DOI:10.1145/2160718.2160740

A Massively Parallel Adaptive Fast Multipole Method on Heterogeneous Architectures

By Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros

Abstract

We describe a parallel fast multipole method (FMM) for highly nonuniform distributions of particles. We employ both distributed memory parallelism (via MPI) and shared memory parallelism (via OpenMP and GPU acceleration) to rapidly evaluate two-body nonoscillatory potentials in three dimensions on heterogeneous high performance computing architectures. We have performed scalability tests with up to 30 billion particles on 196,608 cores on the AMD/CRAY-based Jaguar system at ORNL. On a GPU-enabled system (NSF's Keeneland at Georgia Tech/ORNL), we observed 30× speedup over a single core CPU and 7× speedup over a multicore CPU implementation. By combining GPUs with MPI, we achieve less than 10 ns/particle and six digits of accuracy for a run with 48 million nonuniformly distributed particles on 192 GPUs.

1. INTRODUCTION

N-body problems are the computational cornerstone of diverse problems in mathematical physics,[16] machine learning,[8] and approximation theory.[4] Formally, the problem is how to efficiently evaluate

$$f(x_i) = \sum_{j=1}^{N} K(x_i, u_j) s(y_j),\ i = 1, \ldots, N, \tag{1}$$

which is essentially an $\mathcal{O}(N^2)$ computation of all pairwise interactions between *N* particles. We refer to *f* as the "*potential*," *s* as the "*source density*," *x* as the coordinates of the target particles, *y* as the coordinates of the source particles, and K as the interaction kernel. For example, in electrostatic or gravitational interactions $K(x, y) = K(r) = \frac{1}{|r|}$ with $r = x - y$ and $|r|$ its Euclidean norm. Equation (1) expresses an *N-body problem*.

The challenge is that the $\mathcal{O}(N^2)$ complexity makes the calculation prohibitively expensive for problem sizes of practical interest. For example, understanding blood flow mechanics[21] or understanding the origins of the universe[26] requires simulations in which billions of particles interact.

Given the importance of *N*-body problems, there has been a considerable effort to accelerate them. Here is the main idea: the magnitude and variance of *K*(*r*) decay with $|r|$, and so when the particles or groups of particles are well separated, the interactions can be approximated with small computational cost. But the details are tricky. How can we perform the approximation? Can we provide rigorous error and complexity bounds? And how do we implement these approximations efficiently?

One of the first successful attempts toward fast schemes was the Barnes–Hut method[1] that brought the complexity down to $\mathcal{O}(N \log N)$. Greengard and Rokhlin[3, 10, 22] solved the problem. They introduced the fast multipole method (FMM) that uses a rapidly convergent approximation scheme that comes with rigorous error bounds and reduces the overall complexity (in three dimensions) to $\mathcal{O}(\log(1/\epsilon)^3 N)$, where ϵ is the desired accuracy of the approximation to the exact sum. This result is impressive: *we can evaluate the sum in* Equation (1) *in arbitrary precision in linear time*.

So how does FMM work? Here is a sketch. Given the positions of particles for which we want to compute all pairwise interactions, we build an octree (henceforth we just consider FMM in three dimensions) so that any leaf node (or octant) has approximately a prescribed number of particles; then we perform tree traversals to evaluate the sum. We perform a postorder (bottom-up) traversal followed by a preorder traversal (top-down). The postorder traversal consists of calculations that involve an octant and its children. The preorder traversal is more complex, as it involves several octants in a neighborhood around the octant being visited.

At this level of abstraction, we begin to see how we might parallelize the FMM. When the distribution of particles is uniform, the analysis is relatively straightforward: we perform the traversals in a level-by-level manner, using data parallelism across octants in each level. The longest chain of dependencies is roughly $\mathcal{O}(\log N)$. The challenge lies in the case of nonuniform particle distributions, in which even the sequential case becomes significantly more complicated.[3]

Summary of the method: We describe and evaluate an implementation of the FMM for massively parallel distributed memory systems, including heterogeneous architectures based on multicore CPU and GPU co-processors. We use the message passing interface (MPI) for expressing distributed memory parallelism, and OpenMP and CUDA for shared memory and fine-grain parallelism. There exist many variants of FMM, depending on the kernel and the

The original version of this paper has the same title and was published in the *2009 Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. Some results in this paper also appeared in *Reference Rahimian et al*, 2010.

interaction approximation scheme. Our implementation is based on the *kernel-independent FMM*.[28, 29] However, this choice affects only the construction of certain operators used in the algorithm. It does not affect the overall "flow" of the algorithm or its distributed-memory parallelization. (It does affect single-core performance as vectorization and fine-grain parallelism depend on the FMM scheme.)

Our parallel algorithm can be summarized as follows. Given the particles, distributed in an arbitrary way across MPI processes, and their associated source densities, we seek to compute the potential at each point. (For simplicity, in this paper, we assume that source and target points coincide.) First, we sort the points in Morton order,[24] a kind of space-filling curve, and then redistribute them so that each process owns a contiguous chunk of the sorted array. Second, we create the so-called locally essential tree (LET, defined in Section 3), for each process in parallel. Third, we evaluate the sums using the LETs across MPI processes and using GPU and OpenMP acceleration within each process. All the steps of the algorithm are parallelized, including tree construction and evaluation.

Related work: There is a considerable literature on parallel algorithms for N-body problems. A work–depth analysis of the algorithm gives $\mathcal{O}(N)$ work and $\mathcal{O}(\log N)$ depth for particle distributions that result in trees that have depth bounded by $\mathcal{O}(\log N)$. Greengard and Gropp analyzed and implemented the algorithm using the concurrent-read, exclusive-write (CREW) PRAM model for the case of uniform particle distributions.[9] The implementation is straightforward using level-by-level traversals of the tree and using data parallelism. The complexity of the algorithm, omitting the accuracy-related term, is $\mathcal{O}(N/p)+\mathcal{O}(\log p)$, where p is the number of threads. This result extends to message passing implementations.

The implementation, parallel scalability, and complexity analysis become harder when one deals with nonuniform distributions of particles. Callahan and Kosaraju[2] propose an exclusive read exclusive write (EREW)-PRAM algorithm with $p=N$ processors which is work optimal, takes $\log p$ time, and does not depend on the distribution of the points. To our knowledge, that algorithm has not been used in practice. Our algorithm uses many ideas that first appeared in the seminal work of Warren and Salmon, who introduced space-filling curves using Morton-ordering for partitioning the points across processors, the notion of local essential trees (LET), and a parallel tree construction algorithm.[27] These ideas were widely adopted and analyzed in other works.[7, 23]

Teng was the first to provide a complexity analysis that accounts for communication costs for particle distributions whose tree depth is bounded by $\log N$ (in finite floating point precision, all particle distributions satisfy this assumption).[25] The key idea is to build a communication graph in which graph nodes correspond to tree octants and edges to interactions between octants. Teng shows that FMM communication graphs have a bisector whose edge cut (in three dimensions) is bounded by $\mathcal{O}(N^{2/3}(\log N)^{4/3})$; to compare, the bisector for a uniform grid is $\mathcal{O}(N^{2/3})$. This result shows that scalable FMM calculations are theoretically possible. Teng outlines a divide-and-conquer parallel algorithm to construct and partition the tree. Morton-order sorting is used to build the tree and repartitioning is done using a geometric graph partitioning method that guarantees low edge-cut and good load balancing.[18] Assuming (1) a parallel radix sort is used to partition the points and (2) the depth of the FMM tree grows logarithmically with N, *the overall algorithm scales as* $\mathcal{O}(N/p)+\mathcal{O}(\log p)$. The constants in the complexity estimates involve flop rate, memory latency, and bandwidth parameters of the underlying machine architecture.

Work that is focused more on actual implementations and performance analysis includes Kurzak and Pettitt[15] and Ogata et al.[19] Several groups have been working on GPU acceleration.[5, 11, 12, 20, 30] Finally, for nearly uniform distributions of particles, one can use fast Fourier transforms (particle-in-cell methods) to calculate the sums in $\mathcal{O}(N \log N)$ time.

Contributions: To our knowledge, there are no FMM implementations that demonstrate scalability on thousands of MPI processes for highly nonuniform particle distributions. Our main contribution is to produce and demonstrate the scalability of such an implementation that exploits all levels of parallelism. In Section 3, we discuss the distributed memory parallelism and introduce a novel FMM-specific all-reduce algorithm that uses hypercube routing. In Section 4, we introduce the shared memory parallelism of our implementation, and in Section 5, we discuss our experiments and the scalability results.

2. OUTLINE OF FMM

In this section, we describe the FMM data structures, the main algorithmic steps, and the parallel scalability of the method. Without loss of generality for the rest of the paper, we assume that the source particles y_j and the target particles x_i coincide. The FMM tree construction ensures that every leaf octant contains no more than q particles. The sequential algorithm is simple: the root octant is chosen to be a cube that contains all the particles; we insert the particles in the tree one by one; we subdivide an octant when it contains more than q particles. The parallel construction is more involved, and we describe it in Section 3.

After the tree construction, for each octant β we need to construct its *interaction lists*. These lists contain other octants in the tree. Each list represents a precisely defined spatial neighborhood around β, and for every list we have to compute "interactions" between β and the octants in that list. By "interactions," we refer to floating point operations that correspond to matrix–matrix or matrix–vector multiplications. Once the interaction lists have been constructed, the tree is traversed twice: fist bottom-up and then top-down. In each traversal for each octant, we perform calculations that involve other octants in its interaction lists.

One difficulty in FMM codes is the efficient mathematical representation and implementation of the octant–octant interaction matrices so that we achieve algorithmic efficiency without compromising accuracy. The size of the matrices varies, but typical dimensions are 100–1000, depending on the implementation. FMM algorithm designers seek to reduce the size of the matrices, to employ symmetries to reduce storage needs, to use precomputation, and to use fast approximations (e.g., fast Fourier transforms or low-rank singular value decompositions). All we need to remember here is that once we fix the desired accuracy, the computation of the interactions between an octant β and the octants in its interaction lists can be completed in $\mathcal{O}(1)$ time.

Now let us give the precise definition of these lists. For each octant β in the tree, we create four lists of octants, the so-called U-, V-, W- and X-list.[3] The U-list is defined only for leaf octants. For a leaf octant β, the U-list of β consists of all leaf octants adjacent to β, including β itself. (We say that α is adjacent to β if β and α share a vertex, an edge, or a face.) For any octant β, its *colleagues* are defined as the adjacent octants that are in the same tree level. The V-list of an octant β (leaf or non-leaf) consists of those children of the colleagues of β's parent octant, $P(\beta)$, which are not adjacent to β. The W-list is only created for a leaf octant β and contains an octant α if and only if α is a descendant of a colleague of β, α is not adjacent to β, and the parent of α is adjacent to β. The X-list of an octant β consists of those octants α which have β on their W-list. These definitions are summarized in Table 1.

For each octant $\beta \in T$, we store its level ℓ and two vectors, u and d. The u vector is an approximate representation of the potential generated by the source densities inside β. This representation is sufficiently accurate only if the evaluation particle is *outside the volume covered by β and the colleagues of β*. For the u vector we will use the name "upward equivalent density" or simply "upward density." This terminology is motivated by Ying et al.[28, 29]

The d vector is an approximate representation of the potential generated by the source densities *outside the volume covered by β and colleagues of β*. This approximate representation is sufficiently accurate only if the evaluation particle is *enclosed by β*. For the d vector we will use the name "downward equivalent density" or simply "downward density." For leaf octants ($\beta \in L$), we also store x, s, and f. Given the above definitions, approximately evaluating the sum in Equation (1) involves an upward computation pass of T, followed by a downward computation pass of T, seen more specifically in Algorithm 1. Let us reiterate that the details of the "`interact`" step in Algorithm 1 depend on list type and the underlying FMM scheme.

Table 1. Notation. Here we summarize the main notation used to define the interaction lists for each octant in the FMM tree. (The symbol "\" denotes standard set exclusion.) Note that $\alpha \in \mathcal{W}(\beta)$ need not be a leaf octant; conversely, since $\mathcal{W}(\beta)$ exists only when $\beta \in L$, $\alpha \in \mathcal{X}(\beta)$ implies that $\alpha \in L$. We say that α is adjacent to β if β and α share a vertex, an edge, or a face. See Figure 1 for an example of the U, V, W, and X lists in two dimensions.

α, β	Octants in the FMM tree
ℓ	FMM tree level
q	Maximum number of particles/octant
Octant lists	
T	The FMM tree
L	All leaf octants
$P(\beta)$	Parent octant of β ($\emptyset$ for root of T)
$\mathcal{A}(\beta)$	All ancestor octants of β
$\mathcal{K}(\beta)$	The eight children octants of β ($\emptyset$ for $\beta \in T$)
$\mathcal{D}(\beta)$	All descendant octants of β in T
$\mathcal{C}(\beta)$ (colleagues)	Adjacent octants to β, same level
$\mathcal{J}(\beta)$	Adjacent octants to β (arbitrary level)
U-list: $\alpha \in \mathcal{U}(\beta)$	$\alpha \in L$ adjacent to β (note: $\beta \in \mathcal{U}(\beta)$)
V-list: $\alpha \in \mathcal{V}(\beta)$	$\alpha \in \mathcal{K}(\mathcal{C}(P(\beta)))\backslash\mathcal{C}(\beta)$
W-list: $\alpha \in \mathcal{W}(\beta)$	$\alpha \in \mathcal{D}(\mathcal{C}(\beta))\backslash\mathcal{J}(\beta)$, $P(\alpha) \in \mathcal{J}(\beta)$
X-list: $\alpha \in \mathcal{X}(\beta)$	iff $\beta \in \mathcal{W}(\alpha)$
$\mathcal{I}(\beta)$ (interaction)	$\mathcal{V}(\beta) \cup \mathcal{U}(\beta) \cup \mathcal{W}(\beta) \cup \mathcal{X}(\beta)$

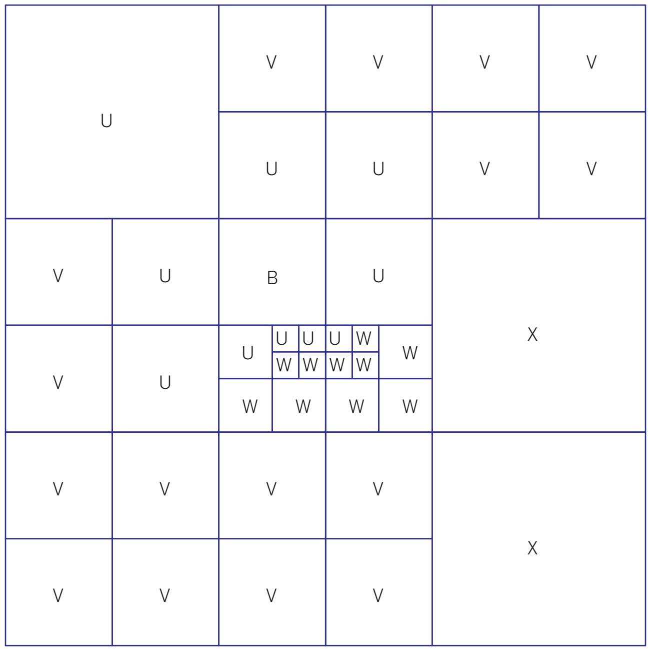
2.1. Parallelizing the evaluation phase

Understanding the dependencies in Algorithm 1 and devising efficient parallel variants can be accomplished by using distributed and shared memory programming models. These models can then be implemented using MPI, OpenMP, and GPU application programming interfaces.

There are multiple levels of concurrency in FMM: across steps (e.g., the S2U and ULI steps have no dependencies), within steps (e.g., a reduction on octants of an octant β during the VLI step), and vectorizations of the per-octant calculations. In fact, one can pipeline the S2U, U2U, and VLI steps to expose even more parallelism. We do not take advantage of pipelining because it reduces the modularity and generality of the implementation.

The generic dependencies of the calculations outlined in Algorithm 1 are as follows: The Approximate Interactions (except for steps (5a) and (5b)) and Direct Interactions parts can be executed concurrently. For the approximate interaction calculations, the order of the steps denotes dependencies, for example, step (2) must start after step (1) has been completed. Steps (3a) and (3b) can be executed in any order. However, concurrent execution of steps (3a) and (3b) requires a concurrent write with accumulation. This is also true for the steps (5a) and (5b), they are independent up to concurrent writes.

Figure 1. FMM lists. Here we show an example of the U, V, W, and X lists in two dimensions for a tree octant *B*. One can verify that $\mathcal{I}(B)$ is inside the domain enclosed by $\mathcal{C}(P(B))$.



Algorithm 1. $\{f_i\}_{i=1}^{N} = \text{FMM}\left(\{x_i, y_i, s_i\}_{i=1}^{N}, \text{octree}\right)$

```
// APPROXIMATE INTERACTIONS
// (1) S2U: source-to-up step
  $\forall\beta \in L$: source to up densities interaction
// (2) U2U: up-to-up step (upward)
  Postorder traversal of $T$
  $\forall\beta \in T$: interact $(\beta, P(\beta))$
// (3a) VLI : V-list step
  $\forall\beta \in T : \forall\alpha \in \mathcal{V}(\beta)$ interact $(\beta, \alpha)$ ;
// (3b) XLI : X-list step
  $\forall\beta \in T : \forall\alpha \in \mathcal{X}(\beta)$ : interact $(\beta, \alpha)$ ;
// (4) D2D: down-to-down step (downward)
  Preorder traversal of $T$
  $\forall\beta \in T$: interact $(\beta, \alpha)$ ;
// (5a) WLI: W-list step
  $\forall\beta \in L : \forall\alpha \in \mathcal{W}(\beta)$ : interact $(\beta, \alpha)$ ;
// (5b) D2T : down-to-targets step
  $\forall\beta \in L$: evaluate potential on $x_i \in \beta$;
//DIRECT INTERACTIONS
// ULI: U-list step (direct sum)
  $\forall\beta \in L : \forall\alpha \in \mathcal{U}_\beta$: interact $(\beta, \alpha)$ ;
```

Our overall strategy is to use MPI-based distributed memory parallelism to construct the global tree and to partition it into overlapping subtrees (the LETs) in order to remove dependencies between steps (3a)/(5a) and (3b)/(5b). We handle the concurrent writes explicitly and we use shared-memory-based parallelism on GPUs within each MPI process to accelerate the direct interactions and steps (1), (3), (5) of the indirect interactions in Algorithm 1. In the following sections, we give the details of our approach.

3. DISTRIBUTED MEMORY PARALLELISM

The main components of our scheme are (1) *the tree construction*, in which the global FMM tree is built and each process receives its locally essential tree and a set of leaves for which it assumes ownership and (2) *the evaluation*, in which each process evaluates the sum at the particles of the leaf octants it owns. This sum has two components: direct interactions (evaluated exactly) and indirect interactions (evaluated approximately).

The input consists of the particles and their source densities. The output of the algorithm is the potential at the particles. Before we describe the algorithm, we need the following definition:[27]

Locally essential tree (LET): Given a partition of L across processes so that L_k is the set of leaf-octants assigned to the process k (i.e., the potential in these octants is computed by process k), the LET for process k is defined as the union of the interaction lists of all owned leaves and their ancestors:

$$\text{LET}(k) := \cup_{\forall\beta \in [L_k \cup \mathcal{A}(L_k)]} \mathcal{I}(\beta).$$

The basic idea[27] for a distributed memory implementation of the FMM algorithm is to partition the leaves of the FMM tree across processes, construct the LET of each process, and then compute the N-body sum in parallel. There are two communication-intensive phases in this approach: the first phase is the LET construction and the second phase is an all-reduce during evaluation. Next, we discuss the main components in these two phases.

3.1. Tree construction

The inputs for the tree construction procedure are the particles. The output is the local essential tree on each process, which is subsequently used in the computation, along with geometrical domain decomposition of the unit cube across MPI processes. The latter is used throughout the algorithm. The tree construction involves (1) the construction of a distributed linear complete octree that contains only the leaf octants and (2) the construction of the per-process LETs.

We start by creating the distributed and globally Morton-sorted array containing all the leaves from the global tree. The algorithms for this task are known (we use the one described in Sundar et al.;[24] see also Hariharan and S. Aluru[13]), and their main ingredient is the parallel Morton-sort of particles. This sort determines the overall complexity of the tree construction.

The distribution of the leaves between processes induces a geometric partitioning of the unit cube: each process controls the volume covered by the leaves it owns. By Ω_k, we will denote the region "controlled" by process k. Each process stores the overall geometric partitioning: we use an `MPI_AllGather` to exchange the first and last octants of their region, which is all is needed to define the partition.

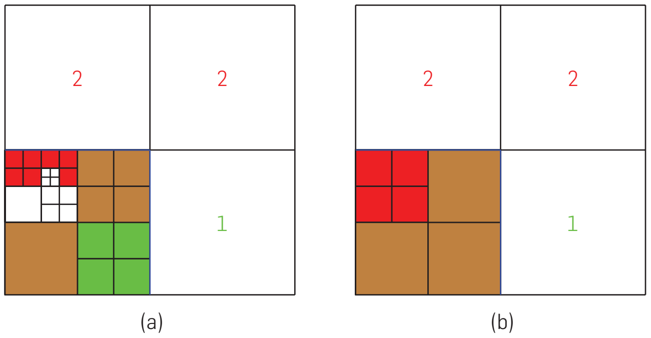
Next, each process adds all ancestor octants to its local leaves, thus creating a local tree. The task of exchanging information about "ghost" octants still remains to be completed. To accomplish that, let us introduce the following definitions:

- "*Contributor*" processes of an octant $\beta \in T$: $\mathcal{P}_c(\beta) := k \in 1 \ldots p : \beta$ overlaps with Ω_k
- "*User*" processes of an octant β: $\mathcal{P}_u(\beta) := k \in 1 \ldots p : P(\beta)$ or $\mathcal{C}(P(\beta))$ overlaps with Ω_k.

Let $I_{kk'}$ be the set of octants to which process k contributes and which process k' uses. Then, process k must send all octants in $I_{kk'}$ to MPI process k'. Figure 2 provides an illustration of this procedure. That is, each process contributor of each octant sends the octant data to all users of that octant. Once the exchange of the $I_{kk'}$ lists has been completed, all MPI processes insert received octants into their local trees. This concludes the construction of the per-process LETs. We summarize the construction of the local essential trees in Algorithm 2.

The correctness of the construction is based on the direct relation between LET and FMM. Consider the potential generated by the sources enclosed by some octant β. In order to evaluate this potential outside the volume covered by $\mathcal{C}(P(\beta))$, one does not need information regarding sources or upward densities associated with β, since the upward density of some ancestor of β would be used instead. This observation can be used to formalize the correctness of the LET construction. See Lashuk et al.[17] for a more detailed discussion.

Figure 2. Communication of ghost octants. Process 0 sends green octants to process 1, red octants to process 2, and brown octants to both 1 and 2. White octants in lower-left corner are "internal" to process 0 and not sent to anyone. The procedure is applied to both leaves and non-leaf octants. (a) Finer level of octree. (b) Coarser level of octree.



Algorithm 2. LET construction

Input: *distributed set of particles x;*
Output: *LET on each process k*

```
1. $L_k$ = Points2Octree($x$)                              //MPI
2. $B_k = L_k \cup \mathcal{A}(L_k)$
3. $I_{kk'} := \{\beta \in B_k : P(\beta)$ or $\mathcal{C}(P(\beta))$ overlaps with $\Omega_{k'}\}$
4. $\forall k' : k' \neq k$
      Send $I_{kk'}$ to process $k'$                        //MPI
      Recv $I_{k'k}$ from process $k'$                      //MPI
      Insert $I_{k'k}$ in $B_k$
5. Return Bk
```

In the last step of Algorithm 2, every process independently builds U, V, W, and X lists for the octants in its LET which enclose the local particles (where the potential is to be evaluated). All necessary octants are already present in the LET, so no further communication is required in this step.

3.2. Load balancing

Assigning each process an equal chunk of leaves may lead to a substantial load imbalance during the interaction evaluation for nonuniform octrees. In order to overcome this difficulty, we use the following load-balancing algorithm.

After the LET setup, each leaf is assigned a weight, based on the computational work associated with its U, V, W, and X lists (which are already built at this particle). Then, we repartition the leaves to ensure that total weight of the leaves owned by each process is approximately equal. We use Algorithm 1 from Sundar et al.[24] to perform this repartitioning in parallel. Note that similarly to the previous section, each process gets a contiguous chunk of global (distributed) Morton-sorted array of leaves. In the final step, we rebuild the LET and the U, V, W, and X lists on each process.

Note that we repartition the leaves based solely on work balance ignoring the communication costs. Such an approach is suboptimal, but is not expensive to compute and works reasonably well in practice. In addition to leaf-based partitioning, the partitioning at a coarser level can also be considered;[25] we have not tested this approach.

3.3. FMM evaluation

Three communication steps are required for the potential evaluation. The first is to communicate the source densities for the U-list calculation. This communication is "local" in a sense that each process typically communicates only with its spatial neighbors. Thus, a straightforward implementation (say, using `MPI_Isend`) is acceptable.

The second communication step is to sum up the upward densities of all the contributors of each octant, the U2U step. During this step (bottom-up traversal), each process only builds *partial* upward densities of octants in its LET. The partial upward densities of an octant do not include contributions from descendants of the octant that belong to other MPI processes. For this reason, we need a third step, a communication step, to collect the upward densities to the users of each octant. This communication step must take place after the U2U step and before the VLI and XLI steps. Once this step is completed, every process performs a top-down traversal of its LET without communicating with other processes.

Algorithm 3 describes a communication procedure that combines the second and the third steps mentioned above. This algorithm resembles the standard "AllReduce" algorithm on a hypercube. What follows is the informal description of the algorithm.

At the start, each processor r forms a pool $\mathcal{S}$ of octants (and their upward densities) which are "shared", that is, not used by r alone. The MPI communicator (set of all MPI processes) is then split into two halves (for simplicity, we shall assume that the communicator size is a power of two) and each processor from the first half is paired with a "peer" processor from the other half.

Now consider those octants in $\mathcal{S}$ which are "used" by some processor from the other half of the communicator (not necessarily a peer of r). If such octants exist, r sends them (together with upward densities) to its "peer". According to the same rule, the "peer" will send some (possibly none) octants and densities to r. The received octants will be merged into $\mathcal{S}$, eliminating duplicate octants while *summing* densities for duplicate octants.

At this point, no further communication between the two halves is required, and the algorithm proceeds recursively by applying itself to each of the halves. We finally note that after each communication round, $\mathcal{S}$ is purged from the "transient" octants which are no longer necessary for communications and not used locally at r.

The time complexity of this algorithm is not worse than $\mathcal{O}(\sqrt{p})$. To be more specific, assuming that no process uses more than m shared octants and no process contributes to more than m shared octants, for a hypercube interconnect, the communication complexity of Algorithm 3 is $\mathcal{O}(t_s \log p + t_w m(3\sqrt{p} - 2))$, where t_s and t_w are the latency and the bandwidth constants, respectively. See Lashuk et al.[17] for a proof.

After the three communication steps, all the remaining steps of Algorithm 1 can be carried out without further communication.

3.4. Complexity for uniform distributions of particles

We have all the necessary ingredients to derive the overall complexity of the distributed memory algorithm for uniform distributions of particles. Let N be the number of particles and let p be number of processes. The number of octants is proportional to the number of particles. The first communication cost is associated with the parallel sort of the input particles. We have used a comparison sort instead of binning, so its time complexity is $\mathcal{O}\left(\frac{N}{p}\log\frac{N}{p} + p\log^2 p\right)$ (combination of sample sort and bitonic sort).[6] Exchanging the "ghost" octants has the same complexity as the reduce-broadcast algorithm described in Section 3.3, that is, $\mathcal{O}(\sqrt{pm})$, where m is the maximal number of "shared" octants between two processes. For a uniform grid, m can be estimated as $\mathcal{O}((N/p)^{2/3})$ divided by p. The communication also includes the exchange of source densities. Assuming that the bandwidth of the system is reasonably high, we can neglect all lower order communication terms. In summary (assuming large enough bandwidth but assuming nothing about latency), the overall complexity of the setup phase is $\mathcal{O}\left(\frac{N}{p}\log\frac{N}{p}\right) + \mathcal{O}(p\log^2 p)$. For the evaluation, we have $\mathcal{O}\left(\frac{N}{p}\right) + \mathcal{O}(\log p)$.

Algorithm 3. Reduce and Scatter

```
Input: partial upward densities of "shared" octants at "contributor" processes
Input: r (rank of current process); assume communicator size is $2^d$
Output: upward densities of "shared" octants at "user" processes
//Define shared octants (for each process):
$\mathcal{S} = \{\beta \in LET : \#(\mathcal{P}_u(\beta) \cup \mathcal{P}_c(\beta)) > 1\}$
//Loop over communication rounds (hypercube dimensions)
For $i := d - 1$ to 0
    //Process s is our partner during this communication round
    1. $s := r$ XOR $2^i$
    2. $u_s = s$ AND $(2^d - 2^i)$
    3. $u_e = s$ OR $(2^i - 1)$
    4. Send to s : $\left\{\beta \in \mathcal{S} : \mathcal{I}(\beta) \cap \left(\cup_{u_s}^{u_e} \Omega_s\right) \neq \emptyset\right\}$
    5. $q_s = r$ AND $(2^d - 2^i)$
    6. $q_e = r$ OR $(2^i - 1)$
    7. Delete $\left\{\beta \in \mathcal{S} : \mathcal{I}(\beta) \cap \left(\cup_{q_s}^{q_e} \Omega_s\right) \neq \emptyset\right\}$
    //Reduction
    8. Recv from s and append $\mathcal{S}$
    9. Remove duplicates for $\mathcal{S}$
    10. Sum up densities for duplicate octants.
```

For a nonuniform particle distribution, the complexity estimates for the setup and evaluation phases include an additional $t_w m\sqrt{p}$ term. Since we do not have a (nontrivial) bound on m, this result is worse than what is theoretically possible by Teng's algorithm. (He partitions the tree using both work and communication costs instead of just using the work at the leaf octants.)

4. INTRA-NODE PARALLELISM

A key ingredient to the efficiency of the overall algorithm is highly tuned intra-node performance. Many current supercomputing systems have heterogeneous nodes, meaning they have both conventional general-purpose multicore CPU processors and more specialized high-speed co-processors, such as GPUs. Consequently, our implementation can exploit both CPUs (via OpenMP) and GPUs (via CUDA).

Shared memory parallelization is common for either type of processor. For the S2U, D2T, ULI, WLI, VLI, and XLI steps, we can visit the octants in an embarrassingly parallel way. Moreover, all octant visits include octant-to-octant or octant-to-particle interactions expressed as matrix–vector multiplications. The matrices are dense except for the VLI calculations, which correspond to a diagonal translation. Thus, overall there are two levels of parallelism: across octants and across the rows of the corresponding matrix. The same approach to parallelism applies on both CPU and GPU co-processors, because they have essentially the same architectural features for exploiting parallelism: shared memory address spaces, a multilevel memory hierarchy, and vector units for regular data parallelism. The U2U and D2D remain sequential in our current implementation, though they could in principle be parallelized using rake and compress methods.[14]

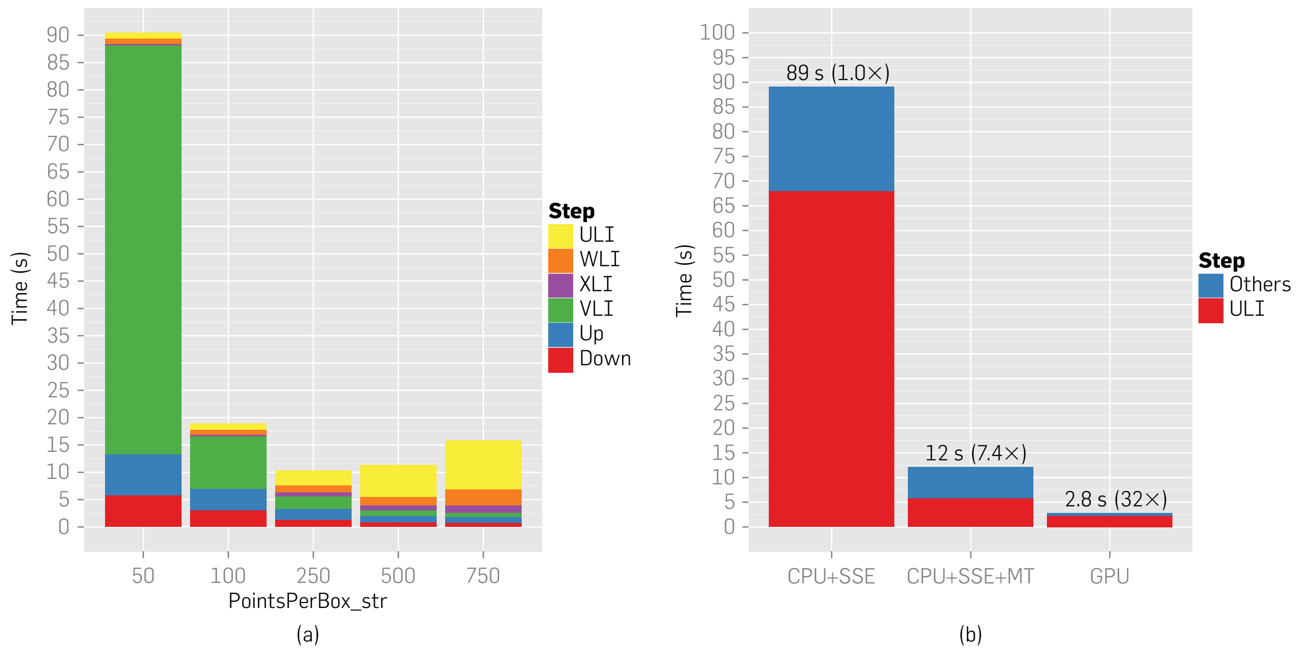
Although the architectural features are similar between CPUs and GPUs, differences in their *scale* directly affect the algorithm and implementation. The main algorithmic tuning parameter for the FMM is the maximum number of particles per octant, q. Increasing q makes the leaf octants larger and the overall tree shorter, which in turn increases the number of flops performed by the compute-bound ULI, WLI, and XLI steps, while decreasing the flop-cost of the memory-bound VLI and other phases. Figure 3(a) shows an example of this behavior. A processor's relative balance of performance and bandwidth will change these curves, thereby changing the optimal value of q.

Indeed, this phenomenon gives rise to the overall performance behavior shown in Figure 3(b), where we compare the single-socket execution times for three single-socket implementations: (i) single-threaded CPU with explicit vectorization using SSE intrinsics; (ii) multithreaded CPU with four threads and SSE; and (iii) GPU using CUDA. Note that combined threading and vectorization yields a 7.4× improvement over the single-threaded code. The GPU code is 32× faster than the single-threaded CPU code and 4.3× faster than the multithreaded CPU code, even including time to send/receive data from the GPU. Interestingly, the GPU win comes not just from parallelization and tuning but because faster processing enables better *algorithmic* tuning of q. In particular, the faster we can perform the compute-intensive ULI step, the larger we can make q thereby reducing the VLI and other memory-bound steps. In other words, there is a synergy between faster processing and algorithmic tuning in which we can trade more flops (ULI step) for less memory communication (e.g., VLI step). Such techniques are likely to apply more generally as many-core processors increase in use.

5. NUMERICAL EXPERIMENTS

This section evaluates the overall scalability of our implementation on two different architectures and in both strong and weak scaling regimes. (We have used the Stokes kernel instead of the Laplacian. $K(r)$ is defined as $1/|r|(1 - r \otimes r)/$

Figure 3. Intra-node optimization. (a) Optimal particles per octant: increasing the leaf sizes increases the relative costs of ULI, WLI, and XLI while decreasing the costs of the other phases. (b) Implementation comparison: solid horizontal bars indicate the observed wall-clock time on each platform. In the hybrid CPU+GPU implementation, ULI runs on the GPU while the other steps run on the CPU, yielding an overall benefit from overlapping.



r^2), where ⊗ is the outer product between two three-dimensional vectors.)

Platforms and architectures: We tested our code on two supercomputing platforms, one based on conventional CPUs and the other based on hybrid CPU+GPU nodes. The CPU system is *Jaguar PF* at the National Center for Computational Sciences (UT/ORNL), (http://www.nccs.gov/computing-resources/jaguar/) a Cray XT5 with 224,256 cores (using 2.6-GHz hex-core AMD Opteron CPUs, 2GB/core), and a three-dimensional-torus network (9.6 GB/s link bandwidth). Jaguar is ranked third on the Top 500 List (www.top500.org) as of June 2011. The GPU system is the *Keeneland* initial delivery system, also at UT/ORNL (http://keeneland.gatech.edu). Keeneland is based on the Hewlett-Packard SL390 servers accelerated with NVIDIA Tesla M2070 GPUs. Keeneland has 120 compute nodes, each with dual-socket, six-core Intel X5660 2.8-GHz Westmere processors and 3 GPUs per node, with 24GB of DDR3 host memory. Nodes are interconnected with single rail QDR Infiniband.

MPI, strong scalability tests on Jaguar: Figure 4(a) summarizes the strong scaling experiments. The problem has 100 million unknowns. We use a nonuniform (line) distribution. On 24,576 cores, the evaluation phase takes just 1.7 s, which constitutes a speedup of 205× when increasing cores by a factor of 512×. The setup phase (tree construction) begins to dominate at this scale in the strong-scaling regime.

MPI, weak scalability tests on Jaguar: Figure 4(b) summarizes the weak scaling results. The problem size per core is kept fixed to approximately 672,000 unknowns. The FMM evaluation phase and setup phase (tree construction) times increase because of several factors. First, since we have highly nonuniform distributions (the max leaf-level is 23 and the min is 4—for a uniform tree, the leaf level would be 9), as we increase the problem size, the tree gets deeper and the cost per core increases (i.e., we observe a $\mathcal{O}(N \log N)$ scaling), as we have not reached the asymptotic $\mathcal{O}(N)$ phase of the FMM. Secondly, for nonuniform trees, it is difficult to load balance all phases of FMM. The solution to these scaling problems is to employ the hypercube-like tree-broadcast algorithm for all of the phases of FMM. (Currently it is used only in the postorder communication phase of the evaluation phase.) Finally, the setup phase is not multithreaded, which is the subject of future work. Nevertheless, we still attain a reasonably good utilization of compute resources: the evaluation phase sustains over 1.2 GFlopsper core (in double-precision).

GPU strong scalability tests on Keeneland: Figure 5 shows strong scaling for a 48 million particle problem on up to 192 GPUs of the Keeneland system. We use 1 MPI process per GPU and 3 GPUs per node (64 nodes total). In the best case, the evaluation completes in just 0.47 s. For comparison, we estimate that the equivalent purely direct ($\mathcal{O}(n^2)$) calculation on 192 GPUs, ignoring all communication, would require approximately 1000× more wall clock time, which confirms the importance of employing parallelism on an algorithmically optimal method.

6. DISCUSSION AND CONCLUSION

We have presented several algorithms that taken together expose and exploit concurrency at all stages of the fast multipole algorithms and employ several parallel programming

Figure 4. Strong and weak scalability on Jaguar PF. (a) Strong Scaling on Jaguar PF. The strong scalability result for 22M particles. Each bar is labeled by its absolute wall-clock time (seconds) and speedup relative to the 48-core case. There are six cores (and six OpenMP threads) per MPI process. The finest level of the octree is nine and the coarsest is three. The evaluation and setup phases show 205× and 48× speedups, respectively, when increasing the core count by 512×. (b) Weak Scaling on Jaguar PF. The weak scalability of the simulation to 196,608 cores, with approximately 672,000 unknowns per core. Each bar is labeled by wall-clock time (seconds) and parallel efficiency. We use one MPI process per socket and all of the six OpenMP threads in each MPI process. The finest to coarsest octree levels range from 24 to 4. Our implementation maintains parallel efficiency levels of 60% or more at scale.

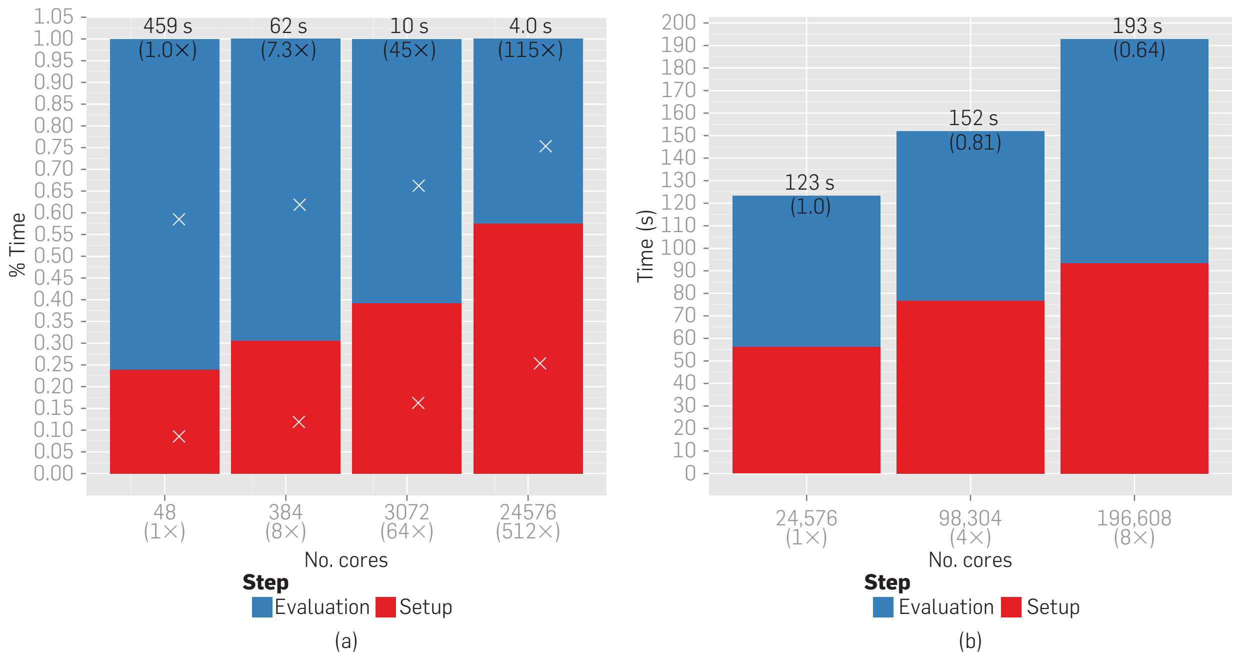
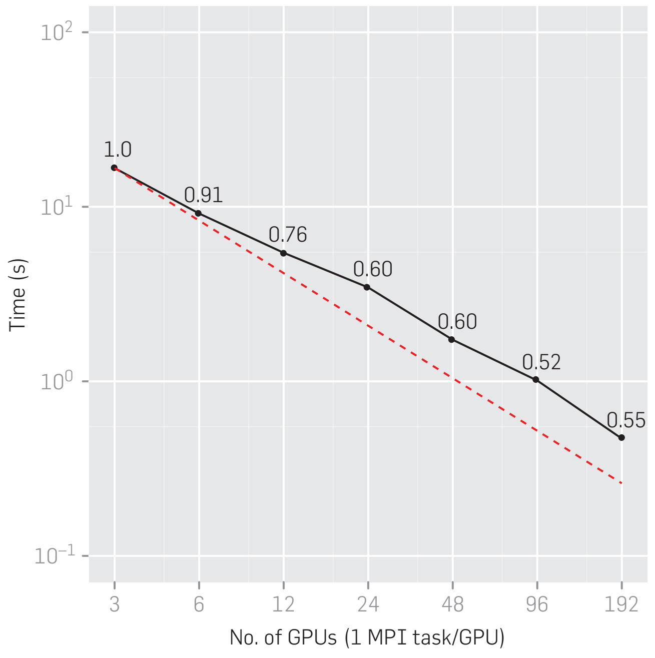


Figure 5. GPU strong scaling. We show strong scaling for FMM evaluation of 48 million particles on up to 192 GPUs of Keeneland system, which completes in as few as 0.47 s. The solid point-line shows the measured wall-clock time, and the dashed line shows the ideal time under perfect speedup. Each point is labeled by its parallel efficiency (ideal = 1).



paradigms. We showed that we can efficiently scale the tree setup with the major cost being the parallel sort, which in turn exhibits textbook scalability. We described a new reduction scheme for the FMM algorithm and we demonstrated overall scalability. We explored per-core concurrency using the streaming paradigm on GPU accelerators with excellent speedups. FMM is a highly nontrivial algorithm with several different phases, a combination of multiresolution data structures, fast transforms, and highly irregular data access. Yet, we were able to achieve significant speedups on heterogeneous architectures.

On our largest run on 196,608 cores, we observed about 220 TFlops/s in the evaluation, about 15% of the Linpack sustained performance. Assuming a 7× speedup on a hybrid machine of similar size means that our FMM would exceed one PetaFlop without further modifications. Our main conclusion is that it is possible to construct highly efficient scalable *N*-body solvers, at least up to hundreds of thousands of cores. But what about one million or many million cores?

One important lesson for us is that hybrid parallelism is absolutely necessary; once we reached several thousands of MPI processes, MPI resource management in the communication-intensive phases of the algorithm became an issue. Employing shared memory parallelism within the socket and combining with accelarators results in a more scalable code. The maintainability overhead, however, can be significant especially for the GPU case and for a complex code like FMM. Currently, we are working on introducing shared-memory parallelism in the tree construction, sharing the workload between CPUs and GPUs (essentially treating the GPU as another MPI process), and introducing shared-memory parallelism in the upward and downward computations.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation (NSF) grants CNS-0929947, CCF-0833136, CAREER-0953100, OCI-0749285, OCI-0749334, OCI-1047980, CNS-0903447 and Semiconductor Research Corporation (SRC) award 1981, U.S. Department of Energy (DOE) grant DEFC02-10ER26006/DE-SC0004915, and a grant from the U.S. Defense Advanced Research Projects Agency (DARPA). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF, SRC, DOE, or DARPA. Computing resources on the TeraGrid systems were provided under the TeraGrid allocation grants ASC-070050N, CCR-090024, ASC-100019, and MCA-04N026. We would like to thank the TeraGrid support staff, and also the staff and consultants at NCSA, TACC, and NICS from whom we have received significant assistance. C

References

1. Barnes, J., Hut, P. A hirerachical O(N logN) force-calculation algorithm. *Nature 324*, 4 (December 1986), 446–449.
2. Callahan, P.B., Kosaraju, S.R. Algorithms for dynamic closest pair and n-body potential fields. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '95* (Philadelphia, PA, USA, 1995), Society for Industrial and Applied Mathematics, 263–272.
3. Carrier, J., Greengard, L., Rokhlin, V. A fast adaptive multipole algorithm for particle simulations. *SIAM J. Sci. Stat. Comput. 9*, 4 (1988), 669–686.
4. Cherrie, J., Beatson, R.K., Newsam, G.N. Fast evaluation of radial basis functions:methods for generalized multiquadrics in r^n. *SIAM J. Sci. Comput. 23*, 5 (2002), 1549–1571.
5. Darve, E., Cecka, C., Takahashi, T. The fast multipole method on parallel clusters, multicore processors, and graphics processing units. *Comptes Rendus Mécanique 339*(2–3) (2011), 185–193.
6. Grama, A., Gupta, A., Karypis, G., Kumar, V. An Introduction to Parallel Computing: Design and Analysis of Algorithms, 2nd edn, Addison Wesley, 2003.
7. Grama, A.Y., Kumar, V., Sameh, A. Scalable parallel formulations of the Barnes–Hut method for n-body simulations. In *Proceedings of the 1994 conference on Supercomputing, Supercomputing '94* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, 439–448.
8. Gray, A., Moore, A. N-Body'problems in statistical learning. *Adv. Neural Inform. Process Syst.* (2001), 521–527.
9. Greengard, L., Gropp, W.D. A parallel version of the fast multipole method. *Comput. Math. Appl. 20*, 7 (1990), 63–71.
10. Greengard, L., Rokhlin, V. A fast algorithm for particle simulations. *J. Comput. Phys. 73* (1987), 325–348.
11. Gumerov, N.A., Duraiswami, R. Fast multipole methods on graphics processors. *J. Comput. Phys. 227*, 18 (2008), 8290–8313.
12. Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K., Taiji, M. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of SC09*, The SCxy Conference Series (Portland, Oregon, November 2009), ACM/IEEE.
13. Hariharan, B., Aluru, S. Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods. *Parallel Comput. 31* (3–4) (2005), 311–331.
14. JáJá, J. An Introduction to Parallel Algorithms, Addison Wesley, 1992.
15. Kurzak, J., Pettitt, B.M. Massively parallel implementation of a fast multipole method for distributed memory machines. *J.Parallel Distr. Comput. 65*, 7 (2005), 870–881.
16. Greengard, L. Fast algorithms for classical physics. *Science 265*, 5174 (1994), 909–914.
17. Lashuk, I. et al. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09* (New York, NY, USA, 2009), ACM, 58:1–58:12.
18. Miller, G., Teng, S., Thurston, W., Vavasis, S. Separators for sphere-packings and nearest neighbor graphs. *J. ACM (JACM) 44*, 1 (1997), 1–29.
19. Ogata, S. et al. Scalable and portable implementation of the fast multipole method on parallel computers. *Comput. Phys. Comm. 153*, 3 (2003), 445–461.
20. Phillips, J.C., Stone, J.E., Schulten, K. Adapting a message-driven parallel application to GPU-accelerated clusters. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (2008), 1–9.
21. Rahimian, A., Lashuk, I., Veerapaneni, S., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampath, R., Shringarpure, A., Vetter, J., Vuduc, R., Zorin, D., Biros, G. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *SC '10: Proceedings of the 2010 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2010), IEEE Press, 1–12.
22. Rokhlin, V. Rapid solution of integral equations of classical potential theory. *J. Comput. Phys. 60* (1985), 187–207.
23. Sevilgen, F., Aluru, S., Futamura, N. A provably optimal, distribution-independent parallel fast multipole method. In *Proceedings of 14th International Parallel and Distributed Processing Symposium, 2000. IPDPS 2000* (2000), IEEE, 77–84.
24. Sundar, H., Sampath, R.S., Biros, G. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM J. Sci. Comput. 30*, 5 (2008), 2675–2708.
25. Teng, S.H. Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation. *SIAM J. Sci. Comput. 19*, 2 (1998).
26. Teyssier, R. et al. Full-sky weak-lensing simulation with 70 billion particles. *Astron. Astrophys. 497*, 2 (2009), 335–341.
27. Warren, M.S., Salmon, J.K. A parallel hashed octtree N-body algorithm. In *Proceedings of Supercomputing*, The SCxy Conference Series (Portland, Oregon, November 1993), ACM/IEEE.
28. Ying, L., Biros, G., Zorin, D. A kernel-independent adaptive fast multipole method in two and three dimensions. *J. Comput. Phys. 196*, 2 (2004), 591–626.
29. Ying, L., Biros, G., Zorin, D., Langston, H. A new parallel kernel-independent fast multipole algorithm. In *Proceedings of SC03*, The SCxy Conference Series (Phoenix, Arizona, November 2003), ACM/IEEE.
30. Yokota, R., Bardhan, J., Knepley, M., Barba, L., Hamada, T. Biomolecular electrostatics using a fast multipole BEM on up to 512 GPUs and a billion unknowns. *Comput. Phys. Comm. 182*, 6 (Jun. 2011), 1272–1283.

Ilya Lashuk (lashuk2@llnl.gov), research scientist, Institute for Scientific Computing Research, Lawrence Livermore National Laboratory, Livermore, CA.

Aparna Chandramowlishwaran (aparna@cc.gatech.edu), graduate research assistant, Computational Science and Engineering Division, College of Computing, Atlanta, GA.

Harper Langston (harper@cc.gatech.edu), postdoctoral associate, Computational Science and Engineering Division, College of Computing, Atlanta, GA.

Tuan-Anh Nguyen (tuananh@cc.gatech.edu), graduate research assistant, Computational Science and Engineering Division, College of Computing, Atlanta, GA.

Rahul Sampath (rahul.sampath@gmail.com), postdoctoral associate, Oak Ridge National Laboratory, Oak Ridge, TN.

Aashay Shringarpure (aashay.shringarpure@gmail.com).

Richard Vuduc (richie@cc.gatech.edu), assistant professor, Computational Science and Engineering Division, College of Computing, Atlanta, GA.

Lexing Ying (lexing@math.utexas.edu), associate professor, Mathematics, The University of Texas at Austin, TX.

Denis Zorin (dzorin@cs.nyu.edu), professor, Courant Institute of Mathematical Sciences. New York University, New York, NY.

George Biros (gbiros@acm.org), W.A. "Tex" Moncrief, Jr. Simulation-Based Engineering Sciences Chair, Institute of Computational Engineering and Sciences, The University of Texas at Austin, TX.