# Design of a Multithreaded Barnes-Hut Algorithm for Multicore Clusters

Junchao Zhang, Babak Behzad, and Marc Snir, *Fellow, IEEE*

**Abstract**—We describe in this paper an implementation of the Barnes-Hut algorithm on multicore clusters. Based on a partitioned global address space (PGAS) library, the design integrates intranode multithreading and internode one-sided communication, exemplifying a PGAS + X programming style. Within a node, the computation is decomposed into tasks (subtasks) and multitasking is used to hide network latency. We study the tradeoffs between locality in private caches and locality in shared caches and bring the insights into the design. As a result, our implementation consumes less memory per core, invokes less internode communication, and enjoys better load-balancing strategies. The final code achieves up to 41 percent performance improvement over a non-multithreaded counterpart. Through detailed comparison, we also show its advantages over other well-known Barnes-Hut implementations, both in programming complexity and in performance.

**Index Terms**—Barnes-Hut, n-body, PGAS, cluster, multicore

## 1 Introduction

The current evolution of supercomputers exhibits several important trends [1], [2]:

- The number of cores per node keeps increasing.
- The amount of memory per core is decreasing.
- One-sided communication (rDMA) is increasingly well supported on the interconnection networks.

The use of one-sided communication has several advantages. Such communication can have less software overhead, since code is executed only on one of the two communicating nodes; this results in lower latency, especially for short messages [3], which are more likely to happen in strong scaling of applications. Also, it is easier to code irregular applications with dynamic communication patterns, using one-sided communication. In such applications, the consumer of a variable often knows the location of that variable, but the owner of the memory containing this variable may not know who is the consumer. Such a pattern embarrasses the sender-initiated two-sided message-passing programming model (i.e., push-model) but fits well to the consumer-initiated shared-memory programming model (i.e., pull-model). Libraries, such as SHMEM [4] and Global Arrays (GA) [5], and PGAS languages, such as UPC [6] or CAF [7], use one-sided communication as their main communication mechanism.

The natural idiom for irregular applications with dynamic communication patterns is to use remote reads or get operations to access remote data. In order to achieve good performance, it is essential to hide the latency of the long round-trip of a remote memory access. Similar to simultaneous or concurrent multithreading in shared-memory environments, such latency hiding is most conveniently achieved by descheduling tasks that are blocked on a remote access and reusing the core to run another, ready-to-execute task. This requires low-overhead task scheduling. Such task scheduling also enables efficient load balancing, ensuring that all cores are used.

Often, that multiple cores on a node will use the same shared structures. Memory pressure can be alleviated by keeping only one copy per node for such structures, a procedure that particularly benefits weak scaling of applications because they require more memory per core. In addition, codes such as the Barnes-Hut (BH) algorithm exhibit significant reuse of remote values; hence, it is advantageous to keep a local copy of data brought from remote nodes, for possible reuse. Effectively, this means using local memory as a software-managed cache for remote memory.

We demonstrate in this paper the use of these techniques in the context of the BH algorithm. The main original contributions of this paper are as follows.

- We give the first BH design that integrates intranode multithreading and internode one-sided communication and uses multitasking to hide network latency.
- We study the interplay in BH between locality in private caches and reuse in shared caches.
- We compare BH implementations done using distinct programming models, and we review how they handle programming challenges on multicore clusters.

- *J. Zhang is with the Mathematics and Computer Science (MCS) Division, Argonne National Laboratory, Argonne, IL 60439. E-mail: jczhang@anl.gov.*
- *B. Behzad is with the Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: bbehza2@illinois.edu.*
- *M. Snir is with the Mathematics and Computer Science (MCS) Division, Argonne National Laboratory, Argonne, IL 60439, and the Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: snir@anl.gov.*

*Manuscript received 16 Jan. 2014; revised 6 June 2014; accepted 10 June 2014. Date of publication 16 June 2014; date of current version 5 June 2015.*
*Recommended for acceptance by X.-H. Sun.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TPDS.2014.2331243*

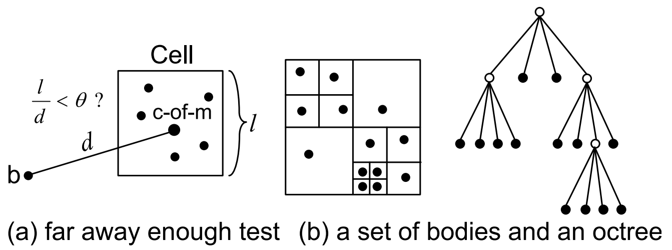

Fig. 1. A Barnes-Hut example in two dimensions.

The remainder of this paper is organized as follows. We describe the basics of the BH algorithm in Section 2, then give details of our design in Section 3. In Section 4 we evaluate and analyze our design. In Section 5 we compare our code with three other BH implementations. Section 6 provides an overview of the library we are designing. Section 7 surveys related work. We conclude in Section 8.

## 2 Barnes-Hut Algorithm

The Barnes-Hut algorithm [8] is a fast algorithm for the n-body problem, which simulates the evolution of a system of $n$ bodies (particles), where bodies apply forces on each other. A simulation consists of multiple time steps of length $\Delta$t. In each time step, forces are computed and body positions updated. A brute-force approach to this problem leads to $\Theta(n^2)$ complexity. To lower the complexity, BH approximates the interaction of a body with a set of other bodies by representing the set with a point located at its center of mass (c-of-m). The approximation is valid when the bodies in the set are *far away enough* from the first body—with *far away enough* being formalized as $l/d < \theta$, where $l$ is the size of the cube containing the bodies in the set, $d$ is the distance from the body to the center of mass, and $\theta$ is a constant called the cell-opening criterion (see Fig. 1a). To increase simulation accuracy by a factor of $s$, one need to scale $n$ by $s^2$ (i.e., increase sampling of the universe), or scale either $\theta$ or $\Delta$t by $1/\sqrt{s}$.

The BH algorithm partitions the 3D space hierarchically into *cells* by using an octree representation. The root of the octree represents the cell that contains all bodies. Each cell is recursively divided into octants, until its number of bodies is below a fixed threshold (see an example in Fig. 1b). In order to compute forces on one body, the procedure begins with the root cell. If the current cell is far enough or contains only one leaf body, then the force is computed, and the procedure stops there. Otherwise, it *opens* the cell and continues, recursively, with each of its children. With this hierarchical approach, BH achieves a computation complexity of $O(n \log n)$ [8].

*Load balancing* and *locality-awareness* are two important issues in parallel implementations of BH. Since the input body distribution is usually nonuniform, each body interacts with a different number of cells. One cannot simply assign bodies to processors evenly. Also in BH, cells accessed during force computation for one body are likely to be accessed again for a nearby body. In systems with hierarchical memories, it is critical to allocate bodies close to each other to one processor in order to exploit this locality.

A shared-memory BH code from the SPLASH-2 benchmark suite [9] handles the issues as follows. It assigns each body a cost, which is the number of forces computed for this body in one time step. Since body locations change slowly, one can use costs in the previous time step as estimates of costs in the current time step. SPLASH-2 BH uses an algorithm called *costzones* to assign bodies to threads. Octree leaves are split into $p$ zones of consecutive leaves of roughly equal total cost, where $p$ is the number of threads. Thread $i$ picks bodies in the $i$th zone. A left-to-right traversal of the octree leaves corresponds to an ordering of the bodies along a space filling curve (SFC); we call this order *SFC order* [10]. Each thread is allocated a segment from the curve.

Zhang et al. presented a distributed-memory BH code in [11]. Their code was implemented in UPC [6] and will be referred to as UPC BH hereafter. UPC BH inherited the above ideas from SPLASH-2. It spawns one UPC thread per core and divides a time step into four phases:

*Build octree*. Threads build subtrees for subspaces assigned to them, compute center of mass of each cell, and then hook subtrees together to form the global octree. See [11] for details.

*Partition octree*. Each thread is assigned an array of bodies, according to the costzones algorithm.

*Compute forces*. Threads compute forces for their bodies by traversing the octree from the root, then update their body costs.

*Advance bodies*. Threads advance their bodies by computing new velocities, positions etc. They also compute the boundaries of the new root cell for the next time step.

The force computation phase and the tree building phase both perform $O(n \log n)$ operations, but the former is much more expensive than the latter and usually dominates the performance; other phases perform $O(n)$ operations. In this paper, we focus on the force computation phase only.

## 3 Multithreaded BH Design

UPC BH is a port of SPLASH-2 BH to UPC. The authors showed that a naive port resulted in abysmal performance, but a sequence of optimizations resulted in dramatic improvement. However, since that code ran a persistent UPC thread (process) per core, it took only limited advantage of shared memory within nodes. In contrast, we now demonstrate that by using multithreading and multitasking within each process, we can get further benefits, such as better latency hiding, improved load balancing, less off-node communication, and less memory use.

It would have been good if we could achieve the above goals by simply extending UPC BH by using the same language, but we found this difficult because UPC is based on C and lacks support for generic programming, which we need in order to abstract common services that are also useful for other applications. Therefore, we designed PPL, a C++ template library atop the Berkeley UPC runtime [12] and implemented a new BH code in it, which is called PPL BH. PPL has the same memory model as UPC. Each process in PPL has its heap divided into two parts: a private heap and a local part of a global heap. While the private heap can be accessed only by threads local to the process, the global heap part can be accessed by any thread. Accesses to the local part of the global heap is much faster than accesses to remote parts. PPL provides a generic global pointer structure that can point to any remote memory locations. Dereferencing global pointers may require remote reads. We use global pointers for the links in the octree. We talk more about intentions of PPL in Section 6.

In this section, we first introduce our test platform and test methodology, which will be used in experiments afterward. Then we give an overview of the force computation in UPC BH for comparison. After that we describe the PPL BH force computation in detail. At the end of this section, we study how to achieve both load balance and cache efficiency within a node.

## 3.1 Test Platform and Test Methodology

We did all experiments on an $\times 86$ Linux InfiniBand cluster. Each compute node has two hex-core Intel Xeon 5,650 CPUs at 2.67 GHz. The six cores on a CPU have a private 32 KB L1 data cache and a private 256 KB L2 cache but share a 12 MB L3 cache. We used gcc4.6.3 as our C/C++ compiler and used the runtime of Berkeley UPC 2.14.2 as the communication library below PPL. The input bodies comprise two galaxies generated by the Plummer model [13], which is an empirical model of galactic clusters. The density of a cluster is very large near the center and diminishes with distance from the center. The octree was built with each leaf having at most 10 bodies. Though a real world simulation may consist of millions of time steps, we ran 22 time steps with the length of a time step $\Delta t = 0.025$ seconds, and timed only the last 20 steps (the first two were used to warm up the simulation). All computations were done in double precision. The number of bodies $n$ varied from $1M$ to $64M$. The cell opening criterion $\theta$ was near $1/\sqrt{3}$ as suggested in [14]. Since the focus of this paper is the force computation phase, we report only the average time per step of this phase.

## 3.2 UPC BH Force Computation

UPC BH has one UPC thread per core in a multicore node. The octree is distributed among threads. For load balancing, bodies are partitioned across all UPC threads by using the costzones algorithm; no distinction is made between threads that belong to the same process and threads that belong to distinct processes. Every UPC thread is assigned an array of bodies sorted in SFC order with equal total cost. Two important optimizations in UPC BH are caching and computation/communication overlapping. Each thread caches cells visited, as these are likely to be reused to compute forces for subsequent bodies. The cached cells form a local partial octree, which is a snapshot of the global octree. Pointer-swizzling is used to have child pointers in a cached cell point to either its children in the original octree or cached copies of these children, depending on whether the children have been cached or not. The cached data is discarded at the end of the force computation phase. UPC threads do not share the cached data even when they are on the same node. If multiple threads on a node need the same off-node cells, they fetch them separately and communicate multiple times.

To overlap computation and communication, UPC BH takes advantage of the two-level parallelism in BH. Force computations for different bodies are independent and can be done in parallel; interactions between a body and different cells are also independent and can be done in parallel, except that all forces acting on one body need to be summed together. At the beginning, each thread caches the octree root locally. Each thread maintains a work list of bodies. To compute forces for a body, a thread traverses the octree from the cached root. If a cell needs to be opened and its children are not cached, the thread will invoke a nonblocking communication to fetch the children; meanwhile the thread traverses other paths in the octree or just picks up another body from the work list. Threads periodically check pending nonblocking communications in order to complete them. For completed ones, threads resume interactions with the cells just fetched back.

## 3.3 PPL BH Force Computation

To make our description easier, we define some terms first. A cell is *localized* if its children have all been cached. Each cell has a `localized` flag to indicate whether it is localized or not; this flag is initially cleared. To localize a cell, we fetch its children, swizzle its global child pointers to local pointers pointing to the cached children, and then set the `localized` flag. The localization is a split-phase operation that includes making a nonblocking communication request and completing the request. So we also add a `requested` flag in each cell to indicate that the cell has been requested but is not yet localized; this prevents making multiple requests for the same cell. The requested flags of all cells are initially cleared.

A *task* consists of the tree traversal and force computation for one body. During the traversal, if a cell needs to be opened but is not localized, we generate a *subtask* to handle the interactions between the body and the subtree rooted at that cell. Thanks to parallelisms in force computation, all tasks and subtasks can be executed in parallel, and synchronization is needed only to properly add together the forces acting on one body. Our objective is to orchestrate tasks and subtasks efficiently.

In PPL BH, we spawn one process per node. Upon entering the force computation phase, each process gets an array of bodies through the costzones algorithm and spawns one thread per core. This approach gives us the opportunity to allocate tasks and subtasks dynamically to threads and to share cached copies of cells across all threads. However, dynamic allocation and sharing can lead to increased synchronization overheads and reduced locality. We study these tradeoffs in Section 3.4. The general rule is that threads take bodies from the array, generate tasks, and execute them. If a thread is blocked in a task's execution by an *unlocalized* cell during tree traversal, it will generate a subtask to encapsulate the context and continue the traversal along other paths if possible; otherwise it will generate new tasks and execute them. Although subtasks from the same task can be executed simultaneously by distinct threads, we do not do so; instead, we execute a task and all its descendant subtasks on the same thread. The reason is that the large

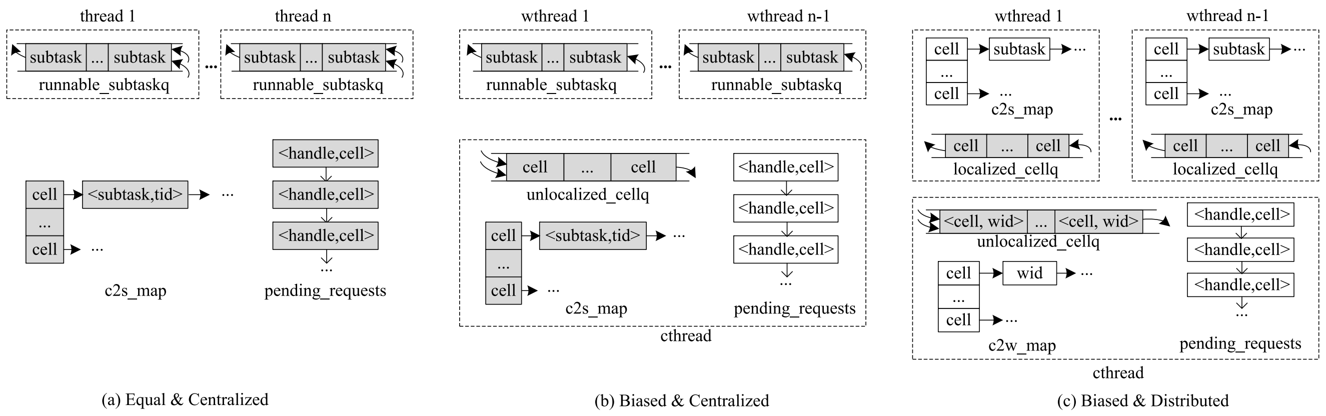

Fig. 2. Data structures used in the three approaches (shaded structures are accessed concurrently).

number of bodies is sufficient to ensure that threads are always busy; moreover, this choice avoids the need to synchronize reductions across multiple threads.

PPL BH adopts the two optimizations in UPC BH: caching and computation/communication overlapping. Additionally, we considered three ways of splitting work across threads, as shown in Fig. 2.

*1) Equal and Centralized (Fig. 2a).* In the first approach, all threads are equally involved in computation and communication. As shown, all threads share a multi-map (`c2s_map`), which stores, for each cell, the list of subtasks blocked on an open request for that cell. The map acts as a hub for subtask registering and releasing. When a thread wants to open an unlocalized cell, it generates a subtask and registers it in the map under that cell. If the cell is not requested, the thread also makes a nonblocking request to fetch children of the cell and puts a handle to the request in a list (`pending_requests`). Threads periodically check the list to see whether any request is completed. For completed requests, they will push registered subtasks to runnable subtask queues (`runnable_subtaskq`) on threads. Each subtask carries a thread id (`tid`) that indicates the thread responsible for executing the subtask—hence the queue the subtask joins when it becomes runnable. Threads query their own `runnable_subtaskq` whenever the task they currently execute becomes blocked.

The biggest problem of this approach is synchronization. First, accesses to the `localized` flags and the `requested` flags must be atomic so that all threads have a consistent view of the cached octree. Second, all data structures, such as `c2s_map` and the `pending_requests` queues, must be concurrent (in other words, support concurrent operations from multiple threads). The overall overhead could be high even with an efficient implementation of these data structures.

*2) Biased and Centralized (Fig. 2b).* In the second approach, one thread is designated as the communication thread (cthread), with the remaining as worker threads (wthreads). All threads share the concurrent map (`c2s_map`) as in the first approach. Only the cthread is in charge of communication. Thus communication management is easier, and data structures such as `pending_requests` need not be concurrent anymore. Every wthread has a runnable subtask queue. When a wthread wants to localize a cell and finds it was already requested, the wthread registers a subtask for the cell in `c2s_map`. Otherwise, if the cell was not requested previously, the wthread will also mark the cell as requested and push it in a concurrent queue (`unlocalized_cellq`). The operation of marking cells as requested need not to be atomic. The same cell can be pushed into `unlocalized_cellq` multiple times by different threads.

The cthread pops cells from `unlocalized_cellq`, checks and updates their `requested` flags again to remove the redundancy, and issues only one request per cell. The cthread also checks pending requests. If a request is completed, it marks the corresponding cell as localized, looks up `c2s_map`, pushes subtasks registered under the cell back to the queues (`runnable_subtaskq`) of their owner wthreads, and then deletes the entry in `c2s_map`. Each subtask has a worker id (`wid`) so that the cthread knows which queue to choose. The cthread and wthreads must be properly synchronized if they are operating on the same entry in `c2s_map`. For example, when a cell in `c2s_map` was deleted by the cthread, no wthread should have the chance to insert it again. In this approach, every `runnable_subtaskq` is now a single-producer single-consumer (SPSC) queue while `unlocalized_cellq` is a multi-producer single-consumer (MPSC) queue.

This approach is similar to thread scheduling in an operating system. Multiple threads can wait for a same signal. Once the signal arrives, all threads registered under this signal are woken up.

*3) Biased and Distributed (Fig. 2c).* This approach has the same cthread and wthreads as in the second approach. But this time the cell-to-subtask map (`c2s_map`) is distributed among threads and is not concurrent anymore. If a wthread wants to localize a cell, it looks up its private `c2s_map` to see whether the cell has already been requested *by itself*. If so, it just registers the subtask in its map; otherwise, it also pushes the cell along with a worker id (`wid`) into a concurrent queue (`unlocalized_cellq`). An unlocalized cell may be pushed into the queue multiple times by different threads. But the cthread makes only one communication request for each cell. The cthread uses a private map (`c2w_map`) to map cells to wthreads that have requested them. The cthread manages communication. When a request is completed, it marks the cell as localized and pushes it back to queues (`localized_cellq`) on wthreads who have requested the cell.

```
// The main loop
while there are unprocessed bodies do
    while !localized_cellq.empty() do
        cell = localized_cellq.pop()
        subtasks = c2s_map.lookup(cell)
        execute each subtask in subtasks
    if current chunk of bodies is empty then
        get next chunk from the body array
    else
        pop next body from current chunk
        generate a task for the body and execute it

// Code to execute a subtask
get <body, cell> of the subtask
for all child of cell do
    if need to open child and !child→localized then
        if no <child, *> in c2s_map then
            unlocalized_cellq.push(<child, wid>)
        generate a new_subtask for (body, child)
        c2s_map.insert(child, new_subtask)
    else
        other recursive tree traversal code (omitted)
```

Fig. 3. Pseudo code on wthreads (id $= wid$) in Biased and Distributed.

```
// The main loop
while !unlocalized_cellq.empty() do
    <cell, wid> = unlocalized_cellq.pop()
    if cell is localized then
        push cell to localized_cellq on wid
    else if cell is requested then
        c2w_map.insert(<cell, wid>)
    else
        handle = a nonblocking call to localize cell
        set cell requested
        pending_requests.insert(<handle, cell>)
        c2w_map.insert(<cell, wid>)

    for all <handle, cell> in pending_requests do
        if handle is completed then
            cell→localized = true
            wids = c2w_map.lookup(cell)
            push cell to localized_cellqs on wids
            pending_requests.erase(<handle, cell>)
```

Fig. 4. Pseudo code on cthread in Biased and Distributed.

On the other side, wthreads pop cells from their queue, look up their map, and execute subtasks registered under the cells. Note that accesses to the `localized` flags of cells need not to be synchronized between the cthread and wthreads. If the cthread set the `localized` flag of a cell, and the new value is not immediately observed by a wthread, the wthread may superfluously push the cell into `unlocalized_cellq`. When the cthread pops up the cell, it will check the cell's flag. Of course, the cthread will find the flag is true because it was set by itself before. If the flag is set, the cthread just rebounds the cell back to its owner wthread. Sooner or later, the new value will be seen by wthreads, thanks to the hardware cache coherence protocol. It turns out that all synchronizations in this approach can be done through either SPSC queues or MPSC queues, which can be implemented efficiently by using lock-free data structures [15]. Pseudo codes on the wthreads and the cthread are given in Figs. 3 and 4 respectively.

In our code, all maps are implemented as hash maps. We use pointers to cells as keys for the hash maps, which we found to be efficient in practice.

## 3.4 Intranode Task Scheduling

We now study how to schedule tasks within a node, in other words, how to distribute bodies to threads, with an aim of achieving both load balance and cache efficiency. A distinguishing feature of multicore CPUs is that they usually have their last-level cache shared among cores on the chip. Therefore, besides private cache efficiency, it is interesting to know whether we can improve shared cache efficiency by synergistic task scheduling.

Two distribution strategies are common: cyclic distribution and block distribution. In the former, body $i$ is assigned to core $i \bmod p$, where $i$ is the body index in the body array, and $p$ is the number of cores. In the latter, each core is assigned a block of consecutive bodies from the body array. The locality in BH means that cells visited by one body are likely to be visited again by nearby bodies. The following experiments were done to measure the locality in these two distributions.

We ran UPC BH with $n = 1M, \theta = 0.5$ and 16 processes (threads in UPC terms) and examined process 8 (P8) in the fourth time step. The process was assigned about 69 K bodies. We tagged cells visited during force computation for the first body. Then we determined how many of them were visited again by the second body, the third body, and so on and calculated a sharing ratio for each body with respect to the first body. The result is shown in Fig. 5, which also includes a zoom-in picture for the first 100 bodies on P8. We can see BH has very good locality when SFC order is used. For example, the second body shares about 90 percent of the cells the first body visited. Even the 100th body still shares more than 50 percent with the first body. The sharing ratio decreases with distance, but the trend is not strict. The reason is that distance in a 1D SFC does not strictly correspond to distance in a 3D space. The curve does not hit zero because top cells of the octree are visited by all bodies. Thus, there is still a good sharing (20-40 percent) even at

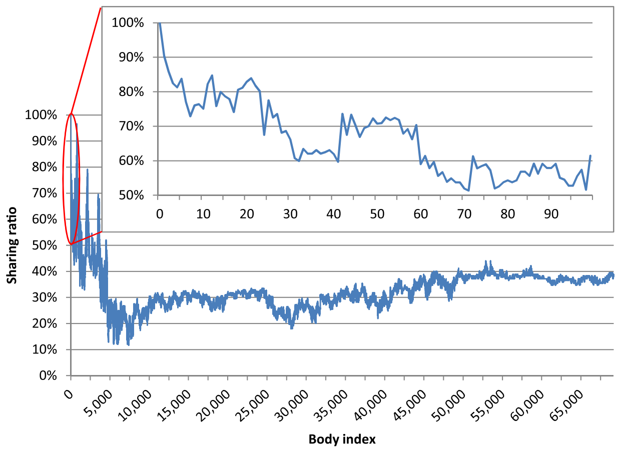

Fig. 5. Sharing ratio for the 69 K bodies on P8.

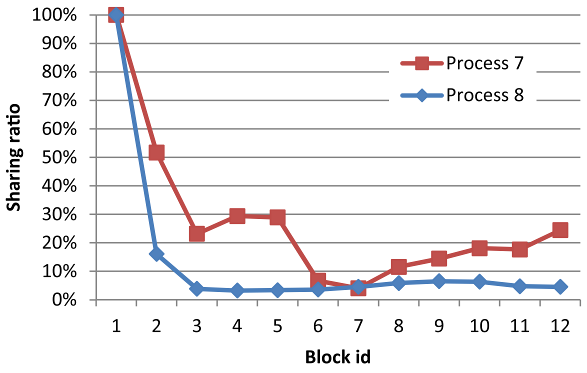

Fig. 6. Block sharing ratio on P7 and P8.

distant parts of the curve. Curves on other processes have similar shapes, so we do not show them here.

Using the same concept, we studied sharing between blocks. The configuration was the same. But this time we cut the 69 K bodies into 12 blocks of equal total costs (note that we have 12 cores per node). We tagged cells visited by the first block of bodies and calculated the sharing ratio of the second block, the third block, and so on with respect to the first block. Fig. 6 shows the block-sharing ratio curves measured on process 7 and 8. Compared with Fig. 5, the ratio drops greatly. This result means blocks of bodies will touch different bottom parts of the octree so that they have less sharing than before. We also notice that the ratio varies a lot between different processes, which indicates that different regions of the space have distinct sharing property. Generally, however, the ratio is low.

Our experiments suggest that block distribution should benefit L1 cache, since computations on successive bodies will reuse the same cells. Similarly, cyclic distribution should benefit L3 cache, since cached cells are likely to be accessed by all threads in a small time interval. To test this hypothesis, we modified SPLASH-2 BH (which uses block distribution), added cyclic distribution, and ran it with 6 threads on one CPU of our platform. We bound threads to cores and did cache profiling with PAPI [16]. Table 3 shows the miss rates for both cases. Note that the table uses *local miss rates* at each level, defined as the miss count of this cache divided by the reference count of this cache. We took the average miss rate on six cores. A surprising result is the slight differences between cyclic distribution and block distribution either in performance or in miss rate. L1 miss rates of both are very high, while L3 miss rates of both are pretty low (in the table, L2 miss rates are high because most of the locality in L2 is absorbed by L1, which makes L2 miss rates less interesting here).

To understand the reason for these miss rates, in the fourth time step, we measured the octree traversal length per body and the total octree size with various $n$ and $\theta$ in number of cells. The traversal length of a body is the number of cells visited by the body during its traversal. Table 1 shows the results.

From Table 1, we can see that, as expected, the tree size is proportional to the number of bodies, while the traversal length is proportional to the log of the number of bodies. In the code, the average size of a cell or body is about two cache lines (i.e., 128 bytes, including fields for 3D positions, mass, and child pointers). So, with $n = 1M, \theta = 0.5$, the

TABLE 1
Average Traversal Length and Octree Size in # of Cells

| $n$ | | $1M$ | $2M$ | $4M$ | $8M$ | $16M$ |
|---|---|---|---|---|---|---|
| Traversal | $\theta = 0.5$ | 1,973 | 2,074 | 2,178 | 2,280 | 2,375 |
| length | $\theta = 0.7$ | 853 | 892 | 928 | 962 | 996 |
| Octree size | $(\times 10^6)$ | 1.4 | 2.7 | 5.4 | 10.8 | 21.6 |

average traversal size is about 246 KB. The number explains why in the previous block distribution test, L1 miss rate was high: As the size of a traversal exceeds L1 capacity (32 KB), accesses to cells in the second traversal generate L1 misses. It also explains why the two experiments have nearly identical L3 miss rates: As the size of a traversal is much smaller than that of L3, it has enough capacity to cache simultaneously many successive traversals of each thread, even if there is less sharing between threads. So we can conjecture that benefit of cache sharing will show up only when the size of a single traversal approaches the L3 quota per core (i.e., 2 MB in our case).

One possibility is to increase $n$ to meet the condition, but $n$ would be an incredibly large number. Instead, we decreased $\theta$. Table 2 shows the result for $n = 1M, \theta = 0.2$ on one CPU. (Note $\theta = 0.2$ is not a realistic choice for BH; we use it only to test the conjecture.) In this test, the average traversal length is about 21,260 cells, or 2.6 MB. Now we can observe a big difference between cyclic distribution and block distribution: the former's L3 miss rate is 1.25 percent, while the latter's is 15.93 percent. This results in a big performance difference, as also shown in Table 2.

These experiments suggest that with reasonable input parameters and hardware, it is *not important* to exploit cache sharing on multicores in BH. What is important is to improve L1 efficiency. To verify this conclusion, we implemented the tiling technique in [17] in SPLASH-2 BH. Block distribution is still used, but each thread now takes a tile of bodies at a time from its block. When visiting a cell, bodies in the tile will interact in turn with that cell. Some bodies may need to open the cell while others may not. We therefore need to dynamically mask member bodies in a tile. As a result, the octree is traversed only once for all bodies in a tile. In this way, when a cell is brought into L1 by one body, it will be reused by other bodies in the same tile that need it. As we saw in Fig. 5, the reuse probability is high. The row "Block w/Tiling" in Table 3 shows the result with a tile size of 128. We can see the dramatic L1 miss reduction and performance improvement. Note that L3 miss rate seems high after tiling. The reason is that the L3 reference count drops greatly (not shown in the table). In reality, the L3 miss count did not increase much. We tried different tile sizes and found that performance is not sensitive to size when it is in range of 64 to 10 K. If bigger than that, the tile itself will overflow the L1, lowering performance. Hence, we choose a tile size of 128 hereafter.

TABLE 2
Performance When Traversal Size > L3, $n = 1M, \theta = 0.2$

| | Time(s) | L1 Miss | L2 Miss | L3 Miss |
|---|---|---|---|---|
| Cyclic | 108.88 | 7.33% | 53.32% | 1.25% |
| Block | 142.36 | 6.91% | 56.66% | 15.93% |

TABLE 3
Performance When Traversal Size < L3, $n = 1M,\ \theta = 0.5$

| | Time(s) | L1 Miss | L2 Miss | L3 Miss |
|---|---|---|---|---|
| Cyclic w/o SFC | 12.66 | 8.19% | 76.77% | 7.65% |
| Cyclic | 10.59 | 8.24% | 69.73% | 0.14% |
| Block | 10.84 | 8.23% | 69.46% | 0.14% |
| Block w/ Tiling | 8.28 | 0.20% | 74.14% | 6.36% |

The row "Cyclic w/o SFC" in Table 3 is also interesting. In this case, we used cyclic distribution, but bodies in the input array were randomly ordered. It can be seen a high L3 miss rate and degraded performance, because the BH locality was not respected. Different parts of the octree are now randomly touched by unrelated bodies, resulting in a much bigger memory footprint in L3 than before.

It is clear that we should increase task size in PPL BH. Fortunately, doing so requires only small changes in the algorithms described in Section 3.3. Now a task computes forces for a tile of bodies. A task has a bit mask to indicate which bodies need to interact with a cell. When a body in a tile needs to open an unlocalized cell, the task generates a subtask and relays the bit mask to it, so that when the subtask is resumed, it knows which bodies to pick up.

Note that the distributions discussed above are static in the sense that bodies are assigned to threads before force computation starts. This approach is fine for the shared-memory SPLASH-2 BH but is not good for the distributed-memory PPL BH because, besides computation, there is communication. With a static block distribution, different threads need different amounts of remote data, resulting in load imbalance. To smooth this variation, we adopted dynamic scheduling in PPL BH. We tried two approaches similar to OpenMP *dynamic* scheduling and *guided* scheduling for work-sharing loops [18]. In the former, worker threads request a fixed-sized chunk of bodies from the body array, work on the chunk, and then request another chunk until no bodies are left. In the latter, worker threads request a chunk of bodies with length proportional to the number of unassigned bodies divided by the number of worker threads. We chose *guided* because it shows a little better performance than *dynamic*.

## 4 Evaluation and Analysis

In this section, we test and analyze PPL BH with the various multithreading approaches discussed in Section 3. Because of the obvious synchronization overhead in the Equal and Centralized (we can see it even in the Biased and Centralized), we implemented only the last two approaches. We quantitatively measure the benefits of PPL BH and present its performance.

### 4.1 Comparison of the Two Biased Approaches

We tested PPL BH with the last two biased approaches with various configurations. We see that the performance of the Biased and Centralized is 1.6 to 3.5 times worse than that of the Biased and Distributed. Both use a data-driven style: Subtasks become runnable only when their requested data arrives. But the first biased approach uses a centralized cell-to-subtask map: All subtasks are registered in the map at the cthread, which notifies wthreads which subtasks become runnable. In the second biased approach, when a cell is localized, the cthread notifies the wthreads that have requested the cell. Then those wthreads look up their private maps to release subtasks depending on this cell. Although this approach can result in the same cell being registered at multiple threads, it reduces the pressure on the cthread, since the wthreads filter requests for the same cell.

We did profiling with $n = 1M,\ \theta = 0.5$ on 8 nodes. In the Biased and Centralized without tiling, on average, dozens to hundreds of subtasks from wthreads are going to be registered under one cell in the cthread's map. The maximal number is huge, ranging from 8,000 to 31,000. In the Biased and Distributed, on the other hand, a maximum of 6 to 11 wthreads request the same cell from the cthread simultaneously; on average, the number is 1. One can easily see that through distributed subtask management, we save much of the traffic between the cthread and wthreads and thus achieve better performance. With tiling, the number of tasks and subtasks decreases so that the phenomenon is not that significant. However, the centralized still lags, likely because of the synchronization overhead in `c2s_map` (note that it is a concurrent map). Because of its superior performance, we use PPL with the Biased and Distributed approach from now on.

TABLE 4
Ratio of Locally Essential Tree (LET) Size to Local Tree Size

| Threads | $n = 1M$ | | $n = 2M$ | | $n = 4M$ | |
|---|---|---|---|---|---|---|
| | $\theta = 0.5$ | $\theta = 0.7$ | $\theta = 0.5$ | $\theta = 0.7$ | $\theta = 0.5$ | $\theta = 0.7$ |
| 16 (1×16) | 1.80 | 1.43 | 1.65 | 1.37 | 1.51 | 1.29 |
| 32 (1×32) | 2.12 | 1.59 | 1.91 | 1.49 | 1.69 | 1.38 |
| 64 (1×64) | 2.58 | 1.81 | 2.21 | 1.64 | 1.92 | 1.49 |
| 192 (12×16) | 3.58 | 2.28 | 2.90 | 1.96 | 2.42 | 1.73 |
| 384 (12×32) | 4.68 | 2.79 | 3.64 | 2.30 | 2.91 | 1.96 |
| 768 (12×64) | 6.37 | 3.54 | 4.76 | 2.81 | 3.66 | 2.30 |

### 4.2 Less Memory Consumption

The octree in UPC BH is distributed among threads. Each thread has a local part of the octree (which we call the *local tree*). Local trees are linked together by shared pointers. In force computation, threads traverse the octree and cache visited cells. The part of the octree visited during force computation for all bodies assigned to a thread is called the thread's *locally essential tree* (LET). Table 4 shows the average ratio of the LET size to local tree size while varying $n$, $\theta$, and the number of threads.

Looking at one of the configurations: 16 nodes, each with 12 cores, for a total of 192 threads for UPC BH. With $n = 1M,\ \theta = 0.5$, the ratio of LET size to local tree size is 3.58. In PPL BH, however, we create only one process per node. Threads spawned by a process share the same LET; this will be the same LET created in UPC BH when one runs 1 thread per node, where the ratio is 1.80. It means that PPL BH saves about one half of the memory consumed by cached remote cells by sharing the LET. By the same reasoning, we see that for $n = 1M,\ \theta = 0.5$, the percentages of memory saved through multithreading with 32 nodes and 64 nodes are $(4.68 - 2.12)/4.68 = 55\%$ and $(6.37 - 2.58)/6.37 = 59\%$, respectively. We could expect bigger savings when more cores are put on a chip. Note that in Table 4, when $n$ increases, the ratio decreases; with bigger $\theta$, which

TABLE 5
Percent of Off-Node Communication Saved in PPL BH With Respect to UPC BH

| Nodes | $n = 1M$ | | $n = 2M$ | | $n = 4M$ | |
|---|---|---|---|---|---|---|
| | $\theta = 0.5$ | $\theta = 0.7$ | $\theta = 0.5$ | $\theta = 0.7$ | $\theta = 0.5$ | $\theta = 0.7$ |
| 16 | 44% | 33% | 36% | 22% | 28% | 13% |
| 32 | 51% | 42% | 43% | 32% | 36% | 25% |
| 64 | 57% | 49% | 50% | 40% | 43% | 32% |

translates into less accuracy and fewer cells opened, the ratio will also decrease.

### 4.3 Less Off-Node Communication

Each thread in UPC BH has its own LET, and the off-node remote cells fetched by one thread will not be reused by other threads on the same node. In PPL BH, this is not a problem anymore. We can quantitatively measure the saving in communication. For example, given 16 nodes and 12 cores per node, we ran UPC BH with 192 threads. We distinguished on-node cells and off-node cells fetched during force computation, and we sumed the number of off-node cells fetched by each thread. Then we ran PPL BH on 16 nodes and collected the same number. Comparing these two numbers, we could know how big the saving is. Table 5 shows the savings on 16, 32, and 64 nodes. For example, with $n = 1M,\ \theta = 0.5$ and 64 nodes, PPL BH saves about 57 percent of the off-node communication compared with UPC BH. In strong scaling, with more nodes, the saving is bigger; with other parameters fixed, increasing $n$ or $\theta$ reduces the saving.

### 4.4 Better Load Balancing

Since the force computation phase is synchronized between processes, its execution time is determined by the longest process. UPC BH inherited from the shared-memory SPLASH-2 BH code the costzones load-balancing algorithm. However, this algorithm is computation-centric. On distributed memory the need to access remote cells can disturb the balance. Because of SFC ordering, boundary processes on a node usually require more remote cells than do interior processes. Considering computation/communication overlapping, the effect is hard to estimate upfront, and thus is better attacked by dynamic scheduling enabled by multithreading. As an example, with $n = 4M,\ \theta = 0.5$ and 64 nodes, we measured the execution time variation of the 12 threads, using the formula $(MaxTime - AverageTime)/AverageTime$ on each node. For UPC BH, it ranges from 0.5 to 71.0 percent. For PPL BH, however, it ranges only from 0.0 to 2.5 percent. Obviously, PPL BH achieves better intra-node load balancing.

### 4.5 PPL BH Performance

We used the optimized UPC BH implementation as the baseline and compared PPL BH with it. The performance improvement is shown in Fig. 7, which also includes SPLASH-2 BH's performance on one node. Surprisingly, UPC BH and PPL BH have much higher single-node performance than SPLASH-2 BH does, even though they perform extra operations. Note that they cache cells even if the cells are in the local part of the octree. (This approach is necessary because otherwise accesses from remote processes to the local part would read invalid swizzled pointers.) While caching the octree entails additional memory copies, it reduces L1 misses, probably because we cache cells in contiguous memory during tree traversal and thus enjoy a better memory layout for later visits. Thus, both UPC BH and PPL BH perform better than SPLASH-2 BH even without tiling. Also, PPL BH significantly outperforms UPC BH. The improvement is larger for higher node counts. For example, with $n = 4M,\ \theta = 0.5$ and 64 nodes, we get the highest improvement, at 41 percent. We can also observe the tradeoff between the benefits of multitasking and their overhead, especially when the computation density degrades. For example, with $n = 1M,\ \theta = 0.7$ and 64 nodes, without tiling, PPL BH degrades about 17 percent. From Table 1, we know that the average cost of a body decreases by more than one half when $\theta$ is changed from 0.5 to 0.7. With tiling, however, by having better locality and fewer tasks, PPL BH's performance goes up.

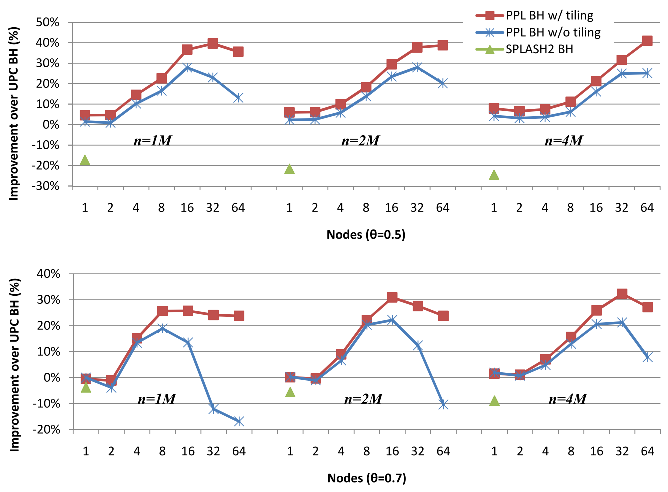

Fig. 7. Performance improvement over UPC BH.

Fig. 8 shows scaling of PPL BH (with tiling). For strong scaling, using performance of one node aa a baseline, at 64 nodes, $n = 4M$, it achieves a speedup of 51, 38 for $\theta = 0.5$ and 0.7, respectively. For weak scaling, we keep $1M$ bodies per node (up to $64M$ bodies for 64 nodes). The computation time increases logarithmically, as expected.

## 5 Comparison with Other BH Implementations

BH is a challenging application that has drawn much attention. It therefore is worth comparing implementations done in different programming models and analyzing how they deal with critical programming issues such as overlapping computation and communication, balancing load,

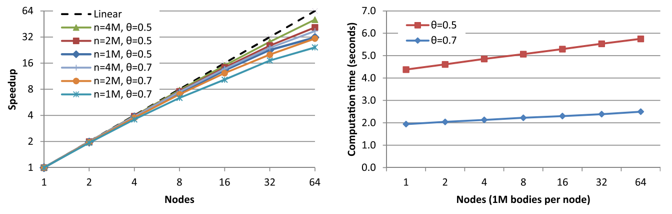

Fig. 8. Strong scaling (left) and weak scaling (right) of PPL BH.

preserving locality, and using multicore nodes. In this section, we compare PPL BH with three other codes—LET BH, PEPC, and Charm++ BH—which are implemented in distinct programming models. We describe and compare each with PPL BH and then summarize our results.

## 5.1 LET BH

The first parallel BH algorithm on distributed memory was invented by Salmon [14]. That work pioneered the locally essential tree concept, which we have already discussed in Section 4.2. In PPL BH, processes build their LET on the fly during force computation, whereas in Salmon's algorithm, LETs are built up front. Since we could not find an open source code of this algorithm of desired quality, we developed one ourselves in C++ and called it LET BH. The code is parallelized with MPI + OpenMP. It follows Salmon's algorithm but with an extension to handle any number of processes, instead of only power of two numbers.

Building LETs is complex in this algorithm. First, processes partition the space by using the orthogonal recursive bisection (ORB) method [14], to get both locality and load balancing (note that particles have a cost as before). This approach results in a binary space partitioning tree (named ORB tree), with each node being a bisection. Then, each process builds an octree using particles in its partition. Next, each process traverses the ORB tree from the root to the process's leaf partition. At each bisection, a process traverses its octree to collect cells (and particles) that *might* be needed by particles locating at the other side of the current bisector and then sends them to a partner process. Generally, processes need to do cell collecting, tree pruning, and tree merging at each bisection. In the end, processes have their LET built. See [14] for more details.

Once LETs are ready, force computation is embarrassingly parallel and can proceed without communication. We use OpenMP `parallel for` to parallelize the force computation loop over particles. But to exploit cache locality as analyzed in Section 3.4, we sort particles in Morton order [10]. Overall, complexities of the algorithm lie in the LET building. However, many of them can be avoided with a global address space. The algorithm does not overlap computation and communication, and it communicates more cells than necessary because of estimation in cell-opening tests. Refer to [19] for an excellent discussion in this regard. Nevertheless, this pure push-model MPI code provides us a good comparison target. Since its force computation phase involves no communication, one might think the code provides an upper performance bound if the phase is measured alone.

## 5.2 Pretty Efficient Parallel Coulomb (PEPC)

The Pretty Efficient Parallel Coulomb solver,[1] written in MPI + F]ortran 2003, is a state-of-the-art parallel BH code developed and widely used at the Julich Supercomputing Center. PEPC was reported to be able to run with up to 2 billion particles and 256 K cores [20]. PEPC has multiple front-ends for various disciplines. They all share the same kernel—a tree code implementing the *hashed octree* scheme pioneered by Warren and Salmon [10]. The scheme computes unique keys for cells and particles according to their positions and then stores them in a hash table. Particles are weighted by their costs as in SPLASH-2 BH and are sorted by their keys, resulting in an SFC. The curve is then partitioned among processors in order to achieve both locality and load balancing. MPI two-sided communication is used to fetch remote cells during tree traversal. The fetching process sends keys of parents to remote processes, which compute child keys, look up their hash tables, and send the children back. Processes do periodic synchronization to exchange data. This approach is well explained in [10].

1. http://www.fz-juelich.de/ias/jsc/pepc.

As multicore cluster emerges, PEPC ships a hybrid MPI + Pthreads tree code [20], for the same reasons as we do in PPL BH. Its design is similar to PPL BH but without one-sided communication, multitasking, and tiling. Usually there is one process per CPU, which spawns multiple *worker threads* while itself continuing as a *communicator thread*. Workers dynamically grab fixed-sized chunks of particles from the list of particles assigned to this process and compute forces for them. For each particle, workers maintain a *todo list*, which contains locally available cells to interact with, and a *defer list*, which contains unlocalized parent cells. Worker threads push into queues at their communicator threads' requests for child cells. Communicator threads communicate with each other in order to satisfy these requests. Once child cells are sent back, communicators cache them locally and tag their parents in the hash table. Workers have to periodically poll tags of cells on defer lists and try to move their children to todo lists. PEPC allows overbooking cores. Communicator threads can periodically yield cores to workers, a function that is not implemented in PPL BH.

Hash keys in PEPC function as global pointers in PPL BH except that dereferencing keys needs hash table lookups and participation of remote threads. In PPL BH, with a global name space and one-sided communication, these added complications are avoided, resulting in less programming complexity and lower runtime overhead. Also, PPL BH abstracts computation into tasks and subtasks and uses a data-driven-style task scheduling in contrast to tag polling in PEPC.

## 5.3 Charm++ BH

Charm++ BH, written in Charm++ [21], is a parallel BH code developed at the University of Illinois. We took the code from the Charm++ benchmarks, which won the 2011 HPC Challenge Class 2 [22]. Charm++ is a C++-based parallel programming system that implements a programming model based on message-driven, migratable objects. It features measurement-based automatic load balancing and automatic computation/communication overlapping, through overdecomposition. The migratable objects are called *chares* in Charm++. Chares are activated by remote invocations and execute without preemption. Communication latency is hidden by having multiple chares for each core. The Charm++ runtime can instrument chares, measure their execution time, and migrate them among processors in order to balance their loads.

Charm++ provides an SMP mode that, when enabled, spawns multiple threads within a node (process, actually).

Each thread becomes a *processing element* (PE) that handles a set of chares. Ideally, Charm++ encourages programmers to think of chares as virtual processors, so that the code is largely independent of the number of physical processors and the number of cores within each. But, for certain optimizations (such as data sharing and message reduction in BH), this ideal model is not feasible. Charm++ provides two language constructs—*group* and *nodegroup*—that are collections of chares; there is one chare per PE in a group and one chare per *process* in a nodegroup. Lacking a global name space, Charm++ BH adopts the hashed octree scheme again and shares its weaknesses.

To lower runtime overhead, Charm++ BH creates a chare for a group of particles instead of one particle. During tree-building, Charm++ BH creates an auxiliary space partitioning tree (similar to the top part of an octree). Each leaf represents a subcube of the space, enclosing a number of particles that is below a threshold set by the user. Each such leaf is handled by one chare (named *TreePiece* or *TP*). There are usually dozens of TPs per PE. TPs build local partial octrees with their particles and compute forces for them by traversing the entire octree. During traversal, TPs may need to access other TPs on the same node. To avoid this intranode communication, Charm++ BH designs a nodegroup (*TreeMerger*). When the SMP mode is enabled, it merges all local trees in a node and forms a larger local octree, which is then shared by all TPs in the node. TPs on the same PE may also need to access the same remote cells. To save this duplicate internode communication, Charm++ BH designs a group (named *DataManager* or *DM*). The TPs are similar to our worker threads, wheras the DM is similar to a communication thread except that the TPs and DM on a PE are executed by a single thread (i.e, the PE itself). The DM maintains a software cache so that duplicate off-node requests from the same PE are screened out. But note that DMs are per-PE objects. Duplicate requests from different PEs on the same node are not filtered. Table 5 shows that this approach can result in a significant amount of superfluous communication. We believe Charm++ BH could design the DM at node level to remedy that, but only after it handles thread synchronization problems as we discussed and fixed in PPL BH.

The Charm++ load balancer is triggered every a few time steps. At the end of such steps, the balancer computes the center of mass of TPs and weights these mass points with the TP costs measured by the runtime. After that, it maps mass points (hence TPs) to PEs using the well-known locality-preserving orthogonal recursive bisection method [14]. With a new TP-to-PE map, chare migration is triggered. The approach is nice, but we found an issue with it in our experiments: Because of particle movement, the space partitioning tree (hence the TPs) can change. If this change happens, it means we will use an outdated TP-to-PE map until the next balancing step, with imperfect load balancing. It is not clear how to balance the size of the TPs and the frequency of load balancing so as to optimize performance.

Thanks to overdecomposition, there are many TPs on a PE. Particles are sorted in an SFC order, and each TP owns a segment. In Charm++ BH, all TPs are active objects. A TP on a PE periodically yields the core to service incoming requests or give its partners a chance to run. From the PE's view, however, particles are handled in a somewhat arbitrary order. As we have shown in Section 3.4, this damages cache locality. It is not clear how the scheduling order could be controlled to improve locality.

TABLE 6
Computation Time(s) of BH Codes, $n = 1M$, $\theta = 0.5$

| Nodes | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| PEPC-mini | 12.19 | 5.69 | 3.10 | 1.82 | 1.14 | 0.81 | 0.67 |
| Charm++ BH | 6.27 | 3.83 | 2.24 | 1.45 | 1.03 | 0.74 | 0.63 |
| LET BH | 5.00 | 2.44 | 1.30 | 0.71 | 0.40 | 0.24 | 0.15 |
| PPL BH | 4.38 | 2.22 | 1.14 | 0.61 | 0.33 | 0.19 | 0.14 |

## 5.4 Comparison

In order to squeeze out performance from BH on multicore clusters, all implementations did nontrivial work. But by leveraging intranode multitasking and internode one-side communication, we encapsulated many complicated issues clearly and gave a simple but high-performance design. Source lines of code (SLOCs) are usually used to measure code complexity, though things are not always that simple. SLOCs of LET BH, PPL BH, Charm++ BH, and PEPC-mini (mentioned later) are about 2,000, 3,300, 8,000, and 25,000, respectively. One reason why the SLOCs of LET BH are fewer than those of PPL BH is that we applied C++11 syntactic sugar and the STL library extensively in LET BH, thus making it a compacted code. In reality, we found the code to be hard. To measure performance, we made all implementations compatible with SPLASH-2 BH so that we could use the same parameters and input files. [2] For LET BH, we used OpenMP guided scheduling and run it with one process per node, 12 threads per process. For PEPC, we started from PEPC-mini, a skeleton molecular dynamics front-end of PEPC. With little effort, we changed its interaction from Coulomb to gravitation and its expansion from multipole to monopole and got a code comparable with ours. The critical PEPC tree code is not modified. We compiled PEPC-mini with the Intel Fortran Compiler 13.1 and ran it with two processes per node, six workers per process (since this configuration achieved the best performance). For Charm++ BH, we used Charm++ 6.4 and triggered the ORB load balancer every five time steps. Test results for $n = 1M$, $\theta = 0.5$ are shown in Table 6. We varied $n$ and $\theta$, but the relative performance did not change much. We can see PPL BH's impressive performance advantage over the three other codes. We guess that the heavy hash table lookups hurt PEPC's performance. Note that the performance of PPL BH is even better than that of LET BH as opposed to what one might think, since PPL BH has better locality optimizations, as we discussed in Section 3.4.

## 6 PPL Library

Through a concrete application, the Barnes-Hut algorithm, we demonstrated the potential of a parallel programing style that combines intranode multitasking with internode one-

2. With one exception, leaves in PEPC's octree can contain only one particle. We configured PPL BH accordingly and found small performance variations so we ignored this.

sided communication and uses task preemption to hide communication latency. Although we worked on only one application, we believe that many of the abstractions (e.g., wthreads/cthread separation, queue-based synchronization, distributed task management, and tagged tree nodes) in our design can be reused for other applications. We are currently designing a library, PPL, to facilitate this kind of programming. PPL is designed as a C++ template library atop one-sided communication. We borrow ideas from the Intel Threading Building Block (TBB) [23]. At the top of the library are parallel algorithms such as parallel_for(), which are used by programmers to express parallelism in their applications. At the middle is the task scheduler, which does load balancing and latency hiding primarily through multitasking. At the bottom are global data structures in a PGAS memory model. Currently, we have implemented primitives such as global variables, global vectors, and global pointers in PPL, which mimic UPC shared variables, shared arrays, and shared pointers, respectively [6].

Generic high-level data structures such as trees can also be defined in a global address space. However, an important difference from data structures on shared memory is that we should also define caching properties for global data structures on distributed memory, such that variables can be cached without changing names to reference them, a capability that is needed for productivity but lacking in current PGAS languages. This is an interesting research area for us.

Besides TBB Task's standard `execute()` interface, tasks in PPL also provide interfaces such as `IsDataReady()` and `ExtractAddress()` to let the runtime know whether the needed data is locally available or, if not, what the global address of the data is so that worker threads can forward their requests to communication threads. We are now refining the interfaces of PPL.

## 7 Related Work

Rein and Liu presented a multipurpose n-body code named REBOUND [24] that incorporates MPI + OpenMP hybrid programming. They used OpenMP to parallelize the *for* loop for force computations for bodies on a node, in order to avoid intranode communication and achieve better load balancing, as we also do in PPL BH. However, the code does not support dynamic internode load balancing and inherits the inefficiency from the LET method.

Dinan et al. [25] introduced a hybrid parallel programming model that combines MPI and UPC. This model consists of UPC groups, and the intergroup communication is done through MPI. Processes in a UPC group can access each other's shared heap as normal UPC programs do, thus in effect increasing the amount of memory accessible to an MPI process. Using this model, Dinan et al. modified a UPC-only BH code and got a twofold speedup at the expense of a 2 percent increase in code size. However, this hybrid scheme's performance comes from replicating the *entire* octree in each UPC group (in other words, from reduced remote data references). The researchers did not discuss optimizations we found crucial to BH's performance. It also is not clear whether this code is scalable.

Dekate et al. [26] described a BH implementation in ParalleX [27]. They suggested four main characteristics of a scalable and high-performance n-body simulation: data-driven computation, dynamic load balancing, data locality, and variable workload. To this end, they have implemented BH in HPX, a C++ implementation of ParalleX, using many light-weight threads to increase parallelism; Work-queues to balance the load on the processors; Interaction lists to improve data locality; and Futures to make use of asynchronous operations. They also use manager threads and communication threads for the force-calculation phase. Some of these ideas are similar to what has been shown in this paper. However, they did evaluations only on shared-memory machines.

Jo and Kulkarni [17] described a point blocking optimization for traversal code, which can be thought as a counterpart of the classic loop tiling transformation for irregular applications. They introduced a transformation framework to automatically detect such optimization opportunities and presented autotuning techniques to determine appropriate parameters for the transformation. Our body tiling optimization was inspired by this work. However, we also measured BH locality in different body array distributions and studied the interplay between locality in private caches and reuse in shared caches, which they did not mention.

Zhang et al. [28] systematically studied the effect of hardware cache sharing on multithreaded programs. They indicated that to explore the potential of cache sharing, we need to write and compile parallel programs in a cache-sharing-aware fashion. Our study of BH shows that sometimes it also depends on input parameters and cache sizes at different levels.

The other well-known n-body algorithm is the fast multipole method (FMM) [29]. A great deal of work has been done on this method; see the 2012 Gordon Bell prize [30] and references therein. Both BH and FMM are hierarchical algorithms. But in addition to the particle-particle or particle-cell interactions in BH, FMM computes cell-cell interactions, thus effectively reducing its complexity to $O(n)$. The accuracy of FMM is controlled by the number of terms in its series expansion, in contrast to the cell-opening criterion $\theta$ in BH. Whether to open a cell is solely determined by its position, regardless of $\theta$ and body distribution in the cell. This implies that a process can know all the cells it needs to access before force computation starts. In other words, the communication is no longer computation dependent. Many FMM codes on distributed memory adopt the LET concept. An interesting question is whether we can transform FMM using one-sided to better overlap computation and communication.

Exploiting multithreading on multicores in high-performance computing has been extensively studied in different contexts. The linear algebra library PLASMA [31], a multicore version of LAPACK [32], has dynamic scheduling as one of its crucial elements [33]. It supports two scheduling strategies: static scheduling and dynamic scheduling. The dynamic scheduling strategy of PLASMA, implemented as QUARK, makes use of queues of tasks from which worker threads pop and execute tasks. DAGuE [34], which extends PLASMA to distributed memory, has another commonality with our work, as it uses a separate thread, called the Asynchronous Communication Engine, for doing internode communication. However, PLASMA and DAGuE have been designed for dense linear algebra, where the communication pattern is known up-front and is regular. Neither is true for BH.

## 8 Summary

We have shown how one-sided communication, message-driven task scheduling and caching of remote data can be combined to implement the Barnes-Hut algorithm with superior performance. We believe that a library implementing this programming model will prove useful for other applications as well, both in terms of ease of programming and performance. We plan to pursue this direction in future work. We have also shown the complex interplay, in multicore systems, between task scheduling and cache hit rate, at different levels of the cache hierarchy. The proper choice of a scheduling policy is extremely dependent on machine parameters and input parameters; on the other hand, the code behavior does not change rapidly across iterations. Hence, it is likely that run-time auto-tuning could be used to properly select the scheduling parameters.

## Acknowledgments

This work was supported by the US Department of Energy (DOE), Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, and by the US DOE Sandia National Lab Grant 1205852.

## References

[1] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, and K. Yelick, “ASCR programming challenges for exascale computing,” Office Sci., U.S. Dept. Energy, Washington, DC, USA, 2011.

[2] R. Stevens, and A. White, “Architectures and technology for extreme scale computing,” in *ASCR Sci. Grand Challenges Workshop Ser.*, 2009.

[3] H. Shan, B. Austin, N. Wright, E. Strohmaier, J. Shalf, and K. Yelick, “Accelerating applications at scale using one-sided communication,” in *Proc. 6th Conf. Partitioned Global Address Space Program. Models*, 1993.

[4] R. Barriuso and A. Knies, “SHMEM user’s guide for C,” Cray Res. Inc., Seattle, WA, USA, Tech. Rep., 1994.

[5] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, “Advances, applications and performance of the global arrays shared memory programming toolkit,” *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 203–231, 2006.

[6] UPC Consortium, “UPC language specifications, v1.2,” Lawrence Berkeley National Lab, Berkeley, CA, USA, Tech. Rep. LBNL-59208, 2005.

[7] *Programming Languages—Fortran*, Int. Org. Standardization, Standard ISO/IEC 1539-1:2010, 2010.

[8] J. Barnes and P. Hut, “A hierarchical O(nlogn) force-calculation algorithm,” *Nature*, vol. 324, p. 446–449, 1986.

[9] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “Methodological considerations and characterization of the splash-2 parallel application suite,” in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.

[10] M. Warren and J. Salmon, “A parallel hashed oct-tree n-body algorithm,” in *Proc. 1993 ACM/IEEE Conf. Supercomput.*, 1993, pp. 12–21.

[11] J. Zhang, B. Behzad, and M. Snir, “Optimizing the barnes-hut algorithm in UPC,” in *Proc. 2011 Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, p. 75–85.

[12] Berkeley UPC. (2014). [Online]. Available: http://upc.lbl.gov

[13] S. Aarseth, M. Henon, and R. Wielen, “A comparison of numerical methods for the study of star cluster dynamics,” *Astron. Astrophys.*, vol. 37, pp. 183–187, 1974.

[14] J. K. Salmon, “Parallel hierarchical n-body methods,” Ph.D. dissertation, Phys., Math. Astron. Dept., California Inst. Technol., Pasadena, CA, USA, 1991.

[15] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Mateo, CA, USA: Morgan Kaufmann, 2008.

[16] P. Mucci, S. Browne, C. Deane, and G. Ho, “PAPI: A portable interface to hardware performance counters,” in *Proc. Dept. Defense HPCMP Users Group Conf.*, 1999, pp. 7–10.

[17] Y. Jo and M. Kulkarni, “Enhancing locality for recursive traversals of recursive structures,” in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2011, pp. 463–482.

[18] OpenMP Archit. Rev. Board, “OpenMP application program interface version 3.1,” 2011.

[19] J. Singh, “Parallel hierarchical N-body methods and their implications for multiprocessors,” Department of Computer Science, Stanford University, Ph.D. Dissertation, Stanford Univ., Stanford, CA, USA, 1993.

[20] 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,” *Comput. Phys. Commun.*, vol. 183, pp. 880–889, 2012.

[21] L. Kale and S. Krishnan, “Charm++: A portable concurrent object oriented system based on c++,” in *Proc. 8th Annu. Conf. Object-Oriented Program. Syst., Languages Appl.*, 1993, pp. 91–108.

[22] L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng, “Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge,” Parallel Programming Laboratory, Tech. Rep. 11–49, 2011.

[23] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. Sebastopol, CA, USA:O’Reilly Media 2007.

[24] H. Rein and S.-F. Liu, “Rebound: An open-source multi-purpose n-body code for collisional dynamics,” *Astron. Astrophys.*, vol. 537, pp. 128–137, 2012.

[25] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur, “Hybrid parallel programming with mpi and unified parallel c,” in *Proc. 7th ACM Int. Conf. Comput. Frontiers*, 2010, pp. 177–186.

[26] C. Dekate, M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, and T. Sterling, “Improving the scalability of parallel n-body applications with an event-driven constraint-based execution model,” *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 3, pp. 319–332, Aug. 2012.

[27] H. Kaiser, M. Brodowicz, and T. Sterling, “Parallex an advanced parallel execution model for scaling-impaired applications,” in *Proc. Int. Conf. Parallel Process. Workshops*, 2009, pp. 394–401.

[28] E. Z. Zhang, Y. Jiang, and X. Shen, “Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs?,” in *Proc. 15th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2010, pp. 203–212.

[29] L. Greengard and V. Rokhlin, “A fast algorithm for particle simulations,” *J. Comput. Phys.*, vol. 73, no. 2, pp. 325–348, 1987.

[30] T. Ishiyama, K. Nitadori, and J. Makino, “4.45 pflops astrophysical n-body simulation on k computer: The gravitational trillion-body problem,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 5:1–5:10.

[31] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, “Numerical linear algebra on emerging architectures: The plasma and magma projects,” in *J. Phys.: Conf. Ser.*, vol. 180, no. 1, pp. 012037–012041, 2009.

[32] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users’ Guide*, 3rd ed. Philadelphia, PA, USA: SIAM, 1999.

[33] J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, and J. Dongarra, “Multithreading in the PLASMA library,” *Multi- and Many-Core Technologies: Programming, Algorithms, and Applications*, New York, NY, USA: Taylor & Francis, 2011.

[34] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, “DAGuE: A generic distributed DAG engine for high performance computing,” *Parallel Comput.*, vol. 38, pp. 37–51, 2011.

**Junchao Zhang** received the PhD degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences, China, in 2005. After graduation, he was a software engineer at ICT. He is currently a computer science assistant at the Mathematics and Computer Science Division at Argonne National Laboratory. His research interests include parallel program optimizations, one-sided communication enabled programming models, and MPI runtime implementation. Before joining Argonne, he was a postdoc research associate at the University of Illinois at Urbana-Champaign from 2010 to 2013.

**Babak Behzad** received the BSc degree from the Sharif University of Technology, Iran. He is currently working toward the PhD degree at the Department of Computer Science, University of Illinois at Urbana-Champaign. He works on different aspects of parallel computing under the supervision of Professor Marc Snir. His main research interests are parallel programming and also parallel I/O.

**Marc Snir** received the PhD degree in mathematics from the Hebrew University of Jerusalem in 1979. He was at New York University (NYU) on the NYU Ultracomputer Project during 1980 to 1982, and at the Hebrew University of Jerusalem during 1982 to 1986, before joining IBM. He is the director of the Mathematics and Computer Science Division at the Argonne National Laboratory and Michael Faiman and Saburo Muroga professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He currently pursues research in parallel computing. He was the head of the Computer Science Department from 2001 to 2007. Until 2001, he was a senior manager at the IBM T.J. Watson Research Center, where he led the Scalable Parallel Systems Research Group that was responsible for major contributions to the IBM SP scalable parallel system and to the IBM Blue Gene system. He was a major contributor to the design of the Message Passing Interface. He has published numerous papers and given many presentations on computational complexity, parallel algorithms, parallel architectures, interconnection networks, parallel languages, and libraries and parallel programming environments. He is an Argonne distinguished fellow, and a fellow of the IEEE, ACM, and the AAAS. He has Erdos number 2 and is a mathematical descendant of Jacques Salomon Hadamard. He recently received the IEEE Award for Excellence in Scalable Computing.