Argobots: A Lightweight Low-Level Threading and Tasking Framework

Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman

Abstract—In the past few decades, a number of user-level threading and tasking models have been proposed in the literature to address the shortcomings of OS-level threads, primarily with respect to cost and flexibility. Current state-of-the-art user-level threading and tasking models, however, either are too specific to applications or architectures or are not as powerful or flexible. In this paper, we present Argobots, a lightweight, low-level threading and tasking framework that is designed as a portable and performant substrate for high-level programming models or runtime systems. Argobots offers a carefully designed execution model that balances *generality* of functionality with providing a rich set of controls to allow *specialization* by end users or high-level programming models. We describe the design, implementation, and performance characterization of Argobots and present integrations with three high-level models: OpenMP, MPI, and colocated I/O services. Evaluations show that (1) Argobots, while providing richer capabilities, is competitive with existing simpler generic threading runtimes; (2) our OpenMP runtime offers more efficient interoperability capabilities than production OpenMP runtimes do; (3) when MPI interoperates with Argobots instead of Pthreads, it enjoys reduced synchronization costs and better latency-hiding capabilities; and (4) I/O services with Argobots reduce interference with colocated applications while achieving performance competitive with that of a Pthreads approach.

Index Terms—Argobots, user-level thread, tasklet, OpenMP, MPI, I/O, interoperability, lightweight, context switch, stackable scheduler.

1 Introduction

Efficiently supporting massive on-node parallelism demands highly flexible and lightweight threading and tasking runtimes. OS-level threads have been long recognized to be inadequate in this regard, primarily owing to their heavy-handed approach in managing arbitration and synchronization, as well as their inflexibility in adapting to the specialization requirements of specific applications. As a result, over the past few decades, a number of user-level threading and tasking abstractions have emerged as more practical alternatives.

- *S. Seo, A. Amer, P. Balaji, P. Carns, and P. Beckman are with Argonne National Laboratory. E-mail: {sseo,aamer,balaji,carns,beckman}@anl.gov*
- *C. Bordage is with Inria Bordeaux. E-mail: cyril.bordage@inria.fr*
- *A. Brooks, P. Jindal, L. Kalé, and M. Snir are with the University of Illinois at Urbana-Champaign. E-mail: {brooks8,jindal2,kale,snir}@illinois.edu*
- *G. Bosilca, D. Genet, and T. Herault are with the University of Tennessee, Knoxville. E-mail: {bosilca,dgenet,herault}@icl.utk.edu*
- *S. Iwasaki and K. Taura are with the University of Tokyo. E-mail: {iwasaki,tau}@eidos.ic.i.u-tokyo.ac.jp*
- *A. Castelló is with Universitat Jaume I. E-mail: adcastel@uji.es*
- *S. Krishnamoorthy is with Pacific Northwest National Laboratory. E-mail: sriram@pnnl.gov*
- *J. Lifflander is with Sandia National Laboratories. E-mail: jliffl2@illinois.edu*
- *H. Lu is with Tencent. E-mail: huiweilv@tencent.com*
- *E. Meneses is with Costa Rica National High Technology Center and the Costa Rica Institute of Technology. E-mail: esteban.meneses@acm.org*
- *Y. Sun is with Google. E-mail: sun51@illinois.edu*
- *S. Seo, A. Amer, and E. Meneses are IEEE members.*
- *L. Kalé and M. Snir are IEEE fellows.*
- *P. Balaji and S. Krishnamoorthy are IEEE senior members.*
- *A. Brooks and A. Castelló are IEEE graduate student members.*
- *P. Beckman is an IEEE affiliate.*

Manuscript received July 28, 2017; revised XXX XX, XXXX.

These lightweight abstractions have successfully served as building blocks for several parallel programming systems and applications. Current state of the art, however, suffers from shortcomings related to how these abstractions handle *generality* and *specialization*. Existing runtimes tailored for *generic* use [1]–[9] are suitable as common frameworks to facilitate portability and interoperability but offer insufficient flexibility to efficiently capture higher-level abstractions. This lack of flexibility often takes the form of transparent decisions on behalf of the user that incur undesired costs or inefficient resource usage. For instance, these runtimes implement transparent and rigid scheduling decisions (e.g., random work stealing) that incur costs (e.g., shared thread pool accesses) and provide no guarantee for optimal scheduling. Unfortunately, these runtimes provide little to no control to the user to overcome these inefficiencies. *Specialized* runtimes are oriented to a specific environment, for example, runtimes targeted at OS task management [10], [11], network services [12]–[14], compiler frameworks [15], specific hardware [16], and parallel programming runtimes [17]–[20]. These are heavily customized with a rich set of capabilities. Such abstractions, however, are virtually unusable outside the target environment because they were not meant for general use. They are often not *portable* (e.g., targeted to specific hardware) and do not offer sufficient user control outside the target environment. The Intel OpenMP Runtime [17] and Nanos++ [18], for instance, are efficient backend runtimes for OpenMP compilers but are hardly usable outside this scope.

We propose, in this paper, to fill this gap with Argobots, a lightweight, low-level threading and tasking framework.

Argobots not only offers a portable library interface that is *broadly applicable* to a number of target domains but also provides a rich set of controls to allow *specialized* runtime management by the user. The first goal of Argobots is to expose sufficient information and capabilities for users to efficiently map high-level abstractions to low-level implementations. The second goal is to allow different software packages to interoperate through Argobots as a lightweight substrate instead of relying on OS-level interoperation.

Argobots honors this high degree of expressibility through three key aspects. First, Argobots distinguishes between the requirements of different *work units*, which are the most basic manageable entities. Work units that require private stacks and context-saving capabilities, referred to as *user-level threads* (ULTs, also called *coroutines* or *fibers*), are fully fledged threads usable in any context. *Tasklets* do not require private stacks. They are more lightweight than ULTs because they do not incur context saving and stack management overheads. Tasklets, however, are restrictive; they can be executed only as atomic work units that run to completion without context switching. This distinction allows users to create the work unit type that fits their purpose. When tasklets are sufficient, performance gains over ULTs are certain. Second, work units execute within OS-level threads, which we refer to as *execution streams* (ESs). Unlike existing generic runtimes, ESs are exposed to and manageable by users. This added level of control offers opportunities for affinity and interoperability improvements (e.g., avoiding oversubscription of OS-level threads). Third, Argobots allows full control over *work unit management*. Users can freely manage scheduling and mapping of work units to ESs and achieve the desired behavior.

In order to ensure fast critical paths despite the rich set of capabilities, Argobots was designed in a modular way to offer configuration knobs and a rich API that allow users to trim unnecessary costs. An in-depth critical path characterization study is also provided, which involved investigating every cache miss and translation lookaside buffer (TLB) miss that occurs on critical paths. In a fully optimized state, Argobots achieved unprecedented performance in the context of lightweight runtimes. Indeed, evaluating Argobots against several highly performing generic lightweight threading libraries, such as Qthreads [5] and MassiveThreads [4], showed that Argobots incurs little overhead and scales better than the other libraries while achieving sustainable performance.

To evaluate the adequacy of Argobots as a substrate runtime and its interoperability capabilities, we present prototype integrations with the most widely used programming systems in high-performance computing (HPC)—OpenMP and MPI—as well as a use case in colocated I/O services. Our OpenMP runtime over Argobots avoids OS-level thread interoperability issues that arise from nesting OpenMP-based software. We demonstrate that OpenMP over Argobots can scale significantly better than existing OpenMP runtimes with synthetic benchmarks and in a fast multipole method (FMM) implementation that suffers from nested parallelism when offloading computation to an external OpenMP-based parallel library. We also show that when interoperating with MPI, Argobots can enable reduced synchronization costs and better latency-hiding capabilities, compared with Pthreads. Moreover, unlike with Pthreads, we show that I/O services over Argobots can readily decouple tuning the level of CPU and I/O concurrency. Consequently, the resulting I/O services lower interference with colocated applications by reducing CPU consumption while achieving performance competitive with that of a Pthreads approach.

2 Design and Implementation of Argobots

This section presents the key components of Argobots.

2.1 Execution Model

Figure 1 illustrates the execution model of Argobots. Two levels of parallelism are supported: ESs and work units. An ES maps to one OS thread, is explicitly created by the user, and executes independently of other ESs. A work unit is a lightweight execution unit, a ULT or a tasklet, that runs within an ES. There is no parallel execution of work units within a single ES, but work units across ESs can be executed in parallel. Each ES is associated with its own scheduler that is in charge of scheduling work units according to its scheduling policy. The scheduler also handles asynchronous events periodically. Argobots provides some basic schedulers, and users can also write their own.

ULTs and tasklets are associated with function calls and execute to completion. However, they differ in subtle aspects that make each of them suited for distinct programming motifs. A ULT has its own stack region, whereas a tasklet borrows the stack of its host ES's scheduler. A ULT is an independent execution unit in user space and provides standard thread semantics at a low context-switching cost. ULTs are suitable for expressing parallelism in terms of persistent contexts whose flow of control can be paused and resumed. Unlike OS-level threads, ULTs are not intended to be preempted. They cooperatively yield control, for example, when they wait for remote data or let other work units make progress for fairness. A tasklet is an indivisible unit of work with dependence only on its input data, and it typically provides output data upon completion. Tasklets do not yield control and run to completion before returning control to the scheduler that invoked them.

2.2 Scheduler

Argobots provides an infrastructure for stackable or nested schedulers, with pluggable scheduling policies, while exploiting the cooperative nonpreemptive activation of work units. Localized scheduling policies such as those used in current runtime systems, while efficient for short execution, are unaware of global policies and priorities. Plugging in custom policies enables higher levels of the software stack to use their special policies while Argobots handles the low-level scheduling mechanisms. In addition, stacking schedulers empowers the user to switch schedulers when multiple software modules or programming models interact in an application. For example, when the application executes an external library that has its own scheduler, it pauses the current scheduler and invokes the library's scheduler. Doing so activates work units associated with the invoked

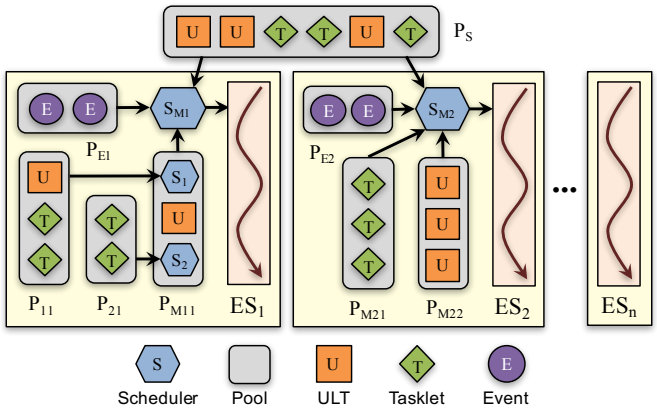


Fig. 1: Argobots execution model. An ES (curved arrow) is a sequential instruction stream that consists of one or more work units. S_M denotes the ES's main scheduler. S_{M1} in ES_1 has one associated private pool, P_{M11}, and S_{M2} in ES_2 has two private pools, P_{M21} and P_{M22}. Arrows indicate associations between schedulers and pools. P_S is shared between ES_1 and ES_2, and thus both S_{M1} in ES_1 and S_{M2} in ES_2 can access the pool to push or pop work units. P_E denotes an event pool. S_1 and S_2 in P_{M11} are stacked schedulers that will be executed by the main scheduler S_{M1}.

scheduler. The control returns to the original scheduler upon completion.

Argobots allows each ES to have its own schedulers. To execute work units, an ES has at least one main scheduler (S_M). A scheduler is associated with one or more *pools* where ready ULTs and tasklets are waiting for their execution. Pools have an access property, for example private to an ES or shared between ESs. Sharing or stealing work units among schedulers (or ESs) is done through shared pools. Each ES also has a special event pool (P_E) for asynchronous events. The event pool is meant for lightweight notification. It is periodically checked by a scheduler to handle the arrival of events (e.g., messages from the network).

When a work unit is in a pool that is associated with a running or stacked scheduler, it is considered ready to execute. Thus, Argobots does not control dependencies between work units. The control is done in the application itself through mechanisms provided by Argobots, such as waiting for completion and synchronization. In order to ensure a particular affinity of a work unit to some data, the application can use the appropriate pool when pushing the work unit. Thus, the work unit will be executed on the ES (or a group of ESs) that pops it from that pool.

Stacking schedulers is achieved through pushing schedulers into a pool. In other words, schedulers in a pool are regarded as schedulable units (e.g., S_1 in Figure 1 is a stacked scheduler that will be executed by S_{M1}). When a higher-level scheduler pops a scheduler from its pool, the new scheduler starts its execution (i.e., scheduling). Once it completes the scheduling, control returns to the scheduler that started the execution. To give control back to the parent scheduler, a scheduler can also yield. To support plugging in different scheduling policies, all schedulers, including the main scheduler, and pools are replaceable by user-provided alternatives.

2.3 Primitive Operations

Argobots defines primitive operations for work units. Since tasklets are used for atomic work without suspending, most operations presented here—except creation, join, and migration—apply only to ULTs.

Creation. When ULTs or tasklets are created, they are inserted into a specific pool in a ready state. Thus, they will be scheduled by the scheduler associated with the target pool and executed in the ES associated with the scheduler. If the pool is shared with more than one scheduler and the schedulers run in different ESs, the work units may be scheduled in any of the ESs.

Join. Work units can be joined by other ULTs. When a work unit is joined, it is guaranteed to have terminated.

Yield. When a ULT yields control, the control goes to the scheduler that was in charge of scheduling in the ES at the point of yield time. The target scheduler schedules the next work unit according to its scheduling policy.

Yield_to. When a ULT calls yield_to, it yields control to a specific ULT instead of the scheduler. Yield_to is cheaper than yield because it bypasses the scheduler and eliminates the overhead of one context switch. Yield_to can be used only among ULTs associated with the same ES.

Migration. Work units can be migrated between pools.

Synchronizations. Mutex, condition variable, future, and barrier operations are supported, but only for ULTs.

2.4 Implementation

We have implemented Argobots in the C language.[1] An ES is mapped to a Pthread and can be bound to a hardware processing element (e.g., CPU core or hardware thread). Context switching between ULTs can be achieved through various methods, such as `ucontext`, `setjmp/longjmp` with `sigaltstack` [21], or Boost library's `fcontext` [22]. Our implementation exploits `fcontext` by default and provides `ucontext` as an alternative when the user requires preserving the signal mask between context switches. Indeed, `fcontext` is significantly faster than `ucontext` mostly because it avoids preserving the signal mask, which requires expensive system calls. The user context includes CPU registers, a stack pointer, and an instruction pointer. When a ULT is created, we create a ULT context that contains a user context, a stack, the information for the function that the ULT will execute, and its argument. A stack for each ULT is dynamically allocated, and its size can be specified by the user. The ULT context also includes a pointer to the scheduler context in order to yield control to the scheduler or return to the scheduler upon completion. Since a tasklet does not need a user context, it is implemented as a simple data structure that contains a function pointer, argument, and some bookkeeping information, such as an associated pool or ES. Tasklets are executed on the scheduler's stack space.

A pool is a container data structure that can hold a set of work units and provides operations for insertion and deletion. Argobots defines the interface required to implement a pool, and our implementation provides a first-in, first-out (FIFO) queue as a pool implementation. A scheduler is

1. The reader can find the Argobots implementation and examples at https://github.com/pmodels/argobots.

implemented similarly to a work unit; it has its own function (i.e., scheduling function) and a stack. Since a scheduler is regarded as a schedulable unit, it can be inserted into a pool and executed as a work unit.

Argobots relies on cooperative scheduling of ULTs to improve resource utilization. That is, a ULT may voluntarily yield control when idle in order to allow the underlying ES to make progress on other work units. Idling occurs when executing blocking operations. Yielding control can be achieved either implicitly, through Argobots synchronization primitives, or explicitly by calling `yield` or `yield_to`. Some Argobots synchronization primitives, such as mutex locking or thread join operations, automatically yield control when blocking is inevitable. ULTs that interact with external blocking resources (such as network or storage devices) are expected to explicitly context switch by using `yield` or `yield_to`. Furthermore, synchronization primitives can be used to resume execution upon completion of external resource operations. This capability will be illustrated in Section 5.3 when coupled with I/O operations.

3 Critical Path Cost Analysis

Argobots is intended for fine-grained dynamic environments, where work unit creation, destruction, and context-switching take place at high frequencies. The rich set of capabilities that Argobots offers, however, can clutter and slow the critical path of Argobots applications. Indeed, supporting such capabilities would require longer code paths and more complex data layouts than a simpler threading runtime would. To allow high flexibility without sacrificing performance, Argobots offers build-time configuration features and advanced API routines that allow capturing efficiently higher-level software requirements. When these features are exploited properly, Argobots' critical path can be competitive or outperform state-of-the-art simpler threading runtimes.

This section presents a cost analysis of basic work unit management primitives, such as creation, joining, and destruction operations, which are found on the critical path of Argobots applications. The goal is for the Argobots user to relate features to costs on the critical path as well as understand the favorable conditions that would bring down these costs. We present experimental results only with ULTs; similar observations apply to tasklets. The exceptions are the join features in Section 3.4, which are applicable only to ULTs since tasklets are not allowed to join other work units, and the data structure organization in Section 3.3, which is insensitive for a tasklet descriptor because it can fit in one cache line.

Methodology. We follow an incremental approach that starts with a basic Argobots implementation and then gradually incorporates features that lower the costs on the critical path. Each step involves a cost analysis and the corresponding feature to lower the cost. In the following, we begin by describing our testbed, a simple microbenchmark that allows us to profile in isolation ULT operations, and then present details about the baseline Argobots implementation.

3.1 Experimental Setup

For all experiments, we used a 36-core (72 hardware threads) machine, which has two Intel Xeon E5-2699 v3 (2.30 GHz) CPUs and 128 GB of memory and runs Red Hat Linux (kernel 3.10.0-327.el7.x86_64) 64-bit. We used `gcc` 4.8.5 for compiling and PAPI [23] for collecting the necessary hardware counter values.

3.2 Baseline and Benchmark Description

Baseline. The baseline Argobots implementation is characterized by use of the default system memory management (i.e., system `malloc/free` and normal pages); semantic organization of data structures (that is, data fields are grouped according to their functionality); a fully fledged context switch mechanism; and a shared pool. Moreover, all Argobots features are build-time enabled.

Benchmark. For simplification, the analysis focuses on spawning and joining ULTs on one ES. That is, we create a large number of ULTs and push them to a shared pool in a bulk-synchronous fashion, join them by the main ULT, destroy them, and repeat the process over 1,000 iterations. Each ULT is created with 16 KiB of stack space. Although this section uses a single ES, our experiments showed similar observations when scaling ESs up to 72; for brevity, we omit including these results. In the following, we report latency results in CPU cycles and show memory-related hardware counters where needed.

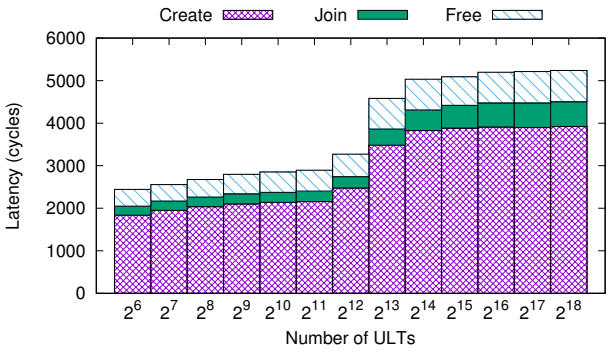


Fig. 2: Performance of the baseline implementation. Create, Join, and Free represent the time spent in creating, executing and joining, and destroying ULTs, respectively. Each bar represents the average latency (arithmetic mean) per iteration per ULT. The standard deviation of the mean is less than 5%.

Figure 2 shows the performance of our baseline implementation according to the number of ULTs that are created in the benchmark. For example, forking and joining 64 ULTs take 2,443 cycles (1.064 μs) for each ULT, where 1,837, 212, and 394 cycles are spent in Create, Join, and Free, respectively. In most cases, about 75% of the time is used for creation, and about 15% of the latency is used for destruction. These results hint at memory management issues and are investigated in the next subsection.

3.3 Memory Management

Work units in Argobots are meant for dynamic fine-grained concurrency. Thus, thread creation and destruction would be frequent. Further analysis of Create and Free reveals that memory allocation and deallocation contribute to 93% and 84% of each latency, respectively. These significant overheads of memory management come from the fact that the baseline implementation relies on `malloc` and `free`

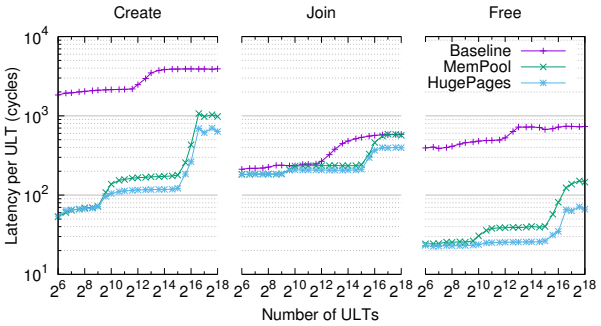


Fig. 3: Effects of using memory pools and huge pages.

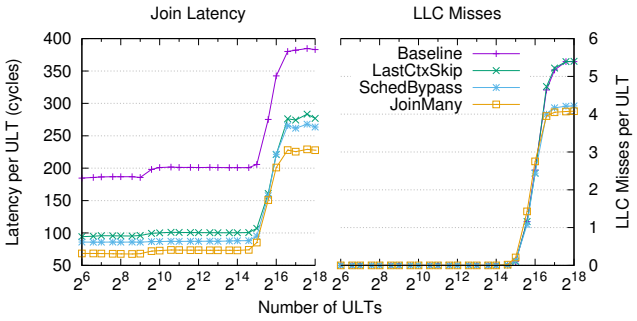


Fig. 4: Effects of the performance improvement techniques for Join.

functions provided in `glibc` to handle dynamic memory allocation.

We developed a custom memory allocator that reduces system calls and thread synchronization overheads. This allocator maintains a memory pool that grows in size with the number of spawned work units. After a work unit terminates, its memory resources are added to the pool or returned to the system if the pool has reached a certain threshold. Since the scalability of a dynamic memory allocator is limited mostly by synchronizations on the shared heap [24], each ES keeps a private memory pool for allocating work units, in order to reduce the number of accesses to the global heap. Basically, if creation and destruction of a work unit occur in the same ES, no synchronization is involved. If the creation and destruction of a work unit take place on different ESs, however, we combine fast-path accesses to local memory pools for scalability and slow-path accesses to remote or global memory pools for load balancing to avoid the heap-blowup problem [25]. Our memory management system also allows the possibility of reducing *memory address translation* overheads by configuring Argobots to use huge pages. This is achieved by allocating 2 MiB huge pages, instead of the normal 4 KiB pages, by using `mmap` until the system runs out of huge pages for explicit allocation. Then, the system reverts to the transparent huge page support [26]. This allows eliminating most of the TLB misses and the corresponding expensive page walks and eventual memory accesses to the page table.

We experimented with the new memory management system and show the latency results in Figure 3. The results are presented incrementally, with the huge pages feature (HugePages) implemented on the top of the custom allocator MemPool. We observe a substantial benefit of the new memory management system, especially for Create and Free, compared with Baseline. Join is less sensitive to these changes because object creation and destruction do not take place on its critical path. We found out, however, that it is sensitive to the layout of critical data structures. Our investigation showed that a performance-oriented data layout[2] could fit critical data in fewer cache lines than a semantic-oriented layout[3] could. Our experiments showed that this optimization lowers the Join latency by up to 7%, which corresponds to the reduction in last-level cache (LLC) misses.

2. A layout that focuses on gathering data according to their contribution to the scheduling critical path.

3. A layout that focuses on gathering data with close semantics or functionality (e.g., identification, scheduling, migration).

3.4 Context Switching

Suspending and resuming control of a thread are frequent operations in threaded environments when yielding control explicitly or implicitly through blocking or synchronization operations. In this section, we investigate the fundamental costs of context switching in Argobots in the context of the Join operation and hint to other operations, such as yield when appropriate.

Context of a Terminating ULT. Context switching comprises two steps: saving the context of the current ULT, which wants to suspend its execution, and restoring the context of the next ULT, which will resume execution. These two steps are usually necessary, but the first step can be omitted if the current ULT terminates, because its context will no longer be used. For this case, we perform only the second part of context switching to execute the next ULT. Since ULTs terminate immediately after they get started in the benchmark, this technique (LastCtxSkip in Figure 4) reduces on average 100 cycles, 45% of the Join latency from Baseline (after the memory optimizations of Section 3.3).

Scheduler Involvement. Since the joiner ULT cannot progress beyond the Join synchronization point until the ULT being joined terminates, it can be blocked and directly context switched to the next ULT to be joined, instead of going through the scheduler. In this case, when the joinee ULT is completed, the control is switched back to the joiner ULT. That is, we can bypass the scheduler in Join. SchedBypass in Figure 4 shows how this modification outperforms LastCtxSkip. The improved version removes context switches from and to the scheduler. In addition, since the joiner ULT can check the state of the joinee ULT right after it is terminated, its data structure is accessed only once by the joiner ULT whereas it is touched twice in the LastCtxSkip version by the scheduler and the joiner ULT. The effect of this technique can be seen as lower LLC miss rates in Figure 4. This approach does, however, have a limitation: it can be applied only to ULTs in the same ES like the yield_to operation (Section 2.3). Although this idea is similar to that presented in [27], the main difference between two approaches is that the target of context switching in our approaches is determined by the user, not the library or kernel.

Joiner ULT Involvement. With the previous improvement on Join, $2 \times N$ context switches are needed in order to join N ULTs, because joining one ULT requires two context switches. To further reduce the number of context switches when joining multiple ULTs at the same time, we devised

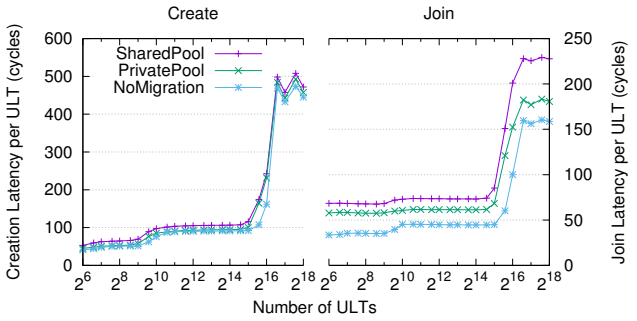


Fig. 5: Effects of using a private pool and disabling migration.

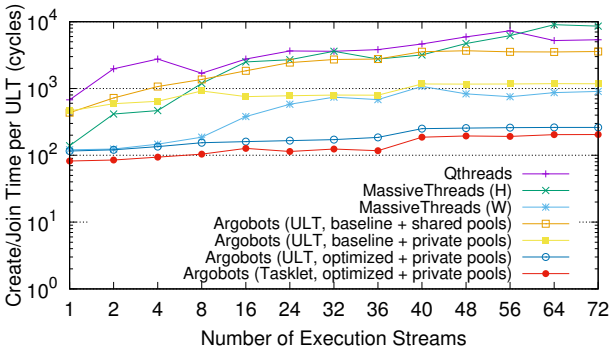


Fig. 6: Average create and join time per ULT with Qthreads, MassiveThreads, and Argobots. The join operation includes both joining a ULT and destroying it. MassiveThreads results include the default work-first scheduling (W) and the help-first scheduling (H) variations. Argobots was run with several variations to cover optimization levels, pool-sharing properties, and work unit types (ULT or Tasklet). These results were presented in a top-down cost-reducing order. Shared pools in Argobots imply *random work stealing*.

the join_many operation. This operation takes a list of ULTs to join and enables each ULT in the list to check the state of the next ULT and to context switch to the next one if it has not finished. Since the join_many operation does not return to the caller until all ULTs in the list terminate and each ULT does only one context switch to the next one, this operation reduces the number of context switches from $2 \times N$ to $N + 1$ and also decreases N Join function calls to a single join_many call. The performance effect of the join_many operation is illustrated in Figure 4 as JoinMany. It reduces the Join latency by an average of 19 cycles from that of SchedBypass.

3.5 Pool Sharing

All experiments so far used a shared pool, which is created by default, even though only one ES was used. The Argobots API exposes pool sharing control to users; a user can chose how many ESs are allowed to push and pull from a pool. If there is no sharing between ESs or only one ES is created, the pool can be created as a private one, which is intended for only sequential access and thus does not use any mutex or atomic instructions in the implementation. Since Create and Join include pushing a ULT to the pool and popping a ULT from the pool, respectively, their latency is improved with the private pool (PrivatePool in Figure 5). On the other hand, Free is not affected by the access property of the pool because it does not involve any pool manipulation.

3.6 Feature Selection

Not all features provided by Argobots are necessarily needed by a user. For instance, Argobots could be packaged into other software that requires only a subset of Argobots features. Unused features may affect the application's performance if their related code (e.g., branches) is part of the performance-critical path although it does nothing useful. To address this issue, Argobots provides configuration options to disable some features, for example, migration and stackable scheduler support. We observed that in the current implementation, disabling migration reduced around 20 cycles in the Join latency (NoMigration in Figure 5); disabling other features was insignificant for this benchmark.

3.7 Cost Analysis Discussion

From the preceding sections, we notice that using memory pools is the most effective for Create and Free while all methods introduced in the preceding subsections collectively influence the performance of Join. Because of the nature of the benchmark (i.e., it is designed to exercise bulk-synchronous ULT operations, and each ULT does nothing in its function), cases with a small number of ULTs can be considered as best scenarios where data structures and stacks fit in the LLC. Those results are difficult to tie to real applications, however, since they might not exhibit such high degrees of cache reuse. We consider the large number of ULT runs more insightful because there is almost no cache reuse, since the working sets do not fit in the LLC and hence reflect a worst-case scenario.

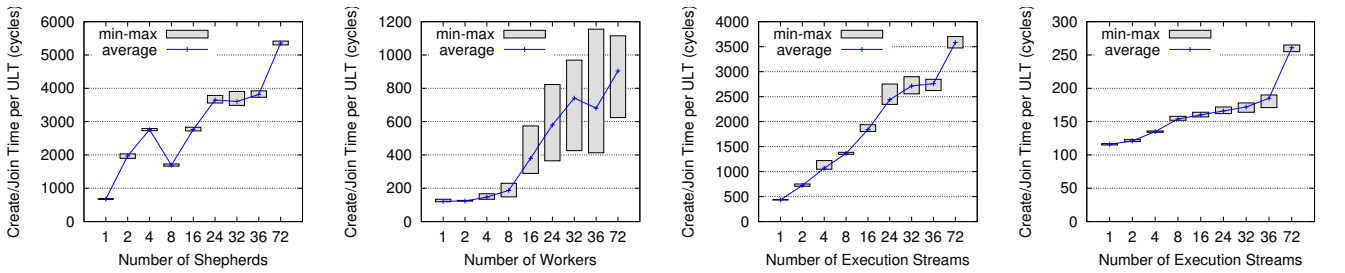
4 Evaluation

We evaluated our Argobots implementation by comparing with two ULT libraries, Qthreads 1.10 and MassiveThreads 0.95, in terms of performance and scalability in the same environment described in Section 3.1. We chose them because they are among the best-performing lightweight threading packages currently used in the HPC community and are available as independent libraries. Moreover, they have been subject to thorough studies by previous works and compared with other lightweight runtimes [4], [5]. All libraries were compiled with -O3 -ftls-model=initial-exec flags. The other build settings of Qthreads and MassiveThreads were left as default; in particular, both libraries maintain their own memory pools and use shared thread pools, which are hidden from the user. Qthreads uses the Sherwood hierarchical scheduler [28], which is locality aware and adopts work stealing for load balancing, and MassiveThreads relies on a Cilk-like last-in, first-out scheduling within a worker and FIFO randomized work stealing between workers [4].

4.1 Create/Join Time

We compared the time taken to create and join a ULT or a tasklet with respect to the number of ESs. For Qthreads and MassiveThreads, the number of workers was set to the same as that for ESs; and one worker in Qthreads was mapped to one shepherd.[4] We created one ULT for each

4. The default hierarchical configuration of one shepherd per chip and one worker per core showed worse results.



(a) Qthreads (b) MassiveThreads (W) (c) Argobots (ULT, baseline + shared pools) (d) Argobots (ULT, optimized + private pools)

Fig. 7: Create/join time tolerance.

ES, and that ULT repeated 1,000 times creating 256 work units, pushing them to the pool associated to its ES, and then joining them. We performed the same pattern for Qthreads and MassiveThreads.

Figure 6 illustrates the average create and join time per ULT for each library from 10 runs of the benchmark. Since MassiveThreads by default utilizes the *work-first* scheduling policy [29] (i.e., pushes the creator to the scheduling queue and executes the spawned thread first), while Qthreads and Argobots adopt the *help-first* principle [30] (i.e., create all threads first), we include results for both the work-first and help-first versions of MassiveThreads. An exhaustive exploration of all combinations of Argobots configurations and features would be excessive; thus we narrow the space exploration to a handful of combinations that incrementally reduce costs: from the baseline Argobots with ULTs, shared pools, and random work stealing, to using private pools, using all the optimizations in Section 3, and using tasklets instead of ULTs.

Ideally, if the ULT runtime is perfectly scalable, the time should be the same regardless of the number of ESs. Usually, however, that is not the case because hardware resources, such as caches, memory, or physical CPU cores, are shared between ESs and synchronizations might exist between ESs to protect shared data. In this benchmark, thread pools are the major resource being shared and thus a potential source of contention.

At the lowest degree of concurrency (i.e., one ES), Qthreads and the baseline Argobots perform the worst. The Argobots optimizations bring down the cost to be competitive with MassiveThreads. At the highest degree of concurrency (i.e., 72 ESs), all help-first scheduling runtimes that use shared pools scale poorly. Argobots, however, performs slightly better than the other runtimes despite not having the optimizations enabled. MassiveThreads with work-first scheduling performs the best in this high-contention regime, thanks to memory optimizations and optimized thread pool manipulation. We observe, however, that all experiments that use shared pools with some form of work stealing exhibit the worst scalability. While this overhead is out of the user's control in Qthreads and MassiveThreads, Argobots offers means to eliminate such interference through private pools, which results in almost perfect scalability. The slight scalability loss starting from 40 ESs is due to hardware threads sharing hardware resources on the same core. With private pools in Argobots, as long as schedulers or ULTs in different ESs do not share a pool or data, there will be almost zero synchronization (even no atomic instructions) between ESs. We also observe that the optimizations in Argobots reduce the overheads by an order of magnitude and that the tasklet abstraction brings down costs even further, making Argobots in this case the fastest and most scalable runtime.

4.2 Create/Join Time Tolerance

We also measured the minimum, maximum, and average time for each ULT on each ES to create and join another ULT. The results are summarized in Figure 7. Because of space limitations, we show only two Argobots combinations: the baseline with shared pools and the optimized setting with private pools. First, percentage-wise MassiveThreads shows the highest degree of variation, followed by Qthreads and the Argobots baseline, which are comparable, and finally by the optimized Argobots, which shows the lowest relative variation. Second, from an absolute variation perspective, MassiveThreads remains the highest (up to 740 cycles), followed closely by Qthreads and the baseline Argobots (up to 400 cycles). The optimized Argobots with private pools incurs the lowest variation, with less than 20 cycles variation across the board. We conclude that Argobots is the only runtime that can achieve both low overheads and sustainable performance; that is, ESs do not interfere with each other without explicit user-controlled interaction. Workers in Qthreads, MassiveThreads, and ESs in Argobots with shared pools interfere with each other; thus, the create/join time per ULT varies significantly when multiple workers or ESs are running, even though they do not interact at all in the user code. These results imply that the design of Argobots can enable users to build their higher-level software without worrying about unnecessary interference caused by the underlying threading runtime from a scheduling perspective.

4.3 Yield Time

The yield time contributes to the ULT create/join time as well. When a ULT tries to join a newly created ULT, it needs to yield control to the scheduler in order to execute the new ULT. The yield latency is also critical for applications that require frequent context switches. We measured the yield overhead for each library with respect to the number of ESs and show the results in Figure 8. For Argobots, we used the same configuration and feature combinations as in Section 4.1 with the exception of omitting tasklet experiments because, conceptually, a tasklet cannot yield. Since Argobots

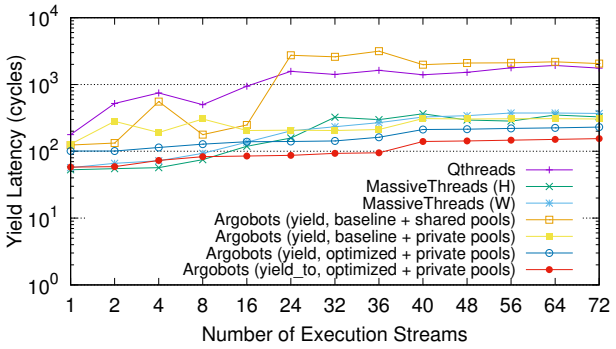


Fig. 8: Yield operation time.

supports the yield_to operation as well as yield (Section 2.3), we included results with the yield_to operation. At low concurrency, Qthreads incurs the highest overhead, followed by Argobots with its normal yield interface. In this case, both Qthreads and Argobots suffer from extra context switches to the scheduler (called *master thread* in Qthreads). MassiveThreads and Argobots with the yield_to interface are the fastest. These bypass the scheduler and effectively reduce the number of context switches by twofold, which can be observed when comparing Argobots with its yield operation variations. At higher degrees of concurrency, we observe similar scalability losses as in Section 4.1. In particular, contention for the Argobots shared pools adds significant overhead. The benefits of the Argobots optimizations are not as pronounced as with the create and join operations, but they are still significant. The yield_to operation reduces the overheads by a constant factor, twofold, regardless of the number of ESs.

4.4 XSBench

XSBench [31] is a proxy application that models the calculation of macroscopic neutron cross-sections of OpenMC, a Monte Carlo particle transport simulation code [32]. The kernel that XSBench simulates is the most computationally intensive part in OpenMC and takes around 85% of the total runtime of OpenMC, according to its documentation. It is written in the C language and is parallelized with OpenMP.

We port the main simulation part of XSBench, namely, the cross-section lookup loop, to Argobots by dividing the iterations of the lookup loop evenly among ESs. One ULT (main) per ES is created, and it creates as many work units as the number of lookups that are assigned to the ES. Each work unit performs one cross-section lookup. Since we noticed that the cross-section lookup code suffers from cache misses due to its irregular memory accesses, our Argobots version takes data locality into account, instead of simply executing the loop iterations as done in the original OpenMP code. We implemented a custom scheduler, using the Argobots scheduler framework (Section 2.2), that executes work units according to the order of the energy indices, which are random values but critical to the memory access pattern. Specifically, main ULTs sort the iterations in ascending order of energy indices and push them to their respective main pools. A scheduler then pulls and executes work units in order, to preserve the energy indices order for better locality. The scheduling begins after creating a certain number (here, 8,192) of work units. Sorting all iterations and creating work units for each iteration at once can lead to significant overhead in the memory usage and thus impact the performance. When its main pool is empty, a scheduler adopts work stealing from neighbors to reduce load imbalance and preserve data locality.

We also implemented XSBench using Qthreads and MassiveThreads. However, since they do not provide the flexibility of writing a user-defined scheduler as Argobots does, we sort the energy indices before creating ULTs (note that Qthreads and MassiveThreads do not support tasklets) and create ULTs according to the energy indices sorted. Then, we rely on their schedulers for the execution.

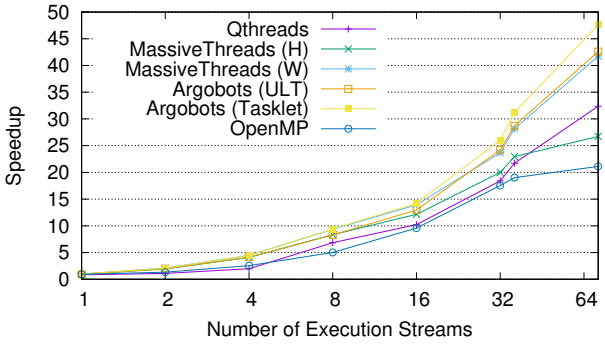


Fig. 9: XSBench performance results.

Figure 9 shows performance results of our XSBench implementations in Argobots, MassiveThreads with the work-first policy, and Qthreads, along with the original OpenMP implementation as a reference. The baseline XSBench used is version 13, dated May 2014; and we used the "large" input size having the default configuration of 355 nuclides, 11,303 grid points per nuclide, and 15 million lookups. Each version was run 12 times, with 5 iterations per run excluding warm-up steps. The figure shows the average result per iteration. The speedups in the graph are obtained by comparing execution times with that of the sequential code without OpenMP pragmas. Execution times with a single ES (OpenMP thread or worker) are 47.94 (Argobots (Tasklet)), 49.96 (Argobots (ULT)), 47.90 (MassiveThreads (W)), 52.04 (MassiveThreads (H)), 57.24 (Qthreads), 51.14 (original OpenMP without presorting), and 47.41 (sequential with presorting) seconds. The results in the figure show that all implementations scale well but that Argobots (Tasklet) achieves the best scalability. MassiveThreads (W) performs better than MassiveThreads (H) and indicates that MassiveThreads suffers in this case from the help-first policy, which stresses thread pool operations and scheduling more than a work-first policy would. Despite Argobots adopting a help-first policy, however, it performs comparably to MassiveThreads with its work-first policy, thanks to data locality scheduling and better pool access performance.

5 High-Level Runtimes

We present here three use cases of Argobots for high-level runtimes: an OpenMP runtime implementation that integrates Argobots as the threading layer, an MPI runtime implementation that interoperates with Argobots, and colocated I/O services that utilize Argobots for better resource

management. Hyperthreading is disabled hereafter because it did not have any positive effects on the experiments that follow.

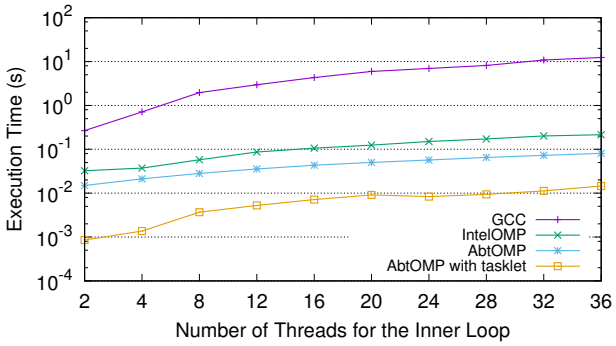
```
1  #pragma omp parallel for
2  for (int i = 0; i < N; i++) {
3    lib_comp(i, range[i], in, out);
4  }
5
6  void lib_comp(i, max, in[][], out[][]) {
7    #pragma omp parallel for
8    for (int k = 0; k < max; k++)
9      out[i][k] = compute(in[i][k]);
10 }
```

Listing 1: Example of OpenMP nested parallelism.

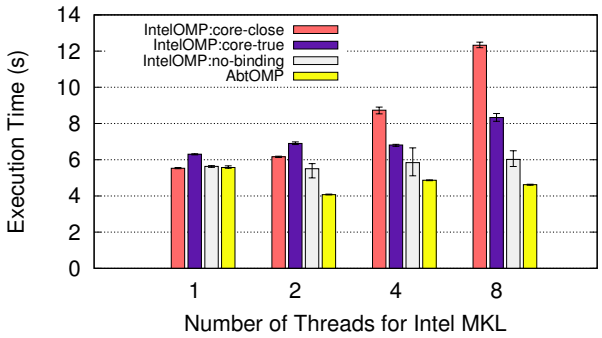
5.1 OpenMP over Argobots

OpenMP implementations, such as GCC OpenMP [33] or Intel OpenMP [17], perform poorly with nested parallel regions, like the case shown in Listing 1, because they use OS-level threads underneath (e.g., Pthreads); performance can drop significantly if the total number of OpenMP threads used for the nested parallel regions is larger than that of CPU cores (i.e., oversubscription). The common workaround found in practice is to avoid oversubscription by suppressing one level of parallelism. However, suppressing parallelism may lose opportunities for performance improvement or hinder programmers from using external libraries that internally use OpenMP. For example, consider Listing 1 that emulates an OpenMP user code (lines 1–4) calling a library routine (`lib_comp`) that internally uses OpenMP. Let us assume that the user uses all processing elements of the machine for the application, as is often the case in practice. In most production OpenMP runtimes, both parallel regions (at lines 1 and 7) would spawn an OpenMP team equal to the full machine size, resulting in a 2x oversubsctiption factor. This issue could be tackled by reducing the size of OpenMP teams. However, tuning the size of the team in the user code as well as in external dependencies to avoid oversubscription while fully utilizing the machine is a daunting challenge.

We designed an OpenMP runtime that exploits Argobots to better deal with nested parallelism. In our runtime, all parallel regions are mapped to Argobots work units (ULTs by default) regardless of the level of nesting. Furthermore, creating many work units does not add much overhead as long as the number of ESs is kept within the number of cores. Our runtime creates at most as many ESs as there are cores. Each ES features a customized scheduler that has one private pool and one shared pool. The private pool is used to schedule work units in an ES in a FIFO manner; the shared pool is used for sharing work units between ESs. ULTs for the first-level parallel regions are distributed to the private pool of each ES. ULTs for the nested parallel regions are pushed to the shared pool of the ES, where the master ULT in the team is running, but they can be stolen by other ESs if there is load imbalance between ESs. This hierarchical scheduling structure enables locality to be improved by binding the first-level ULTs to distinct ESs and enables the workload of ESs to be balanced through work stealing.



(a) Execution time of a nested parallel for loop. The number of threads for the outer loop was fixed at 36, and that for the inner loop was varied. The number of iterations for both outer and inner loops was 2,240, and static scheduling was used. GCC, IntelOMP, AbtOMP, and AbtOMP with tasklet represent results with gcc 6.1.0, Intel compiler 17.0.0 with Intel OpenMP runtime, the same Intel compiler with our OpenMP runtime using only ULTs, and using ULTs for the outer loop and tasklets for the inner loop, respectively.



(b) Execution time of the `Downward` stage in KIFMM with AbtOMP and IntelOMP with different thread bindings. `OMP_NUM_THREADS` was set to 9, and `MKL_NUM_THREADS` was varied.

Fig. 10: Performance results of OpenMP nested parallel loops.

To reduce thread management overheads, our OpenMP runtime, with a user hint, can generate tasklets for compute-only loops, which do not contain any blocking functions call or OpenMP synchronization (e.g., `critical` or `barrier`). In other words, if the code has no possibility of context switching occurring during the execution, the runtime creates tasklets instead of ULTs, since using ULTs adds unnecessary overhead from managing contexts and stacks. We currently provide an API function for the user to give our OpenMP runtime a hint of whether it is compute-only or not. We plan to develop compiler techniques so that this process is automated and thus the advantages of tasklets can be easily accessible. We note that while some previous work used ULTs to overcome nested parallelism issues [28], [34]–[36], our work exploits tasklets as well as ULTs and a custom scheduler that is specialized for OpenMP nested parallelism.

We prototyped our OpenMP runtime by modifying the open-source version of the Intel OpenMP runtime [37] and kept the application binary interface so that it can be used with existing compatible OpenMP compilers, such as Intel compiler, GCC, and LLVM clang. We also evaluated our implementation using one microbenchmark and a work-sharing-based implementation of FMM [38] on the machine described in Section 3.1.

The microbenchmark measures the execution time of a

nested parallel loop, which is similar to the code in Listing 1. Figure 10a illustrates the average execution times over 100 runs. As expected, our OpenMP runtime outperforms other OpenMP implementations because of using lightweight work units. The results imply that utilizing ULTs to implement parallel regions is efficient, and exploiting tasklets further reduces the overhead when it is possible. GCC shows the worst performance because the GCC OpenMP does not reuse threads and instead spawns threads every time it encounters the parallel region. IntelOMP achieves better performance than does GCC by reusing threads, but it has more overhead than our runtime does because of the heavy cost of managing Pthreads and real oversubscription of the machine.

Real-world case. We present here results with a highly tuned implementation of the FMM, a method to solve N-body problems, called kernel-independent FMM (KIFMM). We used the variant that implements each of the five stages that constitute the entire flow using OpenMP work-sharing constructs [38]. KIFMM offloads some compute-intensive operations, such as matrix-vector multiplications (`dgemv`) and fast Fourier transformations, to external libraries. These external packages are free to generate additional parallelization levels that might cause nested parallelism. Here, we focus on one of the stages (`Downward`) that is sensitive to data locality and has parent-children dependencies resulting from the hierarchical domain decomposition. It is also compute intensive and relies extensively on `dgemv` operations computed by linear algebra packages.

KIFMM offloads `dgemv` operations to the Intel Math Kernel Library (MKL) [39] shipped as part of the Intel complier suite. MKL also employs OpenMP internally, but this behavior is disabled if MKL detects that it is being called within an OpenMP parallel region. This is the default behavior and can be overridden by the user with appropriate environment variables. To evaluate the efficiency of the nested parallelism support in IntelOMP and AbtOMP during the `Downward` stage, we used 9 OpenMP threads for the application (outer parallel region) and varied the number of MKL threads (inner parallel region). This approach effectively allows a gradual shift from a non-oversubscribed regime (9 threads on 36 cores) to an oversubscribed one (72 threads on 36 cores). Figure 10b shows different trends for the OpenMP runtimes and binding strategies. IntelOMP clearly performs poorly as performance degrades with more MKL threads. We note that all binding policies result in some degree of oversubscription except when left to the OS (no binding) with IntelOMP. AbtOMP, on the other hand, scales slightly, then stagnates. One factor that affects both runs is the poor data locality resulting from offloading `dgemv` data to other threads that potentially run on remote cores. A second factor is the high thread management overhead, which includes oversubscription, of IntelOMP. We also note that using more than one MKL thread improves scalability over a single-threaded MKL with AbtOMP. Thus, MKL's default strategy of disabling nested parallelism loses parallelism opportunities that could be exploited with an efficient OpenMP runtime.

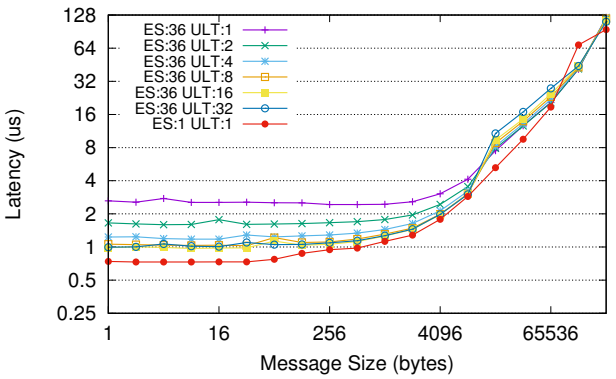


Fig. 11: MPI latency between two Haswell nodes interconnected with a Mellanox FDR fabric. One node hosts a single-threaded sender process while the other hosts a 36-way multi-ES receiver process.

5.2 Interoperability with MPI

Most MPI implementations interoperate with OS-level threads, such as Pthreads, to comply with MPI's threading support requirements. Consequently, shared-memory programming systems, including OpenMP, whose runtimes rely on OS-level threads underneath can interoperate with most MPI runtimes. This coarse-grained interoperability level is heavy, however, and does not allow exploiting upper-layer runtime information to improve synchronization and scheduling decisions. For instance, with existing MPI runtimes, an OpenMP task blocked for MPI communication cannot context switch to another task; thus it loses the opportunity to better utilize computational resources, because these runtimes are oblivious of OpenMP tasks. If the programming system shares a more lightweight and flexible common runtime with MPI, new synchronization and scheduling improvement opportunities will be exposed.

In this work, we investigated an MPI runtime that interoperates with Argobots ULTs instead of OS-level threads. The runtime is based on MPICH 3.2, a fully thread-compliant MPI implementation. MPICH 3.2 drives communication through a single communication context and ensures thread safety with a coarse-grained critical section. The runtime has been shown to be subject to lock management issues, which can significantly degrade performance [40]. The major interoperability challenge is handling thread safety. We exploit in the runtime a custom locking method tailored for Argobots' expressive capabilities and the needs of the MPI runtime. Our lock has three primary components. First, it is built on the advantages of the two-level prioritization scheme described in our prior work [40]; ULTs that are in a waiting state, which occurs in routines with blocking semantics, are demoted in favor of other ULTs in order to avoid waste and improve progress. Second, lock acquire and release operations avoid contention on the critical path. This feature is achieved by blocking a ULT with an unsuccessful acquisition in a low-contention queue corresponding to the ES and the priority level of the ULT. Third, we expose an API routine in Argobots to allow a lightweight lock ownership passing between ULTs in the same ES. Since such ULTs are sequential, lock ownership can be passed with mostly a simple context switch without an expensive fully fledged lock release operation.

To evaluate this runtime, we ran a benchmark that stresses communication latency between two MPI processes: a sender and a receiver. The sender issues a stream of blocking send operations, and the receiver consumes the messages with blocking receive operations. The sender is single-threaded (i.e., one ULT in one ES), and the receiver is multithreaded with Argobots. The goal of this benchmark is to stress the capability of the receiver to keep pace with the sender. Figure 11 shows latency results between two Haswell nodes (Section 3.1) with 36 ESs at the receiver side while scaling the number of ULTs per ES. We observe that the response time of the receiver improves with the number of ULTs per ES until saturation. A total of 288 ULTs (8 ULTs per ES) are sufficient to reach the lowest latency in the multithreaded setting and approach the single-threaded receiver latency. A single ULT per ES results in OS-level threading interoperability, but such an interoperability level is limited by the performance of the lock implementation. The extra benefits obtained from having more ULTs per ES can be obtained only through the interoperation of MPI with an expressive and lightweight runtime that can reduce synchronization costs and improve latency hiding, a feature of great importance for emerging hybrid MPI+threads applications.

5.3 Colocated I/O Services

This section demonstrates the flexibility of Argobots when leveraged by *colocated I/O services*: distributed I/O service daemons that are deployed alongside application processes. This service model can be used to provide dynamically provisioned, compute-node-funded services [41], in situ analysis and coupling services [42], or distributed access to on-node storage devices [43]. The key challenge in this programming model use case is that it must balance three competing goals: programmability (i.e., ensuring that the service itself is easy to debug and maintain), performance for concurrent workloads, and minimal interference with colocated applications.

The most straightforward way to utilize Argobots within an I/O service daemon is to create a new ULT to service each incoming I/O request. Unlike conventional OS-level threads, ULTs are inexpensive to create and consume minimal resources while waiting for a blocking I/O operation. Each ULT can cooperatively yield when appropriate so that other ULTs (i.e., concurrent requests) can make progress, thereby enabling a high degree of I/O operation concurrency with minimal resource consumption. This architecture is designed to realize the performance advantages of an event-driven model while retaining the programmability advantages of a conventional thread model.

We implemented two small extension libraries to help support this use case. The first, *abt-io*, provides thin wrappers for common POSIX I/O function calls such as `open()`, `pwrite()`, and `close()`. From the caller's perspective, these wrappers behave exactly like their native POSIX counterparts. Internally, the wrappers delegate blocking system calls to a separate Argobots pool as shown in Figure 12. The calling ULT is suspended while the I/O operation is in progress, thereby allowing other service threads to make progress until the I/O operation completes.

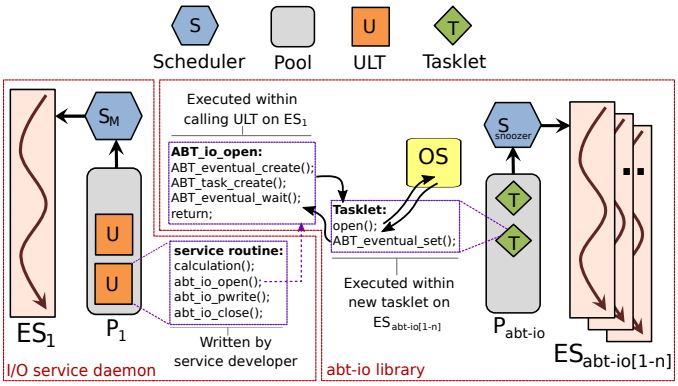


Fig. 12: The abt-io library architecture. Conventional POSIX I/O function calls such as `open()` would block progress of all ULTs on an execution stream. The abt-io library avoids this by delegating these operations to a separately provisioned pool of execution streams, thereby allowing the caller to yield until the operation is completed.

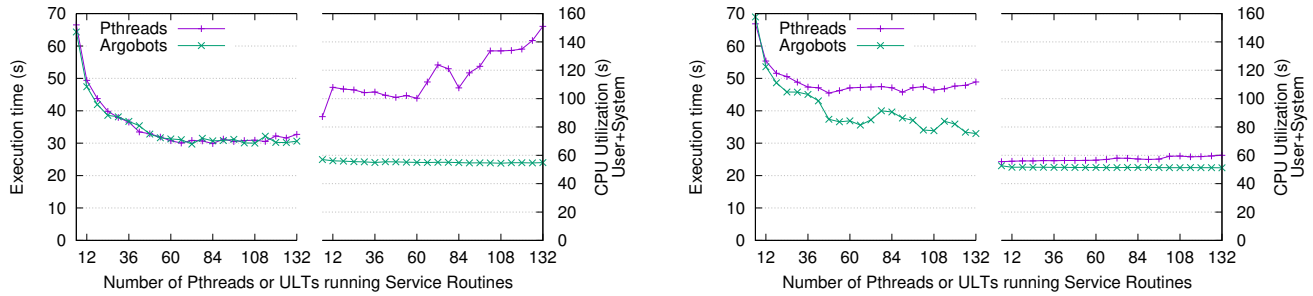
The delegation step is implemented by spawning a new tasklet that coordinates with the calling ULT via an `eventual`, an Argobots future-like synchronization construct. The tasklets are allowed to block on system calls because they are executing on a dedicated pool that has been designated for that purpose. This division of responsibility between a request servicing pool and an I/O system call servicing pool can be thought of as a form of I/O forwarding that allows I/O resources to be provisioned independently without interfering with execution of the primary application routine. This same technique could be applied to any blocking I/O resource. If the I/O resource provides a native asynchronous API (such as the Mercury RPC library [44]), then one need not delegate operations to a dedicated pool; the resource can use its normal completion notification mechanism to signal eventuals.

The second extension library, *abt-snoozer*, implements an I/O-aware Argobots scheduler that causes the ES to block (i.e., sleep) when no work units are eligible for execution and wake up when new work units are inserted. It therefore exchanges a modest latency cost for the ability to idle gracefully when ULTs are waiting for external I/O events, which in turn minimizes interference with other tasks. The scheduler can use the `epoll()` system call to block, and the pool can `write()` to an `eventfd()` file descriptor to notify it when new work units are added. The abt-snoozer library uses the `libev` [45] event loop and asynchronous event watchers to abstract this functionality for greater portability. The abt-io library does not require the use of the abt-snoozer scheduler, but it reduces resource consumption for workloads in which the I/O pool is sometimes idle.

We implemented a synthetic I/O service daemon to serve as a benchmark for empirical comparison of Pthreads and Argobots. The benchmark concurrently executes multiple instances of the service routine shown in Listing 2. The

```
1 calculation(buffer);
2 fd = open(path);
3 pwrite(fd, buffer);
4 close(fd);
```

Listing 2: Benchmark service routine pseudocode.



(a) The benchmarks use all 12 CPU cores.

(b) The benchmarks are constrained to two CPU cores.

Fig. 13: Execution time and CPU utilization for a synthetic benchmark that represents the workload of a colocated I/O service.

service routine contains a sequence of computation, metadata, and I/O steps that are carried out to service each client request. The calculation step in a real-world service daemon may include checksumming, compression, or parity encoding; but in the synthetic benchmark we represent it with `RAND_bytes` function from libcrypto [46], which fills the buffer with a random sequence of bytes. I/O is performed in synchronous, direct I/O mode. In the Pthreads version of the benchmark, a dedicated Pthread is assigned to execute each service routine in its entirety. The Argobots version of the benchmark differs by executing each service routine in a ULT rather than a Pthread and using abt-io wrappers together with the abt-snoozer scheduler to perform I/O.

We executed the benchmark on a 12-core, 2.4 GHz E5-2620 compute node containing a pair of mirrored Seagate ST9500620NS (500 GiB SATA) disk drives. The benchmark was configured to execute 2,048 ULTs, with each ULT processing 1 MiB of data. Therefore, in aggregate it produced and wrote 2 GiB of random data.

Figure 13a shows the results of executing this experiment as we vary the number of Pthreads or ULTs that are allowed to execute simultaneously. In the Pthreads case, the number of threads determines not only the request servicing concurrency but also the compute concurrency and I/O concurrency. Those three parameters cannot be tuned independently. In the Argobots case, the number of threads determines only the request servicing concurrency. The ULTs are executed on a pool shared with 4 ESs (i.e., the desired level of CPU concurrency), and the abt-io tasklets are executed on a pool shared with 36 ESs (i.e., the desired level of I/O concurrency) in all cases. Argobots provides the unique ability to tune these parameters independently without altering the actual ULT service routines.

In the left portion of Figure 13a we see that the Pthreads and Argobots implementations achieve similar performance. Both improve as more concurrent threads are used, until roughly 60 threads are engaged. In the right portion of the graph, however, we see that the Pthreads version consumes significantly more CPU time to achieve this level of performance. The discrepancy grows as more threads are used, because of higher context switching cost and OS overhead in the Pthreads implementation. This is a key metric for I/O services that will be colocated with applications because it directly impacts how much CPU time is available to application processes. We measured the CPU time using the GNU `time` command line utility to collect the number of CPU-seconds consumed by the benchmark itself ("User" time) plus the CPU-seconds consumed by the operating system on behalf of the process ("System" time).

Figure 13b shows the outcome of the same experiment when the Linux `taskset` utility is used to constrain the benchmark to use only the first 2 of 12 cores. This configuration reflects a deployment scenario in which the I/O service is pinned to dedicated cores in order to avoid interfering with application tasks. Performance is degraded slightly in comparison with Figure 13a, but the Pthreads variant is more severely impacted, in some cases taking nearly 8 seconds longer to complete the benchmark. The Pthreads implementation also continues to consume more CPU time than the Argobots implementation does, even though the total CPU consumption is capped by the number of cores assigned to the service.

Overall, Argobots maintains programmability (by expressing I/O service routines as straightforward sequential functions), achieves performance competitive with that of Pthreads, and produces consistently lower resource consumption to minimize interference with co-located application tasks. We note that the Pthreads service implementation could likely be optimized with a more sophisticated threading model (for example, offloading I/O work to a dedicated thread pool). Doing so, however, would require decomposing the service routines into smaller discrete event-driven routines with disjoint stacks, a technique known as *stack ripping* [10]. By maintaining a sequential control flow in each service routine, we significantly reduce the development, debugging, and maintenance burden for system services [47], [48]. The Argobots model accomplishes these tasks while also enabling fine-grained division of work, customizable scheduling policies, and interoperability with a variety of application programming models.

6 Related Work

We discuss here related work in *generic* threading runtimes and *specialized* ones for on-node concurrency.

In the generic runtimes category, we find several threading and tasking packages developed as independent libraries similar to Argobots. Some libraries, such as GnuPth [1] and StackThreads [2], provide ULTs but only within a single OS-level thread. Recent packages, such as Marcel [3], MassiveThreads [4], Qthreads [5], TBB [8], and StackThreads/MP [6], allow scheduling of ULTs on multiple OS-level threads. MPC [9] uses ULTs to support MPI and

OpenMP and provides a lightweight Pthreads interface. While these packages are generic and often efficiently execute certain type of algorithms, such as divide-and-conquer, they provide little control to the user. They often handle scheduling transparently, hide thread pools from the user, and give no control over stack and context-switch requirements of work units. Converse [19], which is still being used as a ULT subsystem in Charm++ [49], inspired the design of Argobots. It was one of the early systems to support ULT abstraction separated from its scheduler and to support scheduling of tasklets and ULTs via a common scheduler. However, it lacks several features—for instance, stackable schedulers, pluggable strategies, ULT migration, and scheduler bypass–and thus is less flexible than Argobots.

Several works exist in the specialized category, given the vast possible environments that require lightweight execution abstractions. Some operating systems provide lightweight threading alternatives to OS-level threads, such as Windows fibers [10] and Solaris threads [11]. Capriccio [12], StateThreads [13], and Li and Zdancewic's work [14] rely on ULTs to handle concurrent network services. Maestro [15] is the target of a high-level language compiler, and TiNythreads [16] is specialized to map lightweight software threads to hardware thread units in the Cyclops64 cellular architecture. These works, however, offer little control to the user and are not portable outside the environments they were meant for; we expect significant efforts will be needed in order to make them portable and available for generic use.

Other lightweight thread packages are tightly coupled with their target parallel programming systems. The Nanos++ runtime [18] provides ULTs that are used to implement task parallelism in OmpSs [50]. The Realm runtime [20] of Legion [51] utilizes ULTs for its event-based tasking model. HPX-5 [52] exposes ULTs for fine-grained execution. Lithe [53] exploits ULTs to support multiple contexts in a single hardware thread. These threading abstractions are heavily optimized for the target programming systems. For instance, threading runtimes under OpenMP compilers, such as Nanos++, offer means to efficiently schedule loop iterations and tasks and to map execution streams to processing units. They also exploit the semantics of the programming system to avoid stack allocation and frequently context switching (e.g., iteration loops executed by multiple OpenMP threads without allocating a stack for each iteration). Because of their lack of generic abstractions, however, these runtimes are hardly usable outside the scope of their programming systems.

From a different perspective, because of the extremely lightweight nature of its work units and given the rich set of capabilities that it offers, Argobots could be positioned at the lowest level in the software stack. We provide Figure 14 as a summary. In other words, all the cited related work can also target Argobots as their underlying runtime. Moreover, several programming languages, such as Cilk [29], X10 [54], Habanero-C [55], Chapel [56], Go [57], and Python [58], can also target Argobots. Cilk, for instance, can map threads to Argobots ULTs similar to the OpenMP example in Section 5.1. Arguably, Argobots, unlike runtime systems such as Realm and HPX, targets exclusively on-node concurrency and does not address multinode execution. As exemplified by the MPI integration in Section 5.2, however, Argobots offers powerful abstractions to efficiently interoperate with internode communication runtimes.

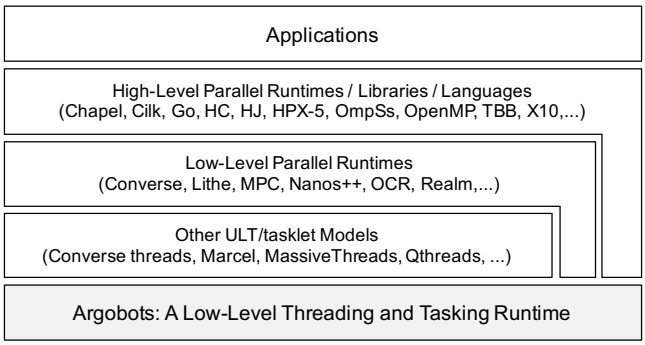


Fig. 14: Argobots as a low-level threading and tasking runtime.

7 Conclusions

We presented Argobots, a lightweight low-level threading and tasking framework that offers powerful capabilities for users to allow efficient translation of high-level abstractions to low-level implementations. We demonstrated that Argobots can outperform state-of-the-art generic lightweight threading libraries. We also presented integration of Argobots with OpenMP and MPI, the most widely adopted programming systems in high-performance computing, as well as colocated I/O services. We showed that our OpenMP runtime over Argobots handles nested parallelism better than existing runtimes do and that an MPI runtime that interoperates with Argobots offers more synchronization-reducing and latency-hiding opportunities than does the commonly adopted interoperation with Pthreads. We also demonstrated that an I/O service with Argobots can manage hardware resources more efficiently and reduce interference with colocated applications better than does such a service with Pthreads.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided by the Laboratory Computing Resource Center at Argonne National Laboratory. The researcher from Universitat Jaume I was supported by Generalitat Valenciana fellowship programme Vali+d 2015.

References

[1] R. S. Engelschall, "GNU portable threads (Pth)," http://www.gnu.org/software/pth, 1999.
[2] K. Taura and A. Yonezawa, "Fine-grain multithreading with minimal compiler support – a cost effective approach to implementing efficient multithreading languages," in *PLDI*, 1997, pp. 320–333.

[3] S. Thibault, "A flexible thread scheduler for hierarchical multiprocessor machines," in *COSET*, 2005.

[4] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond*, 2014, pp. 222–238.

[5] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *MTAAP*, 2008.

[6] K. Taura, K. Tabata, and A. Yonezawa, "StackThreads/MP: Integrating futures into calling standards," in *PPPoP*, 1999, pp. 60–71.

[7] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *SenSys*, 2006, pp. 29–42.

[8] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[9] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *EuroPar*, 2008, pp. 78–88.

[10] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management," in *ATC*, 2002.

[11] C. SunSoft, "Solaris multithreaded programming guide," 1995.

[12] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *SOSP*, 2003, pp. 268–281.

[13] G. Shekhtman and M. Abbott, "State threads library for internet applications," 2009, http://state-threads.sourceforge.net/.

[14] P. Li and S. Zdancewic, "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives," in *PLDI*, 2007, pp. 189–199.

[15] A. Porterfield, N. Nassar, and R. Fowler, "Multi-threaded library for many-core systems," in *MTAAP*, 2009.

[16] J. d. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "TiNy threads: A thread virtual machine for the Cyclops64 cellular architecture," in *WMPP*, 2005.

[17] "Intel OpenMP runtime library," https://www.openmprtl.org/, 2016.

[18] "Nanos++," 2016, https://pm.bsc.es/projects/nanox/.

[19] L. V. Kalé, J. Yelon, and T. Knuff, "Threads for interoperable parallel programming," in *LCPC*, 1996, pp. 534–552.

[20] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *PACT*, 2014, pp. 263–276.

[21] R. S. Engelschall, "Portable multithreading - the signal stack trick for user-space thread creation," in *ATC*, 2000.

[22] "Boost.Context," http://www.boost.org/doc/libs/1_57_0/libs/context/, 2009.

[23] "PAPI: Performance Application Programming Interface," http://icl.cs.utk.edu/papi/, 2016.

[24] S. Seo, J. Kim, and J. Lee, "SFMalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores," in *PACT*, 2011, pp. 253–263.

[25] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *ASPLOS*, 2000, pp. 117–128.

[26] "Transparent Hugepage Support," https://www.kernel.org/doc/Documentation/vm/transhuge.txt, 2016.

[27] K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" in *SOSP*, 2013, pp. 133–150.

[28] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *ROSS*, 2011, pp. 49–56.

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

[30] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IPDPS*, 2009, pp. 1–12.

[31] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR*, 2014.

[32] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith, "OpenMC: A state-of-the-art Monte Carlo code for research and development," *Annals of Nuclear Energy*, vol. 82, pp. 90–97, 2015.

[33] "GOMP: An OpenMP implementation for GCC," https://gcc.gnu.org/projects/gomp/, 2015.

[34] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa, "Performance evaluation of OpenMP applications with nested parallelism," in *LCR*, 2000, pp. 100–112.

[35] P. E. Hadjidoukas and V. V. Dimakopoulos, "Nested parallelism in the OMPI OpenmP/C compiler," in *EuroPar*, 2007, pp. 662–671.

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

[37] "LLVM OpenMP project," http://openmp.llvm.org/, 2015.

[38] A. Amer, N. Maruyama, M. Pericàs, K. Taura, R. Yokota, and S. Matsuoka, "Fork-join and data-driven execution models on multi-core architectures: Case study of the FMM," in *ISC*, 2013, pp. 255–266.

[39] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi*, 2014, pp. 167–188.

[40] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "MPI+threads: Runtime contention and remedies," in *PPoPP*, 2015, pp. 239–248.

[41] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "DeltaFS: Exascale file systems scale better without dedicated servers," in *PDSW*, 2015, pp. 1–6.

[42] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An interaction and coordination framework for coupled simulation workflows," in *HPDC*, 2010, pp. 25–36.

[43] Argonne Leadership Computing Facility, "Aurora," http://aurora.alcf.anl.gov/, 2016.

[44] 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 *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2013, pp. 1–8.

[45] M. Lehmann, "libev," http://software.schmorp.de/pkg/libev.html, 2016.

[46] OpenSSL Software Foundation, "OpenSSL Cryptography and SSL/TSL Tookit," https://www.openssl.org/docs/manmaster/crypto/crypto.html, 2016.

[47] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *HotOS*, 2003.

[48] D. Kimpe, P. Carns, K. Harms, J. M. Wozniak, S. Lang, and R. Ross, "AESOP: Expressing concurrency in high-performance system software," in *NAS*, 2012, pp. 303–312.

[49] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *OOPSLA*, 1993, pp. 91–108.

[50] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of GPU clusters with OmpSs," in *26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012, pp. 557–568.

[51] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC*, 2012, pp. 66:1–66:11.

[52] G. R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu, "ParalleX: A study of a new parallel computation model," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–6.

[53] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with lithe," in *PLDI*, 2010, pp. 376–387.

[54] 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 *OOPSLA*, 2005, pp. 519–538.

[55] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *IPDPS*, 2013, pp. 712–725.

[56] B. L. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *IJHPCA*, vol. 21, no. 3, pp. 291–312, 2007.

[57] F. Schmager, N. Cameron, and J. Noble, "GoHotDraw: Evaluating the Go programming language with design patterns," in *PLATEAU*, 2010, pp. 10:1–10:6.

[58] C. Tismer, "Continuations and stackless Python," in *Proceedings of the 8th International Python Conference*, vol. 1, 2000.

Sangmin Seo is an assistant computer scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. He received his B.S. degree in computer science and engineering and his Ph.D. degree in electrical engineering and computer science from Seoul National University in 2007 and 2013, respectively.

Abdelhalim Amer is a postdoctoral appointee in the Mathematics and Computer Science Division at Argonne National Laboratory. His research falls generally under the parallel and distributed computing landscape.

Pavan Balaji is a computer scientist at Argonne National Laboratory, a fellow of the Northwestern-Argonne Institute of Science and Engineering at Northwestern University, and a fellow of the Computation Institute at the University of Chicago. He leads the programming models and runtime systems group at Argonne. His research interests include parallel programming models and runtime systems for communication and I/O on extreme-scale supercomputing systems, modern system architecture, cloud computing systems, data-intensive computing, and big data science.

Cyril Bordage is a postdoctoral researcher in the team Tadaam at Inria Bordeaux Sud-Ouest in France. He received his M.Sc. degree in 2009 and his Ph.D. degree in 2013 in computer science from the University of Bordeaux.

George Bosilca is a research director and adjunct assistant professor at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. His research interests evolve around the concepts of distributed algorithms, parallel programming paradigms, and software resilience, from both a theoretical and practical perspective.

Alex Brooks is a Ph.D. candidate at the University of Illinois at Urbana-Champaign. He received his B.A. degree in computer science and mathematics from Monmouth College in 2013. His current research interests include parallel programming models, hardware acceleration, and threading/communication interoperability.

Philip Carns is a principal software development specialist in the Mathematics and Computer Science Division at Argonne National Laboratory, a fellow of the Northwestern-Argonne Institute for Science and Engineering, and an adjunct associate professor of electrical and computer engineering at Clemson University. He received his Ph.D. degree in electrical and computer engineering from Clemson University in 2005.

Adrián Castelló is a Ph.D. student in the Departamento de Ingeniería y Ciencia de los Computadores at Universitat Jaume I de Castelló. He received his B.S. degree in computer science and M.S. degree in advanced computer systems from Universitat Jaume I in 2009 and 2011, respectively.

Damien Genet is a postdoctoral researcher at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. He received his Ph.D. degree in 2014 from the University of Bordeaux, France. His focus is on parallel programming paradigms for distributed applications.

Thomas Herault is a research scientist at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. He received his Ph.D. degree from the University of Paris XI, France in 2003. His research interests include fault tolerance, performance modelings, and programming models for distributed algorithms with emphasis on HPC.

Shintaro Iwasaki Shintaro Iwasaki is a Ph.D. candidate at the University of Tokyo in Japan. He received his B.S. and M.S. degrees from the University of Tokyo in 2015 and 2017, respectively. His current research interests include parallel languages, compilers, runtime systems, and scheduling techniques.

Prateek Jindal received his Ph.D. degree in computer science from the University of Illinois at Urbana-Chanpaign in 2013. Then he worked on big data technologies as a software engineer in Yahoo. Subsequently, he joined UIUC as a postdoc where he contributed to the research mentioned in this paper.

Laxmikant V. Kalé received his B.Tech. degree in electronics engineering from Benares Hindu University, India, in 1977; his M.E. degree in computer science from the Indian Institute of Science in Bangalore, India, in 1979; and the his Ph.D. degree in computer science from the State University of New York, Stony Brook, in 1985. He is a full professor at the the University of Illinois at Urbana-Champaign. His current research interests include parallel computing. He is a fellow of the IEEE.

Sriram Krishnamoorthy is a research scientist and the System Software and Applications Team Leader in PNNL's High Performance Computing group. His research interests include parallel programming models, fault tolerance, and compile-time/runtime optimizations for high-performance computing.

Jonathan Lifflander is a research staff at Sandia National Laboratories. He received his PhD degree from the University of Illinois at Urbana-Champaign in 2016.

Huiwei Lu is a senior software engineer at Tencent. He was a postdoctoral appointee in the Mathematics and Computer Science Division at Argonne National Laboratory in 2015. He received his MS and Ph.D. degrees in Computer Architecture from Institute of Computing Technology in 2013.

Esteban Meneses leads the Advanced Computing Laboratory at the Costa Rica National High Technology Center. He also holds a partial appointment at the Costa Rica Institute of Technology. His research interests include fault tolerance and programming models for high-performance computing.

Marc Snir is Michael Faiman Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He is an AAAS Fellow, ACM Fellow, and IEEE Fellow. He has an Erdos number 2 and is a mathematical descendant of Jacques Salomon Hadamard. He recently won the IEEE Award for Excellence in Scalable Computing and the IEEE Seymour Cray Computer Engineering Award.

Yanhua Sun is currently a software engineer working at Google Inc. She received her Ph.D. degree from the University of Illinois at Urbana-Champaign in 2015. Her research interests include parallel programming models, communication optimization, parallel runtime adaptivity, parallel performance analysis and tuning, and molecular dynamics applications.

Kenjiro Taura is an associate professor at the Department of Information and Communication Engineering, University of Tokyo. He received his B.S., M.S., and D.Sc. degrees from the University of Tokyo in 1992, 1994, and 1997, respectively. His major research interests are centered on parallel/distributed computing and programming languages. His expertise includes efficient dynamic load balancing, parallel and distributed garbage collection, and parallel/distributed workflow systems. He is a member of ACM and IEEE.

Pete Beckman is the co-director of the Northwestern University / Argonne Institute for Science and Engineering (NAISE). He leads the Extreme Computing group at Argonne National Laboratory.