Optimizing the Barnes-Hut Algorithm in UPC*

Junchao Zhang, Babak Behzad, Marc Snir†

02/27/2011

Abstract

Partitioned global address space (PGAS) languages like UPC or Fortran provide a global name space to the user. This facilitates the expression of parallel algorithms, since communication is implicit, as compared to message passing where communication is explicit. 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 in PGAS languages. 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 a representative irregular application, namely the Barnes-Hut algorithm. We have implemented a series of optimizations that have improved the performance of this application above 1600 times compared with a baseline, naive, shared-memory style implementation. We then discuss the implications to the programmer, the compiler and PGAS languages themselves.

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

1 Introduction

Parallel programming models are divided into two worlds: shared-memory and message passing. In shared-memory programming models, communication of data between threads is implicit – implied by their use of references to a common memory location; 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 [6]. 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 [15]. One-sided communication libraries, such as SHMEM [3] or GA [18], 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. The tension between programming convenience (equated to shared memory) and program performance (equated to message passing) persists.

PGAS languages, such as Universal Parallel C (UPC) [25] or the latest Fortran (Fortran 2008) [13] 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

*The work is supported in part by NSF grant 0833128

†{jczhang, bbehza2, snir}@illinois.edu, Department of Computer Science, University of Illinois at Urbana-Champaign

Table 1: Communication Patterns: Categories and Examples

	static	dynamic
data-independent (regular)	regular finite-element mesh	?
data-dependent (irregular)	irregular finite-element mesh	adaptive mesh refinement

providing some explicit control of locality. PGAS languages map very naturally to hardware that supports rDMA operations [19]. Communication is identified by the use of global references, thus allowing for compiler optimizations that reduce communication costs by hiding latency (using split-phase communication) and amortizing communication overheads (using message aggregation) [9].

One can categorize the communication patterns of parallel applications as shown in Table 1. The communication pattern can be *regular* – e.g., as defined by a stencil in a Cartesian mesh; or *irregular* – e.g., nearest-neighbor communication in an irregular mesh; it can be *static*, e.g., with a fixed finite-element mesh; or *dynamic*, e.g., with Adaptive Mesh Refinement (AMR). 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 can be expressed with shared-memory models using indirection. They are much harder to code efficiently in a message-passing model: It is hard to express the required scatter-gather operations with message-passing; and compilers fail to optimize communication as it is not amenable neither to compile time optimization (that works for regular, static communications), nor to "inspector-executor" approaches (that work for irregular, static communications) [28].

We submit that *a scalable shared memory programming model must handle irregular, dynamic communication patterns*. Codes with such communication patterns are a large and increasing fraction of HPC codes; and these are the codes where the convenience of shared-memory programming is most sorely needed.

We study in this paper the suitability of PGAS languages for expressing such codes. In particular, we look at the impact of various optimizations that reduce communication overheads. This includes optimizations that have been studied before – namely, *split-phase communication*, *message aggregation* and the *casting* of global pointers that refer to local variables into local pointers. 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, the UPC Barnes-Hut code from Berkeley UPC 2.10.2 release [4], which is almost a literal translation from the SPLASH2 benchmark. This code is highly optimized for shared memory, but has no distributed-memory optimizations. We show why a code highly optimized for shared-memory 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. All these optimizations are done manually, and involve increased user control of communication; they are indicative of UPC language enhancements and compiler and run-time optimizations that would be needed to facilitate the expression of codes such as BH in UPC, while achieving good performance.

Most of the literature on compiler optimizations for PGAS languages study simple benchmarks such as MG, CG etc. in NAS Parallel Benchmarks, or simple synthetic test cases [9]. There are relatively few studies of problems with dynamic, irregular communication; see, e.g., [10]. To our knowledge, this is the first paper that thoroughly studies the Barnes-Hut algorithm in a PGAS language and studies the impact of caching.

The rest of this paper is organized as follows: We briefly introduce the UPC language in section 2, focusing on features that are most relevant to our optimizations. Section 3 introduces the Barnes-Hut algorithm and Section **??** describes its implementation in SPLASH2. Section 4 talks about the base line implementation, i.e, the porting of the program to UPC. Section 5, describes a series of optimizations to improve the performance. Section 6 considers algorithmic changes that are needed to achieve scalability to thousands of threads. In section 7 we discuss how these

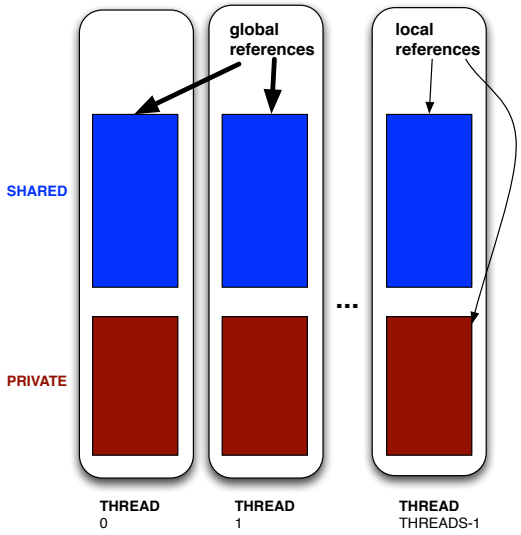


Figure 1: The UPC memory model

optimizations could be eased by better compiler and run-time technology and language enhancements. We survey related work in Section 8 and conclude in Section 9.

2 The UPC language

We briefly introduce in this section the Unified Parallel C (UPC) language. The reader is referred to [25] for a full language description.

UPC is a parallel extension to the C programming language. UPC adopts the same single program multiple data (SPMD) execution model as MPI. 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 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.) Fig.1 shows the UPC memory model.

Global pointers carry more information than local pointers hence dereferencing and pointer arithmetic is more expensive with them. In addition, dereferencing a global pointer to a remote location is much more expensive than dereferencing a global pointer to local memory, because of the need to communicate with another thread, possibly on another node.

Communication is normally effected in UPC by dereferencing global pointers that point to remote locations. UPC also provide functions for explicit, blocking transfer of a block of data from one thread to another (`upc_memget()`,

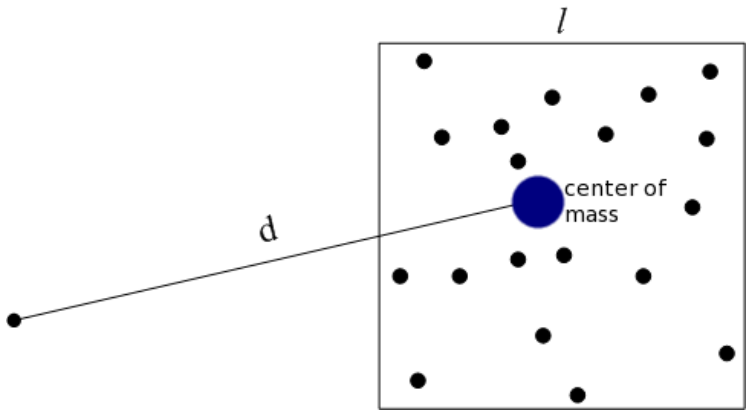


Figure 2: Test for using center of mass

`upc_memput()` and `upc_memcpy()`). Standard UPC (V 1.2) does not have non-blocking communications, but those are supported by some UPC extensions.

3 The Barnes-Hut 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. The most time consuming part is the force computation. A direct method requires computing forces between each pair of bodies, which results in $O(n^2)$ computation complexity. This makes it impractical for simulations involving a very large number of bodies.

Barnes and Hut's observation was that, if a body is far enough from a group of bodies, then we can approximate the force exerted on this body by the bodies in the group by assuming that all bodies in the group are located at the center of mass of the group. "Far enough" is controlled by a parameter θ. As shown in Fig.2, let d be the distance from the body to the center of mass of the group, l be the length of a side of the bounding box of the group. If $l/d < \theta$, then the group in the cell is far enough from the body. (For simplicity, we draw 2D structures; the algorithm uses 3D structures.)

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. Once the tree is computed top-down, the center of mass of each cell can be computed bottom-up. Fig.3 shows a 2D body distribution and the corresponding quad-tree with marks for three bodies a, b, c and their corresponding nodes in the quad-tree.

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 $O(n \log n)$.

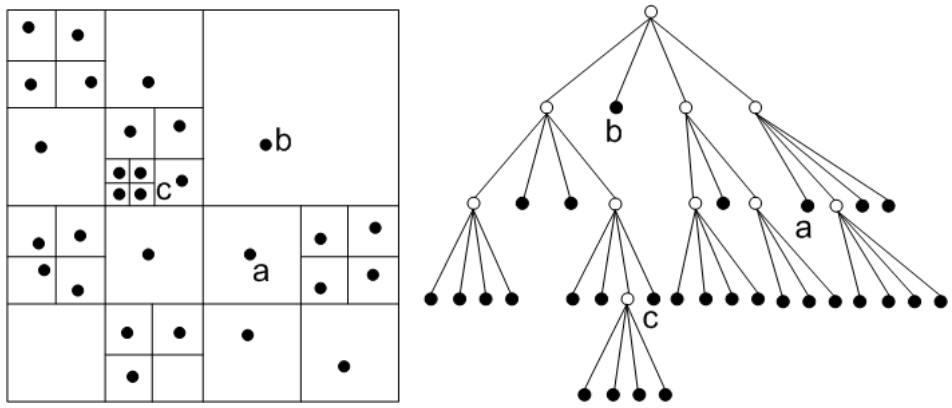


Figure 3: A 2D body distribution and the corresponding quad-tree

4 A baseline UPC implementation

In this section we will describe a UPC implementation of the BH algorithm released in the Berkeley UPC 2.11.4. We use it as a baseline for our further optimizations. The baseline UPC BH is nearly a literal C to UPC translation of the implementation in SPLASH2 that was not coded for performance. Here we focus on shared variable declarations.

Global parameters are declared as shared scalar variables. The dimension of the root cell, `rsize`, is also a shared variable. `bodytab` is a shared pointer-to-shared, pointing to a block-distributed shared memory allocated by thread 0 using `upc_global_alloc()` so that each thread hosts an equal number of bodies in `bodytab[]`. `mybodytab[]` and `mycelltab[]` are local arrays of pointer-to-shared. When a thread creates a new cell during tree-building, it allocates it in its shared memory, and saves its pointer in `mycelltab[]`. The array `subp[]` in the cell struct is now changed to an array of global pointers, pointing to children cells. Because `subp[]` itself is allocated in shared space, these pointers are actually shared pointer-to-shared. Otherwise, the code is almost the same as SPLASH2, except that it uses `upc_lock` and `upc_barrier` for synchronization. Overall it is relatively easy to port the shared-memory BH code to UPC.

4.1 Test environment and approaches

We tested the application on an IBM Power5 cluster with 118 nodes. Each node has 64GB memory and 16 cores running at 1.9GHz. The operating system is IBM AIX V6.1. We used the Berkeley UPC (BUPC) compiler 2.12.0, whose runtime GASNet was built on the IBM LAPI conduit. The back-end compiler is IBM XLC V11.1, with its -O3 option turned on. Except changing the default number of bodies (16K) to larger numbers, we kept the default parameters in SPLASH2: $\theta = 1.0$, and a time-step of 0.025s. As in SPLASH2, we run 4 time-steps and measure time for the last two time steps. The initial body distribution is generated by the Plummer model [1] with M=-4E=G=1, as in SPLASH2.

To take advantage of hardware intra-node shared memory support, BUPC has an option *-pthreads* to build a threaded executable. At runtime the user can specify how many pthreads to spawn per process, with each pthread corresponding to one UPC thread. To simplify the test, in this section and next section, we only use one process per node, with no threading. These two sections focus on weak 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). In section 6, when we study scalability of the application, we will consider strong scaling, with a fixed number of bodies per UPC thread, and

	1		2		4		8		16		32		64		96		112	
	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%
Tree-building	6.0	3.0	285.2	1.3	165.8	0.9	96.1	0.9	53.4	1.0	40.5	1.2	38.9	1.1	38.5	1.2	38.3	1.2
C-of-m Comp.	1.4	0.7	112.1	0.5	69.2	0.4	38.8	0.4	20.6	0.4	11.2	0.3	6.3	0.2	4.6	0.1	4.0	0.1
Partitioning	0.1	0.1	0.1	0.0	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Force Comp.	189.7	95.5	21272.7	96.5	17229.7	97.4	9953.5	97.4	5402.8	97.4	3379.5	97.4	3323.2	97.7	3208.3	97.8	3172.1	97.8
Body-adv.	1.5	0.7	382.3	1.7	224.0	1.3	133.7	1.3	68.2	1.2	38.0	1.1	32.5	1.0	30.5	0.9	29.7	0.9
Total	198.6		22052.4		17688.7		10222.2		5545.0		3469.2		3401.0		3281.8		3244.2	

Table 2: Test results of the baseline UPC BH

run multiple UPC threads per node.

It is worth mentioning that if one uses multiple processes on a node, the performance can be very bad. 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 execution time was 26s. But with -pthreads disabled (16 processes/node), the execution time was more than 36000s. So in this paper, we always enable -pthreads when we want to use multiple cores per node.

4.2 Test results

Table2 shows test results for the baseline UPC BH. It was tested with 2M bodies on 1 to 112 nodes. The table shows the time spent on each phase and its percentage of the total execution time. We can see, compared with one thread, the performance of multiple threads is extremely poor: The code suffers from a significant slow-down, rather than showing speed-ups. Compared with 2 threads, a 32-thread run gets a speedup of about 6.6. Similar results were also reported in [9].

5 Optimizations

We introduce in this section a sequence of optimizations to improve the baseline UPC BH performance. The test environment is the same as in the previous section.

5.1 Replicate shared scalar variables

By specification, UPC shared scalar variables are stored on thread 0. If they are accessed frequently by other threads, it could be beneficial to maintain copies on the other threads. This can be done in at least in three ways: The first is to have runtime maintained caches. The second is to declare local copies that are periodically updated (using a broadcast). The third is to replicate the updates of the shared scalar variables on each thread. The last approach was implemented in the Fortran D compiler[12] and subsequent HPF compilers: It is beneficial if the cost of the redundant computations is much lower than the communication cost and/or if all the other threads would, anyhow, be idle while thread 0 updates the shared variables.

We found the later two approaches can both be used to improve performance. At each time-step, after threads update the positions of their bodies and compute bounding boxes, thread 0 will compute `rsize`, the dimension of the new root cell. In tree-building phase of the next time-step, `rsize` will be repeatedly accessed by every thread

	1		2		4		8		16		32		64		96		112	
	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%
Tree-building	6.1	3.1	160.9	1.4	94.4	0.8	53.0	0.8	28.0	0.7	15.2	0.7	8.5	0.8	6.0	0.8	5.3	0.8
C-of-m Comp.	1.4	0.7	123.6	1.1	68.3	0.6	39.5	0.6	21.0	0.5	11.4	0.6	6.5	0.6	4.7	0.6	4.1	0.6
Partitioning	0.1	0.1	0.1	0.0	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Force Comp.	187.6	95.5	10583.2	94.5	11183.6	97.0	6716.8	97.2	3720.3	97.3	1989.0	97.3	1034.8	97.1	726.1	97.0	658.5	97.1
Body-adv.	1.4	0.7	329.3	2.9	178.2	1.5	100.4	1.5	53.7	1.4	28.2	1.4	15.9	1.5	11.4	1.5	10.1	1.5
Total	196.6		11197.1		11524.5		6909.8		3822.9		2043.8		1065.6		748.2		677.9	

Table 3: Test results with replicating shared scalar variables

when inserting their bodies into the tree. We manually declare a local variable `myrsize` on each thread and copy `rsize` to it.

Some other shared variables, e.g., `tol` (θ) and `eps` (a "potential softening" term) are parameters that are set up at initialization time and do not change afterward, but are accessed frequently. For example, `tol` is used each time to determine whether to open a cell, and `eps` is used in every force computation. The third approach is very appropriate for "write once" variables. We change `tol, eps` etc to be private variables and let every thread parse user's input and initialize those variables.

The test results are shown in Table 3. Compared with the baseline results for 112 threads, the time spent on tree-building, force computation and body-advancing is reduced by 86%, 79% and 66% respectively. The overall compute time is reduced by 79%.

5.2 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 coordinates of each body it "owns" multiple times in every phase of Figure **??**, 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 time-step. 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 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 avoid copying, we use a double-buffer approach. Each thread allocates two buffers in its local shared space and set one of them as the current buffer (`curbuf`). In the body redistribution phase, every thread examines the affinity of each body in their `mybodytab[]`. If it finds remote bodies, it uses an indexed memget, `upc_memget_ilist()` to get all of them, appends them to the end of `curbuf` and replaces remote shared pointers in `mybodytab[]` with pointers to the local copies, accordingly. When `curbuf` fills up, the thread copies all the bodies in `mybodytab[]` to the alternative buffer and switch `curbuf` to it. Given a reasonable buffer size, buffer copying is infrequent.

Table 4 shows the new results. The cost of body distribution is small and performance improves. All phases benefit. In particular, the time spent on center of mass computation and body-advancing is almost totally eliminated. The overall compute time reduction varies from 17% at 2 threads to 4% at 112 threads over the time in Table 3.

	1		2		4		8		16		32		64		96		112	
	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%
Tree-building	4.9	2.6	8.1	0.1	12.4	0.1	8.8	0.1	6.4	0.2	4.5	0.2	3.4	0.3	2.2	0.3	2.2	0.3
C-of-m Comp.	0.8	0.4	0.6	0.0	0.8	0.0	0.6	0.0	0.4	0.0	0.3	0.0	0.3	0.0	0.2	0.0	0.2	0.0
Partitioning	0.1	0.1	0.1	0.0	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Redistribution	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Force Comp.	182.9	96.7	9321.4	99.9	10395.3	99.9	6516.6	99.9	3572.8	99.8	1863.7	99.7	994.1	99.6	699.3	99.6	647.3	99.6
Body-adv.	0.3	0.2	0.2	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Total	189.1		9330.4		10408.6		6526.1		3579.7		1868.6		997.8		701.8		649.8	

Table 4: Test results with body redistribution

The performance gains have several causes. First, we fetch remote bodies only once in a time step instead of several times in different phases (caching). Second, we fetch multiple remote body structures in one remote, coarse-grain operation, instead of having multiple fine-grained accesses to their different fields (aggregation). Third, we can safely do pointer casting to avoid the overhead of global pointer dereferencing (casting).

We simulated four time-steps, and no buffer filled up. Additional experiments showed that the overhead to handle buffer copying is very small since copying is local and occurs infrequently.

5.3 Cache remote nodes

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. In this section we shall develop two methods to take advantage of this. The first builds a separate local tree to cache all accessed cells. The other uses *shadow pointers* so as to cache only remote cells.

5.3.1 Cache using a separate local tree

Cells in the octree are referenced using UPC pointers to shared. The cells can be cached locally by creating a local copy and swizzling pointers to point to the local copy. A flag `Localized` is added to each cell struct. `n.Localized==TRUE`, if all children of cell `n` are stored on the same thread as `n`. Initially, all cells have this flag set to `FALSE`. Before force computation, every thread makes a local copy, `L_root`, of the global root `G_root`. During force computation, every thread traverses the tree starting from its private `L_root`. Force computations are done using local copies of the needed cells. If a thread needs to open a cell `n`, it tests if `n.Localized==TRUE`. If the test succeeds then the children are already cached locally. Otherwise, it first makes local copies of all children of n, replaces pointers in `n.subp[]` with ones to those local copies, sets `n.Localized=TRUE`, and then resumes the work.

The pseudo code is shown in Listing 1[1]. For each body `b`, the forces on `b` are computed by a call to `compute_force(b, L_root)`.

Listing 1: Using local trees

```
1 compute_force(shared_bodyptr_t b, local_cellptr_t n)
2 {
```

[1]We omitted some details such as handling leaf nodes

	1		2		4		8		16		32		64		96		112	
	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%
Tree-building	5.0	3.5	8.1	7.2	12.1	18.1	9.6	23.7	6.0	27.9	4.3	31.6	3.3	26.9	2.3	20.9	2.1	19.1
C-of-m Comp.	0.8	0.6	0.6	0.5	0.7	1.1	0.6	1.5	0.4	2.0	0.3	2.5	0.3	2.2	0.2	1.9	0.2	1.9
Partitioning	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.0	0.2	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3
Redistribution	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1
Force Comp.	136.4	95.6	103.9	92.0	54.1	80.6	30.2	74.5	15.1	69.7	8.9	65.4	8.7	70.4	8.5	76.6	8.5	78.4
Body-adv.	0.3	0.2	0.2	0.1	0.1	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.2	0.0	0.2
Total	142.6		112.9		67.2		40.6		21.7		13.6		12.4		11.1		10.8	

Table 5: Test results with caching by a separate local tree

```
3    if (need_open(b, n)) {
4      if (n->Localized) {
5        for (int i = 0; i < 8; i++) {
6          local_cellptr_t ch = n->subp[i];
7          if (ch != NULL) compute_force(b, ch);
8        }
9      } else {
10       for (int i = 0; i < 8; i++) {
11         shared_cellptr_t ch = n->subp[i];
12         if (ch != NULL) {
13           shared_cellptr_t p = upc_alloc(sizeof(cell));
14           *p = *ch;
15           n->subp[i] = p;
16         }
17       }
18       n->Localized = TRUE;
19       // Same as line 5~8
20     }
21   } else { Compute force between b and n }
22 }
```

There are a few things worth mentioning. `local_cellptr_t` is a normal C pointer type: The cell argument `n` that is passed to `compute_force` in the recursive invocations is always cached locally, so that the access can use a cheaper regular C pointer. At line 6, `n->subp[i]` is a pointer-to-shared, but we can safely cast it to a local pointer because `n->Localized` is `TRUE`. At line 14 `p` is set to point to a newly allocated local cell copy, but has pointer-to-shared type, to avoid an illegal cast on line 15.

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 [21]. The difference is that our local copy is built, demand-driven, during force computation, instead of being built up-front before force computation.

Table 5 shows the performance results. Compared to Table 4, the time spent on force computation is dramatically reduced by 99% (!) for all multithreaded runs. And interestingly, even for one thread, the time is also reduced by 25%, 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.

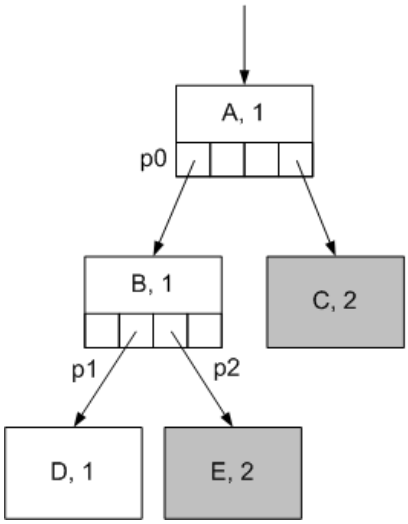


Figure 4: A distributed octree

5.3.2 Cache using a merged local tree

In previous subsection, every thread constructs a separate local tree, which means that nodes of the global tree are cached even if they already are in shared memory local to the current thread. Performance could be improved by avoiding this superfluous copies. Figure 4 shows why a naive implementation of this idea may not perform well. There are five cells `A...E`, which are distributed on two threads, `T1` and `T2`. The notation "`A,1`" means cell `A` is on `T1`. When `T1` processes `A`, it finds that `B` is in its memory so it keeps `pointer p0` unchanged. In the same way, when `T1` processes `B`, it keeps pointer `p1` unchanged, but caches `E` and replaces `p2` with a new pointer. If `T2` reaches `B` after `T1`, then the information of the old pointer `p2` was lost and `T2` will copy `E` from a remote thread, rather than using its local copy.

The problem can be avoided by storing at each cell copy two sets of pointers to children: one is the original set of pointers and the other a set `shadowp[]` of s*hadow pointers* that point to the local copies of the children, or to the original copies, if they already are local. The resulting algorithm as shown in Listing 2. Note that at lines 13–19, we test the affinity of pointers and only cache remote cells. For remote cells, we avoid copying private data, such as the shadow pointers, and the value of `Localized`.

Test results (not shown here) showed little performance improvement over Table 5: The improved algorithm saves some local copying but does not affect global communication and increases the size of cell structures.

Listing 2: Avoiding the caching of local cells

```
1 compute_force(shared_bodyptr_t b, local_cellptr_t n)
2 {
3  if (need_open(b, n)) {
4   if (n->Localized) {
5    for (int i = 0; i < 8; i++) {
6     local_cellptr_t ch = n->shadowp[i];
```

```
7       if (ch != NULL) compute_force(b, ch);
8      }
9     } else {
10     for (int i = 0; i < 8; i++) {
11      shared_cellptr_t ch = n->subp[i];
12      if (ch != NULL) {
13       if (upc_threadof(ch) == MYTHREAD) {
14        n->shadowp[i] = ch;
15       } else {
16        shared_cellptr_t p = upc_alloc(sizeof(cell));
17        // Copy shared portion of ch to p,
18        // notably, without localized, shadowp[]
19        n->subp[i] = p;
20       }
21      }
22     }
23     n->Localized = TRUE;
24    }
25   } else { Compute force between b and n }
26  }
```

5.4 Octree building

In the SPLASH2 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 use a lock to protect the operation. The lock contention 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 11% [23]. In UPC, the overhead is larger, due to the higher cost of global locks and remote accesses. As shown in Table 5, tree-building may consume almost 19% of the total computation time with 112 threads.

Singh suggested a new tree-building algorithm in his thesis [23]. 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 merging.

We implemented this algorithm in UPC. We used pointers to shared when building the local trees because these nodes and pointers will be merged into the global tree. In local tree-building, we can safely cast those global pointers to local pointers, because we know they are pointing to local memory.

This algorithm greatly reduced tree-building time. For example, for 112 threads, the time spent on tree-building was reduced by 83% to 0.37s. Unfortunately, it also increased the center of mass computation. For 112 threads, the time spent on center of mass computation was doubled, to 0.42s.

The center of mass computations for the octree are done in parallel. Each cell has a flag *done* to indicate if the center of mass of this cell is valid. 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.

	1		2		4		8		16		32		64		96		112	
	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%
Tree-building	1.9	1.4	2.1	1.9	2.9	5.1	2.1	6.8	1.7	9.8	1.0	10.2	0.7	7.9	0.7	7.0	0.6	6.4
Partitioning	0.1	0.1	0.1	0.1	0.1	0.2	0.1	0.2	0.1	0.4	0.1	0.6	0.0	0.5	0.0	0.4	0.0	0.4
Redistribution	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1
Force Comp.	136.6	98.3	104.7	97.8	54.1	94.5	28.8	92.8	15.1	89.5	8.9	89.0	8.7	91.4	8.5	92.2	8.5	92.8
Body-adv.	0.3	0.2	0.2	0.1	0.1	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.2	0.0	0.2	0.0	0.2
Total	138.9		107.0		57.2		31.1		16.8		10.0		9.5		9.3		9.2	

Table 6: Test results with new tree-building algorithm

In order to solve this problem and reduce communication, we use an alternative approach. Each thread first compute a local center of 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 $(l.mass \cdot l.\vec{cofm} + g.mass \cdot g.\vec{cofm})/(l.mass + g.mass)$. 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.

Table 6 shows the test results. For 112 threads, compared with Table 5, the time spent on tree-building and center of mass computation was reduced by 74%. And the overall computation time was reduced by 15%. The speedup with 112 threads is about 15.

5.5 Non-blocking communication and message aggregation

Caching remote nodes 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 nodes in one big message instead of multiple small messages.

The BH application has two sources of asynchronism to support non-blocking communication. One is that force computation for one body is independent to that for another body. Threads can compute forces for their bodies 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 nodes of an octree in any order, provided that a parent is visited before any of its children.

We have designed a framework which implements both optimizations. It is similar to the latency hiding approach described in [26]. 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 needed to compute forces for this body. Initially, the only frontier cell for a body is `L_root`, the local copy of the global root. To compute forces for a working body, a thread loops through the body's frontier cells. For each frontier cell `n`, if the cell is far enough, then it is used to computer a force; otherwise if all children of `n` are cached locally (`n.Localized == TRUE`), then the thread deletes this frontier cell and proceeds with the computation with its children recursively. Otherwise, (`n.Localized == FALSE`), then the thread adds the children of `n` to a list of needed remote nodes and marks them as requested (to avoid requesting the same cell more than once). Once all available frontier cells 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. Note that, for simplicity, we bring all children of a cell in the same communication, so that a communication may handle between n_3 and $n_3 + 7$ nodes.

We use the BUPC extension function `bupc_memget_vlist_async()` to bring in nodes. It is a non-blocking gather operation and can get data from multiple remote threads. It returns a handle. The handle can by used in `bupc_trysync()` or `bupc_waitsync()`. The former is a non-blocking test, while the latter is a blocking wait. When the thread cannot make progress, it tests outstanding requests. If a request is completed, the `localized` flag of parents of children in this request are set. The workflow of this framework is shown in Listing 3.

Listing 3: Non-blocking communication and aggregation

```
while (have bodies uncomputed) {
  Fill up the list of working bodies;

  Compute force for working bodies until can't make progress;
  Aggregate msgs and send out a request if it is long enough;
  Delete old frontier cells and add new ones if any;
  Delete a working body if its work is done;

  waitsync/trysync each outstanding request;
  Set localized flag of parents of children in requests;
}
```

Table 7 shows the new results with $n_1 = n_2 = n_3 = 4$. There are a few interesting things to note. First, although there are many possible choices for n_1, n_2 and n_3, we found that the results are not very sensitive to that choice, and 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 `upc_waitsync()` instead of `upc_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. Instead, we take advantage of `bupc_memget_vlist_async()` to fetch data from multiple sources. On one hand, it is more easy to code. On the other hand, it turns out that cells in a request have very good locality. In our testing, with 32 threads, more than 95% of the requests have only one source thread, and about 4% of the requests have two source threads. With 64 threads, the numbers are 93% and 6% respectively.

As the number of threads increases in strong scaling, the number of remote cells a thread will access will increase too. As a result, the performance gain of the optimizations becomes more significant. For example, with 32 threads, the time for force computation is reduced by 40%, and with 112 threads, it is reduced by 81%. For 112 threads, total compute time is reduced by 75%, as compared with Table 6. The speedup on 112 threads is more than 70.

The results of the successive optimizations are summarized in Figure 5 that shows (on a log scale) the speedups achieved by the cumulative application of the successive optimizations (the "subspace" optimization is discussed in the next session). The speedup, with 112 processes is 81.4. Figure 6shows (on a log scale) the time consumed by each phase, for runs with 112 processes. With all optimizations applied, force computation consumes 82.4% of the total computation time.

	1		2		4		8		16		32		64		96		112	
	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%
Tree-building	1.9	1.2	2.0	2.4	3.0	6.7	2.5	10.7	1.7	13.6	1.0	15.8	0.7	20.8	0.6	25.0	0.6	25.9
Partitioning	0.1	0.1	0.1	0.1	0.1	0.3	0.1	0.3	0.1	0.6	0.1	0.9	0.0	1.3	0.0	1.5	0.0	1.5
Redistribution	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.1	0.0	0.2	0.0	0.3	0.0	0.5	0.0	0.5
Force Comp.	159.4	98.6	80.3	97.2	40.7	92.8	20.6	88.7	10.4	85.6	5.3	82.9	2.8	77.2	1.9	72.2	1.6	71.2
Body-adv.	0.3	0.2	0.2	0.2	0.1	0.2	0.0	0.2	0.0	0.2	0.0	0.2	0.0	0.4	0.0	0.7	0.0	0.9
Total	161.8		82.6		43.9		23.2		12.2		6.4		3.6		2.6		2.3	

Table 7: Test results with non-block communication and message aggregation

Figure 5: Speed-up for Cummulative Optimizations

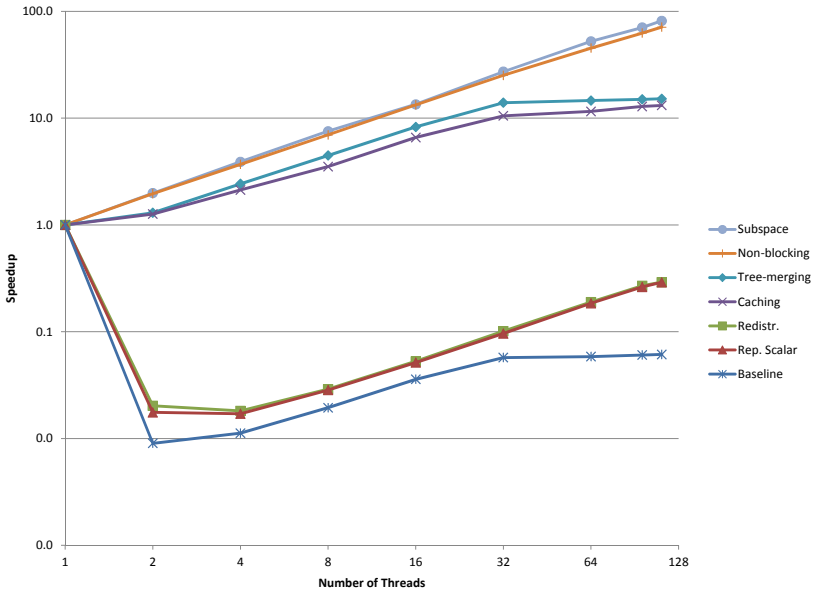
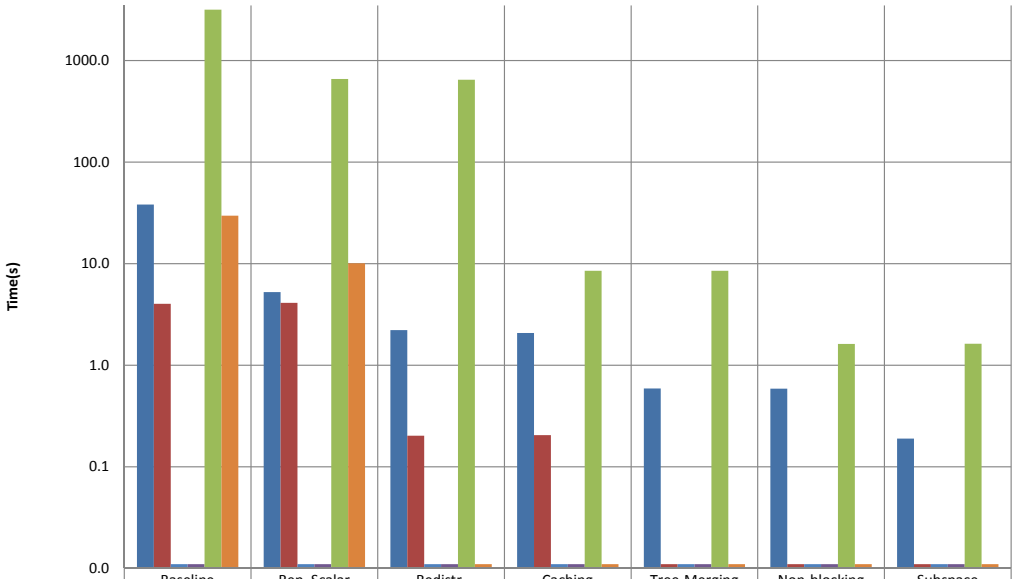


Figure 6: Time for each phase at 112 processes



	Baseline	Rep. Scalar	Redistr.	Caching	Tree-Merging	Non-blocking	Subspace
Tree-building	38.3	5.3	2.2	2.1	0.6	0.6	0.2
C-of-m Comp.	4.0	4.1	0.2	0.2	0.0	0.0	0.0
Partitioning	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Redistribution	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Force Comp.	3172.1	658.5	647.3	8.5	8.5	1.6	1.6
Body-adv.	29.7	10.1	0.0	0.0	0.0	0.0	0.0

Optimizations

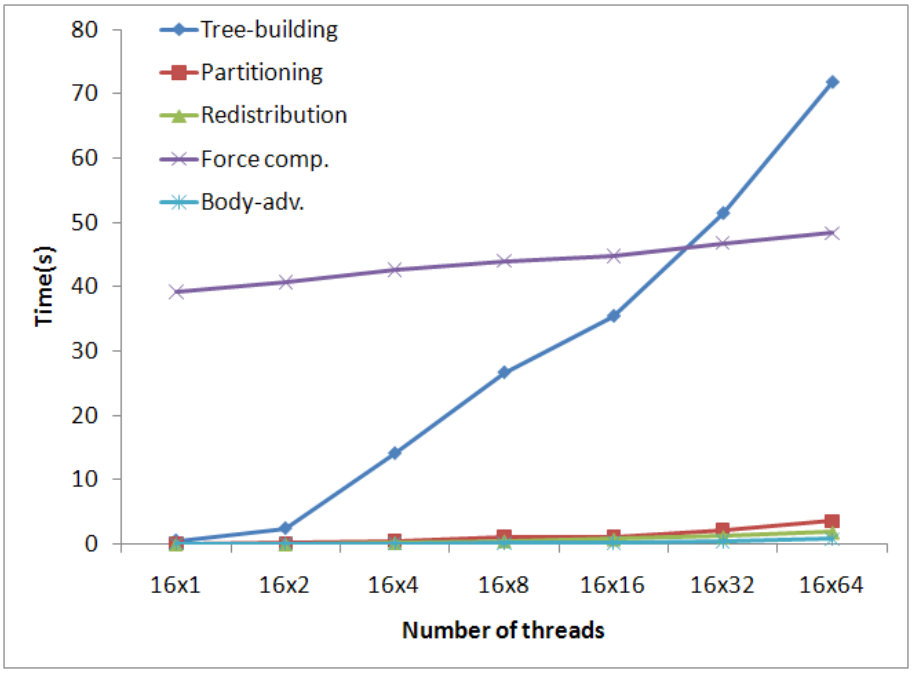


Figure 7: Weak scaling with 250K bodies/thread

6 Scalability

In this section we will study the scalability of the program, using weak scaling. A new bottleneck arises, that requires an algorithm change. We then study weak scaling of the updated code.

6.1 Weak scaling

Fig. 7 shows performance of our code under weak scaling with 250K bodies/thread. We tested it with 16 threads per node and up to 64 nodes (1024 threads). We see that except for tree-building, all other phases scale well. Tree building becomes the most expensive phase above 512 threads. To understands the reason, we studied the time each thread spends in this phase. The phase consists of two sub-phases: local tree building and global tree merging. Fig. 8 shows the time spent on each sub-phase by each thread in a 16x8 thread run. We can see that local tree-building is well-balanced and has a low cost (<0.5s). However, tree-merging is not. Tree merging time varies from 0s to 26s, which is the root of the poor scaling of tree-building.

The reason for this unbalance is the following: when two threads compete to merge their local trees, the winner pays only a pointer redirection to insert a subtree. The loser afterward has to traverse the subtree inserted by the winner to find correct locations to insert its nodes. This tree traversal is a step-by-step remote operation, which can

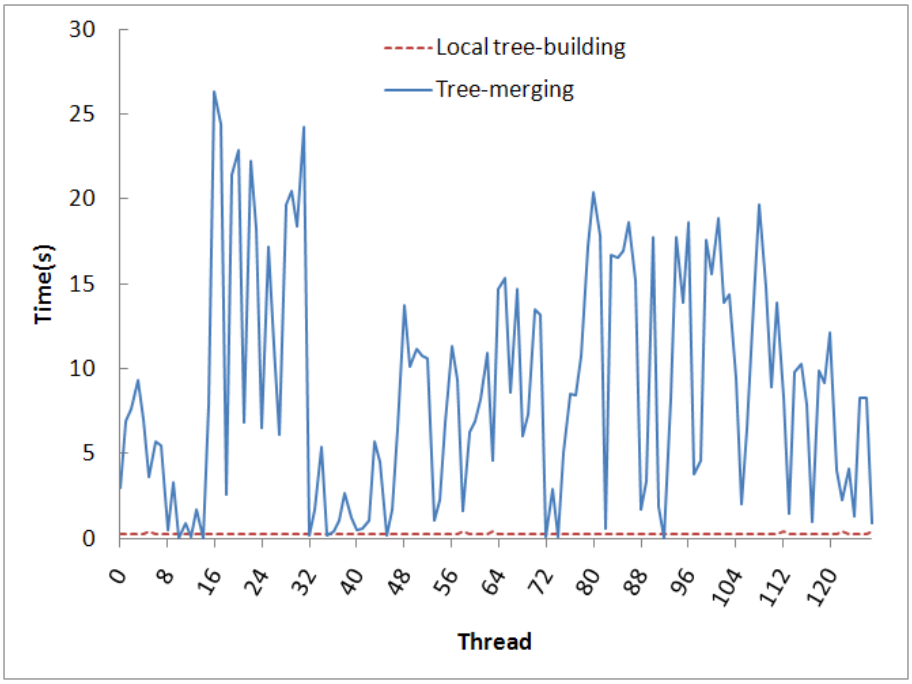
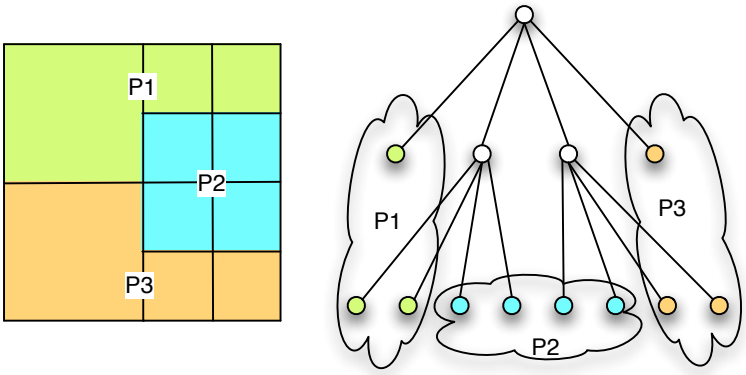


Figure 8: Time distribution of tree-building, total 128 threads, 250K bodies/thread

Figure 9: Modified octree building algorithm



be very costly. If two threads repeatedly conflict on tree merges, then one thread is likely to be repeatedly delayed and fall behind the other.

To alleviate this problem, we seek a merge algorithm that avoids the large number of fine grain merges our current algorithm suffers from – at the possible expense of increasing the work for local tree building. The algorithm is based on an algorithm proposed by Shan and Singh[22]; they studied it for shared memory on up to 30 processors. To achieve scalability, we need some key changes in their algorithm. With these changes, we could scale to 1000's of threads.

The basic idea is for the threads to exchange enough information so as to know what will be the shape of the merged octree. Each thread can then locally build a subforest containing their bodies that has the same structure as the merged tree. The merge process is now much simplified, since it does not require splitting cells, but only requires updating pointers; the updates do not conflict and require no locking. The global octree will be built so that each thread "owns" a set of consecutive leaves of approximately equal total cost. The only global information needed is the cost of subtrees in the octree. The algorithm is illustrated in Figure 9(we show a binary tree, rather than an octree, to simplify graphics). The leaves are divided across 3 threads; the thick edges indicate where global insertions are needed.

To ensure that leaves can be be evenly allocated to threads, we divide a cell if its total cost is larger than a threshold $\tau = \alpha Cost/THREADS$, where $Cost$ is the total cost of all bodies and $THREADS$ is the number of threads. This ensures that leach thread will be allocated a set of leaves with a total cost of at most $(1+\alpha)Cost/THREADS$. (The actual bound is much better, in practice, since it is highly unlikely that a large fraction of the cost of a cell is carried by only one of its 8 children.) We use, in our experiments, $\alpha = 2/3$.

Each thread has two arrays, `subspaces[]` and `costs[]`, which are allocated in the thread's shared memory. Element of `subpsaces[]` are a local representation of a cell in the octree. Among other things, a subspace entry has a list of bodies which are currently owned by this thread and are lying in this subspace. Each subspace has a cost, which equals to the sum of the costs of all bodies in this subspace. We store costs at each thread in `costs[]`. In addition, thread 0 is building a regular octree representation for the shared tree.

`subspaces[]` and `costs[]` are computed recursively at each thread. `subspaces[0]` is the root subspace, which represents the root cell. A thread builds `subspaces[0]` by putting all its bodies into the root subspace and stores

the sum of their costs in `costs[0]`. We then perform a global reduce&broadcast operation to sum the local costs and broadcast the result. All threads now hold in `costs[0]` the cost of the root cell. Now each thread can decide whether the root subspace needs to be divided. If so, each thread creates 8 children subspaces and split the local root bodies among the children.

This procedure is repeated, at each subsequent level, for all cells that have a cost that is $> \tau$, until a level is reached with no "fat" cells. The only communication needed is the reduce&broadcast of cell costs at each level. We use a collective vector reduction to compute global costs for all nodes at a level in one communication.

Each thread can now compute which leaves belong to any other thread. An all-to-all communication is used to distribute bodies to their owners. After that, each thread builds a local sub-forest consisting of all cells containing only leaves allocated to the thread. Once the subforest is built and the center of mass of each node is computed, a thread can directly hook the subtree to Thread 0's octree. Because leaf subspaces are not overlapped, this operation doesn't need any lock and is very fast. After all threads have hooked their subtrees, Thread 0 computes centers of mass for the cells not owned by any thread; this is also very fast, since the number of nodes in the top, shared part of the tree is typically linear in the number of threads (and always $O(p \log p)$, where p is the number of threads).

The difference between our algorithm and the original one of Shan and Singh's algorithm is that we use costs, rather than number of bodies to decide whether to divide a cell, so that the octree structure matches the body partition; and we handle one level at a time, rather than one subspace 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. Keeping costs in a separate array facilitates the use of vector reduction.

Fig. 10 and Fig. 11 shows the results without and with vector-reduction respectively. Compared to Fig. 7, performance of tree-building is greatly improved. We can also see that without the 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 reduction operations. The total number of levels in the tree in this case is 9, so that we do, instead, 9 vector reductions.

We tested the program with 1, 4 or 8 threads per node on 1 to 112 nodes. Fig. 12 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. (Each node has 16 cores, so that we do not expect per core performance to decrease as more threads are put on each node.)

Fig. 12 also has results for "1 process/node" with -pthreads disabled. Disabling pthreads improve performance by about 50%. as compared to "1 thread/node", with pthreads enabled. This would seem to indicate performance problems in the run-time we have been using, possbily related to the interaction of GASNet with pthreads.

6.2 Strong scaling

With the subspace tree-building algorithm introduced above, Table 8 and Table 9 give performance results with 1 process per node and 1 thread per node respectively. We can observe similar phenomena as in Fig.12: process is better than thread, from 50% with 1 node, to 40% with 112 nodes. If we compare the results Table 8 with the baseline implementation, with all optimizations introduced in this paper, we improved the performance from 272 times with 2 nodes, to 1644 times with 112 nodes.

Fig.13 gives the strong scaling speedup curve. 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 at 512 threads, where each thread has about 4K bodies.

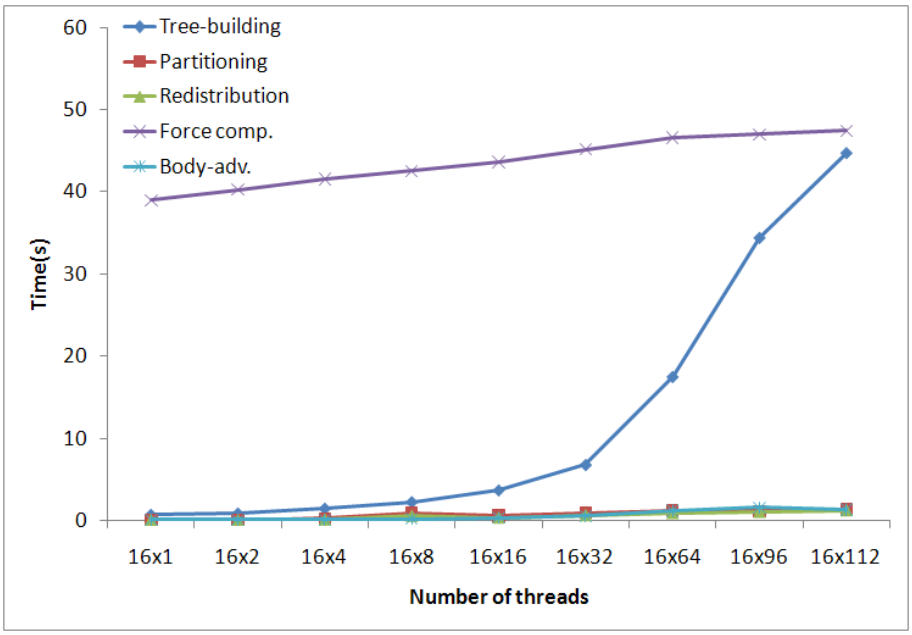


Figure 10: Weak scaling without vector-reduction, 250K bodies/thread

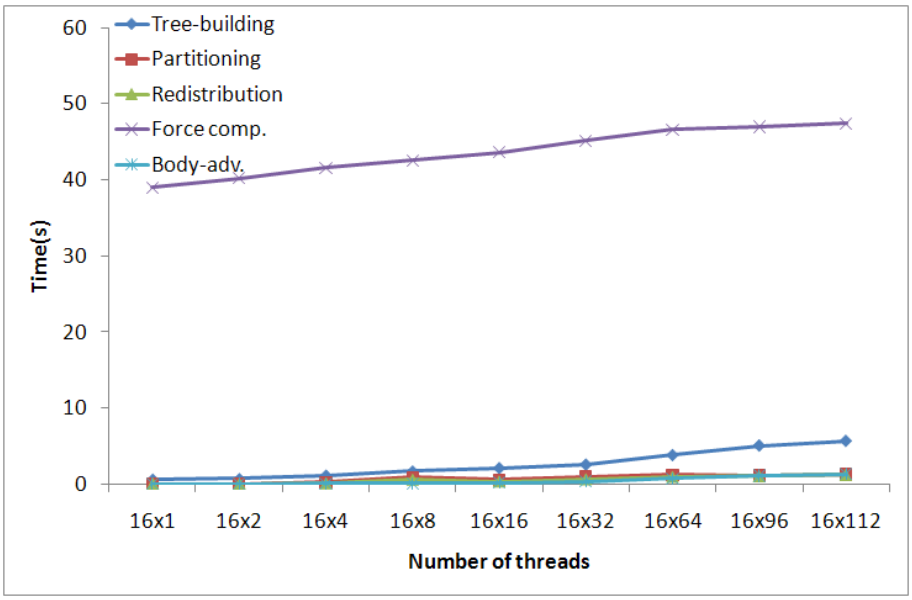


Figure 11: Weak scaling with vector-reduction, 250K bodies/thread

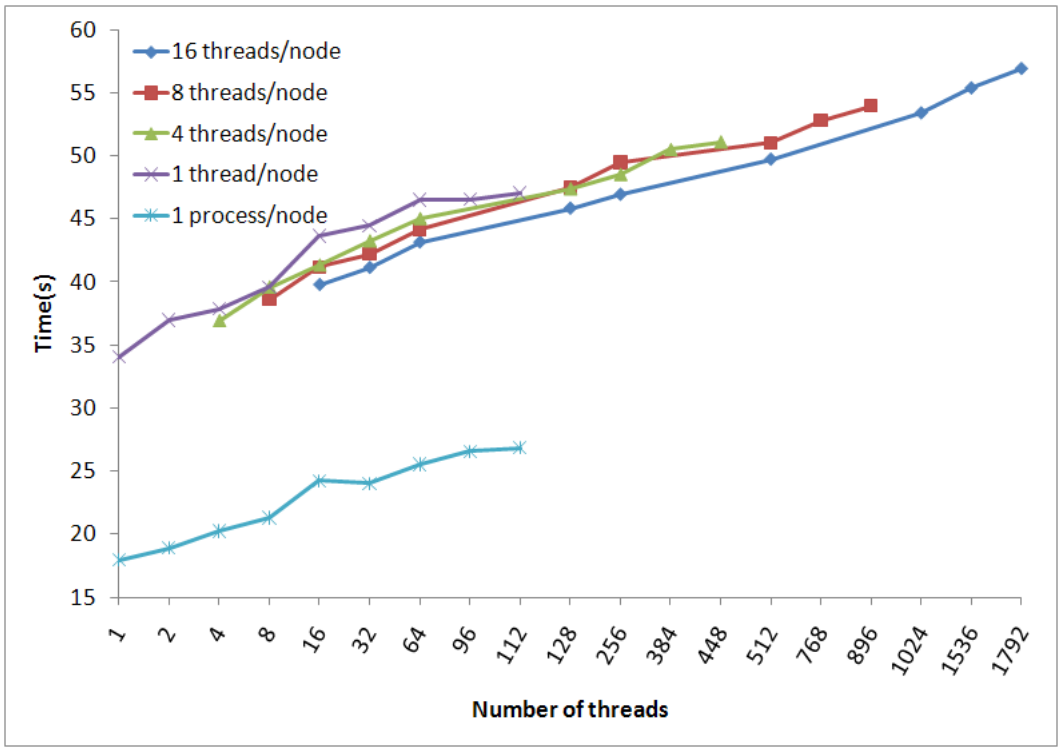


Figure 12: Weak scaling by varying threads per-node, 250K bodies/thread

	1		2		4		8		16		32		64		96		112	
	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%
Tree-building	2.0	1.2	1.1	1.4	0.6	1.5	0.4	2.0	0.4	3.5	0.2	3.8	0.2	5.8	0.2	8.0	0.2	9.6
Partitioning	0.1	0.1	0.1	0.2	0.1	0.2	0.3	1.3	0.6	4.8	0.2	3.9	0.1	3.4	0.1	4.9	0.1	5.0
Redistribution	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.6	0.2	2.0	0.1	1.6	0.0	1.4	0.0	2.1	0.0	2.0
Force Comp.	158.2	98.5	79.5	98.2	40.4	97.9	20.5	95.9	10.7	89.5	5.3	90.5	2.7	88.8	1.9	84.2	1.6	82.4
Body-adv.	0.3	0.2	0.2	0.2	0.1	0.2	0.0	0.2	0.0	0.2	0.0	0.3	0.0	0.5	0.0	0.8	0.0	1.1
Total	160.7		80.9		41.2		21.3		11.9		5.9		3.1		2.3		2.0	

Table 8: Strong scaling with 2M bodies, 1 process/node

	1		2		4		8		16		32		64		96		112	
	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%	t(s)	%
Tree-building	2.9	0.9	1.7	1.1	1.0	1.2	0.6	1.5	0.5	2.6	0.3	2.7	0.2	3.9	0.2	5.3	0.2	6.4
Partitioning	0.2	0.1	0.2	0.1	0.1	0.2	0.3	0.7	0.6	2.8	0.2	2.1	0.1	1.9	0.1	3.0	0.1	3.0
Redistribution	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.2	1.1	0.1	0.8	0.0	0.8	0.0	1.3	0.0	1.2
Force Comp.	309.2	98.9	154.1	98.7	77.8	98.4	39.5	97.4	19.8	93.4	10.0	94.1	5.1	93.1	3.4	89.9	2.9	88.7
Body-adv.	0.3	0.1	0.2	0.1	0.1	0.1	0.1	0.1	0.0	0.1	0.0	0.2	0.0	0.3	0.0	0.6	0.0	0.7
Total	312.6		156.1		79.1		40.5		21.2		10.6		5.5		3.8		3.3	

Table 9: Strong scaling with 2M bodies, 1 thread/node

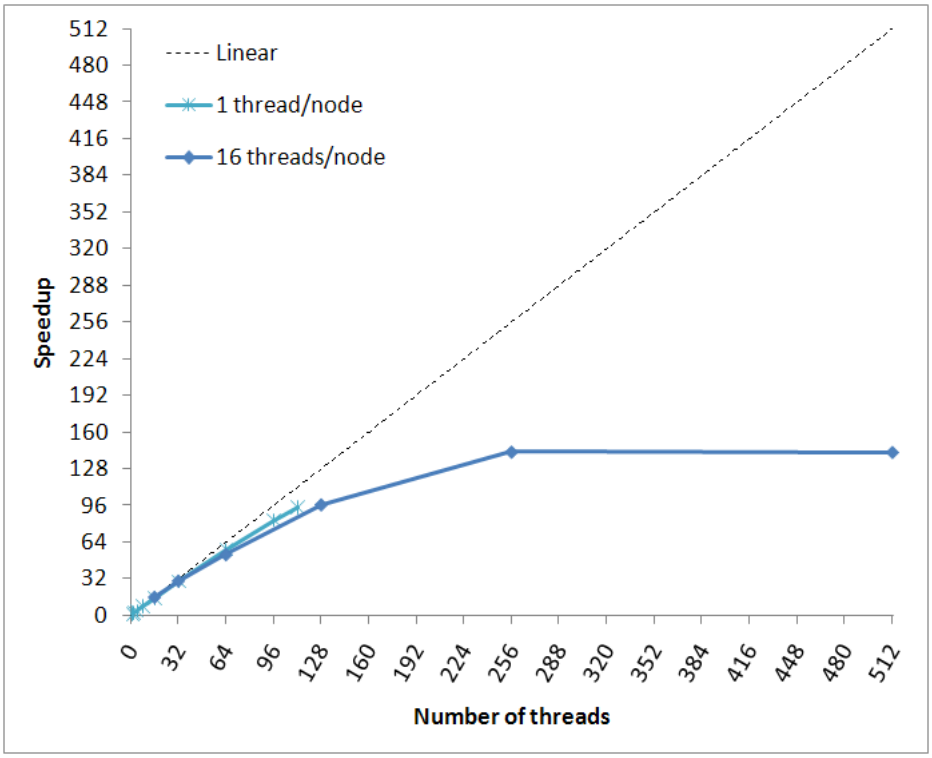


Figure 13: Strong scaling with 2M bodies

7 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, for 64 threads, by a factor of 854 and, for 112 threads, by a factor of 1644. The original code was designed for shared memory performance, and very different optimizations were needed to achieve performance in UPC.

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 5 and 6, in one of three categories:

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

Many of the optimizations done manually in Section 5 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:

1. 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. Furthermore, these objects are accessed via pointer dereferencing; therefore, caching can be handled in software via pointer swizzling.
2. 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 octree construction. 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.
3. 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 as 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 5 and 6.

Replicate write-once scalars [1]: The first transformation described in Section 5.1 consists of replicating write-once shared variables (`tol` and `eps`) at each thread. This is likely to be an optimization that can be automated: Alias analysis is often quite accurate for scalars (especially in languages that restrict aliasing) and codes contain relatively few scalars.

Replicate write-rarely scalars [2]: The second transformation in that section replicates a "write-rarely" variable – `rsize`, the size of the root cell. The variable is replicated whenever it is updated. The transformation

can be done automatically, but it may not be obvious to a static compiler that the transformation improves performance. This depends on the ratio of reads to writes; profile-driven recompilation may be needed to figure this out. It will be easy to effect this transformation if the programmer annotates the code specifying that `rsize` should be replicated. Automating the replication is easy, as `rsize` is a scalar variable. It would be harder if the code did not copy the size of the root cell into a scalar variable, as fields we want to replicate (the size of the root cell) are syntactically indistinguishable from fields we do not want to replicate (the size of a leaf cell). It is reasonable to ask programmers to rewrite their codes so that values to be replicated are syntactically distinct.

Redistribute by ownership [2]: The transformation discussed in Section 5.2 consists of redistributing shared data (bodies) between iterations. Each thread has a list of bodies it will update and the bodies are redistributed accordingly. While the gains from this transformation are moderate, it is key to the gains achieved in the next set of transformations, since it enhances cell reuse in body computation. It seems hard to fully automate this transformation. It will be impossible for a compiler to figure out the right object distribution (that depends on the spatial distribution of the bodies). On the other hand, if the distribution is computed by the user, then it will be hard for a compiler and run-time to analyze what object is updated by each thread and figure out that each object is updated by only one thread. The programmer can facilitate this task by specifying that ownership is exclusive, providing the list of objects owned by each thread, specifying what is the right time for redistribution, and providing hints to help choose the redistribution algorithm; e.g., in our case, that only a small fraction of the data is expected to move. The programmer can also indicate that all accesses will be local. The run-time can move data and swizzle pointers.

Cache Read-Only Data on demand [2]: The transformation that provides most performance improvements is described in Section 5.3: 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 phase, 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]: The next transformation, in Section 6, 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. This transformation is harder to automate for more general updates, when it is not clear that the updates commute. In many cases, a different order of updates may lead to a different result, and it requires deep algorithmic understanding to know that all such results are valid. For example, suppose that, instead of dividing cells into eight sub-cells, we would always divide a cell into two sub-cells, along in a dimension that separates the bodies. Then different execution orders would result into different trees, all valid. 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 5.4 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]: The transformation described in Section 5.5 has a significant effect in performance: It uses the availability of "parallel slack" in the code executed by each thread in order to hide latency and also 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 iterates over all cells that are not "far enough" from the body. The iterations over bodies can be executed in any order, while the iterations

over cells can be executed in any order, provided that a parent is visited before its children. We use this available parallelism to perform "concurrent multithreading": 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 multithreading efficiently. However, a compiler may not be able to identify the available parallelism. The user can indicate that parallelism is available by using a parallel `foreach` iterator, similar to the one used in [17]. (However, in our case, we need to iterate over a partially ordered domain, rather than over an unordered domain with mutual exclusion relations, as in [17]).

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 [14] could be used for UPC, in order to achieve the same effect.

High-level alignment [3]: The last transformation described in Section 6 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 merges of the local fragments become much more efficient.

8 Related works

Graph algorithms usually have dynamic, irregular computation and communication patterns, hence present the same problem as Barnes Hut. The study by Cong et al. [10] 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 Barnes-Hut algorithm in UPC is pursued by Dinan et al. [11]: 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 $\times 50$ 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 asynchronous communication, were first proposed for the BH algorithm by Warren et al. [26]. 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 strcuture for accessing cells, one can use Morton codes as hashing keys. It is interesting to speculate whether such data-dependent storage order and dynamic partitions could be accommodated by extending PGAS shared array distributions.

The MuPC implementation of UPC supports software caching for scalar variables; variables are written back at each synchronization point, to avoid coherence issues [30]. A similar approach has been used to add caching to Berkeley UPC [8]. We could not find an evaluation of the performance 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, and the difficulty of choosing the right caching unit.

9 Conclusion and further work

We analyzed the performance of a UPC implementation of the Barnes Hut 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. We suspect that, with all these changes, the UPC code is as efficient as a similar MPI code. We plan, in future work, to directly compare the performance of this code to the performance of a similar code expressed in MPI.

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. The analysis presented in Section 7 attempts to contribute to this understanding. Our conclusion is that many of the transformation needed to enable caching can be done using existing compiler and run-time technologies, but require help from the user, in the form of information added to the shared memory code. This information does not change the program semantics, but just controls the execution plan. Therefore, it could be added in the form of pragmas, or via source code chanages that can be semi-automated using a suitable refactoring environment. We plan to study such extensions in follow-up research.

The final Barnes Hut code we d;veloped is quite similar to an MPI code implementing the same algorithm. One can question whether the use of UPC has simplified the coding task. We believe this is still the case, for two reasons: (a) The availability of a global address space means that the initial code can be a simpler, shared-memory code; an efficient code can be developed by successive refinements of this initial shared-memory code. (b) The availability of a global address space simplifies communications, as one can use one-sided communication (get needed data, without having it sent), and one has a simpler naming scheme for shared data.

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 Barnes Hut – 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 Barnes Hut, they will suffer the fate of High Performance Fortran [16]: Users will be turned of by PGAS languages before these languages become ready for prime time.

References

[1] Aarseth, S., Hénon, M., and Wielen, R. A comparison of numerical methods for the study of star cluster dynamics. *Astronomy and Astrophysics 37* (1974), 183–187.

[2] Barnes, J., and Hut, P. A hierarchical O(NlogN) force-calculation algorithm. *Nature 324* (1986), 446–449.

[3] Barriuso, R., and Knies, A. SHMEM user's guide for C. Tech. rep., Cray Research Inc, 1994.

[4] Berkeley UPC. `http://upc.lbl.gov`.

[5] Callahan, D., Chamberlain, B., and Zima, H. The cascade high productivity language. In *Proceedings of the Ninth International Workshop onHigh-Level Parallel Programming Models and Supportive Environments* (2004), pp. 52–60.

[6] Cantonnet, F., Yao, Y., Zahran, M., and El-Ghazawi, T. Productivity analysis of the UPC language. In *Proceedings 18th International Parallel and Distributed Processing Symposium.* (2004), pp. 254–260.

[7] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2005), pp. 519–538.

[8] Chen, W., Duell, J., and Su, J. A software caching system for UPC. `http://crd.lbl.gov/~jcduell/cs265/project/UPC-cache.pdf`, 2003.

[9] Chen, W., Iancu, C., and Yelick, K. Communication optimizations for fine-grained UPC applications. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)* (2005), pp. 267–278.

[10] Cong, G., Almasi, G., and Saraswat, V. Fast PGAS Implementation of Distributed Graph Algorithms. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), pp. 1–11.

[11] Dinan, J., Balaji, P., Lusk, E., Sadayappan, P., and Thakur, R. Hybrid parallel programming with MPI and unified parallel C. In *Proceedings of the 7th ACM international conference on Computing frontiers* (2010), pp. 177–186.

[12] Hiranandani, S., Kennedy, K., and Tseng, C. Compiler support for machine-independent parallel programming in Fortran D. In *Languages, compilers and run-time environments for distributed memory machines* (1992), J. Saltz and P. Mehotra, Eds., North-Holland, pp. 139–176.

[13] International Organization for Standardization. Programming Lnanguages – Fortran. ISO/IEC 1539-1:2010 Standard, 2010.

[14] Kale, L. V., and Zheng, G. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[15] Karlsson, S., and Brorsson, M. A comparative characterization of communication patterns in applications using MPI and shared memory on an IBM SP2. In *Network-Based Parallel Computing Communication, Architecture, and Applications*, D. Panda and C. Stunkel, Eds., vol. 1362 of *Lecture Notes in Computer Science*. Springer, 1998, pp. 189–201.

[16] Kennedy, K., Koelbel, C., and Zima, H. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages* (2007), p. 7.

[17] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI)* (2007), pp. 211–222.

[18] Nieplocha, J., Harrison, R., and Littlefield, R. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing 10*, 2 (1996), 169–189.

[19] Nishtala, R., Hargrove, P., Bonachea, D., and Yelick, K. Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2009), pp. 1–12.

[20] Numrich, R., and Reid, J. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum* (1998), vol. 17(2), pp. 1–31.

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

[22] Shan, H., and Singh, J. Parallel tree building on a range of shared address space multiprocessors: Algorithms and application performance. In *Proceedings of 1st Internation Symposium on Parallel and Distributed Processing (IPPS/SPDP)* (1998), pp. 475–484.

[23] Singh, J. *Parallel hierarchical N-body methods and their implications for multiprocessors*. PhD thesis, Stanford University, 1993.

[24] Singh, J., Weber, W., and Gupta, A. SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News 20*, 1 (1992), 44.

[25] UPC Consortium. UPC language specifications, v1.2. Tech. Rep. LBNL-59208, Lawrence Berkeley National Lab, 2005.

[26] Warren, M., and Salmon, J. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (1993), pp. 12–21.

[27] Warren, M., and Salmon, J. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (1993), pp. 12–21.

[28] Wu, J., Saltz, J., Hiranandani, S., and Berryman, H. Runtime compilation methods for multicomputers. In *Proceedings of the International Conference on Parallel Processing (ICPP): Software* (1991), p. 26.

[29] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., et al. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience 10*, 11-13 (1998), 825–836.

[30] Zhang, Z., and Seidel, S. Benchmark measurements of current UPC platforms. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International* (2005), pp. 276–283.