Optimizing the Barnes-Hut Algorithm in UPC

Junchao Zhang, Babak Behzad, Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
{jczhang, bbehza2, snir}@illinois.edu

PGAS languages’ support of a global name space facilitates the expression of parallel algorithms, since communication is implicit. This is especially convenient when writing irregular applications with data-dependent, dynamically changing communication patterns. However, programming in a shared memory style, with no explicit control of communication, may result in poor performance. The problem may be due to weaknesses of current implementations of PGAS languages or limitations inherent in these languages. To clarify which is the case, we discuss an implementation in UPC of the Barnes-Hut algorithm. A literal port of a good quality shared-memory implementation (merely replacing shared arrays with partitioned global arrays) achieves abysmal performance – more than 1000 times worse than a message-passing implementation. We achieve in UPC a performance comparable to message-passing with a series of optimizations. Most of these optimizations could be performed with limited changes in the source code using an enhanced run-time and a few language extensions or pragmas. We discuss the implications to the programmer, the compiler and PGAS languages themselves.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming

General Terms
Languages

Keywords
N-body, Barnes-Hut, PGAS, UPC

1. INTRODUCTION

Parallel programming models are divided into two worlds: shared-memory and message passing. In a shared-memory programming model, communication of data between threads is implicit – implied by their use of references to a shared variable; synchronization is explicit, and usually distinct from communication. In a message-passing model, communication of data is explicit, effected, e.g., by invocations to a message-passing library; the invocation both communicates data and synchronizes the threads involved. It is widely believed that the shared-memory programming model is more convenient, as users do not have to express communication [5]. On the other hand, cache-coherent shared-memory hardware does not scale to a large number of processors, and shared memory emulation atop a distributed memory hardware often results in abysmal performance [13]. One-sided communication libraries, such as SHMEM [3] or GA [15], provide a closer, scalable approximation to the shared-memory programming model and map well onto systems with hardware support for Remote Direct Memory Access (rDMA) – but still require explicit coding of communication.

PGAS languages, such as Unified Parallel C (UPC) [21] or the latest Fortran (Fortran 2008) [10] attempt to resolve this tension by supporting a Partitioned Global Address Space: Communication is implicit, effected by the use of global references – thus providing a shared-memory programming model; references to private variables are syntactically distinct from references to shared variables, so that the support for shared memory does not slow down the execution of explicitly local references; and global arrays are explicitly partitioned by the programmer, thus providing some explicit control of locality. PGAS languages map very naturally to hardware that supports rDMA operations [16].

Codes increasingly evolve toward the use of irregular, dynamic communication, since adaptation of the computation to the structure of the problem in hand can reduce the computational load. Such codes most benefit from the use of a shared-memory programming model: the changing communication pattern is handled in shared memory using indirection, but is much more awkward to express using message-passing. Therefore, we believe that a scalable shared memory programming model must handle irregular, dynamic communication patterns – this is where its impact on productivity will be most significant.

We study in this paper the suitability of PGAS languages for expressing such codes. In particular, we look at the im-
pact of various optimizations that reduce communication overheads. This includes optimizations that have been studied before – namely, non-blocking communication, message aggregation and the casting of global pointers that refer to local variables into local pointers [7]. It also includes an optimization that is essential for shared-memory performance but, has been much less studied for PGAS languages – namely caching.

We use for our study the Barnes-Hut algorithm – which strongly requires dynamic, irregular communication. We use, as a baseline, a UPC Barnes-Hut (BH) code from the Berkeley UPC release 2.11.4 [4], which is almost a literal translation from the SPLASH-2 benchmarks [23]. This code is highly optimized for shared memory, but has no distributed-memory optimizations. This code performs badly on a distributed memory machine. Then we show a series of optimizations that dramatically improve the performance, eventually reducing compute time by a factor of over 1600. The resulting performance is comparable to that achieved by a well-tuned message-passing code. All these optimizations are done manually, but most of them could be largely automated.

There are relatively few studies of the use of PGAS languages for problems with dynamic, irregular communication; see, e.g., [8]. To our knowledge, this is the first paper that thoroughly studies the BH algorithm in a PGAS language and studies the impact of caching. The main contribution of the paper is to indicate what enhancements are needed in PGAS languages, their compilers and their run-time in order to facilitate the expression of codes such as BH, while achieving good performance.

The rest of this paper is organized as follows: Section 2 introduces the BH algorithm, describes its implementation in SPLASH-2 and the baseline UPC implementation. Section 3, describes a series of optimizations to improve the performance. Section 4 considers algorithmic changes that are needed to achieve scalability to thousands of threads. In Section 5 we discuss how these optimizations could be eased by better compiler and run-time technology and language enhancements. We survey related work in Section 6 and conclude in Section 7.

2. THE BARNES-HUT ALGORITHM

2.1 Sequential Algorithm

The Barnes-Hut (BH) algorithm [2] is used to simulate the evolution of a system of \( n \) bodies where each body applies a force on each other body. While this formulation applies to many physical phenomena, we shall assume gravitational forces. The system is simulated by a sequence of time steps. At each time step, the forces exerted on each body are computed and the position and velocity of each body are updated. In order to avoid the \( O(n^2) \) complexity of a naive computation, one approximates the interaction of a body with a set of other bodies, by representing the set with one mass located at its center of gravity. The approximation is valid when the bodies in the set are “far enough” from the first body – with “far enough” being formalized as \( t/d < \theta \), where \( t \) 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.

The BH algorithm partitions the 3D space hierarchically into cells using an octree representation. The root of the octree represents the cell that contains all bodies. Each cell is recursively divided into octants, until a cell has only one body\(^1\). To compute forces exerted on one body, the procedure begins with the root cell. If the current cell is far enough or contains only one body then we compute force with it and stop there. Otherwise, we “open” the cell and continue, recursively, with each of its children. With this hierarchical approach, the BH algorithm reduces the computation complexity to \( O(n \log n) \) [2].

2.2 Parallel Algorithm

An efficient parallel implementation of the BH algorithm has to address two critical issues: load balance and locality. The cost of computing forces for each body is not equal and changes over time. Since body locations change slowly, one can use the cost of a body (i.e., number of interactions between the body and cells) at the previous time-step as an estimate for current cost; the bodies are distributed so that the estimated total cost of each group is roughly equal. Also, because of characteristics of this application, cells accessed during force computation for one body are likely to be accessed again in force computation for a nearby body. So it is better to allocate bodies near to each other to one processor in order to exploit the locality. The shared-memory BH code in SPLASH-2 [20, 19, 23] achieves both goals. It executes in parallel at each time-step 5 phases, separated by barriers. The phases are:

**Build Octree**: Each thread inserts into the shared octree structure the bodies it owns. Each thread stores in a private array a list of pointers to the cells, i.e., tree nodes, it creates (the thread “owns” these cells).

**UpdateCells**: A bottom-up pass on the tree computes the center of mass and cost of each cell. The cost of a cell equals to sum of the costs of bodies it contains. Each thread processes cells it owns, possibly waiting for the children cells to be processed.

**PartitionOctree**: The tree leaves are split into \( p \) segments of consecutive leaves of roughly equal total cost, where \( p \) is the number of threads. Thread \( i \) picks the \( i \)-th segment. Each thread finds independently the bodies in its segment and stores pointers to them in a private array (the thread “owns” these bodies).

**ComputeForces**: Each thread computes forces for bodies it owns. During force computation, threads also compute new costs.

**AdvanceBodies**: Threads advance their bodies by computing new accelerations, velocities and positions. Each thread computes the boundaries of the box containing its bodies; global max and min reductions are then used to compute the boundaries of the new root cell for the next time-step.

\(^1\)In the SPLASH-2 code, cells are partitioned until they contain no more than \( k \) bodies. We use the same approach, and same threshold \( k = 10 \) as SPLASH-2, but will not discuss this detail afterward.
The force computation phase, which perform $O(n \log n)$ operations, usually dominates the performance; other phases perform $O(n)$ operations. This implementation performs and scales well on shared memory machines. Singh got a near linear speedup on a 48-processor system with 32K bodies [19]. The force computation phase consumed most of the execution time ($\sim$90%), next was tree-building ($\sim$10%). Other phases had negligible overhead.

### 2.3 UPC

We briefly introduce the UPC language in this section. The reader is referred to [21] for a full language description.

UPC is a parallel extension to the C programming language. An execution consists of a fixed number of threads, each executing the same program. Each thread has a local memory, which is divided into two areas, one private and the other shared. Variables in the private area can be accessed only by the local thread, while variables in the shared area can be accessed by any thread. Shared scalars are stored in the local memory of thread 0, while shared arrays are distributed across all memories, with a user-specified block-cyclic distribution.

UPC supports two types of references: local references, that point to private or shared locations in local memory, and global references that can point to any location in shared memory. As a result, UPC has three types of pointers: a private pointer to private, a private pointer to shared, and a shared pointer to shared, according to where the pointer is stored and what it refers to. There are no shared pointers to private as such pointers would be useless.

Communication is normally effected in UPC by dereferencing global pointers that point to remote locations. Since remote access at word granularity is very inefficient, compilers attempt to aggregate multiple accesses in one operation and issue accesses early, in order to hide latency. UPC also provides functions for explicit, blocking transfer of a block of data from one thread to another (upc_mempot(), upc_memput() and upc_memcpy()). Standard UPC (V 1.2) does not have non-blocking communications, but those are supported by some UPC extensions.

### 2.4 Baseline Code

We use as baseline a UPC BH implementation released in Berkeley UPC 2.11.4. The code is nearly a literal C to UPC translation of the implementation in SPLASH-2, which was highly optimized for shared-memory. Global parameters are declared as shared scalar variables. The dimension of the root cell, rsize, is also a shared variable. Cells and bodies are stored in shared memory. The bodies are stored in a global array (with the same number of bodies on each thread). Threads allocate space for the cells they create in their local shared memory. Overall it is very easy to port the shared-memory BH code to UPC.

### 2.5 Test Environment and Approaches

We tested the application on Blueprint, an IBM Power5 cluster with 116 compute nodes. Each node has 64GB memory and 16 cores running at 1.9GHz. The operating system is IBM AIX 6.1. We used the Berkeley UPC (BUPC) compiler 2.12.0, because it was the only compiler we had access to that supported versatile non-blocking communication libraries we needed in our optimizations. Its GASNet run-time was built on the IBM LAPI conduit. The back-end compiler is IBM XLC 11.1, with its -O3 option turned on. Except for changing the default number of bodies (16K) to larger numbers, we kept the default settings as in SPLASH-2: $\theta = 1.0$, a time-step $= 0.025s$, and the initial body distribution was generated by the Plummer model with $M = -4E = G = 1$ [1]. Also, as in SPLASH-2, we run four time-steps and timed the last two.

To take advantage of hardware intra-node shared memory support, the user can use the -pthreads option to build a threaded executable, or enable process-shared memory (pshm) in GASNet. On Blueprint, only the former was supported. The user specifies how many pthreads to spawn per process, with each pthread corresponding to one UPC thread. To make interpretation of results easier, in this section and next section, we only use one thread per node. These two sections focus on strong scaling: We keep the number of bodies constant at 2M, and vary the number of nodes (which is equal to the number of UPC threads). We consider weak scaling in section 4, with a fixed number of bodies per UPC thread, and multiple UPC threads per node.

It is worth mentioning that if one uses multiple processes on a node, the performance can be very bad, because all communications between processes go through LAPI. For example, we have tested the baseline UPC BH with 2M bodies and 16 threads on one node. With -pthreads enabled (16 threads/node – one thread per core), the computation time was 26s. But with -pthreads disabled (16 processes/node), the computation time was more than 36000s. So in this paper, we always enable -pthreads when we want to use multiple threads per node.

We tested the code with 1, 2, 4, ..., 64, 96, 112 nodes. Tables with the detailed results of all tests are given in a separate technical report [24]. Due to lack of space, we provide here only a short summary of the results.

The parallel performance of the baseline code is bad. Computation time on 2 threads is 111 times longer than on one thread. Compared with 2 threads, a 32-thread run gets a speedup of about 6.6. Similar results were also reported in [7].

### 3. OPTIMIZATIONS

We introduce in this section a series of optimizations to improve the baseline UPC BH performance. The test environment is the same as in the previous section. We measure the cumulative effect of the optimizations described below.

#### 3.1 Replicate Shared Scalar Variables

UPC shared scalar variables are all stored on thread 0. If they are accessed frequently by other threads, it could be beneficial to maintain copies on the other threads. We use two approaches to do so:

Two scalar variables (problem parameters, one of them is $\theta$) are written only during program initialization but fre-
We replicate them and initialize them at each thread. One scalar variable (i.e., size, the size of the root cell) is initialized once at the end of each iteration and frequently read by all threads in the force computation phase. We replicate it and broadcast the updates.

These three simple changes reduce computation time by 50% at 2 threads and 79% at 112 threads.

### 3.2 Body Redistribution

In the baseline UPC BH code, bodies are distributed evenly among threads during initialization. The distribution is fixed and will not match the dynamic distribution of bodies to threads that is needed to achieve good load balancing and locality. On shared-memory machines, thanks to the hardware caches, the initial data layout has limited performance impact. This is not the case on distributed-memory machines. Since a thread accesses the fields of each body it "owns" multiple times during an iteration, it is profitable to redistribute bodies to threads that own them. We add a body redistribution phase right after the octree partitioning phase; the distribution is preserved until next partitioning. With this approach, a thread only accesses bodies stored in its local shared memory, except during the redistribution phase. The pointers to these bodies can be cast to local, further improving performance.

It may seem costly to redistribute all bodies at each time-step. However, as their locations change slowly, only a small fraction of the bodies are migrated at each time-step. In our experiments, we find that about 2% of the bodies allocated to a thread migrate during a time-step.

To ease memory management, we use a double-buffer approach. Remote bodies copied to the local thread are appended to the end of a buffer; when the buffer fills, the non-empty entries are copied to the other buffer. Our experiments do not measure buffer copying overhead, since the buffer does not fill in 4 steps. However, additional experiments showed the overhead to be very small (copying occurs rarely, for a reasonably sized buffer, and does not involve inter-node communication.)

The new phase has negligible cost (~0s) but notable gains. It reduces computation time by 17% at 2 threads and 4% at 112 threads.

### 3.3 Cache Remote Cells

During force computation the octree is read only, and cells accessed for computing forces acting on one body are likely to be accessed again for a computation involving a nearby body. Therefore, caching cells can significantly reduce communication, without requiring a complex coherence protocol. We cache cells using pointer swizzling. To each cell structure we add a Boolean flag localized to indicate whether the cell’s children are cached locally. The flag is initialized to be 0. The tree root is initially cached at all threads. If a thread needs to open a (locally cached) cell $n$, it checks whether $n$-localized is set. If yes, it knows $n$’s children are cached, and pointers to them can be safely cast. If not, it gets the children, copies them in local memory and resets pointers in $n$ to point to the local copies.

The algorithm builds at each thread a partial local copy of the global octree. This is similar to the the locally essential tree that is constructed at each node in the original algorithm of Salmon [17]. The difference is that our local copy is built, demand-driven, during force computation, instead of being built up-front before force computation.

This optimization has a very significant effect: The time spent on force computation is dramatically reduced by 99% (!) for all multithreaded runs. Interestingly, the time is also reduced by 25% even for one thread, as global pointers are replaced with local pointers. The overall compute time for all multithreaded runs is reduced by 98% or more. For the first time, we get improved performance from parallelism. The speedup with 112 threads is about 13 – still unsatisfactory.

We did not have a cache overflow problem with 2M bodies; we do not expect that to be a problem as each body interacts on average with $O(\log n)$ cells. The caching algorithm might seem inefficient as it caches cells even if they already are in local shared memory. We implemented an alternative algorithm that avoids caching such cells. This requires maintaining in each cell structure two pointers for each child: One points to the “master copy” of the cell structure, and the other points to the locally cached copy, if such exists. This variant did not improve performance – it only saves local copies, at the expense of a more complex cell data structure and more complex logic.

### 3.4 Octree Building

In the SPLASH-2 code, the octree is built by having each thread insert its bodies into a common global tree. If a thread wants to modify a cell, it must lock it. The lock contentions increases with the number of threads and becomes a bottleneck. In Singh’s results, with 48 processors, the fraction of time spending in tree-building is about 10% [19]. In UPC, the overhead is larger, due to the higher cost of global locks and remote accesses. Tree-building consumes almost 27% of the total computation time with 64 threads.

Singh suggested a new tree-building algorithm in his thesis [19]. In this algorithm, each thread first builds a local octree with its bodies. Then threads merge their local octree into the global octree. Building a local octree is a sequential procedure, which does not need any locks. And because each thread’s bodies have good locality, only a small portion of cells will be modified during tree-merging.

We implemented this algorithm in UPC. This modification greatly reduced tree-building time. For example, for 112 threads, the time spent on tree-building was reduced by 83%. Unfortunately, it also increased the cost of cell updating for reasons discussed next. For 112 threads, the time spent on cell updating was doubled, negating much of the gain.

The center of mass computations for the octree are done in parallel. To compute the center of mass of a cell, we have to wait until the center of mass of every child has been computed. The waiting time depends on the order the
centers of mass are computed. In the original tree-building algorithm cells are listed at a thread in the top-down order they were created; traversing this list in reverse, bottom-up order has the side-effect of reducing waiting time. However, the new tree-building algorithm breaks this ordering.

In order to solve this problem and reduce communication, we use an alternative approach. Each thread first computes local centers of mass for the cells in the local octree. This requires no communication and is done fast. Then, during tree merging, whenever two cells are merged, a new center of mass for the merged cell is computed. The center of mass tree merging, whenever two cells are merged, a new center requires no communication and is done fast. Then, during the local centers of mass for the cells in the local octree. This requires no communication and is done fast. Then, during tree merging, whenever two cells are merged, a new center of mass for the merged cell is computed. The center of mass resulting from the merge of two cells \( l \) and \( g \) is computed as \( (l \cdot \text{cof} + g \cdot \text{cof})/(l + g) \). This weighted average computation is associative and commutative, so the merges can occur in any order; the update of the center of mass is done atomically.

While this algorithm increases the time for tree building alone by 60% at 112 threads, it decreases the overall time for tree building and cell updating by 25%. If compared with the code in Section 3.3, it reduces the overall computation time by 5% at 2 threads and 15% at 112 threads. The speedup with 112 threads is about 15.

3.5 Non-blocking Communication and Message Aggregation

Caching remote cells significantly reduces communication in the force computation phase. However, the first “cache miss” is still expensive as the thread waits for the remote access to complete. We consider two optimizations to reduce this cost. The first is to use non-blocking communication to overlap communication with computation. The second is to use message aggregation to bring in multiple cells in one big message instead of multiple small messages. Both require using a form of concurrent multitasking where a thread switches to another task when the current task blocks waiting for a remote access to complete.

The BH application has two potential sources of concurrency to support non-blocking communication. One is that force computations for distinct bodies are independent and can be executed in any order. The other is that the vector summing of forces acting on one body is associative and commutative, so that a body can interact with cells of an octree in any order, provided that a parent is visited before any of its children.

We have designed a framework which leverages both sources. It is similar to the latency hiding approach described in [22]. Each thread maintains a list of \( n_1 \) working bodies that are processed concurrently. The thread maintains for each working body a list of frontier cells, i.e., cells that are cached locally, need to be opened, but whose children have not yet been cached (i.e., their localized flag is false). Initially, the only frontier cell for a body is \( L_{\text{root}} \), the local copy of the octree root. To compute forces for a working body, a thread loops through the body’s frontier cells. For each frontier cell \( n \), if \( n->\text{localized} = \text{false} \), the thread deletes \( n \) from the frontier cell list, opens \( n \) and proceeds recursively. If the thread meets a descendant cell \( d \), which is needed to be opened but \( d->\text{localized} = \text{false} \), then the thread adds \( d \) as a new frontier cell, adds the children of \( d \) to a list of requested cells and marks \( d \) as requested (to avoid requesting children of \( d \) again). Once all frontier cells in the list for a body are processed, the thread then goes to another working body. Periodically, when sufficiently many requests have accumulated, the thread will initiate a non-blocking communication to bring in the requested cells. We maintain at most \( n_2 \) concurrent asynchronous communications and start a new communication only if there are at least \( n_3 \) requested cells to be handled by that communication.

We use the BUPC extension function \texttt{bupc_memget_vlist_async()} to bring in cells. It is an indexed non-blocking gather operation and can get data from multiple remote threads. It returns a handle. The handle can be used in \texttt{bupc_trysync()} or \texttt{bupc_waitsync()}. The former is a non-blocking test, while the latter is a blocking test. When the thread cannot make progress, it tests outstanding requests. If a request is completed, the localized flag of parents of cells in this request are set.

The above optimizations add a little overhead to single thread performance. But with \( n_1 = n_2 = n_3 = 4 \) the total compute time is reduced by 23% for 2 threads and 75%, for 112 threads; the speedup for 112 threads is more than 70. The effect is more significant (in strong scaling) when the number of threads is larger, as communication takes a larger fraction of the total computation time.

We found that the results are not very sensitive to the choice of \( n_1, n_2 \) and \( n_3 \). performance is good even with \( n_1 = n_2 = n_3 = 1 \). This is because of the good locality: most force computations involve local cells, and accesses to local cells can hide communication latency even with a few remote cells accessed at a time. For the same reason, we simply use \texttt{bupc_waitsync()} instead of \texttt{bupc_trysync()} to wait for all outstanding requests to be completed before continuing computation. Most of the time, the communication is completed and the wait call returns immediately.

It might seem possible to improve performance by binning cells according to their sources and generating separate requests for each source. However, it turns out that cells in a request have very good locality. In our testing, with 64 threads, more than 94% of the requests have only one source thread, and about 5% of the requests have two source threads. With 112 threads, the numbers are 93% and 6% respectively. The extra complexity of binning requests does not seem warranted.

The results of the successive optimizations are summarized in Figure 1 that shows (on a log scale) the performance achieved by the cumulative application of the optimizations (the “subspace” optimization is discussed in the next section). Table 1 shows the time consumed by each phase, for runs with 112 processes.

4. SCALABILITY

In this section we will study the scalability of the program, using weak scaling. New bottlenecks arise, that require an algorithm change. We then study weak and strong scaling of the updated code.

4.1 Weak Scaling
forest to the appropriate leaves of the top tree. This global merge only requires updating pointers; the updates do not conflict and require no locking. The idea is illustrated in Figure 4 (we show a quadtree, rather than an octree, to simplify graphics). The space is divided across 3 threads; edges reaching out of the clouds indicate where global insertions are needed.

Each node in the octree represents a cell – a sub-cube of the cube that contain all bodies. We divide a cell into 8 octants if its total cost exceeds a threshold. We use a relatively large threshold, \((2/3) \times \text{total costs}/\text{THREADS}\) in our experiments to avoid unnecessary over-division. The threads build collectively the top levels of the octree, starting from the root (level 0), until a level is reached where the cost of all nodes is below the threshold.

Level 0 is build by having each thread assign all the bodies it owns to the root cell, and sum their costs. A collective all-reduce is then used to compute and distribute the total cost of the root cell.

Assume that the costs of all nodes in the octree at level \(i\) has been computed and each thread has assigned its bodies to the appropriate cell. Then each cell with cost above the threshold is further subdivided at each thread into 8...
sub-cells. Each thread then partitions the bodies it owns that were assigned to that cell and assigns them to one of the 8 sub-cells. The thread also computes the total cost of the bodies assigned to each of the 8 sub-cells. A collective vector all-reduce is then used to compute the total cost of each octree node at level \( i+1 \) and have these costs available at each thread.

The leaves of the resulting tree are now allocated to threads so that each thread “owns” a set of consecutive leaves of approximately equal total cost. Each thread becomes the owner of the bodies contained in the leaf cells allocated to it. Then an all-to-all communication is used to distribute bodies to their owners. The communication volume is small because of persistence across iterations. After that, each thread builds a local sub-forest of the octree that contains the bodies it owns. Once the sub-forest is built and the center of mass of each cell is computed, a thread can directly hook the subtrees to thread 0’s octree. After all threads have hooked their subtrees, thread 0 computes centers of mass for top level cells; this is also very fast, since the number of cells in the top is typically linear in the number of bodies it owns. Once the sub-forest is built and the center of mass of each cell is computed, a thread can directly hook the subtrees to thread 0’s octree.

The difference between our algorithm and the original one of Shan and Singh’s is that we use costs, rather than number of bodies to decide whether to divide a subspace, so that the octree structure matches the body partition; and we handle one level at a time, rather than one cell at a time, thus reducing the number of communications. Vector reduction usually can be implemented very efficiently. We find this is critical to the performance when the number of threads is large. We keep costs in a separate array to facilitate the use of vector reduction.

Figure 5 shows the results of the updated code. Compared to Figure 2, performance of tree-building is greatly improved. We can also see that without vector reduction, tree-building cost becomes prohibitive when the number of threads grows beyond 16x32. With vector reduction, tree-building scales smoothly. To see why, consider that in the 16x112-thread case, we create about 10,400 subspaces in a time-step. Without vector reduction, this means 10,400 scalar reductions. The total number of levels of subspaces in this case is 9, so that we do, instead, 9 vector reductions.

We tested the program with 1, 4, 8 or 16 threads per node on 1 to 112 nodes. Figure 6 shows the results. For each number of threads, configurations that use fewer nodes perform better, but not by much. For example, for 64 threads, “16 threads/node, 4 nodes” is only 7% better than “1 thread/node, 64 nodes”. We think it is because optimizations we introduced in this paper make communication cost in this program no longer significant. Figure 6 also has results for “1 process/node” with -pthread enabled. Disabling pthreads improves performance by about 50% as compared to “1 thread/node”, with pthreads enabled. This would seem to indicate performance problems in the runtime we have been using, possibly related to the interaction of GASNet with pthreads.

Figure 6: Weak scaling with and without vector reduction, 250K bodies/thread

4.2 Strong Scaling

We also tested the strong scaling of the updated code with 1 process per node on 1 to 112 nodes. The results for 112 processes are given in Table 1 (the subspace column). With all optimizations introduced in this paper, comparing with the baseline version, we improved the performance from 272 times with 2 processes, up to 1644 times with 112 processes. With 112 processes, the speedup is about 81; force computation and tree-building consume about 82% and 10% of the total computation time respectively. Figure 7 shows the strong scaling speedup curves with threading. For 1,2,...,64,
96, 112 threads, we run with 1 thread per node. For 16, 32, ..., 512 threads, we run with 16 threads per node. The inflection point happens between 256 and 512 threads, where the number of bodies per thread goes from 8K to 4K.

Figure 7: Strong scaling, 2M bodies

4.3 Comparison to ChaNGa

We compared the performance of our code to the performance of ChaNGa[11], a state of the art production code written in Charm++. Since Blueprint was decommissioned during our testing, we used another platform, namely Ranger, to run the tests in this subsection. Each node of Ranger has four quad-core AMD Opteron CPUs running at 2.3GHz. Nodes are connected by Infiniband. The operating system is CentOS 4.9. The backend compiler is Intel ICC 11.1. GASNet was built on the ibv conduit with pshm support enabled. We used ChaNGa 2.0 and built it with Charm++ 6.2. We run ChaNGa and UPC BH with same quadrupole force computation formula. Three inputs from ChaNGa benchmarks were used: dwf1, dwf1.6144 and lambb, which contain about 5M, 50M and 80M bodies respectively. All tests were run with 16 threads/node, except those with 168 threads, which were run with 8 threads/node. We run 10 steps and measured the average time per step in seconds. The table below shows the results. The performance of a highly optimized UPC BH is very competitive.

<table>
<thead>
<tr>
<th># of threads</th>
<th>dwf1</th>
<th>dwf1.6144</th>
<th>lambb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaNGa</td>
<td>23.8</td>
<td>2.9</td>
<td>8.2</td>
</tr>
<tr>
<td>UPC BH</td>
<td>19.9</td>
<td>28.2</td>
<td>7.2</td>
</tr>
</tbody>
</table>

5. DISCUSSION

The results in this paper show, not surprisingly, that a naive shared memory-style implementation of a code with irregular, dynamic communication patterns in UPC can have abysmal performance. The sequence of optimizations that we described improved performance by a factor of 1644 for 112 threads. The original code was designed for shared memory performance, and very different optimizations were needed to achieve performance in UPC that is competitive with message-passing.

It is important to understand whether this gap reflects weaknesses in current compile and run-time support for UPC or is inherent to UPC and similar PGAS languages; and, in the later case, whether simple extensions to UPC can fill the gap. We attempt to classify each of the program transformations described in Sections 3 and 4, in one of three categories:

1. Transformations that can be fully automated.
2. Transformations that could be enabled by modest additions to the source program, e.g. pragmas.
3. Transformations that are likely to require significant manual code rewrite.

Many of the optimizations done manually in Section 3 are different forms of caching: Variables are moved or copied to take advantage of existing reuse (e.g., caching cell values) or to enhance reuse (e.g., repartitioning bodies). There are several factors that facilitate caching:

- Values are encapsulated into reasonably large objects (cell or body structures) that can be handled as one unit, for the purpose of caching, one “logical cache line”. This results in coarser granularity for data transfers and avoids the false sharing that occurs when “cache lines” are arbitrarily defined. Furthermore, these objects are accessed via pointer dereferencing; therefore, caching can be handled in software via pointer swizzling.
- The “coherence state” of objects of a given type does not change during a computation phase. Thus, cells are read-only while bodies are updated during force computation; and bodies are read-only while cells are updated during tree-building. Exceptions, such as concurrent updates during tree merges or center-of-mass computations are sources of performance bottlenecks and need careful algorithm redesign. This bulk change of coherence state is typical of many parallel codes, such as molecular dynamics, finite-elements or Poisson solvers as it provides a simple scheme to avoid races. The bulk change in state is easier to handle than individual, asynchronous changes that are tracked by normal coherence protocols.
- The ownership of objects being updated does not change during a computation phase and the algorithm uses an “owner compute” discipline. Again, this logic is followed by many parallel codes, as it avoids races and reduces communications. Furthermore, it is often the case that the set of objects “owned” by a thread is known when the phase where these objects are updated starts. This is usually the case when the program controls load balancing and locality by explicitly allocating objects to threads.

We now categorize the different manual optimizations of Sections 3 and 4.

Replicate write-once scalars [1-2]: The transformation described in Section 3.1 consists of replicating write-once or “write-rarely” shared scalar variables to each thread. The transformation can be automated, and is almost always desirable for write-once scalars. On the other hand, one needs profiling information to decide whether replication improves performance for variables that are repeatedly updated. The programmer can provide this information by...
annotating the variable declaration.

**Redistribute by ownership** [2, 12] The transformation discussed in Section 3.2 consists of redistributing shared data (bodies) between iterations. Each thread has a list of bodies it will update and the bodies are redistributed accordingly. It seems hard to fully automate this transformation. The programmer can facilitate this task by providing a list of the bodies owned by each thread and specifying that each body is only updated by its owner: We replace an indexed gather call by an “indexed prefetch” call. The run-time can move data and swizzle pointers.

**Cache read-only data on demand** [2, 12] The transformation that provides most performance improvements is described in Section 3.3. It consists of caching in local memory remotely accessed, read-only data. The caching run-time is relatively simple, since remote accesses are performed using pointers to shared; the access can be redirected to the local cache by pointer swizzling. Again, a fully automated analysis may have hard time figuring out that cells are read-only for a long period, and have heavy reuse. The programmer that understands the Barnes-Hut algorithm would have this knowledge and could annotate the code so as to provide this knowledge to the compiler and run-time; the actual caching can be then fully automated.

**Reduce locally then globally** [2-3, 12] The next transformation, in Section 4, reduces contention. Rather than globally updating the shared octree data structure, one performs the updates locally, next merges the updates. This transformation is often done by compilers for simple reduction operations: One computes a local reduction first, next do a global reduction. The transformation is harder to automate for more general updates, when it is not clear that the updates commute. It seems that, with current compiler technology, such transformations must be performed manually. It might be possible to automate them by having the user identify that updates commute. The improved computation of local center of mass in Section 3.4 uses the same type of transformation, but it is easier to verify that the updates commute (up to floating point rounding errors).

**Use parallel slack for latency hiding** [2, 12] The transformation described in Section 3.5 has a significant effect on performance: It uses the availability of “parallel slack” in the code executed by each thread in order to hide latency and aggregate communication. The code executed by each thread has two levels of parallelism: One iterates over all bodies owned by the thread; and, for each body, one traverses all cells that are not “far enough” from the body. We use this available parallelism to perform “concurrent multitasking”: a thread blocks if it waits for a remote access and is rescheduled when the access is complete. A combination of compiler and runtime can implement such concurrent multitasking efficiently. However, a compiler may not be able to identify the available parallelism. The user can annotate the loop iterating over bodies, and the loop iterating over the 8 children of a cell that is opened to indicate that iterates are independent.

In our case, since most of the accesses are local, we need only a small amount of parallelism within each thread. This could be achieved by using an overdecomposition – having more UPC threads than cores, and dynamically descheduling threads when they are waiting for a remote access to complete. A run-time such as used in Charm++ and AMPI [12] could be used for UPC, in order to achieve this effect.

**High-level alignment** [3, 12] The last transformation described in Section 4 is an algorithm change; it is not clear how a compiler or run-time could facilitate such a change. However, the change follows a pattern that is likely to occur in other parallel codes, as well: We modify a code that constructs in parallel a shared data structure so that each thread produces the same skeleton for the shared data structure (the octree); the merge of the local fragments becomes much more efficient.

### 6. RELATED WORKS

Graph algorithms usually have dynamic, irregular computation and communication patterns, hence present the same problem as BH. The study by Cong et al. [8] examines UPC implementations of connected components and minimum spanning tree algorithms. They achieve an order of magnitude improvement with transformations that coalesce communications and cache data. They also use algorithmic improvements in order to reduce local work and communication volume, avoid hot-spots and load imbalances and better use the underlying hardware – resulting in an algorithm that is better than the original shared memory algorithm, on SMP’s.

An alternative approach to the coding of BH in UPC is pursued by Dinan et al. [9]: They improve the performance of a UPC BH code by adding MPI communications: The repartitioning of bodies at the end of each iteration uses an MPI all-gather operation. The code size is increased by 2%, while performance is improved by about *×50 on 256 processors. (Their base UPC code already includes a body repartitioning phase; the shared-memory code we started with only repartitions pointers to the bodies.)

Some of the optimizations described in our paper, such as caching cells and using asynchronous communication, were first proposed for the BH algorithm by Warren et al. [22]. A fundamental observation in that paper is that if bodies are sorted in the order defined by the Morton code of their spatial coordinates then good locality is achieved by partitioning the sorted list into disjoint segments. Each segment roughly has the same weight, but may contain a different number of bodies. Rather than using an octree pointer structure for accessing cells, one can use Morton codes as hashing keys. It is interesting to speculate whether such a data-dependent storage order and dynamic partitions could be accommodated by extending PGAS shared array distributions.

Most of the optimizations described in our paper have been implemented manually in various production codes, in particular ChaNGa [11], which has shown scalability to 1000’s of processors.

The MuPC implementation of UPC supports software caching for scalar variables; variables are written back at each synchronization point, to avoid coherence issues [25]. A similar approach has been used to add caching to Berkeley UPC [6]. We could not find an evaluation of the perfor-
mance impact of caching on MuPC and the Berkeley project discusses performance only for a very simple benchmark. We suspect that fully transparent caching is unlikely to help the performance of more complex UPC codes, because of the cost of frequent invalidations and flushes due to false sharing and lack of knowledge of the true dependencies.

7. CONCLUSION AND FUTURE WORK

We analyzed the performance of a UPC implementation of the BH algorithm. We started with a naive translation of a good shared-memory algorithm to UPC and showed how successive program transformations can improve the performance of this naive UPC code by a factor of over 1600 – with the resulting code performing as well as a high-quality message-passing code.

Most of the optimizations we considered have to do, in one way or another, with the caching of shared data. We suggest that a shared memory programming model has two essential features:
1. Shared variables can be referenced by any thread.
2. Shared variables can be cached locally without changing the reference used to access them.

The first property significantly simplifies coding: the second is essential for achieving good performance while using global references. The current PGAS languages support the first feature but lack the second feature. Lacking this second feature, good locality is achieved by explicitly copying data from remote locations to local memory; the local, “in-cache” address is distinct from the global address, and much of the advantage of having one global address space is negated, since the same datum is now accessed using a different address when it is cached.

Caching is supported on shared memory multiprocessors by hardware. On distributed memory systems, where such a support is lacking, caching must be achieved by a combination of user code, run-time support and compiler analysis. It is essential to understand to what extent the user coding effort can be minimized by using language enhancements and better compiler and run-time technology. Our conclusion is that many of the transformations needed to enable caching can be done using existing compiler and run-time technologies, but require help from the user, in the form of statements and directives added to the UPC code, as discussed in Section 5. We found the need for directives indicating that variables of a certain type are read-only or write-exclusive within a (compound) statement and for cache prefetch statements. In addition, we either need annotations to specify that parallel iterates are independent, or to use overpartitioning. Finally, annotations that specify that certain updates commute would help in properly parallelizing reductions.

In order to achieve good performance in the BH UPC code, we had to perform transformations similar to those needed for good message-passing performance, thus increasing code complexity. One can question whether the use of UPC simplified the coding task and resulted in a simpler code. We believe this is the case, for three reasons: (a) The availability of a global address space means that the initial code could be much simpler; a BH efficient code could then be developed by successive refinements of this initial shared-memory code. (b) The availability of a global address space simplified communications, as one could use one-sided communication. (c) The global arrays provided simpler naming scheme for shared data. Unfortunately, we cannot directly compare programming efforts or even LOC counts, since production message-passing codes we have access to implement additional options.

We think it is important for PGAS languages to support efficiently shared-memory codes with dynamic, irregular communication and dynamic load balancing. These are the codes that are hard to write using message-passing; hence, these are the codes where the productivity impact of a shared-memory programming model can be most significant. We shall be pleasantly surprised if some of the transformations we classified as hard to automate with no language changes turn out to be within the realm of automated program transformations. However, we feel that it is unwise for the HPC community to wait until such progress occurs: Progress in automatic parallelization has been slow, and the investment in commercial compilers for PGAS languages is limited. Therefore, it is important to enhance UPC and other PGAS languages with a limited set of extensions that will enable the convenient and efficient expression of algorithms such as BH – without requiring heroic programming efforts, nor requiring novel compiler technology. There is now a strong push for parallel programming models that provide more productivity than message passing; PGAS languages seem to be the next step in that evolution. However, if PGAS languages are pushed into broad use with no adequate compiler and run-time support and no adequate functionality for coding conveniently an application such as BH, they will suffer the fate of High Performance Fortran [14]: Users will be turned off by PGAS languages before these languages become ready for prime time.

8. ACKNOWLEDGMENTS

We would like to thank the BUPC community for their help when we developed the code, and thank the anonymous reviewers for their suggestions. The work is supported in part by NSF grant 0833128 and supported in part by the NSF through TeraGrid resources provided by TACC under grant TG-SES090019. Zhang was supported in part by the 973 Program of China under 2011CB302500.

9. REFERENCES


