

# An asynchronous runtime system using RDMA and lightweight tasks

Hoang-Vu Dang and Alex Brooks and  
Nikoli Dryden

Department of Computer Science  
University of Illinois at Urbana-Champaign, IL,  
USA

{hdang8, brooks8, 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

We describe a novel runtime system that integrates a task model with RDMA communication and software caching. For evaluating the runtime system, we design two microbenchmarks and implement three applications: Barnes-Hut, Sparse triangular linear solver, and Monte Carlo particle tracking. The resulting codes are simpler, since load balancing across cores and latency hiding are hidden in the runtime. This also results in them performing better than state-of-the-art implementations of the same algorithms by up to 13, 10.8, and 3 times, respectively.

## 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

Barnes-Hut, Monte Carlo particle transport, Sparse triangular solver, PGAS, Lightweight task, Qthreads, RDMA.

## 1. INTRODUCTION

Future supercomputers will soon run hundreds of physical threads per node. The speed of different nodes and threads within nodes will exhibit a large variability, due to dynamic power management and error recovery [2, 37]. Both inter- and intra-node parallelism must be exploited effectively on such platforms. Moreover, problems such as load balancing, latency hiding, and managing communication worsen at this scale. In order to implement efficient applications, new techniques are necessary.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

Task libraries using dynamic tasking and work stealing are often used to handle large-scale shared-memory parallelism, including systems such as Cilk, Intel TBB, the Qthreads library, and OpenMP 3 [15, 32, 44, 27]. Several research projects have considered extending work stealing and dynamic tasking beyond the node level [12, 22]. An alternative that makes sense for many applications is to use a static, algorithmic partition of work among nodes, and schedule tasks dynamically within nodes. This is the model supported by MPI+OpenMP code and for systems such as DaGuE [4].

An obvious problem with such models is the interaction between the tasking library and the communication library. The usual approach with MPI+OpenMP, namely of communicating only within sequential OpenMP sections, is not scalable to hundreds of threads, and does not help with latency hiding and asynchrony. While MPI supports multithreaded processes, performance diminishes rapidly when the number of threads increases [40]. The problem can be attributed in part to implementation issues (coarse-grain locking and inefficient polling), but it is also inherent to MPI semantics: any thread could, potentially, be the consumer of any arriving message.

An alternative approach is to enable all threads to communicate and use blocking RDMA calls: any thread can execute a remote “get” (read) or “put” (write) and the thread blocks until the remote value is delivered or until the write is committed; compute resources can be reused by another thread if the memory access has a high latency. Since there is no dynamic matching of threads to sends and receives, incoming messages can be easily associated with the waiting thread. MPI endpoints [11] is another approach that works since threads are associated with statically defined channels.

This programming style is very similar to hardware-supported multithreading, with one exception: it does not provide caching. Caching reduces the effective memory latency, because of cache reuse by the same thread, or by other threads (synergistic caching). By analogy, we will want to use the node local memory as a cache for remote values, in order to support reuse by the same thread or by other threads on the same node.

A long history of research on Distributed Shared Memory [17] and on caching runtimes [10] clearly indicates that transparent, coherent caching is not a scalable approach. For caching to work, the user needs (a) to control what a “cache line” is: the system cache objects, rather than consecutive bytes in memory; and (b) the user needs to take care of co-

herence. Both are easy in many HPC applications, since (a) programmers specify which entities are moved across nodes; (b) the status of objects (read or written) is changing infrequently, in a bulk-synchronous manner; and (c) write conflicts are often due to reductions that can be handled as a special case.

Our paper provides an initial exploration for this style of programming, using several irregular parallel application kernels. We have designed and developed PPL, a new C++ runtime library which evolved from our previous work on the Barnes-Hut algorithm using a PGAS model [47, 48]. Our runtime library combines a dynamic tasking system with an RDMA communication library and a caching service. To the best of our knowledge, this is the first system that combines these three ingredients. We demonstrate the convenience and performance of this model by implementing the following applications: an  $n$ -body simulation using the Barnes-Hut algorithm, a Monte-Carlo particle tracking algorithm, and a sparse triangular solver. Our code is simpler and performs significantly better than other alternative implementations of the same algorithms. These applications were chosen because they require irregular computation, stress the memory and communication subsystems, and require extensive synchronization, respectively.

The rest of the paper is organized as follows. Sections 2 and 3 describe our programming model and its current implementation. Section 4 describes the three algorithms we implemented and Section 5 presents the results of our experiments. We end with a description of related work and conclude in Sections 6 and 7.

## 2. PPL DESIGN

PPL is designed for abstracting a model of computation which combines a communication layer using one-sided communication, a threading model supporting task-parallelism, and a caching service.

### 2.1 Tasking Model

We do not distinguish in our design between tasks and threads. The implementation can freely choose a threading library to realize the idea. However, the full benefit of our design is achieved with tasks that are scheduled by the runtime (i.e. a *task model*). Our tasking interface provides the following methods:

- **spawn**: spawn a task to execute a function.
- **future**: spawn a task, but delay execution until a requirement is met.
- **yield**: preempt the executing task.
- **sync**: synchronization primitive for enabling or disabling the execution of a task.

### 2.2 Communication Layer

The communication layer provides an interface for performing one-sided communication. The three basic methods are:

- **memget**: perform a read operation from a local or remote memory address into local memory.
- **mempup**: perform a write operation from local memory into a local or remote memory address.
- **active\_msg**: perform **mempup** and additionally execute a registered function at the remote node.

When **memget** or **mempup** return, they indicate local completion. A handler (**sync**) is associated with each execution of **memget** and **mempup** which can be used to poll for remote completion, either by testing or waiting. If an executing task is required to wait, it will be preempted at least until the communication is finished, providing an opportunity for other *runnable* tasks to be executed; an implementation of the communication layer must provide a mechanism for preempting any communicating task. The **active\_msg** operation is useful in data-dependent problems and can be used together with the **sync** primitive to provide lightweight remote synchronization.

### 2.3 Memory Model

PPL uses a PGAS memory model. Each process has two distinct heaps: a *global heap* and a *local heap*. The local heap contains locations which can be accessed only by locally executing threads, while locations on the global heap can be accessed by any thread. The access to an address on a remote portion of the global heap is done using *global references*.

There are three types of global references described in PPL: global variables (**gvar**), global vectors (**gvec**), and global pointers (**gptr**). A **gvar** is allocated on a single node and provides read-only access to remote nodes. A **gvec** is similar to a Fortran Co-Array [25]. The implementation must ensure that the local chunks have the same base offset from the beginning of the heap on each node, enabling a thread to correctly compute the address of a global vector location from a remote node. A **gptr** can be used to refer to any location in a local heap or remote portion of the global heap.

Locations in the global heap can be accessed using one-sided operations, such as **put** or **get**. These operations may result in an access to the global heap on the executing or a remote node. In the later case, the invocation results in a communication request which will preempt the executing thread.

Caching is essential for good shared memory performance. It becomes increasingly important when nodes run a large number of threads, because of the increasing opportunities for *synergistic caching*: a remote location accessed by one thread may also be accessed by other threads on the same node. Many UPC codes perform caching explicitly by copying data from remote nodes to the local heap. PPL provides implicit caching, where remote memory is cached in local memory; however, it does not provide implicit coherence. Implicit coherence does not scale well and most parallel scientific algorithms proceed by well-defined phases; coherence operations can be associated with the beginning or end of a phase.

PPL provides a cache interface for which any caching policy may be implemented in software. The interface requires the following operations: adding an object to the cache, accessing or updating a cache entry, and removing an object from the cache.

In order to provide an opportunity for optimization with regards to accessing remote data, PPL specifies three types of **get** operations for global pointers: **lget**, **rget**, and **get**. First, **lget** is simply a local **get**; it is assumed there is a cached copy of the variable. The corresponding cache entry is returned, otherwise an exception is thrown. Next, **rget** ignores any cached copies of the **gptr** and always requests a

new copy from the corresponding remote node. This will update the old cached copy and return the new reference once the request is complete. Finally, `get` is a fail-safe generic get operation. If there exists a cached copy of the `gptr`, then the reference is returned. Otherwise, a request for a new copy is sent to the corresponding remote node, updating and returning the new cached copy once the request is complete. Currently there are no changes to the cache when a put operation is performed, for simplicity.

### 3. PPL IMPLEMENTATION

#### 3.1 Tasking Model

In most cases, supporting a large number of threads can be expensive, especially when context-switching is performed regularly. Although the PPL threading model supports any threading library, it is expected that using a lightweight task model will be more efficient than a standard system-level threading model (i.e. POSIX threads). By using lightweight tasks, spawning and context-switching latencies are reduced. Further, applications and libraries are unable to explicitly wake system-level threads, since the scheduler is completely dependent on the operating system. Lightweight task libraries typically implement their own scheduler, which includes a load-balancer and mechanisms for preempting and enabling tasks. For these reasons, our first threading model implementation uses a lightweight task library, specifically the Qthreads library [44].

Qthreads provides a synchronization primitive which allows tasks to wait on the status of a single bit in memory (i.e. a full/empty bit). This method was originally seen in the Denelcor HEP system for guaranteeing correct ordering of memory operations [39] and is still seen in the Cray XMT architecture as a form of low-overhead synchronization for simple parallel programming models [23]. In order to efficiently couple communication completion with thread scheduling, PPL implements `sync` using the full/empty-bit primitive provided by Qthreads. This ensures that a task waiting for communication completion is preempted and efficiently rescheduled when it can continue execution.

Qthreads supports controlling the number of execution units (*workers*) and associating them with task queues (*shepherds*); workers are typically implemented as POSIX threads. It also provides automatic load-balancing via task stealing between shepherds. In PPL, we fix the number of shepherds and assign one worker per shepherd.

#### 3.2 Communication Layer

In order to efficiently support a large number of tasks performing communication, it is necessary to avoid polling on the workers. Hence, we dedicate polling to a different thread. Our implementation splits the execution of communication calls between the calling task and a *communication engine* (CE). The CE is defined as a dedicated service worker that executes parts of the communication code which must be atomic. This avoids the need for locking and mutual exclusion, which is an important reason why thrashing exists when many tasks communicate simultaneously. Further, this design facilitates leveraging more intelligent NICs, since functions can shift between the NIC and CE without affecting other components. We expect that, on future systems, an intelligent NIC will be able to completely replace the CE worker.

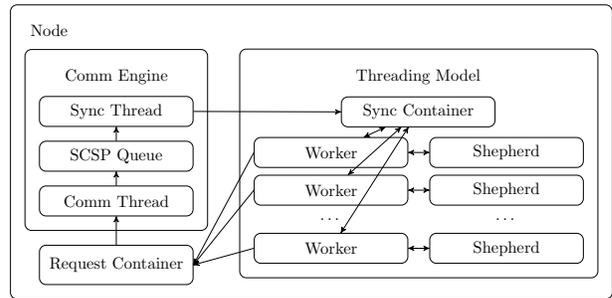


Figure 1: PPL Implementation using Qthreads for the threading model and two threads for the communication engine. *Sync Container* is the Qthreads scheduler.

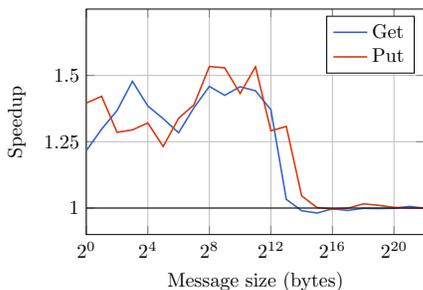
We implement the communication layer assuming that a NIC is serviced by a single CE. As modern NICs provide multiple independent virtual interfaces, this assumption is not too restrictive. Larger systems may require multiple CEs, but a suitable partition of traffic can ensure that there is no synchronization between multiple CEs.

Communicating tasks submit requests to the CE through a *request container* (RC). When a task performs communication, the request is submitted to the RC and the task could wait on the associated flag for completion. When a communication request is complete, the CE notifies the scheduler by changing the state of the flag to re-enable the task.

Since communication is completely offloaded to the CE, the scheduler is able to better manage tasks which are waiting for communication requests. Progress on a request continues even if a task is preempted. Therefore, in the general case, it is most efficient to yield a task once it submits a request to the RC, provided that there are other tasks waiting to be scheduled. Likewise, it is ideal to attempt to reschedule a task once notified by the CE that a communication request is complete. The communication library we use initially for PPL is GASNet [3], in order to simplify development and provide portability. We also have an experimental communicator based upon the *ibverbs* Infiniband library.

Through empirical results, we find that compared to the latency of communication, the overhead of preempting and re-enabling tasks is substantial. Hence, we use two pthreads to implement the CE: one is dedicated to executing the RDMA calls associated with the communication requests (*Comm Worker*); the second is dedicated to managing synchronization (*Sync Worker*). When a request is submitted to the RC, it is labelled with the corresponding RDMA operation. Once the Comm Worker performs the operation, it relabels the request as waiting and places it back into the RC to be tested at later time. Periodically, the Comm Worker will test for any completed requests. When a request is complete, it is placed in a secondary queue so that the Sync Worker can wake the requesting task. This design overlaps submitting and polling communication requests with waking preempted Qthreads tasks and also improves locality. The RC is implemented using a lock-free MPSC queue [21] while an SPSC queue is sufficient for the secondary queue.

The benefits of using two pthreads for the CE are shown using a simple latency test for put and get operations. Figure 2 shows the speed up of using two pthreads versus one pthread in the approach described above. For data sizes



**Figure 2: Speedup for using two threads for CEs in comparison to one thread in a put and get latency experiment.**

up to 8 KB, it improves performance by up to 48% for get and 53% for put. For larger data sizes, the performance is the similar due to the synchronization latency being much smaller than the overall communication latency.

The importance of using two pthreads for implementing the CE does not necessarily come from the improved performance for small messages. For many applications, there may be no benefits. We instead focus our analysis on the benefits with respect to future hardware. Considering an implementation of PPL which places the CE on an intelligent NIC, the latency of notifying the scheduler of a completed communication request should be negligible. By using two pthreads to implement the CE, we can better understand the behavior of future implementations of PPL which use intelligent NICs.

Figure 1 illustrates the interaction between the communication layer and threading model in the current PPL implementation.

### 3.3 Memory Model

At the initialization of PPL, a segment of memory is pre-allocated for one-sided communication and registered to the RDMA device. The preallocated memory is split into two equal segments for the local and global heaps. `umalloc` [31], a memory manager that dynamically allocates space for objects on custom memory addresses, is used for allocating global references to the custom heaps.

A `gptr` is the basic global reference used for interacting with the memory model. It is a class which contains a single structure holding the remote memory address and the associated node id. Coupled with `gptr` is a software cache engine. We currently implement several different cache policies that applications may use: no caching at all, caching without eviction, and least-recently used (LRU) caching. Internally, these caches use a concurrent hash table to store `gptr` data on the local heap. Depending upon the accessor, a query to the cache will determine if the entry is present and perform the appropriate operations.

## 4. APPLICATIONS

### 4.1 Barnes-Hut Algorithm

An  $n$ -body simulation is the problem of modelling the evolution of a system of  $n$  bodies interacting with each other through forces. A direct approach to this problem computes the force of one body with respect to all other bodies,

yielding  $\Theta(n^2)$  time complexity. This is impractical for a simulation involving a very large number of bodies.

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 (cell) deemed *far enough* by considering the collection as a single body, using the center of mass of the collection as its location [35]. The BH algorithm partitions and organizes the 3D space of bodies into an octree. Each cell is recursively divided into child cells that contain the bodies in one octant of the parent cell. The recursion stops when the number of bodies in a cell is below a fixed threshold. Once the tree is built, the center of mass of a group of bodies in a cell can be computed bottom-up. When computing the force of a body, the tree is traversed and the center of mass is used once the cell is deemed far enough. By using this hierarchical partitioning strategy and approximation, BH reduces the complexity of the simulation to  $\mathcal{O}(n \log n)$ .

Our BH implementation uses the algorithm implemented and described in [48] (UPCR-BH), with the exception of the force computation phase being implemented using PPL. After the tree generation phase, it becomes feasible to spawn a task for each body.

---

#### Algorithm 1 BH Force Computation with PPL

---

**Require:**

`node.handler` : `future(Localized(node))`

**procedure** BH-FORCE

**for all** `body`  $\in$  list of local bodies **do**  
   `spawn(ComputeForce(root, b)).sync()`

**end for**

**end procedure**

**procedure** COMPUTEFORCE(`node`, `body`)

**if** `NeedOpen(node)` **then**  
   **if** `!node.local` **then** `node.handler.sync()`  
   **end if**

`OpenCell(node, body)`

**else**

`BodyCellUpdate(node, body)`

**end if**

**end procedure**

**procedure** LOCALIZE(`node`)

**if** `node.local` **then** `return`  
  **end if**

**for** `child`  $\in$  `node.childs` **do**  
   `node.local = &child.gptr.get()`

**end for**

**end procedure**

---

Pseudo-code for the new force computation phase can be seen in Algorithm 1. `BH-FORCE` is the main procedure of the algorithm and is executed each step for each compute node. `ComputeForce` computes the interaction of a single body on a cell. `Localize` is used to localize children of unlocalized cells. Each execution of `ComputeForce` is done by a single spawned task using `spawn`, whilst `future` is used for spawning cell localization tasks. `NeedOpen` checks a body against the current cell and decides if it should be opened. Depending on the decision, `OpenCell` computes the appropriate calculation on the cell and recursively performs `ComputeForce`

on child nodes, while `BodyCellUpdate` updates the body position. The use of `futures`, `gptrs`, and the caching subsystem ensure that redundant communication is prevented.

## 4.2 Monte Carlo Particle Transport

Monte Carlo particle transport uses a random sampling process to approximate the diffusion of particles through a structure – e.g., the diffusion of neutrons through a nuclear reactor. Such codes are widely used and typically involve frequent lookups of very large, static cross-section data that cannot be stored on each node.

---

**Algorithm 2** The EBMS tracking algorithm.

---

```

while nalive < npars do ▷ until all particles absorbed
  for 0 ≤ n ≤ nbands do ▷ for each energy band
    retrieve En ▷ get energy band n
    for p do ∈ particlesn ▷ for each particle in band n
      repeat ▷ until absorbed or leaves band
        randomly sample interaction
        update particle
      until absorbed(p) or p ∉ particlesn
    end for
  end for
end while

```

---

A current state-of-the-art approach in the case of neutron transport is the energy band memory server (EBMS) algorithm [14]. This algorithm advances all particles in a loosely bulk synchronous fashion. The neutrons begin at high energy and tend to gradually lose energy until they are absorbed. Thus, if the cross-section data is decomposed into energy bands, accesses will be concentrated in one or a few bands at a time. The energy bands are distributed across a set of memory servers, from which processors tracking particles can request data as needed. In the implementation described, there is a separation between tracking nodes that compute interactions of neutrons and memory nodes that store and provide access to cross section information: a memory node runs a single-threaded server process whereas as many tracking processes as possible are placed on the tracking nodes.

The basic EBMS tracking algorithm is presented in Algorithm 2. Tracking processors retrieve an energy band then iterate over the contained particles until they leave the band. In practice, it takes two to three iterations over the energy bands to complete the algorithm [14].

PPL is suited to this problem due to its ability to naturally manage and cache the cross-section data and interleave the communication and computation to minimize latency. This allows us to have hybrid memory/tracking servers for no additional effort, and thus make better use of available computational power. Cross-section data is accessed via global pointers, allowing easy remote retrieval and taking advantage of the LRU cache policy to manage the memory on each node.

We have implemented two versions of this tracking algorithm. The first approach is a simple adaptation of the original EBMS algorithm, incorporating hybrid memory/tracking servers. The primary difference is that a lightweight task is spawned for each particle to handle the computations of the inner loop.

The second approach implements “prefetching” of energy bands and restructures the algorithm. A lightweight task is

spawned for each particle that persists until it is absorbed. These tasks now request cross-section data as needed, and are managed appropriately to avoid unnecessary duplication. This enables data to be retrieved in parallel based upon memory constraints.

## 4.3 Sparse Triangular Linear System Solver

We have implemented a multi-threaded sparse triangular linear solver application. Solutions of such 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 to parallelize and achieve high efficiency.

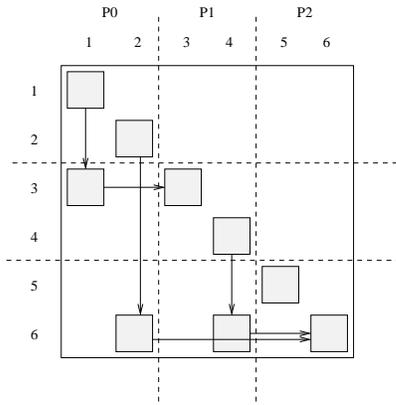
The problem is formulated as follows. Given a sparse linear system  $Lx = b$  such that  $L$  is an  $n \times n$  lower triangular matrix, solution vector  $x$  is computed using forward substitution by the recurrence:

$$x_i = b_i - \sum_{j=1}^{i-1} l_{ij}x_j/l_{ii}, i = 1, 2, \dots, n.$$

From the formula, each solution variable  $x_i$  depends on all previous  $x_j$ ; however due to the sparseness of  $L$ , most of  $l_{ij}$  are zero and  $x_i$  only depends on a small number of previously computed variables.

In a recent paper, Ehsan et al. [41] present an algorithm implemented in Charm++ which shows significant improvement over traditional solvers such as SuperLU [18] on a BlueGene/P. The method utilizes different levels of parallelism by partitioning the matrix and choosing the order in which non-zero entries are computed. It does so after an initialization phase to determine the dependencies and then dynamically generates working blocks (namely `chares`) for each process. The ordering in a `chare` is for computing the entries that satisfy the most dependencies, enabling more parallelism. A simple round-robin load-balancing is used to map `chares` to nodes. A process begins executing a `chare` when all of its dependencies are satisfied. These dependencies include the rows on the blocks to the left that need to be accumulated and the diagonal blocks at the same set columns. The Charm++ implementation uses a busy-waiting style such that when a process has no work, it waits for incoming data until receiving indication that dependencies are satisfied. Figure 3 shows an example of a triangular matrix and the dependencies between its entries.

In PPL, the data dependency problem is simplified, thus producing a simpler algorithm. Each node is assigned a stripe of contiguous columns. Within each node, the stripe is divided into sections of contiguous rows, where each section has a similar number of non-zeroes; each section is assigned to a task. This simple partitioning mechanism is shown to be efficient enough in the later evaluation section. A task thus has dependencies from tasks processing diagonal entries as well as from remote nodes having the same assigned rows. It is worth noting that sparse triangular solver performance is heavily dependent on the locality of memory accesses, since the solver has a low computation intensity. Hence a task is started when all of its dependencies are satisfied and executes until completion. This also reduces the overhead due to task preemption. For this purpose, we use a `counter` implementation of the `sync` which enables a task only when certain number of synchronizations are completed. A pro-



**Figure 3: Example of a triangular matrix of size  $6 \times 6$ . The arrows represent the dataflow when partitioning the matrix for 3 processes.**

cess which computes the entry at diagonal  $l_{ii}$  holds the result for solution component  $x_i$ , while the partial results need to be sent in a fashion as shown in Figure 3 using the PPL active message feature.  $x$  and the partial results are accessed using global pointers.

## 5. EVALUATION

We conducted evaluations on two systems. The first is Taub, a cluster at the University of Illinois at Urbana-Champaign [26]. Each compute node has at least 24 GB of RAM and two six-core processors. The network uses a QDR InfiniBand interconnect. On Taub, PPL was run with 10 shepherds.

The second system is the Stampede supercomputer, located in the Texas Advanced Computing Center [6]. It consists of 6400 compute nodes each with 32 GB of memory and two eight-core processors. Nodes are interconnected with a 56 GB/s FDR InfiniBand interconnect. Each node also has at least one Intel Xeon Phi co-processor, but these were not used in our evaluation. On Stampede, PPL was run with 12 shepherds.

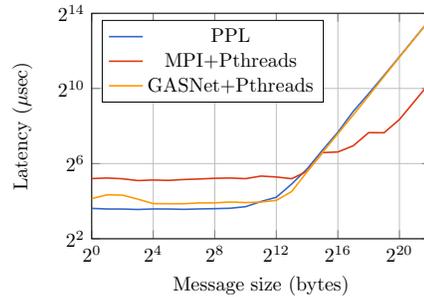
On both systems, the C/C++ compiler was GCC 4.7.1, the GASNet version was 1.22.4, and the MVAPICH2 version was 2.0

### 5.1 Microbenchmarks

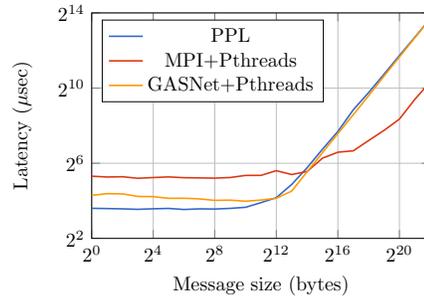
We conducted two microbenchmark experiments to evaluate the performance of PPL’s one-sided communication in comparison with MPI+Pthreads and GASNet+Pthreads. The experiments perform large amounts of remote put operations for varying size data transfers in a multi-threading environment. The experiments were run on the Taub system and averaged over five runs.

The first benchmark is designed to run on two nodes, spawning threads on each node. The threads are then paired and one performs put operations. The second benchmark is designed to run on two or more nodes. A designated node spawns one thread for each other node. Each thread then performs put operations to its associated node.

For the MPI+Pthreads benchmarks, an MPI memory window is used for each thread. The synchronization is done on each window after each remote put to ensure remote completion. There is no explicit synchronization for GASNet since



(a) Put latencies for 10 pairs of threads.



(b) Put latencies for 10 threads to distinct nodes.

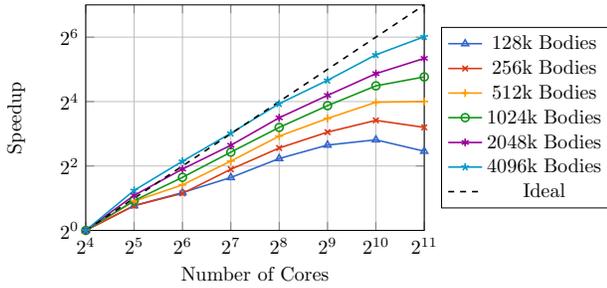
**Figure 4: Put latencies for data sizes ranging from 1 byte to 4 MB for the two microbenchmarks.**

we use the blocking put procedure; GASNet’s specification ensures remote completion when this function returns.

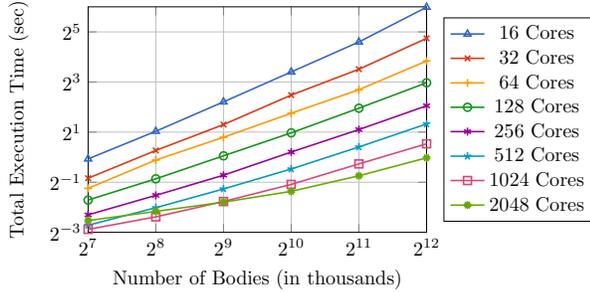
Figures 4a and 4b show the results of the first and second microbenchmarks, respectively. For message sizes smaller than 4 KB, PPL averages 34% and 48% smaller latency than GASNet+Pthreads, averaging 8% and 9% higher latency thereafter, respectively. Further, for message sizes up to 16 KB, PPL averages 2.7 and 3 times lower latency than MPI+Pthreads, respectively.

Since PPL uses GASNet as the underlying communication layer, it is reasonable that GASNet+Pthreads performs better than PPL for large messages. Communication latency can be split into two categories: submission time latency and network time latency (i.e. actual communication latency). The later is measured through the amount of polling performed until communication is complete. For small messages, the dominating factor is submission latency. However, at larger messages, it is typically necessary to poll more than once before communication completion. Since polling in PPL requires traversing the RC and polling for each individual request, the network time latency is larger compared to GASNet. It is presumed that this occurs for message sizes greater than 4 KB because the message transfer unit of the Infiniband device is 4 KB.

There are a few reasons which could explain why MPI+Pthreads performs better at larger sizes. First, for message sizes larger than a certain threshold, many MPI implementations, including MVAPICH, split messages and change communication modes, resulting in smaller overall latency for transfer setup and improved scalability on clusters with InfiniBand interconnects [19]. Second, the polling method for communication completion is performed in nearly every MPI



(a) PPL BH speedup compared to ideal in strong scaling tests.

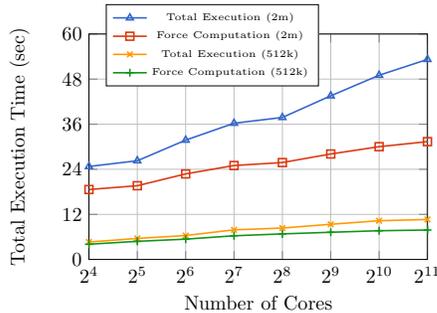


(b) PPL BH execution time in strong scaling tests.

**Figure 5: PPL BH strong scaling results.**

function invocation. Conversely, PPL only supports polling specific communication requests due to GASNet’s network polling specification. It is expected that the advantage of MVAPICH over PPL for large messages will reduce if PPL uses the same communication layer used by MVAPICH.

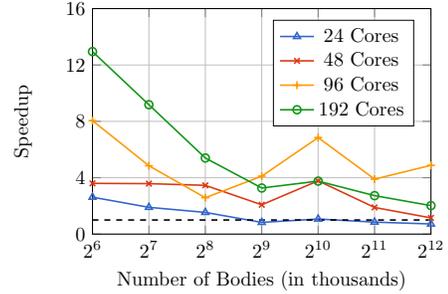
## 5.2 Barnes-Hut Results



**Figure 6: PPL BH weak scaling results.**

For our evaluation, input bodies were generated with the Plummer model [1]. We ran four time steps and excluded the first two, averaging the remainder. All computations involved a time step of 0.025 seconds, a tolerance of 0.5, and were done in double precision. We performed weak and strong scaling tests on Stampede, and compared our implementation with UPCr-BH on Taub.

Figure 5a shows the strong scaling results for PPL on Stampede, using one node as a baseline. For 4096k bodies, we achieve nearly linear speedup until we reach  $2^9$  cores (8k atoms per core), after which we drop slightly below the ideal speedup. However, as can be seen in Figure 5b, we continue



**Figure 7: Speedup in total execution time of PPL BH relative to UPCr-BH implementation.**

to achieve reasonable speedup in total execution time at any combination of bodies and cores.

Our weak scaling results for 512k bodies per node (32k per core) and 2048k bodies per node (128k per core) are presented in Figure 6. In both cases, we see that PPL continues to scale well as the problem size and core count increase. Computation time grows roughly logarithmically, as expected.

Finally, we compare PPL with UPCr-BH in Figure 7 using total execution time. Both implementations were run for problem sizes with total body counts ranging from 64k to 4096k on two to sixteen nodes on the Taub system. We find that PPL outperforms UPCr-BH for all body counts once we reach four nodes or more; at sixteen nodes, PPL is between 2 and 13 times faster than UPCr-BH, depending upon the problem size. Our performance increase primarily comes from improvements in the force computation phase. This is mainly due to the use of lightweight threads to overlap computation more efficiently, and hence make better use of the available processor resources.

## 5.3 Particle transport results

|               | Reference | Simple | Prefetch |
|---------------|-----------|--------|----------|
| Total runtime | 301.2     | 117.5  | 103.8    |
| Speedup       | 1.0       | 2.56   | 2.9      |
| Communication | 290.6     | 117.0  | n/a      |
| Tracking      | 10.6      | 0.53   | 1.40     |

**Table 1: Monte Carlo particle transport results: average running, communication, and tracking times, in seconds. Speedup is relative to the reference.**

We compare our Monte Carlo particle transport implementation with a reference implementation in MPI from [14] on a benchmark representative of typical problems. The benchmark is for 100 processes, 10 memory servers and 90 tracking. There are 10 energy bands, each 10 GB in size and 900,000 particles distributed evenly among the tracking processes. The comparison was performed on the Stampede system. We used our experimental PPL implementation atop iverbs due to problems with GASNet supporting the memory requirements.

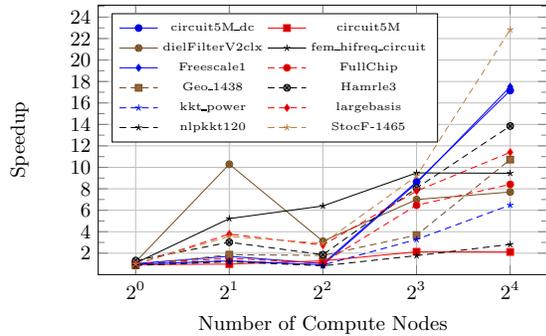
Due to memory requirements, the reference implementation was run with two tracking processes per physical node, requiring a total of 45 nodes for the tracking processes. Our implementation is able to make better use of the memory

on each node, and thus requires only six nodes. Considering the memory processors—which in our implementations are hybrid—brings the node counts required for the MPI and PPL versions to 55 and 16, respectively.

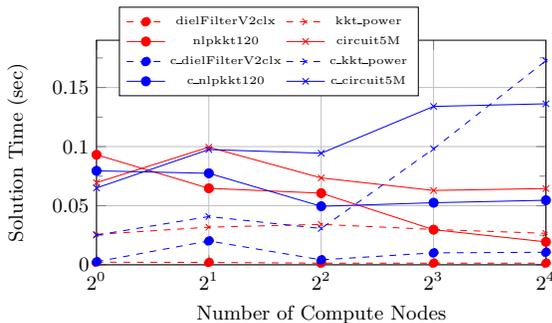
The benchmark was run five times and total running, communication, and tracking times were recorded and averaged. Total running time is the time required for all particles to be absorbed; communication time is the time spent waiting to receive cross-section data; and tracking time is the time spent performing tracking computations. These results are presented in Table 1. We do not report the communication time for the prefetch algorithm, since it may be retrieving multiple energy bands in parallel.

The prefetch version is 2.9 times faster than the reference implementation. Much of this improvement is a result of reduced communication time. The tracking time is significantly reduced due to the combination of employing hybrid memory/tracking servers and the lightweight tasking being better able to take advantage of available resources. The prefetch version spends significantly more time in tracking than the simple version—although still 7.5 times less than the reference version—due to the increased overhead of synchronization related to the prefetching. Note that with the simple version, we see a 2.5 times performance improvement with modest implementation effort.

## 5.4 Sparse Triangular Linear Solver results



(a) PPL implementation speedup for various matrices.



(b) PPL vs. Charm++; *c\_* prefix in the matrix name indicates the Charm++ implementation.

**Figure 8: Performance comparison between PPL and Charm++ sparse triangular solver implementations.**

We conducted experiments on the Taub cluster using the same set of matrices as [41] which come from an incomplete

LU factorization of matrices from various problems. The comparison between PPL and Charm++ implementations are shown in Figure 8. For Charm++, since its granularity is at a processing unit, we specified 1 to 192 processes with one process per core (i.e. 12 processes per node). The results are averaged over 5 runs of the same experiment. In Figure 8a, we observe that when the number of nodes increases, PPL speedup increases for all matrices; the average speedup at 16 nodes is 10.8. In Figure 8b, we compare the runtime for a few matrices with the same magnitude of runtime. For matrices that are identified as having high parallelism, such as *nlpkkt120*, PPL makes better use of overlapping synchronization and thus has better performance and scaling. The overhead of synchronization and communication in Charm++ is quickly worse, as can be seen in other matrices. Although the results shown in [41] are better on BlueGene/P for Charm++, which we suspect that is mainly due to better communication bandwidth. Based on our experiments, we expect PPL to perform with similar or better results.

## 6. RELATED WORK

On large scale systems, hybrid programming models have become part of mainstream research in recent years. A very common approach is to incorporate an MPI implementation with a threading library such as OpenMP [29], Habanero-C [9], and SMPSS [20]. MPI implementers are also looking to improve MPI performance by integrating threading libraries with the communication layer [38]. The hybrid MPI+Pthread library approach has shown improvements for several algorithms on multi-core distributed memory clusters. *n*-body simulations are among those studied, as shown in [13, 30]. These implementations, however, are based on the locally essential tree initially proposed in [35], which does not allow overlapping computation and communication very effectively. Several comparisons to different BH implementations on other programming models such as Charm++ [16] and PEPC [45] have been investigated in [48, 47].

The idea of “bringing the data to the particle” is suggested in [5]. PGAS has been employed to achieve this in the context of quantum Monte Carlo applications [24]. Similarly, decomposing cross-section data is seen in [33] for Monte Carlo particle transport. PPL’s key achievement in this respect is to apply these with lightweight tasking to better interleave communication and computation, and easily implement hybrid memory/tracking processors.

It is shown by Michael et al. [46] that synchronization such as `barrier` is among the most important factors that affect performance of hybrid MPI+Pthread sparse triangular solvers. Using PPL, we have efficient lightweight synchronization. Other research on locality improvement through better partitioning, such as in [34, 36], are complementary optimizations which could be applied on top of a PPL implementation.

PGAS programming languages such as Chapel [7] and X10 [8] have dedicated threading mechanisms, while UPC [42] and CAF [25] have added multi-threading support through language extensions. Chapel and X10 do not provide lightweight threading support; however, several efforts have been made to add lightweight threading support, such as in [43, 28]. Since lightweight thread features are added later to the design, they are still adhoc and not widely adopted. The PPL library natively supports lightweight threads and can be easily extended to any threading library through class

inheritance.

## 7. CONCLUSION

We have presented the design and an implementation of PPL, a new C++ parallel runtime system that combines RDMA technology, caching, and asynchronous tasks. We have demonstrated how PPL can reduce communication overhead for high task counts. We have implemented in PPL three applications of different types, achieving better performance with simpler code. The performance comes from better utilization of multi-tasking to more efficiently interleave communication and computation, while avoiding contention overheads. We simplify the code by using tasks to represent a natural unit of work without any explicit load-balancing and hand-tuned optimization for synchronization and locality. These are handled internally by the PPL runtime with workstealing, cache management, and one-sided communication.

As supercomputers advance toward exascale, future parallel applications will need to make greater use of inter- and intra-node parallelism in the face of increasing communication delays. The use of C++ allows extending and reimplementing features of PPL easily through class inheritance and polymorphism. This makes it an attractive platform for experimenting with different combinations of programming models. We plan to continue to improve PPL, extending it to support different underlying communication and threading mediums, while evaluating its performance on a wide range of problems.

## 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. The work was also supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

We thank Andrew Siegel for his help in providing and explaining the EBMS code.

## 8. REFERENCES

- [1] S. Aarseth, M. Henon, and R. Wielen. A comparison of numerical methods for the study of star cluster dynamics. *Astronomy and Astrophysics*, 37:183–187, 1974.
- [2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, 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.
- [3] D. Bonachea. GASNet specification, v1.1. *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*, 2002.
- [4] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.
- [5] F. B. Brown. Recent advances and future prospects for monte carlo. In *Proc. Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2010 (SNA+ MC2010)*, pages 17–21, 2010.
- [6] T. A. C. Center. Stampede user guide. <https://portal.tacc.utexas.edu/user-guides/stampede>. Accessed: 2015-01-13.
- [7] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
- [8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.
- [9] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with MPI. In *Parallel and Distributed Processing (IPDPS) 2013, IEEE 27th International Symposium on*, pages 712–725. IEEE, 2013.
- [10] W. Chen, J. Duell, and J. Su. A software caching system for "UPC". *Memory*, 1:P2, 2003.
- [11] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling mpi interoperability through flexible communication endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 13–18. ACM, 2013.
- [12] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.
- [13] T. V. T. Duy, K. Yamazaki, K. Ikegami, and S. Oyanagi. Hybrid MPI-OpenMP paradigm on SMP clusters: MPEG-2 encoder and n-body simulation. *CoRR*, abs/1211.2292, 2012.
- [14] 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.
- [15] 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.
- [16] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.
- [17] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter*, volume 1994, 1994.
- [18] X. S. Li. An overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.
- [19] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and

- implementation of MPICH2 over InfiniBand with RDMA support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 16. IEEE, 2004.
- [20] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero. Overlapping communication and computation by using a hybrid MPI/SMPs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 5–16. ACM, 2010.
- [21] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.
- [22] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, 2011.
- [23] D. Mizell and K. Maschhoff. Early experiences with large-scale Cray XMT systems. In *Parallel and Distributed Processing (IPDPS) 2009. IEEE International Symposium on*, pages 1–9. IEEE, 2009.
- [24] Q. Niu, J. Dinan, S. Tirukkovalur, L. Mitas, L. Wagner, and P. Sadayappan. A global address space approach to automated data management for parallel quantum monte carlo applications. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10. IEEE, 2012.
- [25] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.
- [26] U. of Illinois. Illinois campus cluster program. <https://campuscluster.illinois.edu/>. Accessed: 2015-01-13.
- [27] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, pages 49–56. ACM, 2011.
- [28] J. Paudel, O. Tardieu, and J. N. Amaral. Hybrid parallel task placement in X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, pages 31–38. ACM, 2013.
- [29] R. Rabenseifner, G. Hager, G. Jost, and R. Keller. Hybrid MPI and OpenMP parallel programming. In *PVM/MPI*, page 11, 2006.
- [30] H. Rein and S.-F. Liu. REBOUND: Multi-purpose N-body code for collisional dynamics. *Astrophysics Source Code Library*, 1:10016, 2011.
- [31] A. Righi. umalloc. [http://minirighi.sourceforge.net/html/umalloc\\_8c.html](http://minirighi.sourceforge.net/html/umalloc_8c.html). Accessed: 2014-10-17.
- [32] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in tbb. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.
- [33] P. K. Romano, B. Forget, and F. Brown. Towards scalable parallelism in monte carlo particle transport codes using remote memory access. In *Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo*, volume 2010, pages 17–20, 2010.
- [34] E. Rothberg and A. Gupta. Parallel ICCG on a hierarchical memory multiprocessor—addressing the triangular solve bottleneck. *Parallel Computing*, 18(7):719–741, 1992.
- [35] J. K. Salmon. *Parallel hierarchical N-body methods*. PhD thesis, California Institute of Technology, 1991.
- [36] J. H. Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM journal on scientific and statistical computing*, 11(1):123–144, 1990.
- [37] 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.
- [38] E. Saule, K. Kaya, and Ü. V. Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In *Parallel Processing and Applied Mathematics*, pages 559–570. Springer, 2014.
- [39] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *25th Annual Technical Symposium*, pages 241–248. International Society for Optics and Photonics, 1982.
- [40] R. Thakur and W. Gropp. Test suite for evaluating performance of multithreaded mpi communication. *Parallel Computing*, 35(12):608–617, 2009.
- [41] E. Totoni, M. T. Heath, and L. V. Kale. Structure-Adaptive Parallel Solution of Sparse Triangular Linear Systems. Technical Report 12-42, Parallel Programming Laboratory, 2012.
- [42] UPC Consortium and others. UPC language specifications v1. 2. *Lawrence Berkeley National Laboratory*, 2005.
- [43] K. B. Wheeler, R. C. Murphy, D. Stark, and B. L. Chamberlain. The Chapel tasking layer over qthreads. *CUG 2011*, 2011.
- [44] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Parallel and Distributed Processing (IPDPS) 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.
- [45] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and P. Gibbon. A massively parallel, multi-disciplinary barnes-hut tree code for extreme-scale n-body simulations. *Computer Physics Communications*, 183(4):880–889, 2012.
- [46] M. M. Wolf, M. A. Heroux, and E. G. Boman. Factors impacting performance of multithreaded sparse triangular solve. In *High Performance Computing for Computational Science-VECPAR 2010*, pages 32–44. Springer, 2011.
- [47] J. Zhang, B. Behzad, and M. Snir. Optimizing the barnes-hut algorithm in UPC. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 75. ACM, 2011.
- [48] J. Zhang, B. Behzad, and M. Snir. Design of a multithreaded barnes-hut algorithm for multicore clusters. Technical report, Tech. Rep. ANL/MCS-P4055-0313, MCS, Argonne National Laboratory, 2013.

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.