PPL: An Abstract Runtime System for Hybrid Parallel Programming

Alex Brooks and Hoang-Vu Dang and Nikoli Dryden
Department of Computer Science
University of Illinois at Urbana-Champaign, IL, USA
{brooks8, hdang8, dryden2}@illinois.edu

Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign, IL, USA
Mathematics and Computer Science Division
Argonne National Laboratory, IL, USA
snir@mcs.anl.gov

ABSTRACT

Hardware trends indicate that supercomputers will see fast growing intra-node parallelism. Future programming models will need to carefully manage the interaction between inter- and intra-node parallelism to cope with this evolution. There exist many programming models which expose both levels of parallelism. However, they do not scale well as per-node thread counts rise and there is limited interoperability between threading and communication, leading to unnecessary software overheads and an increased amount of unnecessary communication. To address this, it is necessary to understand the limitations of current models and develop new approaches.

We propose a new runtime system design, PPL, which abstracts important high-level concepts of a typical parallel system for distributed-memory machines. By modularizing these elements, layers can be tested to better understand the needs of future programming models. We present details of the design and development implementation of PPL in C++11 and evaluate the performance of several different module implementations through micro-benchmarks and three applications: Barnes-Hut, Monte Carlo particle tracking, and a sparse-triangular solver.

Categories and Subject Descriptors

D.3 [**Programming Language**]: D.3.3 Language Constructs and Features — Frameworks, Concurrent programming structures; D.3.4 Processors — Runtime environments.

General Terms

Algorithms, Design, Performance.

Keywords

Distributed-memory parallelism, Programming models, PGAS, RDMA, One-sided communication, Multithreading.

ESPM2 2015, November 15-20, 2015, Austin, TX, USA
Copyright 2015 ACM 978-1-4503-3996-4/15/11...$15.00
DOI: 10.1145/2832241.2832246.

1. INTRODUCTION

Future supercomputers are expected to run hundreds of physical threads per node. The speed of nodes and threads within nodes will exhibit large variability, due to dynamic power management and error recovery mechanisms [7, 26]. In order to achieve even moderate efficiency on such machines, both inter- and intra-node parallelism must be carefully exploited. Moreover, it becomes more difficult to address problems such as load balancing, latency hiding, and communication management at this scale.

Researchers are addressing these problems by exploring better implementations of current programming systems and designing new programming models. In particular, many projects have explored the use of a partitioned global address space (PGAS) across nodes [32, 9], the use of fork-join models with lightweight threads [14], and coarse-grain dataflow models within and across nodes [22, 6].

Both approaches will require the design of new runtimes. Typical of these runtimes are:

1. Support for RDMA communication, which leverages modern network capabilities, reduces the software overhead of communication, and matches the needs of PGAS models.
2. Support for active messages. This is required for more complex one-sided communication that is not directly supported in hardware, and for remote method invocation as used by many models [18, 9].
3. Support for lightweight threads (tasks). Fast task activation and preemption is essential for many of the emerging programming models. With it, each synchronization point can become a potential scheduling point. This avoids busy waiting and facilitates latency hiding.
4. Support for event-driven task scheduling. The mechanism that identifies which tasks become runnable when a synchronization event occurs and schedules a runnable task when a physical thread become idle has to be very efficient. The most important events are those used for producer-consumer synchronization, both intra-node (full-empty bits and counters) and inter-node (completion of a communication).

In many cases, new programming models do not expose their underlying runtime. In a few cases, such as GASNet [8], OCR [19], and HPX [17], the underlying runtime exposes an API that can be used to implement micro-benchmarks and applications, so that the runtime can be evaluated on its own. However, many of these runtimes deal only with shared memory (OCR) or with inter-node communication (GASNet), not both. Hence, they do not address the critical interaction between communication events and scheduling. HPX deals with both.

The ability to create copies of data objects on multiple nodes, and to synchronize such copies under software control, is essential for many applications and libraries. For example, the ability to repartition meshes or update halos is key to a mesh library such as Zoltan [11] and to adaptive mesh refinement applications. None of the cited runtimes provide such memory services.

We believe such core memory services should be closely integrated with communication, coordination, and task management services into a low-level runtime that can be used directly by libraries and applications, or to support higher level programming models, with their specialized runtimes. In this paper, we propose and evaluate such a runtime, *PPL*.

The design of PPL abstracts the important high-level concepts, called *modules*, of a parallel runtime system for distributed-memory machines: a memory management system, a communication layer, and a threading layer. All of the modules are designed to work closely together, so as to increase interoperability between the modules and minimize the amount of runtime overhead. Our goal is to provide researchers with a flexible tool for developing future runtime systems. By designing an API which modularizes common runtime system elements, it becomes simple to test combinations of layers which otherwise may not have been feasible.

In this paper, we present details of our runtime system design and evaluate an implementation in C++11. We evaluate our implementation in two ways. First, we analyze potential runtime overhead which may result from implementation choices. Second, we implement and discuss several options for each module and test combinations using three mini-applications: the Barnes-Hut algorithm [36], a Monte Carlo particle-tracking algorithm [13], and a sparse-triangular linear solver [31].

The remainder of this paper is organized as follows. Section 2 discuss related work. Section 3 presents the design of the PPL runtime system, with details on each available module. Section 4 describes our implementation of PPL, including details on the module choices. Section 5 contains the evaluation and analysis of our implementation. We present concluding remarks and ideas for future work in Section 6.

2. PROGRAMMING MODELS AND RUNTIME LIBRARIES

In this section, we provide some background on several available programming models for distributed-memory machines, and the runtime features they require.

2.1 Message Passing

Almost all scientific HPC applications today use the message passing model for interprocess communication via the Message Passing Interface (MPI) [20]. This model has been widely popularized due to its high performance, scalability, and portability. Highly tuned implementations of MPI are available for each of the current HPC platforms.

Historically, as core density per node increased, performance and scalability of MPI began to suffer due to its inability to cope with decreasing per-core resources and effectively exploit larger amounts of intra-node parallelism. A common solution is a hybrid approach, where MPI is coupled with a threading model such as OpenMP [22] or Intel Threading Building Blocks [23]. In these programming models, the purpose is to use MPI for interprocess communication while the threading model handles shared-memory parallelism.

Now as core density continues to grow and per-core resources continue to shrink, performance of these hybrid solutions is degrading. This is largely due to how implementations deal with thread safety requirements. In virtually every implementation today, this is guaranteed through a combination of atomic operations and critical sections [3]. Many efforts have been made to decrease the overhead in providing thread safety in MPI implementations [12, 4, 29, 28]. However, improving multithreaded performance requires very careful thought and analysis in order to make any noticeable impact in performance and to remain portable [5, 15]. As a result, many implementations resort to simple solutions, such as coarse-grained locking for critical sections. This has been shown to greatly hinder performance.

Moreover, there is no agreed standard for the coupling of MPI and threading systems. In OpenMP, threads are not first-class citizens; they are execution units that execute language constructs such as parallel iterates or tasks. Consequently, the association of MPI threads with OpenMP threads does not lead to a convenient programming model. In practice, MPI is only called from sequential OpenMP sections, causing sequential bottlenecks.

MPI was initially developed for single-threaded processes. The need for computation/communication overlap led to the introduction of non-blocking communication calls, implemented by running MPI code as a coroutine. The use of nonblocking calls is more bug prone than the use of blocking calls and also complicates the MPI logic. This style seems superfluous in an environment that supports many tasks. A runtime with the four properties listed in the introduction will efficiently support an MPI+task model, where MPI calls are blocking, and multitasking is used to hide latency.

2.2 PGAS

Although message passing is the dominant programming model, writing shared-memory parallel programs is often viewed as being simpler and easier to maintain. The main obstacle for performance and scalability of shared-memory models has been their inability to effectively exploit locality [10]. The Partitioned Global Address Space (PGAS) model was developed to address this issue.

In the PGAS model, the address space is shared and partitioned among the available processes and threads such that a portion is local to each process or thread. The primary benefit of PGAS is that it provides a middle ground between shared- and distributed-memory models by allowing direct access to remote memory and clearly distinguishing between local and remote accesses [16]. There have been many efforts focused on providing solutions which use a global address space model, including Unified Parallel C (UPC) [32], Co-Array Fortran (CAF) [21], and, more recently, X10 [9].

The PGAS model is a very good match to communication based on RDMA and active messages. Indeed, UPC, as well as other PGAS languages, have been implemented atop GASNet [8], a communication layer built atop active messages and RDMA.

Many PGAS systems provide no support for local multithreading. Another weakness is that compilers have limited knowledge of access patterns for codes that use indirect addressing; they cannot optimize access to remote data by aggregating multiple individual accesses and caching remote copies for local reuse. Programmers end up doing such optimizations manually, reverting to explicit communication.

2.3 High-Performance ParalleX

High-Performance ParalleX (HPX) [17] is one of the first systems to provide a complete and modern solution for inter- and intra-node parallelism. It couples an adaptive PGAS with a lightweight threading subsystem. The system defines *local control objects* which can be used to express many types of parallelism and synchronization, including futures, data-flow objects, traditional synchronization (i.e. mutexes, condition variables, etc.), and thread suspension [17]. Finally, all network communication is performed using a form of active messages, where an object, function, and optional continuation are provided [17]. Although HPX addresses the issue of coupling threads and communication, it is missing a closely integrated memory subsystem which we believe is necessary for efficiently managing synchronization and locality of data copies.

2.4 Other

Some additional related models include Charm++ [18], Legion [6], and Open Community Runtime [19].

Charm++ is a parallel programming framework which provides asynchronous message-driven execution via migratable objects [18]. The user defines their application using specially designated objects called *chares* and *chare arrays* which describe work and data units. These objects can be migrated by the runtime system and also be the subject of message-driven events [2]. Work units are automatically and adaptively mapped by the runtime system, allowing for careful management and adaptive load balancing. Although Charm++ performs well across many application domains, it is a language which requires some experience and effort to use with existing codes. In contrast, it is much easier for existing codes to adopt a library.

Legion is a data-centric parallel programming model which uses *regions* to describe properties of data [6]. The user is able to define the organization, partitioning, privileges, and coherence properties of each region, allowing Legion to implicitly extract parallelism (tasks) and issue appropriate data movement operations based on the data mapping [6]. A *software out-of-order processor* (SOOP) is used for defining a physical task mapping and scheduling tasks using deferred execution coupled with low-level runtime event systems for managing dependencies [6]. Although preliminary experiments show good performance for a circuit simulation application [6], it is unclear how well this model will scale beyond the tested 16 compute nodes and 96 GPUs.

Open Community Runtime (OCR) is a recent effort to design and develop a research runtime system which supports a variety of features required for extreme-scale computers [19]. Similar to Charm++ and Legion, the fundamental idea behind OCR is to consider computation as a dynamically generated directed acyclic graph of tasks which operate on relocatable data blocks [19]. Task execution is event-driven based on preconditions on the execution, which can include data block and event dependencies. OCR defines a memory model which manages the movement and modification of data blocks by defining *happens-before* relations which follow a *Release Consistency* memory model [19]. However, OCR does let the programmer write programs in which two or more tasks can write to a single data block simultaneously, resulting in a potential data race which must be resolved by the application. Since this research system is relatively new, little has been published evaluating it. As it matures, it will be interesting to compare it to PPL.

3. PPL DESIGN

PPL is designed for abstracting an execution model which combines three modules: memory management, communication, and threading.

3.1 Memory Management

PPL uses a partitioned global address space and specifies distinct *local* and *global* heaps. The local heap may only be accessed by locally-executing threads, whereas the global heap may be accessed by any thread. The local heap has two purposes. First, it stores objects that are not intended for remote access. For example, the vertices of a distributed graph are often accessed only by the node to which they are assigned. The second purpose is for the caching of remote data, discussed later. The layout of the global heap is identical across nodes, and thus any object on any node may be accessed by changing the base address of a pointer.

PPL defines three types of global objects: global pointers (`gptr`), global variables (`gvar`), and global vectors (`gvec`). A `gptr` can be used to reference an address in the local heap or any global heap, facilitating any pattern of one-sided communication. `gptr` is the basis for `gvar` and `gvec`. A `gvar` is allocated on a single node and provides read-only access to remote nodes, e.g. for write-once variables such as configuration parameters. A `gvec` is like a Fortran Co-Array [21] and is useful for collective operations such as reduce or gather.

To take advantage of locality for communicated objects, PPL's memory system defines a software caching mechanism. This is critical for good shared-memory performance, especially as the number of threads per node increases, due to the greater opportunity for *synergistic caching*: a remote object retrieved by one thread may be accessed by other threads on the same node. The caching mechanism provides implicit caching, but in order to maintain scalability, does not provide implicit coherence. Most parallel scientific algorithms proceed by well-defined phases, where conflicting accesses to the same variable do not occur within a phase (except for the special case of accumulators). Such codes do not need implicit coherence; instead, explicit coherence operations can be associated with the start or end of a phase.

The caching mechanism is policy-agnostic and requires only three basic operations: obtaining, updating, and removing an entry from the cache. Each operation is provided with a tuple specifying the memory address and node of the object, and must be thread-safe.

To provide an opportunity for optimizing remote data accesses, PPL specifies three types of get operations on `gptr`s: `lget`, `rget`, and `get`. `lget` is a local get that returns the locally cached copy of the object or raises an exception if a cached copy does not exist. Next, `rget` always fetches a remote copy of the object and updates its entry in the cache. Finally, `get` is a generic get operation that returns a cached copy if present, and behaves like `rget` otherwise.

Updates to objects may be done with `put` operations. For simplicity, these currently do not update the cache.

3.2 Communication

The communication module provides an interface for performing one-sided communication, available in three forms: put, get, and active message. All three operations act on a global object. Completion of an operation indicates local completion. In order to test for remote completion, a synchronization handler is associated with each communication operation to trigger an event on the remote node, such as decrementing a counter.

The communication module comes as an application-scoped *communicator* object. To send an active message, the respective function must be registered with the communicator upon creation. In addition, the communicator provides a method for querying information about the underlying network, such as the local rank of the communicator, the total number of nodes, and RDMA memory information.

3.3 Threading

Our threading interface provides support for spawning threads and futures, as well as performing blocking synchronization which causes the caller to yield control. Futures are identical to threads, except that their execution is delayed until a synchronization event occurs. This supports basic data-flow programming, where dependencies can be enforced by synchronization events. For example, a counter can be used for a task to wait for all its inputs to be available: each input producer atomically decrements the counter, and the consumer task is scheduled when the counter is zero.

Each thread and future has an associated synchronization primitive. This allows threads to wait on or check completion of other threads. The intention is to provide a mechanism which allows fine-grained thread management without busy-waiting.

4. PPL IMPLEMENTATION

We implemented PPL in C++11, which allowed modules to make use of templates, inheritance, and virtual functions.

4.1 Memory Management

PPL's heaps are implemented using the user-space memory allocator `umalloc` [24] to allocate space within regions that are appropriately mapped for RDMA access. Allocation is performed on the global heap upon creation of a `gptr` and on the local heap to cache a remote object.

The caching service currently implements three different policies: `nocache`, `noevict`, and `evict`. The `nocache` policy is a pass-through policy designed for development and testing that does no caching. `noevict` and `evict` both cache data. When a request is made for an object that is not in the cache, space is allocated on the local heap for the object, and then a get is performed to retrieve the data to this location. Local accesses to the object then make use of this data unless `rget` is subsequently used or the entry is evicted.

The `noevict` policy caches every remote request and does not perform evictions; instead, the user has the option to explicitly evict entries of cached global objects. The object's origin and address are stored, along with a local pointer, in a concurrent dictionary which is consulted for future lookup and removal operations. This policy is useful for applications with low memory pressure, or that have well-defined phases after which explicit memory management and coherence can be performed.

Lastly, the `evict` policy is similar in implementation to the `noevict` policy, except that when local cache space fills, an unused entry is evicted. If every entry is in use, the requestor yields until an entry can be evicted. This makes for a good "general purpose" policy for the case where an application has significant memory requirements.

4.2 Communication

We implemented two different communication layers, one using GASNet [8], and one using custom-built RDMA functions via the InfiniBand Verbs API [1] called RDMAX, which is optimized to provide lower overhead for operations specific to the design of PPL. RDMAX is a header-only API which utilizes the C++11 template and closure systems, providing the compiler with more opportunity for critical code path optimization. One example where this might be beneficial is in inlining the processing code when a communication request is complete. Traditionally, function pointers are used, giving the compiler little information with which to perform appropriate transformations.

Further, the communication layer implementation supports using dedicated thread(s) to send data over the network. To do this, communication requests are submitted to an MPSC queue which is accessed by a communication thread. A synchronization handle is attached to each request, allowing the requesting task to wait on completion of the request. Once complete, the communication thread performs the associated synchronization operation on the handle to wake the requesting task. When possible, we use the threading layer to implement the dedicated threads, otherwise resorting to Pthreads.

4.3 Threading

The threading module is implemented to be flexible enough to use any underlying threading library. This is done in two ways. First, we do not place restrictions on the type of threading model (i.e. kernel or lightweight). Second, thread objects are templated with the function and arguments in such a way as to expose the *true* function header to the programmer, instead of the typical approach where an opaque pointer is passed as a single parameter, forcing the programmer to pack and unpack thread arguments.

We implemented two different threading layers, one using Qthreads [34] and one using Argobots [27]. We chose both implementations to use a lightweight threading runtime due to the small amount of scheduling and context-switching overhead. This allows our applications to use very fine-grained execution in order to produce massive amounts of concurrency for hiding latency and improving load-balance.

In both implementations, the synchronization primitive has full/empty-bit (FEB) semantics. This works very nicely with Qthreads's implementation, as the Qthreads runtime already utilizes FEBs for all thread management and synchronization. However, Argobots does not directly provide an interface for FEBs. Instead, it provides more common synchronization primitives such as mutually-exclusive locks and condition variables; we utilize these primitives to build the FEB synchronization in the Argobots implementation.

5. EVALUATION

We conducted experiments using the Stampede supercomputer, located in the Texas Advanced Computing Center [30]. It consists of 6400 compute nodes each with two

	`nocache`	`noevict`	`evict`
1 B	7.4%	15.3%	17.7%
1 KiB	8.3%	6.6%	14.8%
1 MiB	0.8%	0.2%	0.6%

Table 1: Cache overhead for different policies and data sizes.

Intel Xeon E5-2680 (eight-core) processors and 32 GB of memory. Each node also has at least one Intel Xeon Phi coprocessor, but were not used in our evaluation. The C/C++ compiler was GCC 4.7.1, the GASNet version was 1.22.4, the UPC version was 2.20.2, and the MVAPICH2 version was 2.0. Unless otherwise mentioned, experiments show PPL using RDMAX. Further, all of our experiments use two dedicated communication threads, one for managing network events and one for managing thread synchronization. Finally, error bars representing one standard deviation are included for every graph except for the sparse triangular solver for sake of readability.

5.1 Runtime overhead

We conducted several experiments to evaluate the overhead of our PPL implementation. These tests are divided by module and are described in the following subsections.

5.1.1 Memory Management

Since setting up the heaps is a one-time cost, our focus is on evaluating the overhead of the software caching mechanism. We performed a large number of one-sided get operations between two nodes with each of PPL's caching policies, and compared this to the time to perform the same number of gets directly. Only one of the nodes performed communication, to reduce network noise; however, network and scheduling noise remain present. Multiple data sizes were used to verify that data size has little impact. These results are presented in Table 1. Our overhead is small in every case and decreases as data size increases due to memory allocation dominating, although some variation remains. The greater overhead of the `nocache` policy in the 1 KiB and MiB cases is an artifact of our implementation. The results indicate that if there is even moderate data-reuse in an application, the overhead will not outweigh the savings of reduced communication.

5.1.2 Communication

To evaluate the overhead of providing one-sided communication through objects and a dedicated communication thread, we ran experiments which test `gptr` get operations against simply performing the communication operation through the underlying communication layer implementations. The results for these tests are presented in Figure 1, showing the overhead of the PPL communication module implementations and raw performance results using RDMAX and GASNet without PPL. The overheads range from 3 to 4.5 μs for most data sizes, and is attributed to two factors: data movement associated with the communication request, and descheduling and rescheduling requesting threads. PPL using RDMAX has more incurred overhead compared to PPL using GASNet because RDMAX alone has less features than GASNet, most of which are implemented in the RDMAX communication module for PPL due to necessity. Our results indicate that RDMAX outperforms GASNet by at least

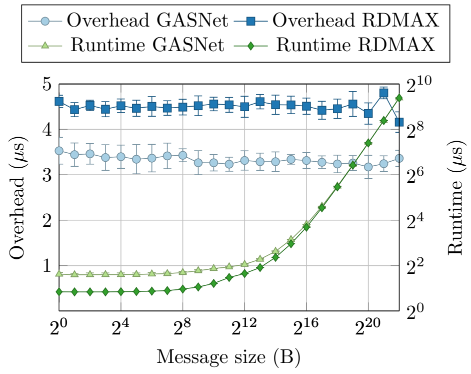

Figure 1: Performance and overhead of communication module implementation for GASNet and RDMAX.

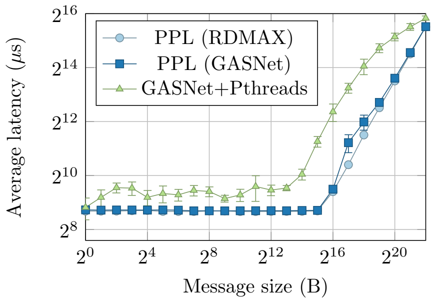

Figure 2: Communication latency for PPL and GASNet+Pthreads with 64 threads communicating simultaneously. PPL (RDMAX) outperforms PPL (GASNet) by 8 and 720 μs, on average, for small ($<$ 64 KiB) and large ($\geq$ 64 KiB) messages, respectively.

1 μs for nearly every message size.

Although the overhead is relatively high for small messages, our communication approach is beneficial when multiple threads communicate simultaneously, as indicated in Figure 2. At 64 threads, PPL outperforms GASNet+Pthreads by 2.7x on average. Some of this is attributed to the greater overhead of context-switching kernel threads (a few μs [33]). By using dedicated threads for communication and synchronization, we better coordinate large numbers of communicating threads, thus reducing contention in place of accessing highly concurrent data structures. We expect to further reduce the overhead through continued development.

5.1.3 Threading

We evaluated the overhead of the threading module via creating and joining empty threads. The test used one core to isolate both methods and to eliminate potential measurement inaccuracies. The results in Figure 3 show the incurred overhead of the PPL threading module for thread creation and joining. That is, the results indicate the added per-thread execution time of both methods with respect to the corresponding library implementation. For example, the total time of joining a single thread in PPL using Qthreads is between 40 and 50 ns higher than joining a thread in Qthreads. The overhead of using Argobots is largely due to the implementation of our synchronization object. This is not an issue for Qthreads since we utilize its FEB prim-

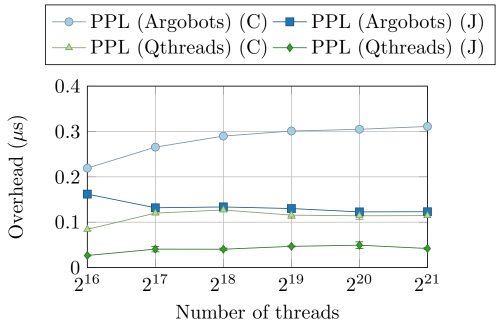

Figure 3: Per-thread incurred overhead of PPL for creation and joining.

Number of Bodies	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
PPL (Qthreads)	1.3	2.9	4.9	6.5	5.1
PPL (Argobots)	2.6	8.0	13.0	13.4	8.8

Table 2: Speedup of PPL over UPCR-BH on 1024 cores.

itives. We expect to reduce the overheads through better implementations of our module and through tighter integration with the threading libraries.

5.2 Application performance

To evaluate PPL as a whole, we have chosen three microbenchmark applications to use for experiments: the Barnes-Hut algorithm, a Monte Carlo particle tracking algorithm, and a sparse triangular linear system solver.

5.2.1 Barnes-Hut

The Barnes-Hut (BH) algorithm is an approximation algorithm to the n-body problem. BH approximates the interaction of a body with a collection of other bodies (cells) deemed *far enough* by considering the collection as a single body, using its center of mass as its location [25]. The algorithm partitions and organizes the 3D space of bodies into an octree, where each cell is recursively divided into child cells, representing one octant of the parent cell. This recursion stops once the number of bodies in a cell reaches a fixed threshold. Once the octree is constructed, for each body, the tree is traversed top down until nodes representing "far-enough" cells are reached in order to calculate the forces acting on the body and progress the simulation.

Our BH implementation uses the algorithm implemented and described in [36] (UPCR-BH) with the exception of the force computation phase being implemented using PPL with **gptrs** as tree nodes. By spawning a thread per body, we are able to better load-balance large amounts of threads and more effectively hide latency-prone operations. Further, the caching subsystem ensures redundant communication is prevented through the use of the `noevict` cache policy. This algorithm was chosen for its property of stressing the threading module with up to millions of threads per node. It also shows the benefit of a high incidence of synergistic caching, since bodies stored at the same process tend to be nearby and tend to interact with the same cells. Additionally, the resulting code is much simpler compared to UPCR-BH which hand-tuned several techniques for optimization.

The results of our BH implementation for PPL are presented in Figure 4. The tests were run for 65K to 4M bodies on up to 1024 cores (64 nodes). We were unable to gather results for 4M bodies for Argobots on 2 nodes due to per node memory requirements surpassing the available memory. Table 2 presents speedup of our two implementations against UPCR-BH on 1024 cores.

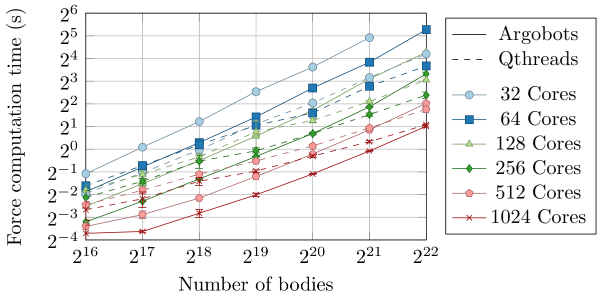

Figure 4: Force computation time for PPL using Qthreads (dashed) and Argobots (solid) on 32 to 1024 cores.

The trends in Figure 4 indicate that the use of Qthreads results in better scaling as the number of bodies increase. This is because Qthreads natively uses a work-stealing thread scheduler and Argobots does not; we simply perform round-robin scheduling with Argobots. Thus, Qthreads is better equiped to remedy the imbalanced work loads of BH. On the other hand, there are instances where Argobots outperforms Qthreads, primarily for large node counts (i.e. 32 and 64 nodes). This is due to the added overhead in communication synchronization for Qthreads which internally spawns a new thread to perform the operation when attempted from a non-Qthreads entity. In contrast, Argobots allows us to synchronize without spawning a new thread.

5.2.2 Monte Carlo particle tracking

Monte Carlo particle tracking (MCPT) algorithms use random sampling processes to approximate the diffusion of particules through a structure. Such problems typically involve frequent lookups of very large, static cross-section data that is too large ($\geq$100GB) to be stored on a single compute node. Our MCPT implementation is designed for neutron transport problems and is based on the Energy Band Memory Server (EBMS) algorithm [13]. The key idea is to decompose the cross-section data into energy bands that are distributed across a set of memory servers from which data is requested as needed. By processing particles within an energy band, data locality can be exploited.

Our PPL implemention makes use of the memory model to combine memory and tracking servers thus simplifying access to cross-section data. Further, implicit caching using the `evict` policy enables easy exploitation of locality through synergistic caching. We compared this with an MPI-3 implementation [35] of the EBMS algorithm which also combines memory and tracking servers. Data is stored using MPI-3 shared memory methods, providing access for every MPI process on a node. There is no threading, but particles are distributed among every rank, and communication makes use of non-blocking collective operations.

The results of the PPL and MPI-3 implementations of EBMS are presented in Figure 5. These were run using 50 million particles and 128 GB of cross-section data in 128 en-

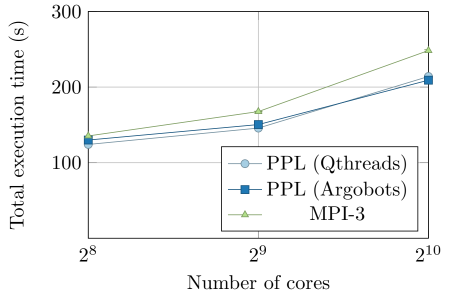

Figure 5: Total execution time of EBMS implementations for PPL and MPI-3.

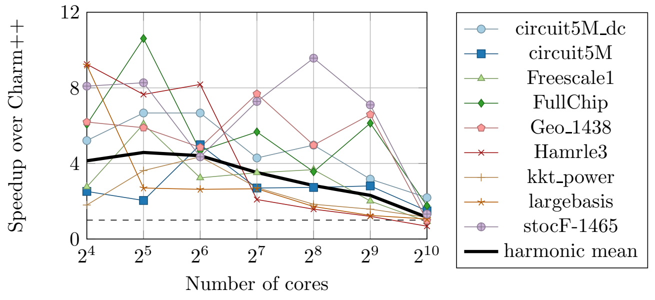

Figure 6: Speedup of sparse triangular system solver implementation in PPL using Qthreads over Charm++ 6.6.1 on SMP for various matrices.

ergy bands at the largest scale, and scaled to have a constant problem size per processor. We did not test at fewer than 256 cores because the small amount of communication does not accurately represent the problem. PPL is slightly faster than the MPI-3 version in all cases. Qthreads is slightly faster than Argobots except at the largest scale; but in general, both have very similar performance. The PPL versions spend additional time tracking particles due to using dedicated communication threads, but caching and better communication/computation overlap help performance.

To better understand the effectiveness of the software cache, we also examined its hit rate for our EBMS implementation. Hit rates are consistently above 97%, which we attribute to good synergistic caching and locality.

Finally, a key point is that implementing EBMS using PPL's `gptr`s and caching is both natural and simple, whereas the MPI-3 implementation is complex.

5.2.3 Sparse triangular linear system solver

Solutions to sparse triangular linear systems are often the kernel for many numerical applications that arise in science and engineering simulations. However, due to the lack of concurrency from structural dependencies in the matrix and the small computation per non-zero entry, it is difficult efficiently parallelize. We have implemented a system modeled after an algorithm presented in [31], which is implemented in Charm++. To provide a data-driven execution model needed for this type of problem, we use active messages.

Figure 6 presents speedups of our implementation over the Charm++ implementation for a number of the matrices as described in [31] (the performance of Argobots is relatively similar, thus is not shown). We see at least 2x speedup over Charm++ in most cases, and 3.3x on average. PPL's advantage over Charm++ degrades for a large number of nodes due to a decrease in on-node parallelism and an increase in communication. For both PPL and Charm++, communication will eventually become the dominant factor in performance.

6. CONCLUSION

We have presented the design and an in-development implementation of PPL, a new C++11 parallel runtime for studying future programming models. It supports RDMA communication, active messages, lightweight threading, event-driven task scheduling, and implicit caching, which greatly simplify the development of many algorithms. These high-level concepts also offer opportunities for performance improvements through better utilization of multitasking to interleave communication and computation while avoiding unnecessary communication.

The implementation of PPL is recent and we expect to improve its performance in future work. In particular, we expect that the overheads in the communication layer can be reduced by improving the interaction between communication threads and communicating tasks. Further, we plan to reduce the overhead of the threading module to be more competitive with directly using Qthreads or Argobots and to improve the performance of our software cache under the load required by memory- and communication-intensive applications, like EBMS at scale.

Finally, we plan to develop better performance models for PPL and other similar runtimes: the close interaction between scheduling and communication implies that performance models which study each in isolation are no longer adequate.

Acknowledgments

The research presented in this paper was funded through the NSF CISE CCF grant 1337217 and the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories, in the context of the Extreme Scale Grand Challenge LDRD project. The work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575. Finally, our thanks to Andrew Siegel for providing an MPI EBMS implementation.

7. REFERENCES

[1] Mellanox technologies. `http://www.mellanox.com`.

[2] B. Acun, A. Gupta, N. Jain, et al. Parallel programming with migratable objects: Charm++ in practice. In *Supercomputing 2014*, pages 647–658. IEEE, 2014.

[3] A. Amer, H. Lu, Y. Wei, et al. MPI+Threads: Runtime contention and remedies. In *PPoPP 2015*, pages 239–248, New York, NY, USA, 2015. ACM.

[4] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward efficient support for multithreaded mpi communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 120–129. Springer, 2008.

[5] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Fine-grained multithreading support for hybrid threaded mpi programming. *International*

Journal of High Performance Computing Applications, 24(1):49–57, 2010.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *Supercomputing 2012*, page 66. IEEE Computer Society Press, 2012.

[7] K. Bergman, S. Borkar, D. Campbell, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[8] D. Bonachea. GASNet specification, v1.8. `http://gasnet.lbl.gov/#spec`, November 2008.

[9] P. Charles, C. Grothoff, V. Saraswat, et al. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPlan Notices*, 40(10):519–538, 2005.

[10] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, et al. An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In *PPoPP 2005*, pages 36–47. ACM, 2005.

[11] K. Devine, E. Boman, R. Heaphy, et al. Zoltan data management services for parallel dynamic applications. *Computing in Science & Engineering*, 4(2):90–96, 2002.

[12] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. Enabling concurrent multithreaded MPI communication on multicore petascale systems. In *Recent Advances in the Message Passing Interface*, pages 11–20. Springer, 2010.

[13] K. G. Felker, A. R. Siegel, K. S. Smith, P. K. Romano, and B. Forget. The Energy Band Memory Server Algorithm for Parallel Monte Carlo Transport Calculations. In *Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo*, 2013.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[15] W. Gropp and R. Thakur. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595–604, 2007.

[16] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley upc compiler. In *ICS '03*, pages 63–73. ACM, 2003.

[17] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014.

[18] L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *OOPSLA '93*, pages 91–108, New York, NY, USA, 1993. ACM.

[19] T. Mattson, R. Cledat, Z. Budimlic, et al. OCR: The open community runtime interface version 1.1.0. `http://xstack.exascale-tech.com/git/public?p=xstack.git;a=blob;f=ocr/spec/ocr-1.0.0.pdf`, June 2015.

[20] MPI Forum. MPI: A message-passing interface standard version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, Sept. 2012.

[21] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[22] OpenMP Architedcture Review Board. OpenMP application program interface version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 213.

[23] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2007.

[24] A. Righi. umalloc.c file reference. `http://minirighi.sourceforge.net/html/umalloc_8c.html`. Accessed: 2014-10-17.

[25] J. K. Salmon. *Parallel hierarchical N-body methods.* PhD thesis, California Institute of Technology, 1991.

[26] V. Sarkar, W. Harrod, and A. E. Snavely. Software challenges in extreme scale systems. In *Journal of Physics: Conference Series*, volume 180, page 012045. IOP Publishing, 2009.

[27] S. Seo, A. Amer, P. Balaji, P. Beckman, C. Bordage, G. Bosilca, A. Brooks, A. CastellÃş, D. Genet, T. Herault, P. Jindal, L. V. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, and Y. Sun. Argobots: A lightweight low-level threading/tasking framework. `http://collab.mcs.anl.gov/display/ARGOBOTS/`, 2015.

[28] M. Si, A. J. Peña, P. Balaji, et al. MT-MPI: Multithreaded MPI for many-core environments. In *ICS '14*, pages 125–134. ACM, 2014.

[29] H. Tang and T. Yang. Optimizing threaded MPI execution on SMP clusters. In *ICS '01*, pages 381–392. ACM, 2001.

[30] Texas Advanced Computing Center. Stampede. `portal.tacc.utexas.edu/user-guides/stampede`. Accessed: 2015-01-13.

[31] E. Totoni, M. T. Heath, and L. V. Kale. Structure-adaptive parallel solution of sparse triangular linear systems. *Parallel Computing*, 40(9):454–470, 2014.

[32] UPC Consortium. UPC language specifications v1.3. `http://upc.lbl.gov/publications/upc-spec-1.3.pdf`, 2013.

[33] V. M. Weaver. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, page 80, 2013.

[34] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS 2008*, pages 1–8. IEEE, 2008.

[35] J. Zhang. MPI-3 EBMS. `http://github.com/ANL-CESAR/EBMS`, 2015.

[36] J. Zhang, B. Behzad, and M. Snir. Design of a multithreaded Barnes-Hut algorithm for multicore clusters. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):31–36, 2015.