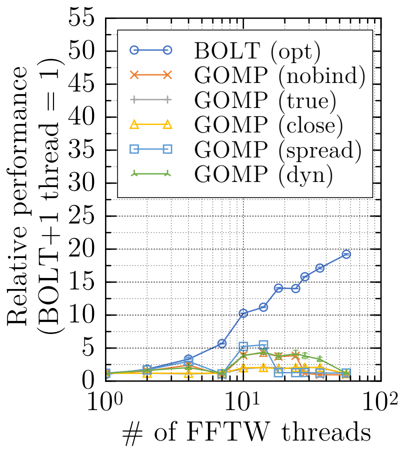
(e) 96 atoms + 32 MPI processes



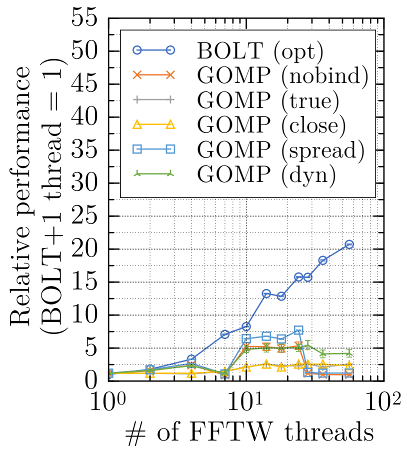
(f) 128 atoms + 32 MPI processes



(g) 64 atoms + 48 MPI processes

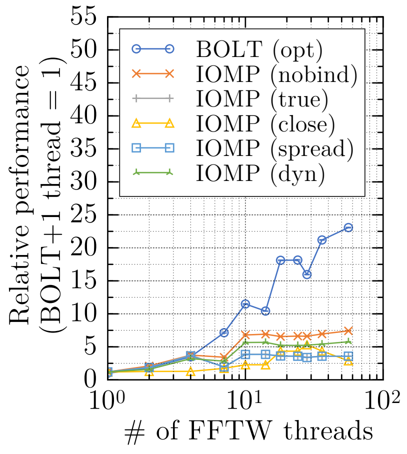


(h) 96 atoms + 48 MPI processes

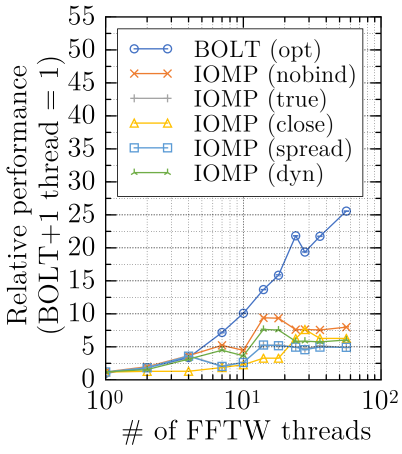


(i) 128 atoms + 48 MPI processes

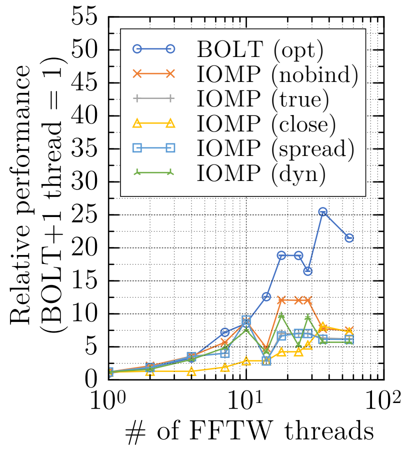
Figure 5.10: Performance of the 3D FFT kernel in Qbox with a 3D FFTW plan compiled with GNU C Compiler.



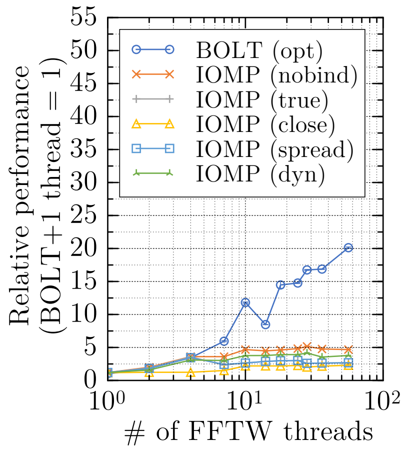
(a) 64 atoms + 16 MPI processes



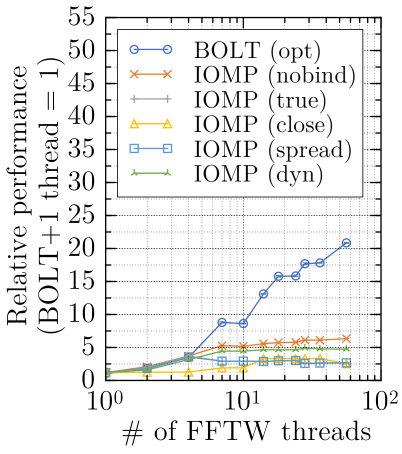
(b) 96 atoms + 16 MPI processes



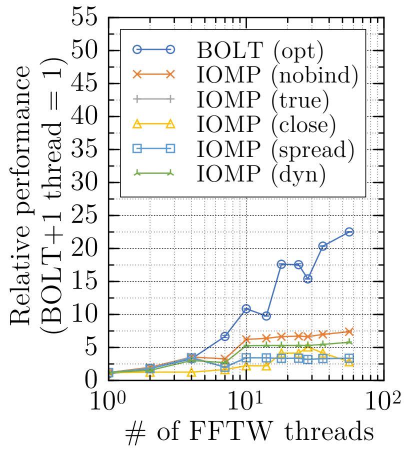
(c) 128 atoms + 16 MPI processes



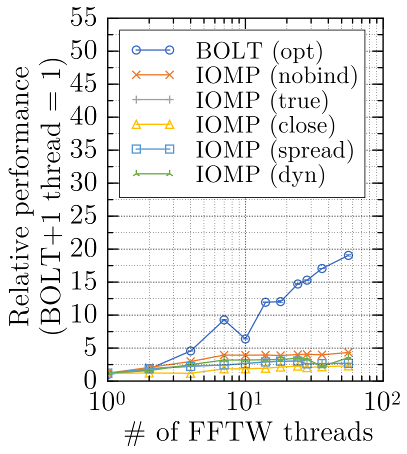
(d) 64 atoms + 32 MPI processes



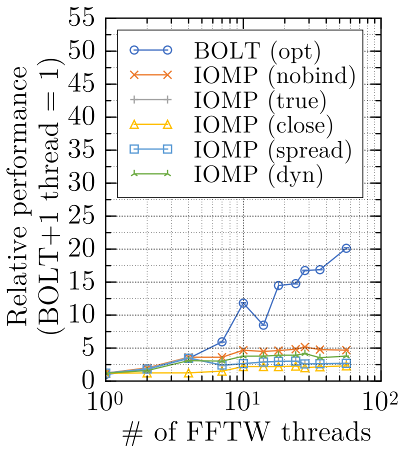
(e) 96 atoms + 32 MPI processes



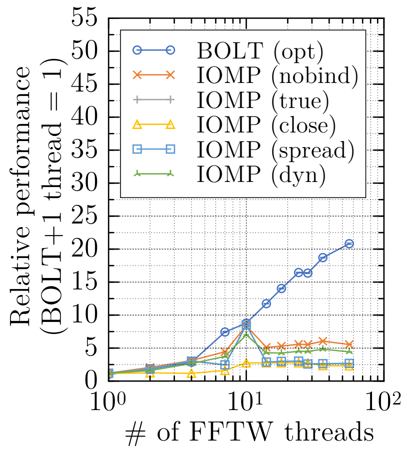
(f) 128 atoms + 32 MPI processes



(g) 64 atoms + 48 MPI processes

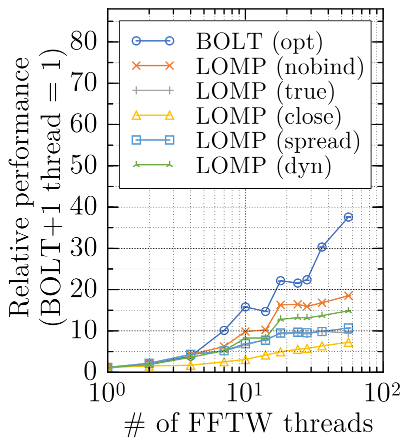


(h) 96 atoms + 48 MPI processes

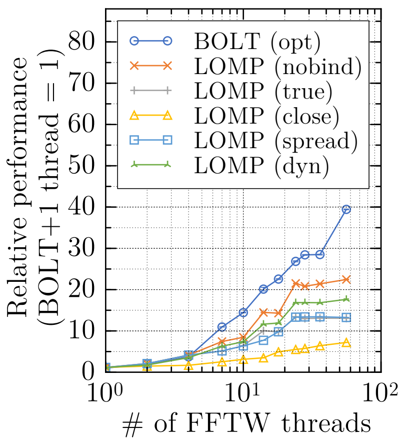


(i) 128 atoms + 48 MPI processes

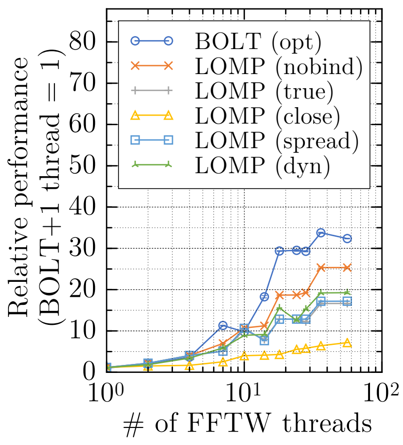
Figure 5.11: Performance of the 3D FFT kernel in Qbox with a 3D FFTW plan compiled with Intel Compilers.



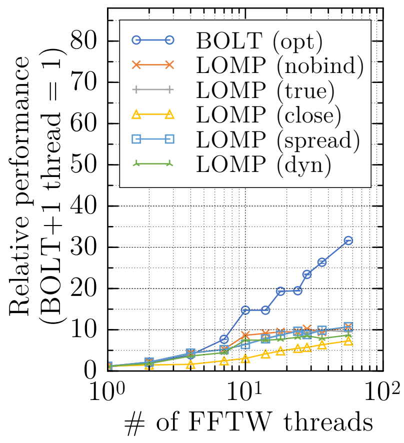
(a) 64 atoms + 16 MPI processes



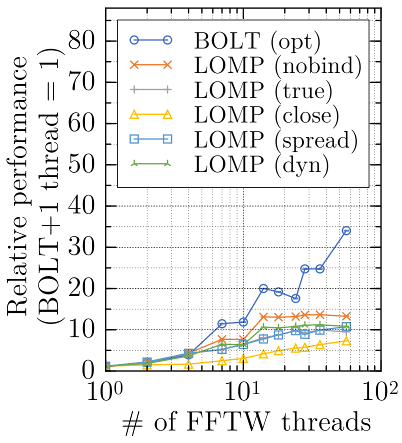
(b) 96 atoms + 16 MPI processes



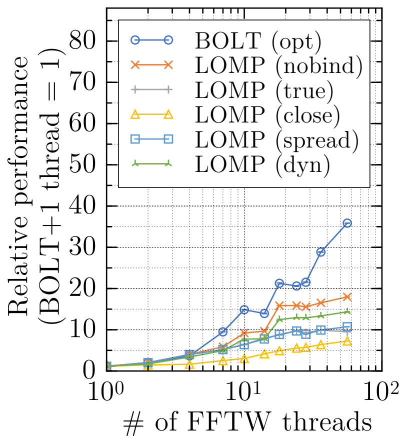
(c) 128 atoms + 16 MPI processes



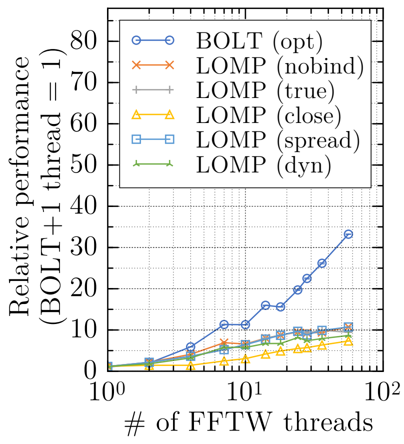
(d) 64 atoms + 32 MPI processes



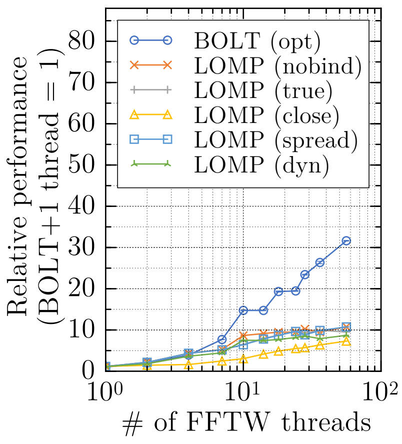
(e) 96 atoms + 32 MPI processes



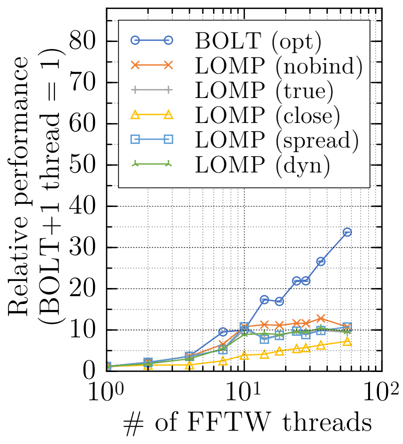
(f) 128 atoms + 32 MPI processes



(g) 64 atoms + 48 MPI processes

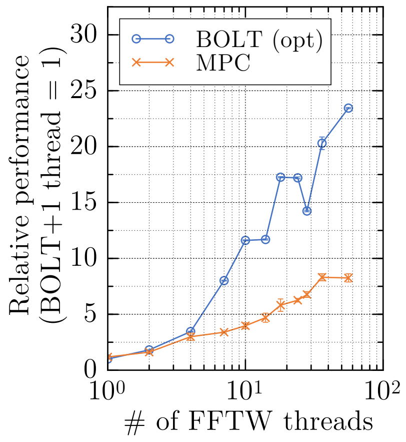


(h) 96 atoms + 48 MPI processes

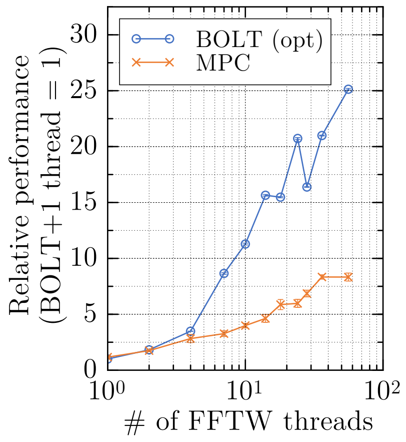


(i) 128 atoms + 48 MPI processes

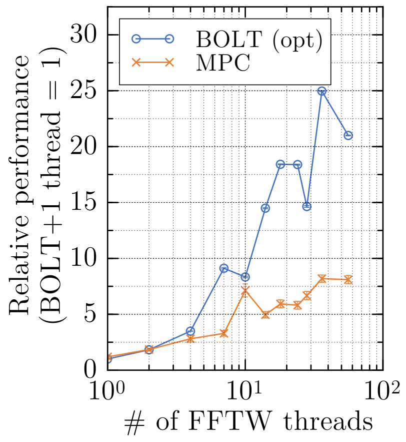
Figure 5.12: Performance of the 3D FFT kernel in Qbox with a 3D FFTW plan compiled with Clang/LLVM.



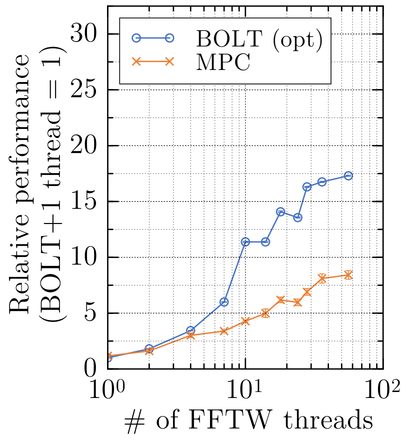
(a) 64 atoms + 16 MPI processes



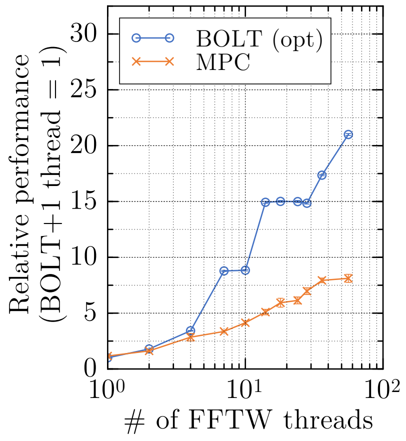
(b) 96 atoms + 16 MPI processes



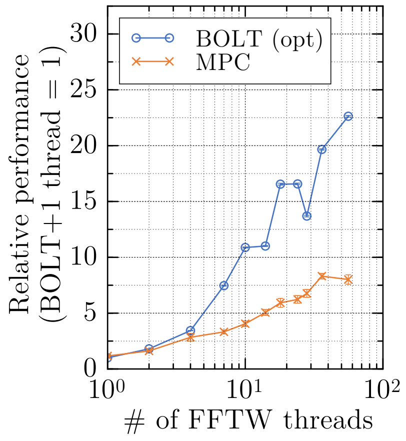
(c) 128 atoms + 16 MPI processes



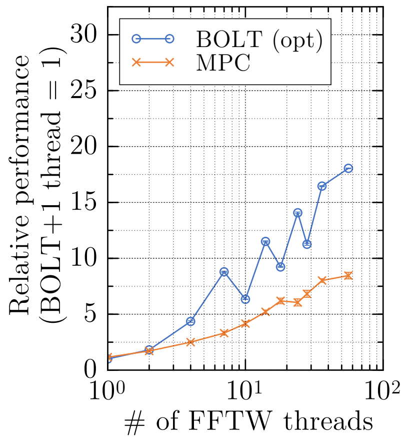
(d) 64 atoms + 32 MPI processes



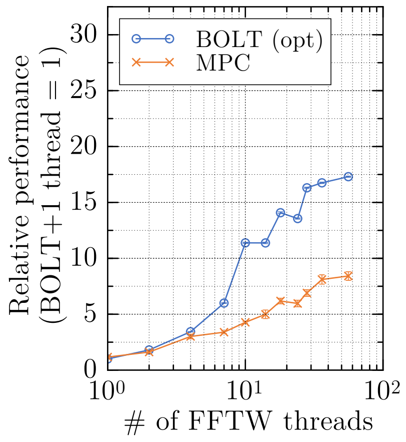
(e) 96 atoms + 32 MPI processes



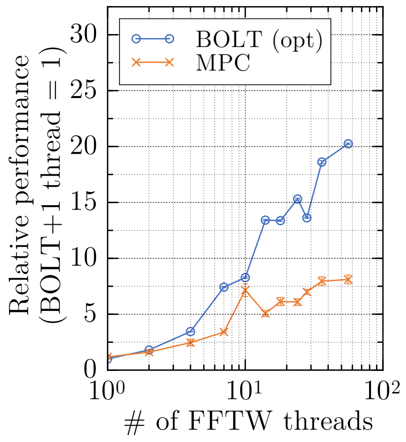
(f) 128 atoms + 32 MPI processes



(g) 64 atoms + 48 MPI processes

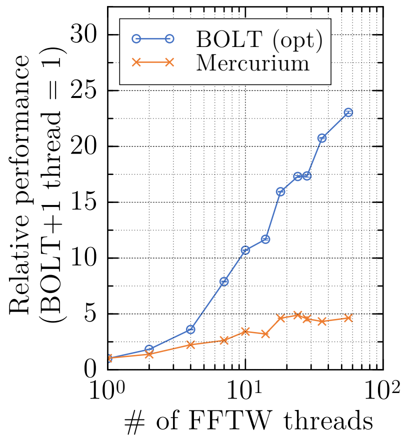


(h) 96 atoms + 48 MPI processes

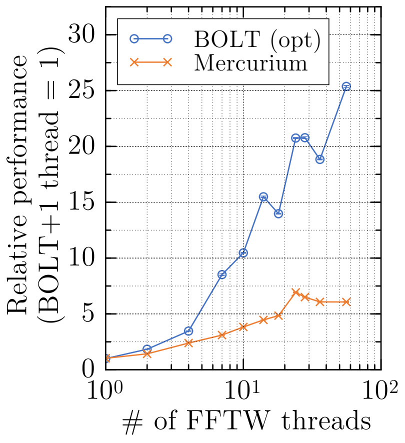


(i) 128 atoms + 48 MPI processes

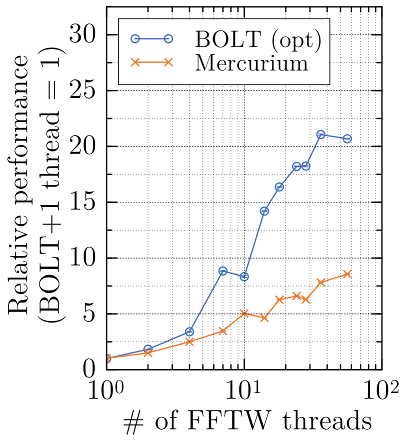
Figure 5.13: Performance of the 3D FFT kernel in Qbox with a 3D FFTW plan compiled with the MPC compiler. Since MPC 3.3.0 is based on GCC 6.2, we used GCC 6.2 for BOLT for a fair comparison.



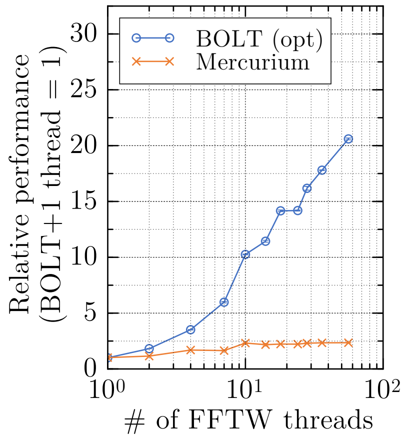
(a) 64 atoms + 16 MPI processes



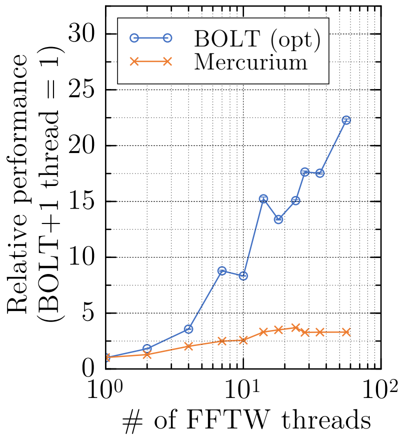
(b) 96 atoms + 16 MPI processes



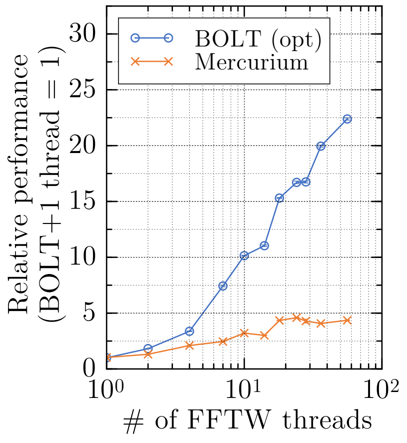
(c) 128 atoms + 16 MPI processes



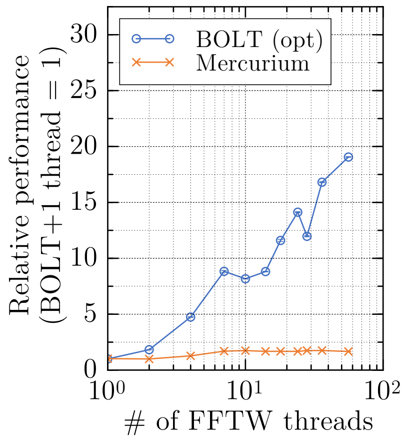
(d) 64 atoms + 32 MPI processes



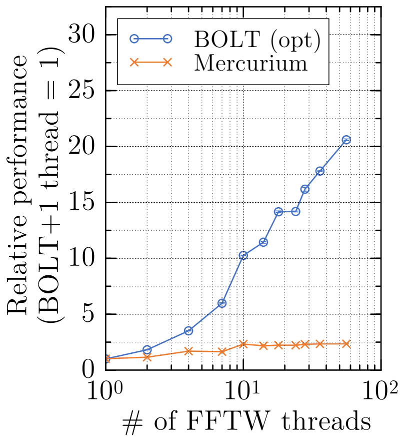
(e) 96 atoms + 32 MPI processes



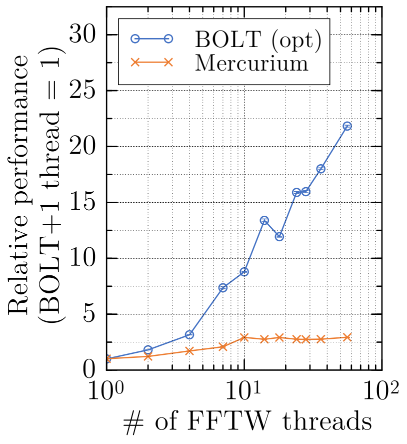
(f) 128 atoms + 32 MPI processes



(g) 64 atoms + 48 MPI processes



(h) 96 atoms + 48 MPI processes



(i) 128 atoms + 48 MPI processes

Figure 5.14: Performance of the 3D FFT kernel in Qbox with a 3D FFTW plan compiled with the Mercurium compiler. Since Mercurium used GCC 4.8.5 as a backend compiler, we used the same version of GCC for BOLT for a fair comparison.

5.3 Evaluation of Lightweight Threading Techniques in BOLT

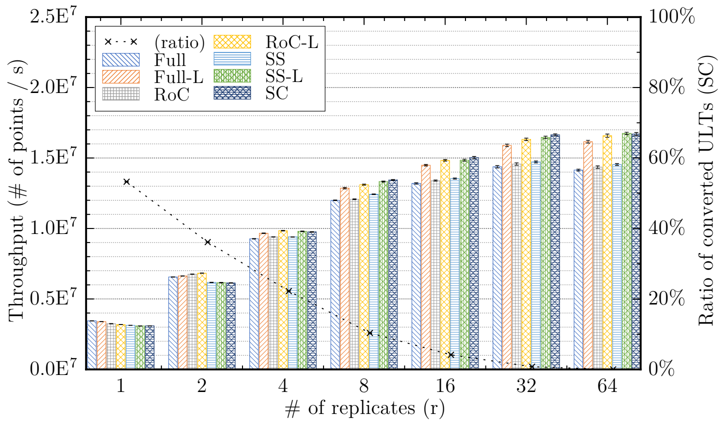
In the last section of this chapter, we evaluate the effect of choosing optimal user-level threading techniques in BOLT. To evaluate the performance of BOLT, we used an OpenMP-parallelized KMeans.

5.3.1 OpenMP-Parallelized KMeans

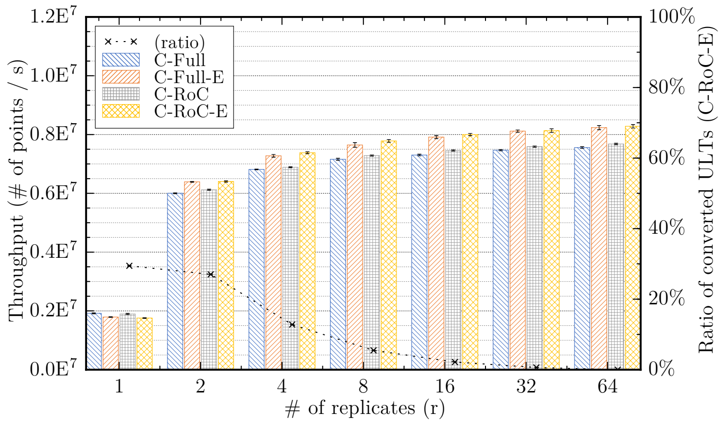
For multithreading, OpenMP offers threads and tasks as parallel work units. OpenMP threads are suspendable parallel units since they need to support several thread-thread synchronization operations such as `barrier` and `single`. OpenMP tasks are not required to be suspendable, while the specification provides an interface to hint at runtime about scheduling. Specifically, according to the standard `taskyield` creates a user-defined scheduling point, which can be implemented as no operation while several studies pointed out the usefulness of suspendable tasks [23, 146]. In this evaluation, we assume tasks yield at `taskyield`, so they were created as ULTs in BOLT. However, not all OpenMP threads and tasks encounter deviations in real programs (e.g., no task scheduling during execution). The dynamic promotion techniques are expected to improve performance when deviations rarely happen. We used OpenMP-parallelized KMeans for evaluation.

KMeans is a machine learning algorithm that partitions N data points into K clusters. Our benchmark is based on the KMeans implementation found in NU-MineBench [125]. In the KMeans algorithm, a point is considered belonging to a cluster with the nearest center. The algorithm first randomly distributes each center of K clusters and repeats updating the cluster centers to the centroids of their points until the positions of the centers get stable enough. The computation of the new centroids is parallelized by a simple method adopted by Chabbi et al. [46]; in our benchmark, each of N ULTs is associated with a data point and updates the partial sum of the centroid of the nearest cluster. At the end of an iteration a master ULT sums up the partial results. The partial sums are shared among workers, so the updates are protected by locks to avoid data race.

To control the lock granularity, we artificially change the number of replications per cluster, which we denote by r. When $r = 1$, each cluster has one partial sum protected by a corresponding lock, so any attempt to update the partial sum of the same cluster incurs lock contention. Creating multiple partial sums increases the reduction cost at the end of iterations but alleviates contention. When $r > 1$, every cluster has r partial sums each of which is accessed by only r/W workers, where W is the number of workers. Accordingly, no contention occurs if $r = W$.



(a) Parent-first scheduling



(b) Child-first scheduling

Figure 5.15: Throughput of KMeans using 64 cores. The ratio of converted ULTs is calculated by dividing the number of promoted ULTs by the number of created ULTs during execution. We obtain these results with **SC** for parent-first scheduling and **C-RoC-E** for child-first scheduling. Other dynamic promotion techniques show similar results.

Table 5.5: Experimental environment of KMeans

Processor	Intel Xeon Phi 7210	Architecture	Knights Landing
Frequency	1.3 GHz	# of sockets	1
# of cores	64	# of HWTs	128
Memory	198 GB	OS	Red Hat 7.5
Compilers	Intel Compiler 17.2.174		

The kernel of the original KMeans was parallelized with OpenMP by a nested parallel loop; the outer loop creates W OpenMP threads each of which spawns N/W tasks in the inner loop. We used KNL for the evaluation presented in Table 5.5, so W was set to 64 in this benchmark. We built the program with Intel compilers, while we needed to apply manual vectorization to the compute kernel to exploit SIMD units in KNL. We used the first 10% data of KDD Cup 1999 [109]; our experiment classifies $N = 5.0 \times 10^5$ points, each of which has 41 floating-point features,[2] into $K = 24$ clusters as instructed by the original problem statement. We changed r from 1 to 64 and measured the performance with different ULT types.

To exploit better locality, we set the OpenMP's `close` affinity for the parent-first threading techniques. The affinity of ULTs can be implemented by limiting the access of a specific pool; as we have seen, BOLT implements the OpenMP's affinity by limiting ULTs associated with OpenMP threads to be scheduled by a specific worker associated with a specific core. Although this strategy works well in a parent-first case, such an affinity setting inhibits dynamic load balancing in a child-first case. Consider a case where r is 64 and no deviation happens in innermost ULTs (an inner OpenMP tasks). Under the child-first scheduling policy, not a child ULT but a parent ULT (i.e., an outer OpenMP thread) is made stealable. Because of the affinity setting, however, a parent ULT cannot be scheduled other than by a specific worker, disabling dynamic load balancing across workers. Thus, we disabled the affinity setting for the child-first threading techniques.

Figure 5.15 shows the average throughputs of 64 executions each of which repeats the KMeans algorithm five times after a warm-up (one execution). An increase in replicates alleviates lock contention and reduces the deviation probability, elevating overall performance. At a larger r, LSA and ESR (**-L** and **-E**) enhances performance. Reducing context-switching overheads (**SC**, **SS-L**, **RoC-L**, and **C-RoC-E**) further improves throughputs. With fewer replicates, fully fledged techniques perform better, but the absolute performance is worse because of significant lock contention. The results show that the dynamic promotion techniques speed up programs if deviations happen infrequently, whereas the threading overheads often get negligible when deviations are frequent because the causes of deviations become the performance bottleneck. We also note that although the dynamic promotion methods improve performance with both parent- and child-first scheduling policies, the KMeans algorithm favors parent-first scheduling because it suits the OpenMP's affinity setting.

[2]We arbitrarily mapped string-typed values to floating-point values.

6 Practicality of Argobots and BOLT

The thesis mainly discusses performance of Argobots and BOLT since performance is the most important metric for high-performance computing. In practice, however, software maturity is also extremely important to let people outside the project use our research products and impact the real world. This chapter briefly discusses Argobots and BOLT from a practicality perspective. We emphasize that, since both are open-source projects, these runtime systems have been and are developed by not only the author but also numerous collaborators and anonymous users.

6.1 Suitability of Argobots for Mainstream Use

Although countless lightweight threading libraries have been proposed, many research products suffer from several problems caused by lack of engineering efforts such as ill-written documents, limited supported architectures, and poor interoperability in addition to compile-time errors and run-time bugs. The Argobots project has been developed with awareness of these problems commonly seen in research projects.

First and foremost, supporting various environments is crucial since computing environments are getting more and more diverse. For example, majority of personal laptops and enterprise servers use Intel processors, while state-of-the-art supercomputers use non-Intel architectures; it has been announced that Fugaku, which will be the top Japanese supercomputer, will employ ARM-based processors while the Summit supercomputer at Oak Ridge National Laboratory adopted IBM POWER9. Different environments have different compilers; GCC and Clang are commonly used on personal laptops, while commercial compilers such as Intel Compilers and IBM XLC Compilers can keenly optimize programs for specific architectures. Supporting various environments is often challenging because of differences of compiler's behaviors and instruction sets. Unfortunately, there is no silver bullet to support all the architectures with ease; Argobots tries to avoid using platform-dependent features and language extensions. Argobots includes context-switching codes that are written in assembly languages and thus highly depends on compiler and architecture, so they are maintained for popular architectures. The Argobots project adopted continuation integration that tests all the changes with several compilers including GCC (versions 4.8, 6.5, and 8.3), Clang (versions 3.4, 7.0) and ICC 19 with several sets of optimization flags. Besides, Argobots is occasionally tested on architectures other than 64-bit Intel machines with their associated compilers in addition to open-source compilers; the environments include 64-bit ARM, POWER 8 and POWER 9, and 32-bit Intel architecture. This effort is necessary to allow developers to use Argobots on various machines in a portable manner. Since the compilation can be often a barrier to use a software package, in addition to a configure-and-make option, we provide a Spack package [74] for developers who want to try Argobots and users who do not directly use Argobots but their applications depend on Argobots and thus need to install Argobots.

Lack of documents and sample programs is commonly seen in research products, discouraging users from using these software packages. We carefully maintain them so that users can easily access necessary information to start programming with Argobots. Another effort is holding a tutorial to educate how to parallelize programs with lightweight threads. The project also has its own website, GitHub, and mailing list to announce and discuss important topics and issues.

Interoperability plays a key role in running a complicated program that contains several software stacks. Since all the environments by default support their OS-level threads, most software components assume OS-level threads as "threads" at present. Lack of interoperability with other runtime libraries has been a traditional issue of user-level threading libraries. We found several proposals on interoperability layers for lightweight threading models [45, 93], but in our understanding, no attempt succeeded; there exists no generic threading layer that is widely used by other runtime systems. The approach taken by the Argobots project is collaborating with other runtime developers to implement an interoperability layer in each of them. There exists ongoing work including Cilk [32], OmpSs [60], PaRSEC [36], and XcalableMP [113], which try to support Argobots as "threads". In this thesis, nonetheless, we focus on the two parallel programming models: OpenMP and MPI.

6.1.1 Interoperability with OpenMP

OpenMP [130] is a simple programming extension for multithreading and successfully parallelizes numerous softwares. As OpenMP is supported and optimized by most research and production compilers, not only applications but also high-level parallel programming models such as C++ executor and Kokkos [43] and math libraries such as FFTW3 [70] Open BLAS [128], ATLAS [172], Intel MKL [100], Intel MKL-DNN [101], Eigen [65], and SLATE [76] use OpenMP as a backend multithreading library. Interoperability of each multithreading layer highly impacts the overall performance since contemporary software contains multiple software components to improve modularity, maintainability, and portability.

In Chapter 4 we discuss the interoperability of multiple OpenMP-parallelized components. BOLT solves the oversubscription issue by using the lightweight Argobots ULTs. Our evaluation shows that BOLT can exploit fine-grained parallelism introduced by multiple parallel software packages. Although we successfully showed that BOLT can achieve high composability **within** the context of OpenMP, one of the performance concerns is mixing multiple multithreading programming models. For example, using an OpenMP-parallelized math library in an Argobots-parallelized program might cause a conflict of resource management since these two software packages are unaware of each other. Better scheduling across multiple independent parallel libraries is our future work.

6.1.2 Interoperability with MPI

Nowadays, node-level parallelism is indispensable to achieving absolutely high performance since the increase of CPU frequency has no longer been sustained. MPI [121] is the de facto standard programming model for internode parallelism. In addition to process-level parallelism, MPI supports hybrid parallelism called MPI+Thread to allow multiple threads to asynchronously call MPI functions. Most MPI runtimes including MPICH [123], Open MPI [73, 80, 81], and MVAPICH [123] assume OS-level threads as "Thread" in MPI+Thread, however. Hence, programs multithreaded by ULTs might cause a data race or a deadlock, which has been one of the major barriers to prevent a mainstream use of ULTs in the field of high-performance computing. To address this issue, we have implemented and are implementing an abstracted threading layer for Argobots in these MPI runtimes so that users can choose Argobots threads as "Threads" on compilation.

Specifically, MPICH [123], which is one of the most widely used implementations of MPI runtimes, is an MPI implementation we used for the evaluation of distributed Graph500 described in Section 5.1.2. MPICH has an abstracted threading layer to support several threading packages including Argobots in addition to OS-level threads such as POSIX threads [95], Solaris threads [154], and Windows threads. Unlike OS-level threads, ULTs do not support cooperative preemption, so MPICH is being developed so that a thread explicitly yields in all the possible busy loops. This change has been merged to the master branch so anyone who wants to use lightweight ULTs over MPI can try it. With researchers at Sandia and Los Alamos National Laboratories, we are also working on introducing the thread interoperability layer in Open MPI [73], which should widen the applicability of lightweight ULTs on supercomputers.

Using lightweight threads over MPI runtimes does not only provide the convenience for ULT users, but also is a promising direction to alleviate lock contentions in MPI runtime systems. Because a multithreaded MPI performs poorly in general, existing work has aimed at improving the thread-safety mechanism in MPI with OS-level threads [15, 16, 18, 57]. However, ULTs that support lightweight context switching and flexible user-level scheduling have a potential to dramatically improve performance especially when the communication is fine grained [72, 117]. Investigating this direction is also our future work.

6.1.3 Summary

As we discussed in this section, Argobots has been and is being developed to maintain high software quality. We collaborate with other projects with respect to interoperability. The software maturity of Argobots has been highly valued and thus it is used in several projects in both academia and industry; Argobots has been used by not only research projects (e.g., Margo [119], a wrapper for the Mercury library [150]), but also some industrial projects such as Intel DAOS [37]. However, several areas especially regarding interoperability have not been fully explored as we mentioned, which is our primary future work.

6.2 Suitability of BOLT for Mainstream Use

Chapter 4 shows that BOLT can exploit fine-grained parallelism introduced by unintentional nested parallel regions without hurting performance of a flat parallel region. Because most popular OpenMP runtime systems utilize OS-level threads as OpenMP threads, however, one might doubt the suitability of BOLT for mainstream use from specification and practicality perspectives. As far as we examined, BOLT does not break the current OpenMP 4.5 specification. In reality, however, BOLT fails to run some programs which assume an OS-level thread as a native thread implementation and thus sometimes requires workarounds. This supplemental section details these issues.

6.2.1 Standard Compliance of BOLT

Because OS-level threads are dominantly used for OpenMP threads and existing ULT-based OpenMP runtimes stagnate at the old OpenMP specification (to the best of our knowledge, the newest is OpenMP 3.1), the current specification might implicitly or explicitly assume OS-level threads as OpenMP threads. As far as we examined the standard, BOLT successfully follows the OpenMP 4.5 semantics [129]. Specifically, BOLT complies with the OpenMP specification with respect to its use of ULTs since the OpenMP standard defines a *thread* as an execution entity with a stack and thread local storage, which an Argobots ULT satisfies. BOLT integrates new OpenMP features introduced in OpenMP 4.0 and 4.5 (e.g., data-dependent task and accelerator offloading) without breaking the standard. The functionality of BOLT has been verified through various test suites with GCC, Intel, and Clang/LLVM compilers. Although the current standard requires some extensions to fully utilize underlying lightweight ULTs as described in this thesis, we believe that BOLT is a standard-compliant implementation of OpenMP 4.5. We also believe that BOLT can follow the latest OpenMP 5.0 [130] specification as well. Supporting OpenMP 5.0 is our future work.

6.2.2 Suggested OpenMP Changes

Despite the compliance of the OpenMP specification, we proposed several extensions to fully utilize lightweight threads in BOLT. The first extension is support of **unset** as a keyword of `bind-var`, which clears `bind-var` and `place-partition-var`. Since **unset** allows dynamic load balancing across cores (i.e., by random-work stealing in BOLT), this keyword allows users to express that dynamic load balancing is preferred for a specific parallel region. Although such dynamic load balancing should be more effective with lightweight threads, this keyword should be beneficial for OS-level thread-based runtimes.

Another suggestion is a wait policy keyword that allows a runtime to decide the best strategy, which we call `auto` and `runtime` in Chapter 4. Although the OpenMP standard does not specify the implementation, the current two keywords, `active` and `passive`, are extremely opposite. If either is set, an OpenMP runtime is urged to assume extreme situations, although, as our evaluation shows, there might exist a better scheduling technique such as `hybrid`. This change would be also beneficial for OS-level thread-based runtimes since some of these systems including GCC and LLVM OpenMP implement an intermediate wait behavior enabled only when a wait policy is not set. With the new keyword, users can explicitly choose this default behavior.

As explained in Section 5.3.1, OpenMP tasks in BOLT yield at `omp taskyield`, while the current OpenMP specification allows the runtime to ignore all scheduling points [23]. In reality, most production OpenMP runtimes perform nothing at `omp taskyield`. One possible definition of a yieldable OpenMP task is a task that guarantees that a task yields to another task at scheduling point if a task encounters scheduling points a finite number of times. Such an implementation should be useful especially for runtime systems that can block (e.g., MPI). For example, Schuchart et al. reported that an OpenMP implementation with yieldable tasks are beneficial for the performance of MPI communications [146]. However, it should require large changes in the existing OpenMP runtimes since they implement OpenMP tasks as stackless threads. Our work on lightweight user-level threading techniques show that some ULT implementations can perform as good as stackless threads; this discovery encourages major OpenMP runtimes that implement tasks as stackless threads to use yieldable ULTs (e.g., **RoC**).

Finally, as suggested in [146], task scheduling significantly impacts the performance of OpenMP tasks. An extension to customizable scheduling for loops (e.g., `parallel for`) has been proposed [24],[1] but customizable scheduling for tasking remains an open question in terms of interface since OpenMP tasks can express more complex parallelism. BOLT allows users to change schedulers at runtime via the Argobots layer while this is not portable as the default scheduler implementation in BOLT is hidden to

[1]Loop chunks created by `omp for` are neither spawned as ULTs nor scheduled as tasks since, conceptually, they are work-shared among OpenMP threads.

users. Devising a portable but powerful interface to control task scheduling is our future work.

6.2.3 Using BOLT in the Real World

Since BOLT correctly implements the specification, it works well with most OpenMP-parallelized programs and libraries without any modification and transparently achieves good performance for both flat and nested parallelism. Because OS-level thread-based runtimes dominate largely in production environments, however, OpenMP users are accustomed to OS-level mapping of OpenMP threads. This situation has led to the emergence of user codes that assume OpenMP threads are always mapped to OS-level threads. We emphasize that this is not an issue of BOLT, while in practice we have to deal with such complexities in order to run complex real-world programs. As briefly mentioned in Section 5.2.1, we sometimes need to work around such assumptions using clever tricks outside the BOLT implementation, but doing so might not always be feasible. This section discusses the two most common user practices we encounter that assume OS-level thread-based implementations.

Synchronization with Non-OpenMP Functions

Some programs assume OpenMP threads are running on OS-level threads and thus implement their own synchronization algorithms with non-OpenMP functions (e.g., a hand-written barrier instead of `omp barrier`). We experienced this case when we ran MKL's `dgemv()` used in Section 5.2.1; `dgemv()` synchronizes OpenMP threads with its own barrier function, not one provided by Intel OpenMP. This barrier function is implemented with busy-waiting calling `sched_yield()`, but ULTs cannot be switched to others by the OS-level scheduling function, so the program hangs if more OpenMP threads than the number of schedulers attempt to perform the barrier. We overcame this issue by overriding these MKL functions by `LD_PRELOAD`; the barrier function was replaced with one using a ULT-aware algorithm. We also observed a similar issue in MKL's `dgemm()`; it also performs a synchronization based on busy-waiting, which may hang with BOLT since ULTs are nonpreemptive and thus a context switch never happens. Note that BOLT works with these applications if not nested or, technically, when the total number of OpenMP threads is not greater than the number of schedulers, since all ULTs are mapped to schedulers, and thus OS-level threads, one by one. However, in oversubscription cases, especially under nested parallelism, BOLT requires modification to user programs.

TLS Access

Some libraries try to create, use, and destroy TLS via the OS-level thread API (e.g., `pthread_specific` or compiler-level TLS), under the assumption of an underlying threading library of OpenMP threads. MKL's `dgemv()` internally accesses TLS with `pthread_getspecific()` to manage some environmental settings. Because ULTs do not individually support Pthreads' TLS, a value accessed via `pthread_-getspecific()` is one of the scheduler, not one of the ULTs. We addressed this issue by replacing a function using Pthreads' TLS with a function returning intended values correctly. Unfortunately, to make matters worse, some compilers, including Clang 7.0, translate OpenMP TLS (i.e., `threadprivate`) into compiler-level TLS. In this case, we need to patch a compiler and recompile programs.

Other Features Specific to OS-Level Threads

In addition to those problems, a signal mask and a CPU mask can be problematic because most ULT implementations including Argobots do not support them, although we have not encountered such situations. We believe that BOLT can run OpenMP-parallelized programs without any modification if the programs properly use OpenMP features.

7 Related Work

The multithreading overhead inhibits good scalability on modern highly parallel computing systems. A tremendous amount of literature has proposed thread-scheduling strategies to mitigate resource contentions, improve locality, and address load imbalance, while the threading overhead fundamentally imposes a penalty on fine-grained decomposition of work. The threading cost prevents programmers from parallelizing work, which caps the possible speedups of multithreaded programs [63]. Moreover, such coarse-grained parallelization limits the scheduling flexibility as the number of schedulable threads is decreased, rendering scheduling optimizations insignificant. The efficient execution of programs that contain fine-grained parallelism has been a traditional problem and thus numerous studies exist.

One of the research directions to tackle threading overheads in fine-grained parallelism is letting programmers write fine-grained multithreaded programs and compilers and runtime systems make threads coarse grained. Several studies proposed static techniques using compile-time information [10, 106], dynamic techniques using runtime systems [31, 61, 169], hybrid techniques that combine static analysis and a run-time adaptive approach [147, 162, 171], or using an autotuning strategy [20, 105] or a machine-learning technique [141] to choose the best granularity parameters. Importantly, these techniques are orthogonal to our optimizations of threading overheads; more lightweight threads allow these techniques to leave threads fine grained, reducing risks of adverse serialization and providing more flexibility to scheduling algorithms to achieve better dynamic load balancing, locality, and resource management.

This chapter discusses precedent work of lightweight threads from viewpoints of a user-level threading library (Section 7.1) and parallel runtime systems including OpenMP implementations (Section 7.2). As we discussed previously, because OS-level threads are heavyweight due to system calls, lightweight threads are mostly implemented in the user space. The following sections explain lightweight user-level threading libraries and parallel systems using ULTs.

7.1 User-Level Threads

Although numerous parallel systems have adopted ULTs as an implementation of lightweight parallel units, the focus of the past papers on ULT-based systems is not a threading technique but other components such as programmability, usability, portability, abstraction, and other performance optimizations such as scheduling and task queue implementations. The performance comparison between these parallel systems [137, 139] only measured the overall performance and fundamentally lacked a detailed performance analysis of threading techniques. Another existing direction was a survey on functionalities of multithreading and multitasking frameworks [161], while this work argues only the interface, not the implementation. This section describes notable studies out of countless research on ULTs from the aspect of user-level threading techniques.

7.1.1 Fully Fledged Threads

Fully fledged threads are widely used to implement ULTs with full threading capabilities. For example, Qthreads [173], Converse [108], MassiveThreads [124], and Argobots [1] are well-known threading libraries that use fully fledged threads. Their stack management policies are different, however. For example, Converse 6.9.0, MassiveThreads 1.00, and Argobots 1.0rc2 (the latest stable version) employ a parent-first fully fledged ULT without LSA while Qthreads 1.15 and Nanos++ 0.15 implement it with LSA. MassiveThreads 1.00 also supports a child-first thread, which is implemented with ESR. The trade-off disclosed by this paper would be helpful for these runtimes to choose the best thread implementation based on their assuming workloads. Evaluating the performance impact of choosing the optimal technique in these libraries is lifted as our future work.

7.1.2 Saving Registers

Our techniques do not include techniques that do not change stacks but only save (callee-) registers. Nevertheless, some previous studies including LazyThreads [79], StackThreads/MP [157, 158], and Fibril [176], proposed such techniques for child-first scheduling. Cilk 1.0-3 over Tapir/LLVM [144] is also a child-first multitasking framework that adopts this idea. As we explained in Section 3.6.1, however, these techniques require compiler modifications or special assumptions on program executions, so they cannot be a building block of a generic threading library. We thus did not evaluate these methods.

7.1.3 Stack Separation

Some studies have proposed methods that omit register manipulations but change only the stacks. Their approaches are different from ours in that a thread invocation function adopts a special calling convention that only marks registers for a stack pointer and an instruction address as callee-saved (e.g., Intel CilkPlus [99]). This approach can be seen as a technique that utilizes a calling convention to save all the necessary registers (including registers marked as callee-saved in widely adopted ABIs). This approach requires patching a compiler to recognize a custom calling convention, whereas neither **SS** nor **SS-L** requires compiler modification.

7.1.4 Scheduler Creation

A few parallel systems have adopted the scheduler creation technique. Chores [62] and Wool [68] are parent-first threading libraries that utilized this method to reduce threading overheads. Concurrent Cilk [180] is a child-first threading library that adopted this technique to implement a yield feature in Intel CilkPlus [115]. The past work, however, solely implemented the scheduler creation technique and thus lacked performance comparison and analysis of programming constraints. We also note that their approaches specially handle ULTs that encountered a deviation, so promoted ULTs are differently scheduled from unpromoted ULTs. Our techniques uniformly schedule all ULTs including promoted **SC** threads.

7.1.5 Stackless Threads

Numerous runtime systems including Filaments [116], Qthreads [173],[1] and Argobots [1] support a run-to-completion thread in order to eliminate all the cost associated with user-level context switch. OpenMP *task* implementations found in the popular OpenMP runtimes [102, 126, 132] and `task` in Intel TBB [143] are essentially classified as stackless threads but not "run to completion" in a narrow sense because they can wait for the completion of children.

In general, stackless threads are lightweight and easy to implement, but its constraint significantly limits the applicability because it cannot perform a context switch at an arbitrary point. Several papers have argued yieldable threads in non-yieldable threading packages from performance and programmability perspectives. For example, Zakian et al. [180] showed that a yield operation in Cilk [71] enables efficient blocking communication and synchronization while several papers on OpenMP [23, 146] reported the same benefits of yieldable OpenMP tasks. Graph500 in our evaluation is a good example that a stackless thread cannot execute; removing a yield operation from a polling loop in the MPI runtime might cause a deadlock.

7.1.6 Other Threading Techniques

Several threading techniques cannot be classified into the categories above. Sivaramakrishnan et al. [148] proposed MultiMLton, which allows relocation of function stacks. This technique might be applicable to functional languages, but it can hardly support C/C++ programs. Cilk-M [112] enables stack copying

[1]Qthreads executes a thread on top of the scheduler's stack when `QTHREAD_SPAWN_SIMPLE` is specified. No blocking operation (i.e., `qthread_yield()`) inside threads is allowed.

by modifying OS to expose the same address space so that a pointer reference to a call stack is valid after copying a stack. This technique requires OS modification. Tascell [88] is a compiler-based technique adopting a lazy task creation policy. This technique invokes threads in a sequential manner and lazily creates logical threads if necessary by backtracking call stacks. Acar et al. [9] propose a threading technique that lazily creates parallel threads at a *heartbeat*. This method requires the cactus stack implementation [79], which breaks the interoperability with precompiled libraries and thus is not suitable for a generic threading library.

7.2 Parallel Programming Models Using ULTs

7.2.1 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 [118] and Omni/ST [155] 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 [86] adopted a hierarchical scheduling algorithm to execute innermost threads on close cores, in order to improve locality [87]. ForestGOMP [39], an OpenMP runtime library over a lightweight threading library called Marcel [160], adopted a BubbleSched scheduler, which is a NUMA-aware hierarchical scheduling policy based on hardware locality information [38]. 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. For example, libKOMP [40], OpenMP over Qthreads [127], and OmpSs [60] utilized ULTs for efficient task parallelism, although their mapping of OpenMP threads is 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 [27], keeps an OpenMP-compatible option. Several researchers tackled the interoperability issue. The developers of MPC [135] integrated their ULT-based OpenMP runtime with their MPI implementation. OpenMPIR [153] is an extension of an LLVM intermediate language [111] for OpenMP to optimize OpenMP codes with the Tapir backend [144]. Programs compiled with OpenMPIR internally ULTs, while this study covers limited OpenMP features because of the poor expressiveness of Tapir. Some work mapped OpenMP threads to ULTs for distributed programming models, for example, OpenMP over Charm++ [25, 26], 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 the existing ULT-based systems to improve the performance of most OpenMP applications since many real-world OpenMP programs are parallelized by parallel regions.

7.2.2 ULTs in Parallel Programming Models

Several high-level parallel programming models, such as Cilk [71], CilkPlus [115], Charm++ [107], and X10 [50], 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.

7.2.3 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 [134] and DoPE [142], 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 [163, 175]. Our work focuses solely on the interoperability issue within OpenMP.

7.2.4 Mitigating Overheads of Nested Parallel Regions

Oversubscription of threads is a traditional performance issue and has been intensively studied [56, 94]. The OpenMP specification offers several interfaces to avoid the oversubscription as we described. In the context of OpenMP, this issue is specifically critical when parallel regions are nested. Several studies, therefore, have tried to mitigate the cost of nested parallel regions by using OS-level threads. ForestGOMP [39] and Intel OpenMP [164] store threading resources of nested parallel regions so that succeeding nested parallel regions can be efficiently created. LLVM OpenMP employs this mechanism as a nested hot team, which improves the performance of both BOLT and LLVM OpenMP. When OpenMP threads are mapped to OS-level threads, caching OpenMP threads results in wasting OS resources, so aggressive caching is harmful when the total number of threads is limited. The hot team optimization can be applied safely in BOLT, however, since caching ULTs increases only the memory footprint.

Even if fork-join overheads are mitigated, however, waking up OS-level threads is costly. To reduce this cost, several OpenMP implementations have options to keep threads running for a certain duration after finishing threads. Our evaluation tweaked such values (e.g., `GOMP_SPINCOUNT` and `KMP_BLOCKTIME`) for the benefits of existing libraries. Unfortunately, no automatic way exists to determine the optimal value without running it on a real machine. BOLT users do not need to tweak this parameter since the context-switching cost of ULTs is cheaper than that of OS-level threads.

8 Conclusions and Future Work

As the number of cores per processor is increasing, multithreading becomes essential to exploiting modern processors. Overheads in thread implementations become increasingly important to efficiently run complex fine-grained parallel programs on highly parallel systems. However, widely adopted thread implementations adopt either (1) a lightweight stackless thread for performance-sensitive programs or (2) an OS-level thread out of fear of missing capabilities required by multithreading programming models. As a result, some runtimes use stackless threads with unnecessarily restricted threading features and lose the potential scheduling points, narrowing scheduling flexibility and expressiveness of programs. Other runtimes rely on fully capable but heavyweight OS-level threads and lose parallelism, leading underutilization of cores.

We pose a question to this prevailing idea and suggest a ULT for an efficient and capable thread implementation, which is as fast as a stackless thread but supports most critical threading features such as suspension as an OS-level thread employs. This work first shows a lightweight threading library, Argobots, and several user-level threading techniques to explore a ULT that can be as fast as a stackless thread with a context-switching capability. Our comprehensive analysis of possible implementations of ULTs covers both parent- and child-first scheduling, revealing more than ten user-level threading techniques each of which has a different trade-off between performance and functionality. Specifically, **RoC-L** and **C-RoC-E** are ones that perform best in most cases: their fork-join overheads are close to that of stackless threads while they have the same functionality that widely used fully fledged ULTs have.

We adopted ULTs to one of the most widely used multithreading programming models, OpenMP. Based on our highly optimized user-level threading library, we build a lightweight OpenMP runtime, called BOLT. Our finding is that, although a ULT can be easily substituted for a thread abstraction in OpenMP, several optimizations are required to fully unleash ULTs. With our in-depth performance analysis, we applied several optimizations such as scalable resource management, ULT friendly affinity implementation, and a thread coordination algorithm, all of which are necessary to attain high performance for both flat and nested parallelism regardless of the level of oversubscription. BOLT showed unprecedented performance if parallelism is nested by creating lightweight ULTs as OpenMP threads, while transparently achieving the best performance if parallelism is flat.

Our resulting software products, Argobots and BOLT, are publicly available and supported by numerous contributors. Most of the optimizations explained in this work have been already merged to the master branches of Argobots and BOLT after minor updates to make them production ready. Particularly, dynamic promotion techniques have been merged and were included in Argobots from version 1.0b1 whereas BOLT contains most optimizations explained in this thesis from version 1.0rc2.

Our quest is a comprehensive understanding of designs and implementations of lightweight threading libraries. This thesis mainly investigates the fork-join performance of threads in Argobots and OpenMP. Arguably, other factors including schedulers and thread pools are known to highly affect the overall performance. Argobots can take the advantage of its low-level design; because schedulers and thread pools are designed as customizable, changing these components is relatively easy. Investigating their design and performance in Argobots by developing efficient scheduling policies (e.g., implementing [8] and [83]) and in BOLT by enhancing scheduling in OpenMP for data-flow tasking [136] is our future work.

We cannot ignore a viewpoint of software quality assurance. Although Argobots and BOLT are rigorously tested with their continuous integration process and checked with several compilers and machines, these tests are rather ad-hoc and not thorough. BOLT passes most tests bundled with LLVM OpenMP, while using other several OpenMP validation and verification testsuites[58, 59, 170] for BOLT should be naturally our next step. In addition to further engineering efforts for empirically advanced testing, formal verification on threading libraries (such as model checking for Qthreads [66]) is a futuristic direction.

Standard compliance is essential to BOLT since it should work with major compilers including GCC, Intel Compilers, and Clang. In order to fully support OpenMP 5.0, the implementation of OpenMP Tool and Debug interface [64] would be challenging, which is lifted as part of our future work. Contributing the OpenMP community from the perspective of a ULT-based OpenMP runtime is as important as making BOLT standard compliant by implementing new OpenMP features. For example, a keyword `unset` for OpenMP affinity should be useful for not only ULT-based implementation but also traditional OS-level thread-based runtimes. Attending the OpenMP language committee and explaining the necessity of these changes is time-consuming but important.

Last but not least, we need to establish an ecosystem to build a software stack that is fully aware of ULTs. To utilize ULTs, we often require code changes in other software pieces; for example, as we have seen, ULT support for MPI requires code changes in MPI implementations. Initial ULT support has been merged into the MPICH master [123], while the effort to Open MPI [73] is ongoing. Similar efforts are required for other communication layers such as OpenSHMEM [49] and GASNet [34]. If we look at a non-HPC application, machine learning becomes an increasingly important workload, so trying ULTs in a deep-learning framework would be promising. Since real deep-learning frameworks are extremely complex, oversubscription is easily caused by highly stacked software components. For example, Tensorflow [6] utilizes CPU resources by using Eigen’s thread pools [65], Intel MKL [100], and Intel MKL-DNN [101]. Investigating this direction is one of our most important future plans.

List of Publications

[1] 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*, PP(99):512–526, Oct. 2017.

[2] S. Iwasaki, A. Amer, K. Taura, and P. Balaji. Lessons learned from analyzing dynamic promotion for user-level threading. In *Proceedings of the 2018 IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 23:1–23:12, Nov. 2018.

[3] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji. BOLT: Optimizing OpenMP parallel regions with user-level threads. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*, PACT '19, pages 29–42, Sept. 2019. (**Best Paper Award at PACT '19**)

[4] S. Iwasaki, A. Amer, K. Taura, and P. Balaji. Optimistic Threading Techniques for MPI+ULT (Poster). In *the 24th European MPI Users' Group Symposium*, EuroMPI '17, Sept. 2017.

References

[5] 64-Bit ELF V2 ABI Specification Power Architecture Workgroup Specification Revision 1.4. `http://openpowerfoundation.org/wp-content/uploads/resources/leabi/leabi-20170510.pdf`, May 2017.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vigas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[7] U. A. Acar, V. Aksenov, A. Charguéraud, and M. Rainey. Provably and practically efficient granularity control. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 214–228, Washington, D.C., USA, Feb. 2019.

[8] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12, Bar Harbor, Maine, USA, July 2000.

[9] U. A. Acar, A. Charguéraud, A. Guatto, M. Rainey, and F. Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, pages 769–782, Philadelphia, Pennsylvania, USA, June 2018.

[10] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle Scheduling: Controlling granularity in implicitly parallel languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 499–518, Portland, Oregon, USA, Oct. 2011.

[11] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Monterey, California, USA, June 2002.

[12] 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*, IWOMP '12, pages 210–222, Rome, Italy, June 2012.

[13] A. Aiken, M. Bauer, and S. Treichler. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceeding of the 23rd International Conference on Parallel Architecture and Compilation Techniques*, PACT '14, pages 263–275, Edmonton, Alberta, Canada, Aug. 2014.

[14] S. Akiyama and K. Taura. Uni-Address Threads: Scalable thread management for RDMA-based work stealing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 15–26, Portland, Oregon, USA, June 2015.

[15] A. Amer, C. Archer, M. Blocksome, C. Cao, M. Chuvelev, H. Fujita, M. Garzaran, Y. Guo, J. R. Hammond, S. Iwasaki, K. Raffenetti, M. Shiryaev, M. Si, K. Taura, S. Thapaliya, and P. Balaji. Software combining to mitigate multithreaded MPI contention. In *Proceedings of the 33rd ACM International Conference on Supercomputing*, ICS '19, pages 367–379, Phoenix, Arizona, USA, June 2019.

[16] A. Amer, H. Lu, P. Balaji, M. Chabbi, Y. Wei, J. Hammond, and S. Matsuoka. Lock contention management in multithreaded MPI. *ACM Transactions on Parallel Computing*, 5(3), Jan. 2019.

[17] A. Amer, H. Lu, P. Balaji, and S. Matsuoka. Characterizing MPI and hybrid MPI+Threads applications at scale: Case study with BFS. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '15, pages 1075–1083, Shenzhen, China, May 2015.

[18] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka. MPI+Threads: Runtime contention and remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 239–248, San Francisco, California, USA, Feb. 2015.

[19] J. Ang, B. Barrett, K. Wheeler, and R. Murphy. Introducing the Graph 500. Cray User Group (CUG), May 2010.

[20] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, Dublin, Ireland, June 2009.

[21] ARM Cortex-A Series Version: 1.0 Programmer's Guide for ARMv8-A, Mar. 2015.

[22] 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*, IWOMP '07, pages 1–12, Beijing, China, June 2007.

[23] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, Mar. 2009.

[24] S. Bak, Y. Guo, P. Balaji, and V. Sarkar. Optimized execution of parallel loops via user-defined scheduling policies. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, Kyoto, Japan, Aug. 2019.

[25] 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*, ESPM2'17, pages 4:1–4:7, Denver, Colorado, USA, Nov. 2017.

[26] S. Bak, H. Menon, S. White, M. Diener, and L. Kale. Multi-level load balancing with an integrated runtime approach. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '18, pages 31–40, Washington, D.C., USA, May 2018.

[27] 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*, EWOMP '04, pages 103–109, Stockholm, Sweden, Oct. 2004.

[28] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Salt Lake City, Utah, USA, Nov. 2012.

[29] D. E. Bernholdt, S. Boehm, G. Bosilca, M. G. Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee. A survey of MPI usage in the US Exascale Computing Project. *Concurrency Computation: Practice and Experience*, Sept. 2018.

[30] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski. OpenMP for accelerators. In *Proceedings of the Sixth International Workshop on OpenMP*, IWOMP '11, pages 108–121, Chicago, Illinois, USA, June 2011.

[31] J. Bi, X. Liao, Y. Zhang, C. Ye, H. Jin, and L. T. Yang. An adaptive task granularity based scheduling for task-centric parallelism. In *Proceedings of the 2014 IEEE International Conference on High Performance Computing and Communications*, HPCC '14, pages 165–172, Paris, France, Aug. 2014.

[32] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, pages 207–216, Santa Barbara, California, USA, July 1995.

[33] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[34] D. Bonachea. GASNet Specification Version 1.8.1, Aug. 2017.

[35] Boost C++ Libraries. `https://www.boost.org/`.

[36] G. Bosilca, B. A., D. A., F. M., H. T., and J. Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *IEEE Computing in Science and Engineering*, 15(6):36–45, Nov. 2013.

[37] M. S. Breitenfeld, N. Fortner, J. Henderson, J. Soumagne, M. Chaarawi, J. Lombardi, and Q. Koziol. DAOS for extreme-scale systems in scientific applications. *arXiv –CoRR*, abs/1712.00423:1–10, Dec. 2017.

[38] 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*, PDP '10, pages 180–186, Pisa, Italy, Feb. 2010.

[39] 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*, 38(5):418–439, Oct. 2010.

[40] 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*, IWOMP '12, pages 102–115, Rome, Italy, June 2012.

[41] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, Aug. 2000.

[42] 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*, SAC '04, pages 220–224, Nicosia, Cyprus, Mar. 2004.

[43] H. Carter Edwards and C. R. Trott. Kokkos: Enabling performance portability across manycore architectures. In *Proceeding of the 2013 Extreme Scaling Workshop*, XSW '13, pages 18–24, Boulder, Colorado, USA, Aug. 2013.

[44] 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*, 84:22–31, July 2018.

[45] 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*, ICPP '17, pages 60–69, Bristol, UK, Aug. 2017.

[46] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High performance locks for multi-level NUMA systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 215–226, San Francisco, California, USA, Feb. 2015.

[47] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, Aug. 2007.

[48] 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*, SPAA '12, pages 182–184, Pittsburgh, Pennsylvania, USA, June 2012.

[49] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, page 2, New York, New York, USA, Oct. 2010.

[50] 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*, OOPSLA '05, pages 519–538, San Diego, California, USA, Oct. 2005.

[51] Q. Chen and M. Guo. Locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures. *ACM Transactions on Architecture and Code Optimization*, 12(2), July 2015.

[52] Q. Chen, M. Guo, and Z. Huang. CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 163172, San Servolo Island, Venice, Italy, June 2012.

[53] Q. Chen, Z. Huang, M. Guo, and J. Zhou. CAB: Cache aware bi-tier task-stealing in multi-socket multi-core architecture. In *Proceedings of the 40th International Conference on Parallel Processing*, ICPP '11, pages 722–732, Taipei, Taiwan, Sept. 2011.

[54] Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`.

[55] Intel® Fortran Compilers — Intel® Software. `https://software.intel.com/en-us/fortran-compilers`.

[56] M. Curtis-Maury, X. Ding, C. D. Antonopoulos, and D. S. Nikolopoulos. An evaluation of OpenMP on current and emerging multithreaded/multicore processors. In *Proceedings of the First International Workshop on OpenMP*, IWOMP '05, pages 133–144, Eugene, Oregon, USA, June 2008.

[57] H.-V. Dang, S. Seo, A. Amer, and P. Balaji. Advanced thread synchronization for multithreaded mpi implementations. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, pages 314–324, Madrid, Spain, May 2017.

[58] J. M. Diaz, K. Friedline, S. Pophale, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran. Analysis of openmp 4.5 offloading in implementations: Correctness and overhead. *Parallel Computing*, 89, Nov. 2019.

[59] J. M. Diaz, S. Pophale, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran. OpenMP 4.5 validation and verification suite for device offload. In *Proceedings of the 13th International Workshop on OpenMP*, IWOMP '18, pages 82–95, Barcelona, Spain, Sept. 2018.

[60] 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*, 21(02):173–193, Mar. 2011.

[61] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '08, pages 36:1–36:11, Austin, Texas, USA, Nov. 2008.

[62] D. L. Eager and J. Jahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Transactions on Computer Systems*, 11(1):1–32, Feb. 1993.

[63] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, Mar. 1989.

[64] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. OMPT: An OpenMP tools application programming interface for performance analysis. In *Proceedings of the Eighth International Workshop on OpenMP*, IWOMP '13, pages 171–185, Canberra, Australia, Sept. 2013.

[65] Eigen. `http://eigen.tuxfamily.org/index.php`.

[66] N. Evans. Verifying Qthreads: Is model checking viable for user level tasking runtimes? In *Proceedings of the 2018 IEEE/ACM Second International Workshop on Software Correctness for HPC Applications*, Correctness '18, pages 25–32, Dallas, Texas, USA, Nov. 2018.

[67] F18 — F18 is a front-end for Fortran intended to replace the existing front-end in the Flang compiler. `https://github.com/flang-compiler/f18`.

[68] K.-F. Faxén. Wool – A work stealing library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, June 2009.

[69] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3 of *ICASSP '98*, pages 1381–1384 vol.3, Seattle, Washington, USA, May 1998.

[70] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb. 2005.

[71] 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*, PLDI '98, pages 212–223, Montreal, Quebec, Canada, June 1998.

[72] T. Fukuoka, W. Endo, and K. Taura. An efficient inter-node communication system with lightweight-thread scheduling. In *Proceedings of the 21st International Conference on High Performance Computing and Communications*, HPCC '19, Zhangjiajie, China, Aug. 2019.

[73] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings of the 11th European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, EuroPVM/MPI '04, pages 97–104, Budapest, Hungary, Sept. 2004.

[74] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack package manager: Bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 15, Austin, Texas, USA, Nov. 2015.

[75] G. R. Gao, T. L. Sterling, R. L. Stevens, M. Hereld, and W. Zhu. ParalleX: A study of a new parallel computation model. In *Proceedings of the First Next Generation Software Workshop*, NGS '12, pages 1–6, Long Beach, California, USA, Mar. 2007.

[76] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra. SLATE: Design of a modern distributed and accelerated linear algebra library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, Denver, Colorado, USA, Nov. 2019.

[77] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`.

[78] P. Ghosh, Y. Yan, D. Eachempati, and B. Chapman. Prototype implementation of OpenMP task dependency support. In *Proceedings of the Eighth International Workshop on OpenMP*, IWOMP '13, pages 128–140, Canberra, Australia, Sept. 2013.

[79] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, Aug. 1996.

[80] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, CLUSTER '06, pages 1–9, Barcelona, Spain, Sept. 2006.

[81] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings of the 6th international conference on Parallel Processing and Applied Mathematics*, PPAM '05, pages 228–239, Poznań, Poland, Sept. 2005.

[82] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '09, pages 1–12, Rome, Italy, May 2009.

[83] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of the 24th IEEE International Symposium on Parallel Distributed Processing*, IPDPS '10, pages 1–12, Atlanta, Georgia, USA, Apr. 2010.

[84] F. Gygi. Architecture of Qbox: A scalable first-principles molecular dynamics code. *IBM Journal of Research and Development*, 52(1.2):137–144, Jan. 2008.

[85] 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*, EuroPar '07, pages 662–671, Rennes, France, Aug. 2007.

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

[87] P. E. Hadjidoukas, G. C. Philos, and V. V. Dimakopoulos. Exploiting fine-grain thread parallelism on multicore architectures. *Scientific Programming*, 17(4):309–323, Dec. 2009.

[88] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa. Backtracking-based load balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 55–64, Raleigh, North Carolina, USA, Feb. 2009.

[89] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa. Process-in-process: Techniques for practical address-space sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 131–143, Tempe, Arizona, USA, June 2018.

[90] R. D. Hornung and J. A. Keasler. The RAJA portability layer: Overview and status. *LLNL-TR-661403*, Sept. 2014.

[91] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '03, pages 306–322, College Station, Texas, USA, Oct. 2003.

[92] J. Hubička, A. Jaeger, M. Matz, and M. Mitchell. System V application binary interface AMD64 architecture processor supplement. `https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf`, Oct. 2013.

[93] A. Huynh, C. Helm, S. Iwasaki, W. Endo, B. Namsraijav, and K. Taura. TP-PARSEC: A task parallel PARSEC benchmark suite. *Journal of Information Processing*, 27:211–220, 2019.

[94] 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*, IPDPS '10, pages 958–968, Atlanta, Georgia, USA, Apr. 2010.

[95] IEEE Standard for Information Technology–Portable Operating System Interface (POSIX®) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language). *IEEE Std. 1003.1c*, 1995.

[96] S. Imam and V. Sarkar. Load balancing prioritized tasks via work-stealing. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing*, Euro-Par '15, pages 222–234, Vienna, Austria, Aug. 2015.

[97] Intel 64 and IA-32 Architectures Software Developers Manual Volume 2, Sept. 2016.

[98] Intel® C++ Compilers — Intel® Software. `https://software.intel.com/en-us/c-compilers`.

[99] Intel® Cilk™ Plus Application Binary Interface Specification. https://www.cilkplus.org/sites/default/files/open_specifications/CilkPlusABI_1.1.pdf, Dec. 2011.

[100] Intel Corporation. Intel® Math Kernel Library (Intel® MKL) — Intel® Software. https://software.intel.com/en-us/mkl.

[101] Intel MKL-DNN. https://github.com/intel/mkl-dnn.

[102] Intel® OpenMP* Runtime Library. https://www.openmprtl.org/.

[103] Intel® Software Development Emulator — Intel® Software. https://software.intel.com/en-us/articles/intel-software-development-emulator.

[104] *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. International Organization for Standardization, Geneva, Switzerland, Feb. 2011.

[105] S. Iwasaki and K. Taura. Autotuning of a cut-off for task parallel programs. In *Proceedings of the IEEE 9th International Symposium of Embedded Multicore/Many-core Systems-on-Chip*, MCSoC '16, pages 353–360, Lyon, France, Sept. 2016.

[106] S. Iwasaki and K. Taura. A static cut-off for task parallel programs. In *Proceedings of the 25th International Conference on Parallel Architectures and Compilation Techniques*, PACT '16, pages 139–150, Haifa, Israel, Sept. 2016.

[107] 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*, OOPSLA '93, pages 91–108, Washington, D.C., USA, Sept. 1993.

[108] L. V. Kalé, J. Yelon, and T. Knuff. Threads for interoperable parallel programming. In *Proceedings of the Ninth International Workshop on Languages and Compilers for Parallel Computing*, LCPC '96, pages 534–552, San Jose, California, USA, Aug. 1996.

[109] KDD Cup 1999 Data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[110] M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending openmp* with vector constructs for modern multicore SIMD architectures. In *Proceedings of the Eighth International Conference on OpenMP*, IWOMP '12, pages 59–72, Rome, Italy, June 2012.

[111] 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*, CGO '04, pages 75–86, San Jose, California, USA, Mar. 2004.

[112] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 411–420, Vienna, Austria, Sept. 2010.

[113] J. Lee and M. Sato. Implementation and performance evaluation of XcalableMP: A parallel programming language for distributed memory systems. In *Proceedings of the 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 413–420, San Diego, California, USA, Sept. 2010.

[114] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, Orlando, Florida, USA, Oct. 2009.

[115] C. E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, San Francisco, California, USA, July 2009.

[116] D. K. Lowenthal, V. W. Freeh, and G. R. Andrews. Efficient support for fine-grain parallelism on shared-memory machines. *Concurrency: Practice and Experience*, 10(3):157–173, Mar. 1998.

[117] H. Lu, S. Seo, and P. Balaji. MPI+ULT: Overlapping communication and computation with user-level threads. In *Proceedings of the 17th International Conference on High Performance Computing and Communications*, HPCC '15, pages 444–454, New York, New York, USA, Aug. 2015.

[118] 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*, 12(12):1205–1218, Oct. 2000.

[119] Margo — Argobots bindings to the Mercury RPC API. `https://xgitlab.cels.anl.gov/sds/margo`.

[120] X. Martorell, J. Labarta, N. Navarro, and E. Ayguadé. A library implementation of the nano-threads programming model. In *Proceedings of the second European Conference on Parallel Processing*, EuroPar '96, pages 644–649, Lyon, France, Aug. 1996.

[121] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard 3.1, June 2015.

[122] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 185–197, Nice, France, June 1990.

[123] MPICH — High-Performance Portable MPI. `https://www.mpich.org/`.

[124] J. Nakashima and K. Taura. MassiveThreads: A thread library for high productivity languages. *Lecture Notes in Computer Science – Concurrent Objects and Beyond*, 8665:222–238, Jan. 2014.

[125] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. MineBench: A benchmark suite for data mining workloads. In *Proceedings of 2006 IEEE International Symposium on Workload Characterization*, IISWC '06, pages 182–188, San Jose, California, USA, Oct. 2006.

[126] D. Novillo. OpenMP and automatic parallelization in GCC. In *Proceedings of the GCC Developers' Summit*, pages 23–24, Ottawa, Ontario, Canada, June 2006.

[127] 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*, 26(2):110–124, May 2012.

[128] OpenBLAS: An optimized BLAS library. `https://www.openblas.net/`.

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

[130] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 5.0, Nov. 2018.

[131] OpenMP C and C++ Application Program Interface Version 1.0, Oct. 1998.

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

[133] P. Osmialowski. How the Flang frontend works: Introduction to the interior of the open-source Fortran frontend for LLVM. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM-HPC '17, pages 1:1–1:14, Denver, Colorado, USA, Nov. 2017.

[134] 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*, PLDI '10, pages 376–387, Toronto, Ontario, Canada, June 2010.

[135] 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*, Euro-Par 08, pages 78–88, Las Palmas de Gran Canaria, Spain, Aug. 2008.

[136] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, Cluster '08, pages 142–151, Tsukuba, Japan, Sept. 2008.

[137] A. Podobas, M. Brorsson, and K.-F. Faxén. A comparative performance study of common and popular task-centric programming frameworks. *Concurrency and Computation: Practice and Experience*, 27(1):1–28, Jan. 2015.

[138] Power ISA$^{\text{TM}}$ Version 2.07 B, Jan. 2018.

[139] G. W. Price and D. K. Lowenthal. A comparative analysis of fine-grain threads packages. *Journal of Parallel and Distributed Computing*, 63(11):1050–1063, Nov. 2003.

[140] Procedure Call Standard for the ARM 64-Bit Architecture. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf`, May 2013.

[141] A. Qawasmeh, A. M. Malik, and B. M. Chapman. Adaptive OpenMP task scheduling using runtime APIs and machine learning. In *Proceedings of 14th IEEE International Conference on Machine Learning and Applications*, ICMLA '15, pages 889–895, Miami, Florida, USA, Dec. 2015.

[142] 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*, PLDI '11, pages 26–37, San Jose, California, USA, June 2011.

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

[144] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 249–265, Feb. 2017.

[145] F. Schmager, N. Cameron, and J. Noble. GoHotDraw: Evaluating the Go programming language with design patterns. In *Proceedings of the First Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 10:1–10:6, Reno, Nevada, USA, Oct. 2010.

[146] J. Schuchart, K. Tsugane, J. Gracia, and M. Sato. The impact of taskyield on the design of tasks communicating through MPI. In *Proceedings of the 13th International Workshop on OpenMP*, IWOMP '18, pages 3–17, Barcelona, Spain, Sept. 2018.

[147] B. S. Siegell and P. A. Steenkiste. Automatic selection of load balancing parameters using compile-time and run-time information. *Concurrency: Practice and Experience*, 9(4):275–317, Apr. 1997.

[148] K. Sivaramakrishnan, L. Ziarek, R. Prasad, and S. Jagannathan. Lightweight asynchrony using parasitic threads. In *Proceedings of the Fifth ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, pages 63–72, Madrid, Spain, Jan. 2010.

[149] L. Sommer, J. Korinth, and A. Koch. OpenMP device offloading to FPGA accelerators. In *Proceedings of the IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, ASAP '17, pages 201–205, Seattle, Washington, USA, July 2017.

[150] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross. Mercury: Enabling remote procedure call for high-performance computing. In *Proceedings of the 2013 IEEE International Conference on Cluster Computing*, Cluster '13, pages 1–8, Indianapolis, Indiana, USA, Sept. 2013.

[151] D. Spoonhower. Scheduling deterministic parallel programs. *Ph.D. Thesis, Carnegie Mellon University*, May 2009.

[152] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper. Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 91–100, Aug. 2009.

[153] G. Stelle, W. S. Moses, S. L. Olivier, and P. McCormick. OpenMPIR: Implementing OpenMP tasks with Tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM-HPC '17, pages 3:1–3:12, Denver, Colorado, USA, Nov. 2017.

[154] SunSoft Inc. Solaris multithreaded programming guide. *Sun Microsystem Press/Prentice Hall*, 1995.

[155] 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*, LCR '00, pages 100–112, Rochester, New York, USA, May 2000.

[156] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama. A task parallelism meets fast multipole methods. In *Proceedings of the Third Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '12, pages 617–625, Salt Lake City, Utah, USA, Nov. 2012.

[157] K. Taura, K. Tabata, and A. Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 60–71, Atlanta, Georgia, USA, May 1999.

[158] K. Taura and A. Yonezawa. Fine-grain multithreading with minimal compiler support - a cost effective approach to implementing efficient multithreading languages. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 320–333, Las Vegas, Nevada, USA, June 1997.

[159] 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*, IWOMP '13, pages 1–14, Canberra, Australia, Sept. 2013.

[160] 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*, EuroPar '07, pages 42–51, Rennes, France, Aug. 2007.

[161] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, Apr. 2018.

[162] P. Thoman, H. Jordan, and T. Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 164–177, Aachen, Germany, Aug. 2013.

[163] 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*, HIPS '03, pages 47–55, Nice, France, Apr. 2003.

[164] 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*, 31(10-12):960–983, Oct. 2005.

[165] C. Tismer. Continuations and stackless Python. In *Proceedings of the Eighth International Python Conference*, IPC '00, Arlington, Virginia, USA, Jan. 2000.

[166] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Transactions on Programming Languages and Systems*, 36(3):10:1–10:51, Sept. 2014.

[167] I. E. Venetis and T. S. Papatheodorou. Tying memory management to parallel programming models. In *Proceedings of the 12th European Conference on Parallel Processing*, EuroPar '96, pages 666–675, Dresden, Germany, Aug. 2006.

[168] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the Ninth Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS '03, pages 4:1–6, Lihue, Hawaii, USA, May 2003.

[169] M. Voss and R. Eigenmann. Reducing parallel overheads through dynamic serialization. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing.*, IPPS/SPDP 1999, pages 88–92, San Juan, Puerto Rico, USA, Apr. 1999.

[170] C. Wang, S. Chandrasekaran, and B. Chapman. An OpenMP 3.1 validation testsuite. In *Proceedings of the Eighth International Conference on OpenMP*, IWOMP '12, pages 237–249, Rome, Italy, June 2012.

[171] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P.-C. Yew. An adaptive task creation strategy for work-stealing scheduling. In *Proceedings of the Eighth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 266–277, Toronto, Ontario, Canada, Apr. 2010.

[172] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, San Jose, California, USA, Nov. 1998.

[173] K. B. Wheeler, R. C. Murphyand, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS '08, pages 1–8, Miami, Florida, USA, Apr. 2008.

[174] J. Wiegert, G. Regnier, and J. Jackson. Challenges for scalable networking in a virtualized server. In *Proceedings of the 16th International Conference on Computer Communications and Networks*, ICCCN '07, pages 179–184, Honolulu, Hawaii, USA, Aug. 2007.

[175] 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*, IWOMP '16, pages 187–202, Nara, Japan, Oct. 2016.

[176] C. Yang and J. Mellor-Crummey. A practical solution to the cactus stack problem. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 61–70, Pacific Grove, California, USA, July 2016.

[177] L. Ying, G. Biros, and D. Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2):591–626, May 2004.

[178] R. Yokota. An fmm based on dual tree traversal for many-core architectures. *Journal of Algorithms & Computational Technology*, 7(3):301–324, Sept. 2013.

[179] R. Yokota and L. A. Barba. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *Int. J. High Perform. Comput. Appl.*, 26(4):337–346, Nov. 2012.

[180] C. S. Zakian, T. A. Zakian, A. Kulkarni, B. Chamith, and R. R. Newton. Concurrent Cilk: Lazy promotion from tasks to threads in C/C++. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, LCPC '15, pages 73–90, Raleigh, North Carolina, USA, Sept. 2016.