Argobots: A Lightweight, Low-Level Threading and Tasking Framework

Sangmin Seo* Abdelhalim Amer* Pavan Balaji* Cyril Bordage† George Bosilca‡
Alex Brooks† Adrián Castelló§ Damien Genet‡ Thomas Herault‡ Prateek Jindal†
Laxmikant V. Kalé† Sriram Krishnamoorthy¶ Jonathan Lifflander† Huiwei Lu‖
Esteban Meneses** Marc Snir* Yanhua Sun† Pete Beckman*

* Argonne National Laboratory, {sseo,aamer,balaji,snir,beckman}@anl.gov
† University of Illinois at Urbana-Champaign, {cbordage,brooks8,jindal2,kale,jliffl2,sun51}@illinois.edu
‡ University of Tennessee, Knoxville, {bosilca,dgenet,herault}@icl.utk.edu
§ Universitat Jaume I, adcastel@uji.es ¶ Pacific Northwest National Laboratory, sriram@pnnl.gov
‖ Social Network Group, Tencent, huiweilv@tencent.com ** Costa Rica Institute of Technology, emeneses@ic-itcr.ac.cr

***Abstract*—In this paper, we present a lightweight, low-level threading and tasking framework, called Argobots, to support massive on-node parallelism. Unlike other threading models, Argobots provides users with efficient threading and tasking mechanisms, not policies, so that users can develop their own solutions. To achieve this goal, Argobots supports two kinds of work units: user-level threads and tasklets. The former have an associated stack and allow blocking calls, while the latter do not but provide fast context switching. Argobots also exposes hardware resources (e.g., cores) as execution streams (ESs) and provides mapping mechanisms between work units and ESs. Moreover, Argobots supports lower-level control of scheduling and stackable scheduling with pluggable strategies. Along with the design, this paper describes the implementation and optimizations of Argobots. The evaluation results with benchmarks on many-core machines show that Argobots incurs low overhead with sustainable and scalable performance and enables users to develop their own efficient solutions.**

I. Introduction

The number of CPU cores used in high-performance computing (HPC) systems has been increasing steadily. Figure 1 shows that supercomputers in the Top500 list [1] are adopting more cores per socket (bar) and that the total number of cores in the first-ranked supercomputer has increased during the past 11 years (line), with the current #1 supercomputer utilizing more than 3 million cores. Based on this trend, we envision that exascale systems are likely to comprise hundreds of millions of arithmetic units. Accordingly, future applications on such systems are expected to use billions of threads or tasks to exploit the massive concurrency supported.

Achieving billion-way parallelism requires highly dynamic computational and data scheduling as well as lightweight threading and tasking methods. Supporting these with traditional threading models is likely to be difficult, however, because of their heavyweight context mechanism. Moreover, the context switching overhead is noticeable [2]: creating and joining an OS-level thread involve significant overhead, especially when the number of threads needed to be created is larger than the number of physical cores (i.e., oversubscription). For instance, when 36 Pthreads on a 36-core machine (Section IV) create a number of Pthreads and join them, the average time for creating and joining one Pthread increases from 245 μs (one Pthread per Pthread, 36 Pthreads in total)

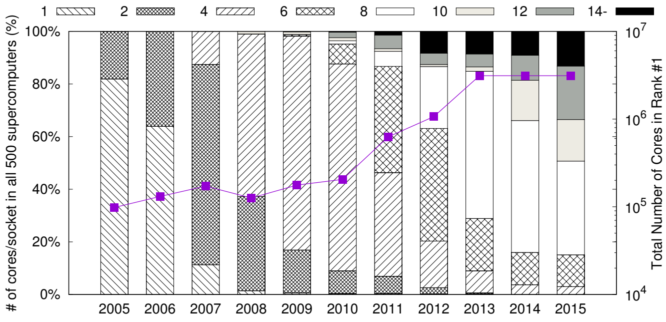


Fig. 1: Distribution of number of cores per socket in all 500 supercomputers (bar) and total number of cores in the #1 supercomputer (line) from the past 11-year Top500 lists.

to 2,645 μs (10 Pthreads per Pthread, 360 Pthreads in total), while the average time for creating and joining one user-level thread (ULT) stays at around 150 ns even though the number of ULTs created is increased. The time for context switching is on average 1.64 μs for Pthreads and 60 ns for ULTs. Consequently, with conventional threading models, programmers always must be concerned about the number of threads to be created; but expressing massive parallelism with a limited number of threads can be challenging.

Increasingly, dynamically scheduled asynchronous work units (i.e., tasks) are seen as being beneficial for dynamic adaptivity by HPC researchers. Two types of work units are common: *user-level threads* and *tasklets*. ULTs have a stack associated with them, while the tasklets do not. Tasklets allow for rapid context switching but are not allowed to block. ULTs allow such blocking and are only slightly slower than tasklets. Support for both types ensures that a wide variety of asynchronous models (sometimes called "tasking models") can interoperate while even interleaving their execution under the control of a common runtime system.

Unfortunately, many existing or proposed lightweight threading models support *either* ULTs *or* tasklets. They try to provide users with programming interfaces similar to conventional threading models while supporting low-overhead context switching between work units. We believe that attempting to meet the requirements of future applications with existing lightweight threading models is not ideal because their smart policies (e.g., work-stealing scheduling) sometimes conflict with the characteristics and demands of applications.

In this paper, we present a lightweight, low-level threading and tasking framework, called Argobots, to support the massive parallelism required for applications on exascale systems. Unlike other lightweight threading models [3], [4], the design goal of Argobots is to provide users with efficient threading and tasking mechanisms, not policies, so that users can develop their own solutions. Argobots defines abstractions for both ULTs and tasklets and supports them with a robust and lean implementation. Argobots provides mapping mechanisms between the work units and each hardware resource (e.g., a core) called an "execution stream" (ES). Scheduling of these work units presents another challenge. To enable flexibility in usage as well as to minimize scheduling costs, Argobots supports an innovative notion of *stackable schedulers* with pluggable queuing strategies. Specifically, by allowing stacking of schedulers associated with each ES, we can use a general scheduler in most cases but switch to a low-overhead but less-general scheduler when needed. Along with the design, this paper describes the implementation and optimizations of Argobots as a user-level runtime system. The evaluation results with microbenchmarks and applications on many-core architectures show that Argobots incurs low overhead with sustainable and scalable performance and enables users to write their own efficient solutions.

The major contributions of this paper are as follows.

- The Argobots execution model design offers an extreme level of flexibility and control to higher level runtimes and programming models by leveraging state-of-the-art concepts and optimizations in a common framework. This flexibility allows users to translate higher-level abstractions into efficient low-level implementations.
- A lightweight design can be heavy if not implemented efficiently. We performed an in-depth analysis that involved investigating every cache miss and TLB miss that occurs in the critical path. This led to implementation optimizations and API extensions that achieved unprecedented performance in the context of lightweight runtimes.
- We evaluate Argobots on a 36-core machine using microbenchmarks and compare its performance and scalability with those of other lightweight threading libraries, such as Qthreads [3] and MassiveThreads [4]. This evaluation shows that Argobots imposes little overhead and scales better than other libraries while achieving sustainable performance.
- We conduct a performance study with three applications. The results show that the Argobots versions perform better than the original applications because Argobots enables more efficient implementations thanks to its lightweight mechanisms.

II. Related Work

ULTs, also called *coroutines* or *fibers*, are usually referred to as lightweight threads with low context-switching overhead. The ULT has thread semantics similar to the semantics of the OS-level thread but running in the user space. In addition, more than one ULT can be mapped to a single OS-level thread. Hence, ULTs may not be executed concurrently, and cooperative scheduling may be required in order to execute multiple ULTs in an interleaved manner. Compared with conventional threads (e.g., Pthreads), ULTs are more suitable for expressing massive parallelism and for overlapping computation and communication (or I/O) because of their lightweight context mechanism.

To leverage these benefits of ULTs, researchers have proposed various threading models, as well as OS supports such as Windows fibers [5], [6] or Solaris threads [7]. Converse threads [8] are designed for the Converse framework [9] as a threading subsystem and support both ULTs and tasklets, incorporating a hierarchical scheduling model. Argobots is highly motivated by the Converse threads, but it delivers more flexible and deterministic threading and tasking primitives by allowing users to control every detail of Argobots. Qthreads [3] provides a large number of ULTs with full-/empty-bit semantics; a ULT can wait for any word of memory until it is marked either full or empty. MassiveThreads [4] is a lightweight thread library focusing on scheduling recursive task parallelism. Maestro [10] provides lightweight threads and synchronization operations, but it is designed to be the target of a high-level language compiler or source-to-source translator. Nanos++ runtime [11] provides ULTs that are used to implement task parallelism in OmpSs [12]. GnuPth [13] supports ULTs on a single kernel-space thread while focusing on portability. StackThreads [14] provides multithreads only within a single processor; StackThreads/MP [15] extends this capability by supporting dynamic thread migration on shared-memory multiprocessors. Marcel [16] is enhanced with hierarchical scheduling of ULTs on NUMA machines. On the other hand, Stackless Python [17] and Protothreads [18] are more focused on stackless threads, that is, tasklets.

Lightweight threads have also been utilized for special purposes, especially for hiding I/O or communication latency. Capriccio [19] is a ULT package for high-concurrency servers, such as the Apache web server; however, it supports multiple ULTs only in single-threaded applications, not in a multithreaded environment. StateThreads [20] provides a threading API for writing Internet applications, such as web servers or proxy servers, with an event-driven state machine architecture. Li and Zdancewic [21] combined the lightweight threading model and the event-driven model to build massively concurrent network services. MPC [22] exploits ULTs to deal with communications and synchronizations. TiNy-threads [23] is specialized to map lightweight software threads to hardware thread units in the Cyclops64 cellular architecture.

Scheduler activations [24] and dispatchers in K42 [25] are similar to ESs in that they virtualize hardware resources and ULTs are mapped to them when executed. However, their goal is to avoid blocking in a scheduler activation (or dispatcher) by creating a new scheduler activation and switching to it by preemption in order to execute a different ULT. They provide kernel interfaces to ULT libraries for this. On the other hand, the Argobots ES does not interact with the OS kernel to avoid blocking. Instead, it relies on cooperative multitasking between ULTs (i.e., the ULT has to voluntarily yield when it blocks). Therefore, the ideas presented in [24], [25] are orthogonal and complementary to our work.

The main difference between Argobots and other approaches is that Argobots is designed primarily to be an underlying threading and tasking runtime for high-level runtimes or libraries. It provides low-level primitives with which we can even build other ULT libraries, whereas current ULT libraries cannot do so or lose important features in the attempt. First, Argobots exposes two levels of parallelism, ESs and work units, that can give a better chance to optimize locality and to deterministically schedule work units. Arguably, explicitly mapping ESs and work units may present a burden to new users, but it can enable advanced users to better control the locality and the scheduling by precisely assigning work units to specific ESs. In addition, unlike other models, Argobots seeks to provide efficient mechanisms, not policies, for users to develop their own solutions. Argobots also supports lower-level control of scheduling and stackable scheduling framework with pluggable strategies. Since this approach prevents Argobots from conflicting with upper-layer runtimes, it enhances the sustainability and stability of performance. We believe that all these efforts make Argobots a better fit for various high-level runtimes or domain-specific libraries.

In Section V we compare Argobots with popular Qthreads and MassiveThreads in terms of performance and scalability because they are among the best-performing lightweight threading models currently used in the HPC community. Moreover, these are available as independent libraries that are not integrated in the programming model runtimes, and their performance was well studied in previous works [3], [4].

III. Design and Implementation

This section presents the design and implementation of Argobots.

A. Execution Model

Figure 2 illustrates the execution model of Argobots. Argobots explicitly supports two levels of parallelism: ESs and work units. An ES is a sequential instruction stream that consists of one or more work units. When an ES is bound to a hardware processing element (PE), it can also be regarded as a software-equivalent or OS-level thread. ESs are explicitly created, and each ES is executed independently. ESs have implicitly managed progress semantics, which guarantees that one blocked ES cannot block other ESs. A work unit is a lightweight execution unit, such as a ULT or tasklet, and gets associated with a specific ES when it is running. There is no concurrent execution of work units in a single ES, and thus only one work unit runs in an ES at a certain point. However, work units in different ESs can be executed concurrently. 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 scheduler.

ULTs and tasklets are associated with function calls and execute to completion. However, they differ in subtle aspects that make each of them better suited for some programming motifs. For example, a ULT has its own persistent 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 pauses and

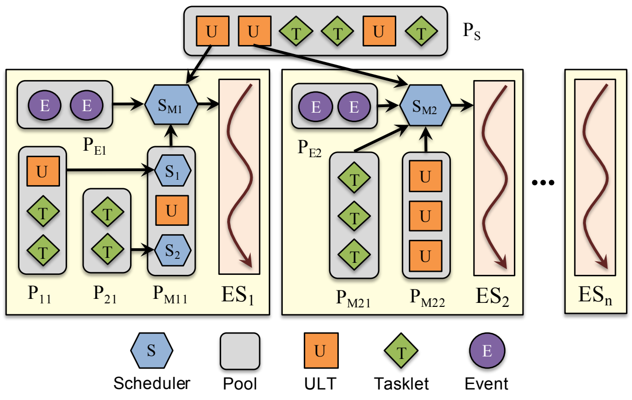


Fig. 2: Argobots execution model.

resumes based on the flow of data. A common example is an overdecomposed application that uses blocking receives to wait for remote data. Unlike OS-level threads, ULTs are not intended to be preempted. They cooperatively yield control, for example, when they wait for remote data or just let other work units make progress. When ULTs run in an ES, their execution may be interleaved inside an ES because they can yield control to the scheduler or another ULT. 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 explicitly yield control but run to completion before returning control to the scheduler that invoked them.

The explicit management of ESs and work units differentiates Argobots from other ULT libraries [3], [4], [10]. Instead of merely relying on the underlying scheduler of a thread library, users of Argobots can control which work units can run concurrently by managing the mapping between ESs and work units. This low-level control enhances cooperative multitasking because work units that involve much communication and need frequent context switch can be mapped together to ESs. In addition, work units that are compute bound and do not benefit from cooperative multitasking can run in an ES without frequent context switching. Moreover, this approach enables users to easily manage the data locality of work units.

B. Scheduler

The design principle for the scheduler is to provide a framework for stackable or nested schedulers with pluggable strategies while exploiting the cooperative, nonpreemptive activation of work units. Localized scheduling strategies such as those used in current runtime systems, while efficient for short execution, are unaware of global strategies and priorities. Adaptability and dynamic system behavior must be handled by scheduling strategies that can change over time or be customized for a particular algorithm or data structure. Argobots supports plugging in custom strategies so that higher levels of the software stack can use their special policies while Argobots handles the low-level scheduling mechanism.

The Argobots scheduling framework also permits stacking schedulers specific to a programming model or an application component. For example, the framework can accept a scheduler for each module in an application; and based on dependencies or relative module priorities, the higher-level scheduler may invoke one of the stacked schedulers. Doing so activates work units associated with the invoked scheduler on the managed hardware and yields control back upon completion. Similarly, stacked schedulers might be a result of multiple programming models interacting in the context of Argobots.

Argobots allows each ES to have its own schedulers. To execute work units, an ES has at least one main scheduler, denoted by S_M in Figure 2. A scheduler is associated with one or more *pools* where ready ULTs and tasklets are waiting for their execution. In the figure, for instance, S_{M1} in ES_1 has one associated local pool, P_{M11}, and S_{M2} in ES_2 has two local pools, P_{M21} and P_{M22}. 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. For example, P_S in the figure 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. Each ES also has a special event pool, called P_E, for asynchronous events. The event pool is for utilizing lightweight notification and is periodically checked by a scheduler to handle the arrival of events (e.g., messages from the network).

In Argobots, 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 synchronizations (see Section III-C). In order to ensure a particular affinity of a work unit to some data, the application simply needs to use the right pool when pushing the work unit. Thus, the work unit will be executed on the ES (or a group of ESs) that can pick from this pool.

Stacking schedulers is realized through pushing schedulers into a pool. In other words, schedulers in a pool are regarded as schedulable units in Argobots. For example, S_1 and S_2 in P_{M11} in Figure 2 are stacked schedulers, which will be executed by the main scheduler 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 of the completed scheduler. To give control back to the parent scheduler, a scheduler can also yield.

C. Primitive Operations

Argobots defines primitive operations for work units. Since tasklets are used for atomic work without blocking, 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 with the 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 created may be scheduled in more than one ES.

Join. ULTs and tasklets 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. Since Argobots does not adopt preemptive scheduling, ULTs must cooperatively yield control in order to enable progress of other work units. The scheduler, which receives the control from the ULT, schedules the next work unit according to its scheduling policy.

Yield_to. To reduce context switch overhead in the yield operation, Argobots provides the yield_to operation for ULTs. When a ULT calls yield_to, it yields control to a specific ULT instead of the scheduler. Since yield_to avoids the scheduler in the flow of control, it can eliminate the overhead of context switching to the scheduler and scheduling another ULT. This feature is useful when the user knows the exact ULT that needs to be executed after the current ULT. Yield_to can be used only among ULTs associated with the same ES.

Migration. Argobots supports migration of work units between different pools. Basically, all ULTs, which are created by the user, can be migrated unless they are terminated. A running ULT can be migrated when it yields. However, tasklets can be migrated only if they have not started the execution. The migration operation can be blocking or nonblocking depending on the request. And, if a callback function is set, it will be invoked when the migration happens.

Synchronizations. Mutex, condition variable, future, and barrier are supported. Only ULTs are expected to use these operations. *Mutex* enables mutual exclusion between ULTs. When more than one ULT competes for locking the same mutex, only one ULT is guaranteed to lock the mutex. Other ULTs are blocked and wait until the mutex is unlocked. *Condition variable* is a signal/wait synchronization mechanism. A ULT waits on a condition variable until it is signaled by another ULT. It is also possible that many ULTs wait on the same condition variable and a different ULT signals (broadcasts) to all waiting ULTs to wake them up. *Future* is a mechanism for passing a value between ULTs, allowing a ULT to wait for the value to be set asynchronously by another work unit. Argobots provides a general form of this mechanism that accompanies a number of compartments for the value. Each compartment is set by a contributing work unit, and ULTs waiting on a future blocks until all the compartments are set. *Barrier* enables multiple ULTs to wait until all of them reach the barrier.

D. Implementation

We have implemented Argobots in the C language as a user-level library and a low-level runtime so that high-level programming models or domain-specific languages can easily integrate Argobots into their runtime. An ES is implemented with Pthread and is bound to a hardware PE, for example, a CPU core or hardware thread. Argobots considers one ES per PE because oversubscription of ESs is not recommended.

ULTs are implemented by using user context mechanisms, such as `ucontext`, `setjmp/longjmp` with `sigaltstack` [26], or Boost library's `fcontext` [27], which provide means to create a user context and to switch between different user contexts. The user context includes CPU registers, stack pointer, and instruction pointer. Our implementation exploits `fcontext` by default. When a ULT is created, we create a ULT context that contains a user context, 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. As described in Section III-A, tasklets are executed by using the scheduler's stack space.

A pool is a container data structure that can hold a set of work units and provide operations for insertion and deletion. Argobots defines the interface required to implement a pool, and our implementation provides a first-in, first-out queue as a pool implementation. The pool has a property for access mode; and depending on its access mode, it can be shared between different ESs or private to a single ES.

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

In order to compensate for the waiting time spent on blocking operations, a ULT that has called a blocking Argobots operation is context switched. For example, when a ULT tries to lock a mutex, if the mutex has already been locked by another ULT, the caller ULT has to wait until the mutex is unlocked. Instead of waiting, the caller ULT is context switched to the scheduler. Similarly, when a ULT tries to join another ULT, the caller ULT implicitly yields control if the target ULT is not joinable at the call time.

IV. Performance Analysis and Optimizations

This section investigates subtle implementation details that can greatly affect the latency of primitive operations for ULTs, such as creation, join, and destruction. We do not include the analysis and optimizations for tasklets here because their overhead comes mainly from memory allocation and deallocation and it is mostly mitigated by using memory pools (Section IV-B). Instead, the performance of tasklets will be discussed in Section V-A.

We used a 36-core (72 hardware threads) machine, which has two Intel Xeon E5-2699 v3 (2.30 GHz) CPUs and 128 GB memory and runs Red Hat Linux (kernel 3.10.0-327.el7.x86_-64) 64-bit, in this section and Section V. We used `gcc` 4.8.5 for compilation and PAPI [28] for collecting necessary hardware counter values.

A. Benchmark Description and Baseline Performance

For simplification, the analysis focuses on spawning and joining ULTs on a single ES. That is, we create a large number of ULTs on one ES in a bulk synchronous fashion, join them by the main ULT, and then destroy them. Each ULT is created with 16 KB of stack space. Since single-threaded performance improvements are often translated into improved multithreaded execution, we expect multi-ES environments to benefit from the optimizations introduced in this section. In the following, we report latency results in CPU cycles and show memory-related hardware counters where needed.

Figure 3 shows the performance breakdown of our baseline implementation according to the number of ULTs that are created in the benchmark. It illustrates the average latency (arithmetic mean) per ULT in CPU cycles from 1,000 executions of each case. We note that the standard deviation of latencies measured in the experiment is less than 5%. Create, Join, and Free in the figure represent the time spent for creating, executing and joining, and destroying work units, respectively. 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 for Create, Join, and Free, respectively.

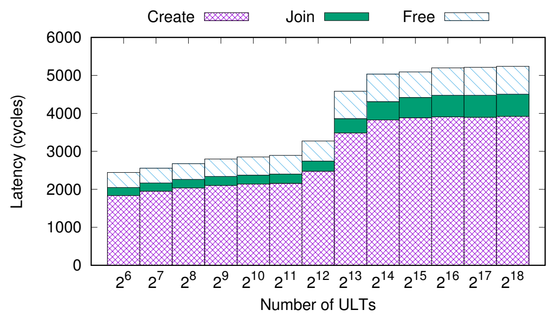


Fig. 3: Performance of the baseline implementation.

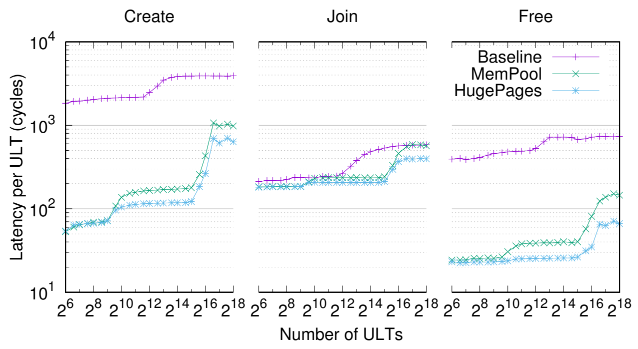


Fig. 4: Effects of using memory pools and huge pages.

B. Using Memory Pools

Work units in Argobots are meant for dynamic fine-grained concurrency. Thus, memory allocation and deallocation take place at high frequencies. In each case of Figure 3, about 75% of the time is used for creation, and about 15% of the latency is used for destruction. A 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` functions provided in `glibc` to handle dynamic memory allocations.

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 [29], 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 work unit is freed in a different ES, however, the memory used for the work unit is returned to the ES that has allocated it, in order to avoid the heap-blowup problem [30].

We experimented with the new memory management system and measured the latency of the Create, Join, and Free operations. The results are shown in Figure 4 as MemPool. We observe a substantial benefit of the memory pool, especially

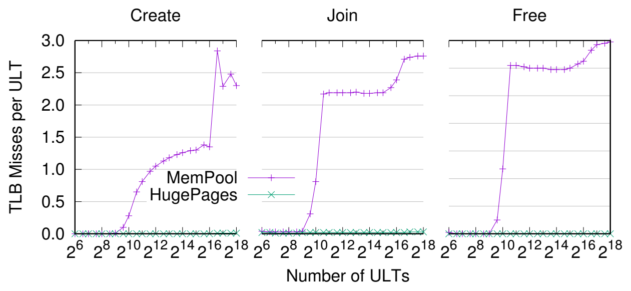


Fig. 5: TLB miss reductions with huge pages.

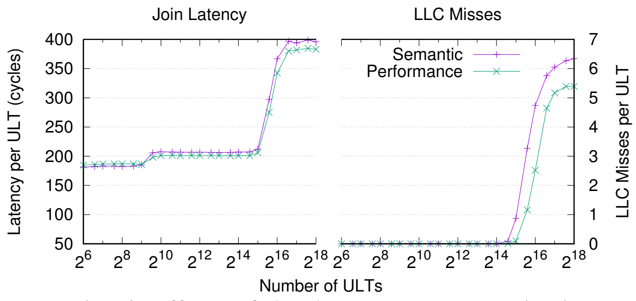


Fig. 6: Effects of the data structure organization.

for Create and Free, compared with Baseline that uses the glibc allocation system.

C. Using Huge Pages

Looking into hardware counters after incorporating the custom memory allocator, we noticed that TLB misses are not negligible, especially for a large number of ULTs, as depicted in Figure 5. When dealing with more than 2^{12} ULTs, each ULT experiences one or more TLB misses for all Create, Join, and Free operations. Given the expensive costs of page walks and eventual memory accesses to the page table, we naturally sought a way to reduce them by using larger page sizes.

For this purpose, we allocated 2 MB huge pages, instead of the normal 4 KB pages, by using mmap until the system ran out of huge pages for explicit allocation. Then, we reverted to the transparent huge page (THP) support [31]. We modified the memory allocator in Section IV-B to exploit huge pages. We report the results in Figure 5. HugePages in the figure was obtained by using huge pages with memory pools. It indicates that the TLB misses go down to almost zero with huge pages, which is translated into up to 39%, 33%, and 57% reduced cost for the creation, join, and destruction of a ULT in Figure 4 (see HugePages), respectively.

D. Data Structure Organization

When building a new structure that contains several fields, programmers often organize them following a semantic approach where fields close to each other have close semantics. In the context of ULTs, the fields of the corresponding data structure may be grouped according to the functionality (e.g., identification, scheduling, migration). Unfortunately, such organization does not always perform well with the pattern and frequency of accessing those fields. We investigated an alternative organization that clusters together fields that contribute to the critical path during scheduling. The creation and destruction of a ULT are not sensitive to the organization of the fields. The opposite statement holds, however, for other scheduling operations such as Join.

Creating a ULT accompanies allocation of an internal data structure that keeps the necessary information for scheduling properly the ULT. This structure fits in three cache lines. During our investigation, we observed that not all fields are touched along the scheduling critical path and that the frequently accessed data can be clustered and fit into two cache lines. Thus, we conjecture that in a performance-critical environment, the data should be organized in a cache-friendly manner by reducing the number of cache lines accessed on the critical path and arranging them in a way that favors spatial locality. Figure 6 shows the consequence of the performance-oriented data layout in the benchmark in terms of reduced cache misses and overall performance. The results indicate that the Join latency has been reduced by up to 7%, which corresponds to the reduction in last-level cache misses.

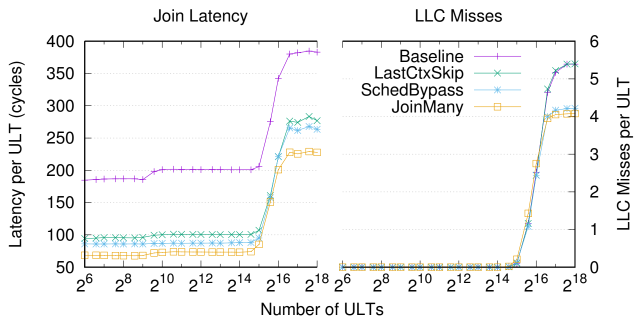


Fig. 7: Effects of the Join optimizations.

E. Optimizing the Join Operation

Not saving the context of terminating ULT. Context switching of ULTs consists of two steps: saving the context of the current ULT, which wants to suspend its execution, and restoring the context of the next ULT, which needs to resume its execution. These two steps are usually necessary in order to switch two ULTs; but the first step, saving the context, can be omitted if the 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. LastCtxSkip in Figure 7 illustrates the effect of this optimization on the Join operation over Baseline (i.e., Performance in Figure 6). Since ULTs terminate immediately after they get started in the benchmark, this optimization can reduce on average 100 cycles (i.e., 45%) of the Join latency from Baseline.

Bypassing the scheduler. Blocking operations in Argobots often involve context switching to the scheduler in order to allow other work units to get scheduled. For Join, since the caller cannot progress beyond the Join synchronization point until the ULT being joined terminates, it need not context switch to the scheduler. Instead, the caller can be blocked and directly context switched to the next ULT to be joined. When the joinee ULT is completed, the control is switched back to the joiner ULT. That is, the scheduler is bypassed in Join. Although this idea is similar to that presented in [32], the main difference between two approaches is that the target of context switching in our optimizations is determined by the user, not the library or kernel. We note that the Argobots runtime does not make a decision for the next ULT.

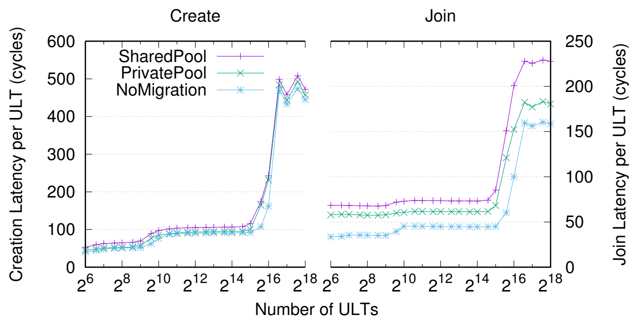


Fig. 8: Effects of using a private pool and disabling migration.

SchedBypass in Figure 7 shows how this optimization outperforms LastCtxSkip. In LastCtxSkip, the joiner ULT jumps to the scheduler, and the scheduler executes all ULTs and switches back to the joiner ULT. The optimized 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 while it is touched twice in the LastCtxSkip version by the scheduler and the joiner ULT. The effect of this optimization can be seen in the lower miss rate of the last-level cache in Figure 7. This approach does, however, have a limitation: it allows direct context switching to occur only within the same ES, because the yield_to operation can be applied only to ULTs in the same ES (Section III-C).

Join_many. With the previous Join optimization, $2 \times N$ context switches are needed in order to join N ULTs, because joining one ULT contains two context switches. To further reduce the number of context switches when joining multiple ULTs at the same time, we devised 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 7 as JoinMany. It reduces the Join latency by an average of 19 cycles from that of SchedBypass.

F. Other Optimizations

This subsection describes two optimizations that can be selectively applied to applications.

Using a private pool. All experiments so far used a shared pool for the scheduler even though only one ES was used. If there is no sharing between ESs or only one ES is created, the pool used by the scheduler 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. PrivatePool in Figure 8 shows the latency of each operation when using a private pool, while SharedPool is the case when a shared pool is used. All previous optimizations are applied to both cases. Since Create and Join operations include pushing a ULT to the pool and popping a ULT from the pool respectively, their latency is improved with the private pool. On the other hand, the Free operation is not affected by the private pool because it does not involve any pool manipulation.

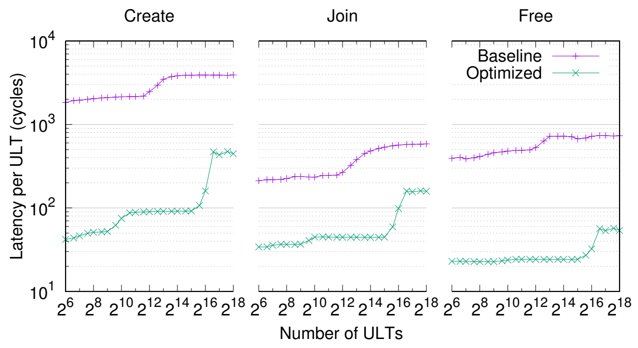


Fig. 9: Latency comparison between the baseline implementation and the fully optimized implementation.

Disabling features. Often not all features provided by a library are used in the application, but all of them have to be included in the applications unless the library supports disabling some features. The problem is that 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 or stackable scheduler. We observe that in the current implementation disabling migration could reduce around 20 cycles in the Join operation latency, which is shown as NoMigration in Figure 8. We note that disabling other features was insignificant, and thus we do not show their effect here.

G. Putting It All Together

Figure 9 shows the improved latencies of Create, Join, and Free operations with all the optimizations introduced in the preceding subsections. In the figure, Baseline means the latency of each operation in the baseline implementation, which is also shown in Figure 3, and Optimized represents the results of fully optimized implementation. In summary, the latencies of Create, Join, and Free were significantly reduced by our optimizations. For instance, the latency of each operation per ULT for 2^6 ULTs was decreased from 1,837 cycles to 42 cycles for Create, from 212 cycles to 34 cycles for Join, and from 394 cycles to 23 cycles for Free. For 2^{18} ULTs, the numbers were changed from 3,921 cycles to 446 cycles for Create, from 584 cycles to 159 cycles for Join, and from 733 cycles to 54 cycles for Free. From analyses and various optimizations, we notice that using memory pools is the most effective for Create and Free operations while all optimization methods introduced in the preceding subsections collectively influence the performance of the Join operation.

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 last-level cache. 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 last-level cache and hence reflect a worst-case scenario.

V. Evaluation

This section shows performance and scalability results of the Argobots implementation with microbenchmarks and applications using the same environment described in Section IV.

A. Microbenchmarks

We compared the performance and scalability of Argobots, which is optimized with all techniques introduced in Section IV, with those of two ULT libraries, Qthreads 1.10 [3] and MassiveThreads 0.95 [4]. Since Qthreads and MassiveThreads are already optimized and it is not simple to apply our optimization techniques to them because of the design differences, we used them as they are without modifying their implementation. All libraries were compiled with -O3 -ftls-model=initial-exec flags. Since Pthread is not suitable for oversubscription and frequent context switching because of its high overhead, we did not include Pthread in the evaluation.

Create/join time. We compared the time taken to create and join a ULT or a tasklet with various numbers of ESs. For Qthreads and MassiveThreads, the number of workers was set to the same as that of ESs; and one worker in Qthreads was mapped to one shepherd. We created one ULT for each ES, and that ULT repeated creating and joining 256 ULTs or tasklets in a private pool of its associated ES 1,000 times. We performed the same pattern of creating and joining ULTs for Qthreads and MassiveThreads although they do not provide the same concept of pool or ES.

Figure 10 illustrates the average create and join times per ULT for each lightweight thread library from 10 runs of the benchmark. The join operation here includes both joining a ULT and destroying it. Since MassiveThreads by default utilizes the *work-first* scheduling policy [33] (i.e., pushes the creator to the scheduling queue and executes the spawned thread first), while Qthreads and Argobots adopt the *help-first* principle [34] (i.e., create all threads first), we include results for both the work-first and help-first versions of MassiveThreads, denoted by MassiveThreads (W) and MassiveThreads (H) in Figure 10, respectively. For Argobots, ULT and tasklet results are depicted as Argobots (ULT) and Argobots (Tasklet), respectively.

The results in the figure show that Argobots achieves better performance than either Qthreads or MassiveThreads does. 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. Argobots shows similar ULT create/join overhead irrespective of the number of ESs, whereas Qthreads and MassiveThreads do not. The big increase in the time at 40 is because two hardware threads share a single physical core in the target CPU. The better scalability of Argobots comes from the design of adopting a scheduler and scheduling pools for each ES and its synchronization-free implementation. 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. Although MassiveThreads (W) shows overhead similar to that of Argobots with a small number of ESs (up to eight), its ULT create/join time increases or fluctuates with a large number of ESs because it uses a shared queue with a work-stealing policy. Qthreads and MassiveThreads (H) do not achieve scalable performance; and with a large number of ESs (e.g., 72 ESs), their create/join is more than 20 times slower than that of Argobots.

As described in Section III-A, Argobots also provides a more lightweight work unit, tasklet, along with a ULT. Argobots (Tasklet) in Figure 10 shows the time taken to create and join an Argobots tasklet. Since the tasklet's context is more lightweight than that of ULT and it does not involve context switching (i.e., it is directly executed by the ES's scheduler), it has less overhead than does a ULT in both creation and execution. Thus, Argobots (Tasklet) shows the smallest cycles among the cases. In addition, the tasklet achieves very scalable performance, as does the Argobots ULT, because it also takes the same advantage of the Argobots design.

Create/join time tolerance. We also measure the minimum, maximum, and average time for each ULT on each ES to create and join another ULT. Figure 11 shows results for Qthreads, MassiveThreads with the work-first policy, and the Argobots ULT. The results indicate that only Argobots achieves very sustainable performance, which means that ESs do not affect each other's execution if they do not interact. However, workers in Qthreads and MassiveThreads interfere with each other, and thus the create/join time per ULT varies significantly when multiple workers 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 runtime without worrying about the conflict with underlying threading runtime from the perspective of scheduling.

Yield time. The yield time contributes to the ULT create/join time as well. When the ULT that has created a new ULT tries to join the new ULT, it needs to yield control to the scheduler in order to execute the new ULT. The worst performance of Qthreads in Figure 10 is due mainly

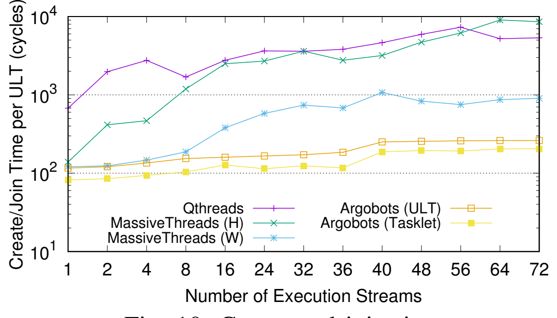


Fig. 10: Create and join time.

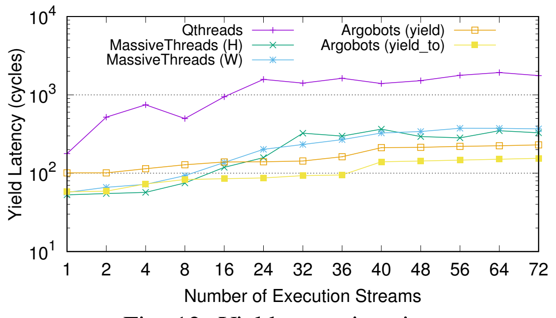
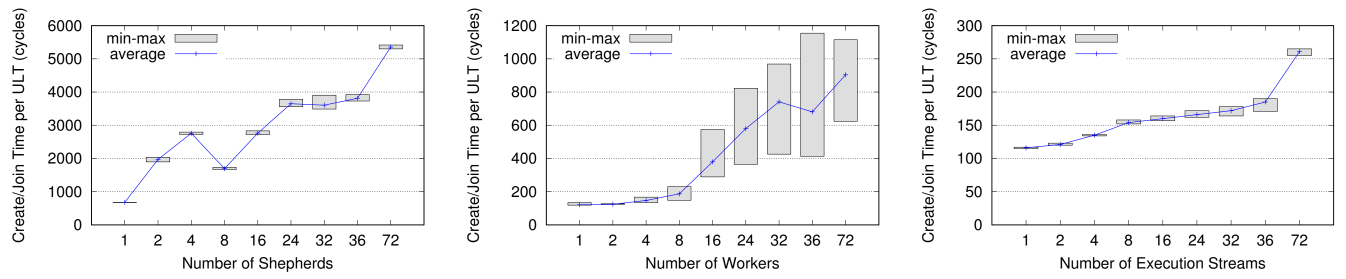
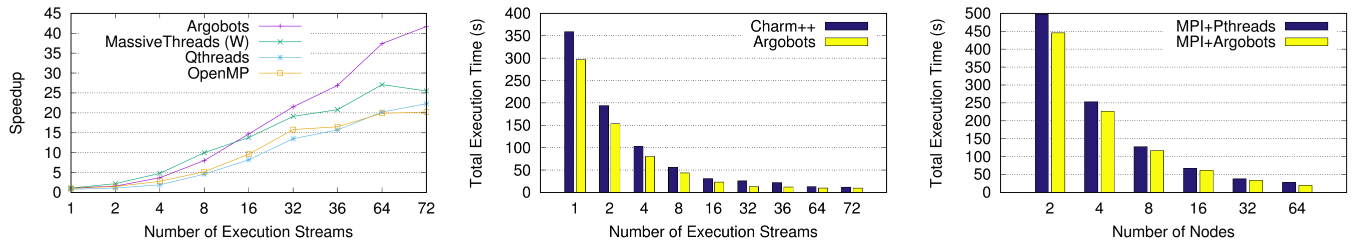


Fig. 12: Yield operation time.



(a) Qthreads (b) MassiveThreads (W) (c) Argobots (ULT)

Fig. 11: Create/join time tolerance.



(a) XSBench (b) LeanMD (c) 2D Poisson Solver

Fig. 13: Application performance results.

to its high yield overhead. The yield latency is also critical for applications that require frequent context switches. We measured the yield overhead for each ULT library according to the number of ESs. Figure 12 depicts the results. Since Argobots supports the yield_to operation as well as yield (Section III-C), the figure shows the overhead of both yield and yield_to. For both operations, Argobots demonstrates better scalability than Qthreads and MassiveThreads do, because of its low synchronization overhead. Figure 12 also shows that, as expected, yield_to incurs almost half the overhead of yield because yield_to does not pass through the scheduler.

B. Applications

We here demonstrate how applications can take advantage of Argobots features and mechanisms.

1) XSBench: XSBench [35] is a proxy application that models the calculation of macroscopic neutron cross-sections of a Monte Carlo particle transport simulation code, OpenMC [36]. 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 actual cross-section lookup loop, to Argobots. Basically, in our implementation the iterations of the lookup loop are evenly divided into ESs. One ULT per ES is created, and it creates as many tasklets as the number of lookups that are assigned to the ES. Each tasklet performs one cross-section lookup. Since we notice that the cross-section lookup code suffers from cache misses due to its irregular memory accesses, our Argobots version takes account of data locality instead of simply executing the loop iterations as done in the original OpenMP code. We implement a custom scheduler, using the Argobots scheduler framework (Section III-B), that executes tasklets according to the order of energy indices, which are random values but critical to the memory access pattern. In other words, the scheduler sorts tasklets in ascending order of energy indices and executes them. This approach can achieve better performance because it can improve the spatial locality of accessed data in the cache. The scheduling begins after creating a certain number (here, 8,192) of tasklets instead of creating all tasklets, because holding and sorting too many tasklets can lead to significant overhead in the memory usage and thus impact the performance. To overcome the load imbalance between ESs, the scheduler adopts work-stealing when its main pool is empty.

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 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 scheduler for the execution. We did not modify the OpenMP code because mimicking the behavior of the Argobots version with OpenMP is not simple.

Figure 13a shows performance results of our XSBench implementations in Argobots, MassiveThreads with the work-first policy, and Qthreads, along with the OpenMP result. The baseline XSBench used is the 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 five times; the figure shows the average result. 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 47.67 (Argobots), 47.67 (MassiveThreads (W)), 61.10 (Qthreads), 51.76 (OpenMP), and 50.51 (sequential) seconds. The results in the figure show that all implementations scale well but that Argobots achieves the best scalability. These results indicate that the scheduling of work units in the

Argobots version is effective in handling this kind of workload and incurs little overhead. Although MassiveThreads (W) shows better performance on a smaller number of ESs because of its aggressive work-stealing policy, it suffers from contention and overhead on a large number of ESs.

2) LeanMD: LeanMD is a molecular dynamics simulation benchmark written in Charm++ [37]. It simulates the behavior of uncharged atoms using the Lennard-Jones potential, and it has been used in many studies [38], [39], [40].

To examine whether Argobots can natively support a mature parallel application, we implemented LeanMD in Argobots and compared its performance with that of the highly optimized original LeanMD code written in Charm++. In our Argobots implementation of LeanMD, each ES is equipped with a custom work-stealing scheduler that first executes work units from its local pool and then steals work units from pools associated with other ESs when the local pool is empty. We create as many pools as there are ESs, and all these pools can be accessed by any ES (i.e., these are shared pools). However, since the scheduler on each ES always accesses its local pool first, our implementation naturally avoids the contention on pools while resolving the load imbalance between ESs. A cell is managed by one ULT, called the cell ULT; and the interaction between two cells is managed by a tasklet, called the interaction task. The cell ULT is responsible for creating the interaction tasks with half its neighbors. Since the interaction between two cells is symmetric, the other half of the interaction tasks will be created by neighbors of the cell. After spawning the interaction tasks, the cell ULT waits for the completion of all the interaction tasks in which the corresponding cell is involved. The waiting mechanism is implemented by using futures: a cell ULT waits for a future that has as many compartments as there are neighbors of the cell, and an interaction task signals and contributes to the futures associated with both the cells that it manages. When a future is ready, a callback function reduces all the forces in the future's compartments, and the corresponding cell ULT is awakened. The cell ULT then updates the kinematic parameters of the atoms contained in the cell and signals all the neighbors of the cell (using future mechanism) once the updates are completed. When the cell ULT receives the notification from all its neighbors about the completion of updates, it proceeds with the next iteration.

Figure 13b compares the performance of Argobots and Charm++ (using Converse threads) for LeanMD. In this figure, we vary the number of ESs from 1 to 72 along the x-axis. The y-axis shows the total execution time for 20 steps of LeanMD simulation for a 3D cell array of dimensions $7\times7\times7$. In our simulation, we used 1-away XYZ configuration and 1,000 atoms per cell. Thus, the entire simulation space has a total of 343,000 atoms. Execution times shown along y-axis were obtained by averaging across 10 runs. The results show that Argobots achieves better absolute performance than does Charm++ in all the cases. For example, when we used all the hardware threads available on the machine (i.e., 72), Argobots takes a total of 9.6 seconds compared with 11.7 seconds taken by Charm++. With regard to speedup, Argobots performs slightly better than Charm++: 30.9 vs 30.7 for 72 ESs, normalized with respect to the one ES case for each implementation. The better performance of Argobots compared with Charm++ demonstrates the efficiency of Argobots even for fine-grained applications. The results in this subsection further demonstrate that the scheduling and synchronization mechanisms provided by Argobots are sufficient for handling complex and mature parallel applications. We note that the goal of this comparison is to show that Argobots has lightweight but fully fledged mechanisms that can be used to implement more complicated algorithms, not to advocate that Argobots is better than other programming model runtimes.

3) 2D Poisson Solver: We adapted a 2D Poisson solver implemented in MPI [41] to use Pthreads and Argobots. This application uses the Jacobi iterative method to solve the linear system. The $N \times N$ region is divided into strips between the MPI processes. Cells that lie on the boundary of a strip must communicate with neighboring processes in order to receive the appropriate data to perform the local computation. Thus, each MPI process sends and receives data twice in each iteration because of the data layout.

Our adaption of the application uses MPI for interprocess communication and different threading models for shared-memory parallelism. In the implementation with Pthreads, only the master thread takes charge of the MPI communication, and all threads including the master thread participate in the computation. The rows in the strip are equally divided into all threads. All communication done by the master thread is nonblocking in order to take advantage of any overlap in communication and computation.

The basic parallelization approach in the Argobots version is similar to that of the Pthreads version. That is, each ULT is spawned on each ES and executes part of the computation, but only one ULT handles all MPI communication. However, our Argobots version further exploits lightweight threading and tasking of Argobots by creating a ULT for communication and tasklets for computation. The communication ULT initiates nonblocking send and receive operations and then yields control after testing whether the communication operations are not complete. When it is scheduled later, it tests the posted operations and decides whether to yield again or to proceed. Once the receive calls are done, the row calculation for the associated data is carried out by the same ULT to utilize the data locality. While the communication ULT is dedicated to a specific ES, computation tasklets are shared between ESs to avoid the potential load imbalance that may be caused by the communication ULT.

Figure 13c shows the results of the application for $N = 16384$ on the Blues cluster at Argonne National Laboratory. Each compute node on Blues has two Intel Xeon E5-2670 CPUs (i.e., 16 cores) with 64 GB memory and uses CentOS 6.7 (kernel 2.6.32-573.7.1.el6.x86_64). All nodes share a GPFS file system and are connected with the QLogic QDR InfiniBand network; gcc 5.2.0 and MPICH 3.2 are used for compilation and MPI communication, respectively. In the figure, MPI+Pthreads and MPI+Argobots denote the total execution time of our implementations with Pthreads and Argobots, respectively. The number of nodes is varied from 2 to 64, and 16 ESs (or Pthreads) are created on each node. The results in the figure indicate that MPI+Argobots achieves better performance than does MPI+Pthreads, because using ULT and its lightweight context switching enables communication and computation to be more overlapped. In addition, our parallelization with tasklets facilitates work sharing among ESs while keeping the low overhead of task management.

VI. Conclusions and Future Work

Argobots is a lightweight, low-level threading and tasking framework to efficiently exploit the massive on-node parallelism provided by current and future many-core architectures. Although many lightweight threading models have been proposed, their focus has been on how to build the best threading runtime system rather than how to be a threading and tasking building block for higher-level runtimes. The Argobots approach is to serve as an enabling technology providing efficient mechanisms, not a policy maker, so that users can implement and optimize their runtime on top of Argobots. With this goal in mind, this paper presents the design and implementation of Argobots as well as a performance analysis and optimizations of Argobots. Evaluation results with microbenchmarks and applications indicate that Argobots incurs low overhead in its operations and achieves sustainable and scalable performance while providing diverse and configurable mechanisms.

Future work on Argobots will include programming model study, memory hierarchy optimizations, and resource-aware threading. First, we will implement diverse programming models over Argobots and study the pros and cons of each approach as well as the application performance. Second, we plan to investigate Argobots optimizations regarding deep memory hierarchy and long latency from unconventional memory such as NVRAM. Third, since the number of cores and the number of hardware threads per core are increasing in HPC platforms, hardware resources per ES and ULT, such as memory or power, will be decreasing; thus, we plan to devise techniques to address this challenge in the context of Argobots.

VII. Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) and Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory. Researcher from Universitat Jaume I was supported by the FPI Program (Vali+d) of the Generalitat Valenciana.

References

[1] "TOP500 Supercomputer Sites," http://www.top500.org/.

[2] V. M. Weaver, "Linux perf_event features and overhead," in *Proceedings of the 2nd International Workshop on Performance Analysis of Workload Optimized Systems*, ser. FastPath '13, 2013.

[3] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Proceedings of the 2008 Workshop on Multithreaded Architectures and Applications*, ser. MTAAP '08, April 2008.

[4] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, vol. 8665, pp. 222–238.

[5] Microsoft MSDN Library, "Fibers," https://msdn.microsoft.com/en-us/library/ms682661.aspx.

[6] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management," in *Proceedings of the 2002 Usenix Annual Technical Conference*, June 2002.

[7] "Programming with Solaris Threads," https://docs.oracle.com/cd/E19455-01/806-5257/6je9h033n/index.html.

[8] L. V. Kalé, J. Yelon, and T. Knuff, "Threads for interoperable parallel programming," in *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '96, August 1996, pp. 534–552.

[9] L. V. Kalé, M. A. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, "Converse: An interoperable framework for parallel programming," in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS '96, April 1996, pp. 212–217.

[10] A. Porterfield, N. Nassar, and R. Fowler, "Multi-threaded library for many-core systems," in *Proceedings of the 2009 Workshop on Multithreaded Architectures and Applications*, ser. MTAAP '09, May 2009.

[11] BSC, "Nanos++," https://pm.bsc.es/projects/nanox/.

[12] ——, "The OmpSs programming model," http://pm.bsc.es/ompss/.

[13] "GNU Pth - The GNU Portable Threads," http://www.gnu.org/software/pth/.

[14] 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*, ser. PLDI '97, 1997, pp. 320–333.

[15] 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*, ser. PPoPP '99, 1999, pp. 60–71.

[16] S. Thibault, "A flexible thread scheduler for hierarchical multiprocessor machines," in *Proceedings of the Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, June 2005.

[17] "Stackless Python," http://www.stackless.com.

[18] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06, October 2006, pp. 29–42.

[19] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, October 2003, pp. 268–281.

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

[21] P. Li and S. Zdancewic, "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, June 2007, pp. 189–199.

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

[23] J. d. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "TiNy threads: A thread virtual machine for the Cyclops64 cellular architecture," in *Proceedings of the Fifth Workshop on Massively Parallel Processing*, ser. WMPP '05, April 2005.

[24] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 53–79, February 1992.

[25] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, "Scheduling in K42," Tech. Rep., 2002.

[26] R. S. Engelschall, "Portable multithreading - the signal stack trick for user-space thread creation," in *Proceedings of the 2000 Usenix Annual Technical Conference*, June 2000.

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

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

[29] S. Seo, J. Kim, and J. Lee, "SFMalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11, October 2011, pp. 253–263.

[30] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IX, November 2000, pp. 117–128.

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

[32] K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, November 2013, pp. 133–150.

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

[34] 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&Distributed Processing*, ser. IPDPS '09, May 2009, pp. 1–12.

[35] 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 - The Role of Reactor Physics toward a Sustainable Future*, 2014.

[36] "The OpenMC Monte Carlo Code," http://mit-crpg.github.io/openmc/.

[37] "LeandMD," http://charmplusplus.org/benchmarks/#leanmd.

[38] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using projections performance analysis tool," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 347–358, 2006.

[39] L. V. Kale and J. Lifflander, "Controlling concurrency and expressing synchronization in charm++ programs," in *Concurrent Objects and Beyond*. Springer, 2014, pp. 196–221.

[40] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni *et al.*, "Parallel programming with migratable objects: Charm++ in practice," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, 2014, pp. 647–658.

[41] "Parallel 2D Poisson Equation Solver, Using MPI," http://people.sc.fsu.edu/~jburkardt/c_src/poisson_mpi/poisson_mpi.html.