Scalable Distributed Fast Multipole Methods

Qi Hu *, Nail A. Gumerov †, Ramani Duraiswami‡
* † ‡*University of Maryland Institute for Advanced Computer Studies (UMIACS)*
* ‡*Department of Computer Science, University of Maryland, College Park*
† ‡*Fantalgo LLC, Elkridge, MD*
**huqi,†gumerov,‡ramani@umiacs.umd.edu*

***Abstract*—The Fast Multipole Method (FMM) allows $O(N)$ evaluation to any arbitrary precision of N-body interactions that arises in many scientific contexts. These methods have been parallelized, with a recent set of papers attempting to parallelize them on heterogeneous CPU/GPU architectures [1]. While impressive performance was reported, the algorithms did not demonstrate complete weak or strong scalability. Further, the algorithms were not demonstrated on nonuniform distributions of particles that arise in practice. In this paper, we develop an efficient scalable version of the FMM that can be scaled well on many heterogeneous nodes for nonuniform data. Key contributions of our work are data structures that allow uniform work distribution over multiple computing nodes, and that minimize the communication cost. These new data structures are computed using a parallel algorithm, and only require a small additional computation overhead. Numerical simulations on a heterogeneous cluster empirically demonstrate the performance of our algorithm.**

***Keywords*-fast multipole methods; GPGPU; N-body simulations; heterogeneous algorithms; scalable algorithms; parallel data structures;**

I. Introduction

The N-body problem, in which the sum of N kernel functions Φ centered at N source locations $\mathbf{x}_i$ with strengths q_i are evaluated at M receiver locations $\{\mathbf{y}_j\}$ in $\mathbb{R}^d$ (see Eq. 1), arises in a number of contexts, such as stellar dynamics, molecular dynamics, boundary element methods, vortex methods and statistics. It can also be viewed as a dense $M \times N$ matrix vector product. Direct evaluation on the CPU has a quadratic $O(NM)$ complexity. Hardware accelerations alone to speedup the brute force computation, such as [2] using the GPU or other specialized hardware, can achieve certain performance gain, but not improve its quadratic complexity.

$$\phi(\mathbf{y}_j) = \sum_{i=1}^{N} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i), \quad j = 1, \ldots, M, \quad \mathbf{x}_i, \mathbf{y}_j \in \mathbb{R}^d, \tag{1}$$

An alternative way to solve such N-body problems is to use *fast algorithms*, for example, the Fast Multipole Method [3]. The FMM reduces the computation cost to linear for any specified tolerance ε, up to machine precision. In the FMM, Eq. 1 is divided into near-field and far-field terms given a small neighborhood domain of the evaluation point $\Omega(\mathbf{y}_j)$

$$\phi(\mathbf{y}_j) = \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i) + \sum_{\mathbf{x}_i \in \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i), \tag{2}$$

in which the (second) near-field sum is evaluated directly. The far-field sum is approximated using low rank representations obtained via kernel expansions and translations that employs a construct from computational geometry called "well separated pair decomposition" (WSPD [4]) with the aid of recursive data-structures based on octrees (which usually have a construction cost of $O(N \log N)$).

There are several different versions of distributed FMM algorithms in the literature, such as [5], [6], [7], [8], [9], [10]. The basic idea is to divide the whole domain into spatial boxes and assigned them to each node in a way that the work balance can be guaranteed. To obtain correct results with such a data distribution several issues have to be accounted for (Fig. 1). The first is domain overlap: while receiver data points can be mutual exclusively distributed on multiple nodes, source data points which are in the boundary layers of partitions need to be repeated among several nodes for the near-field sum. The algorithm should not only determine

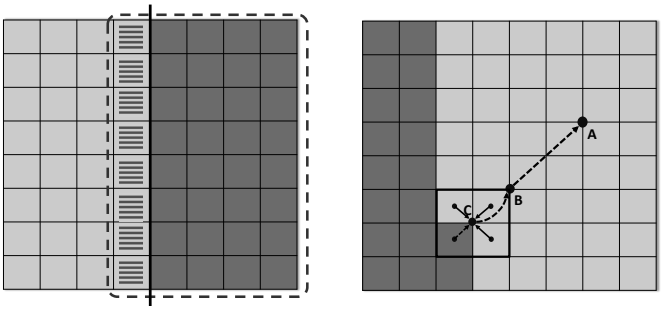

Figure 1. Problems in distributing the FMM across two nodes. Left: lightly-shaded boxes are on node 1 (Partition I) and darkly shaded boxes are on node 2 (Partition II). The thick line indicates the partition boundary line and the dash line shows the source points in both partitions needed by Partition II. The hashed boxes are in Partition I but they have also to be included in Partition II to compute the local direct sum. Right: light boxes belong to Partition I and dark boxes belong to Partition II. The multipole coefficients of the box with thick lines (center at C) is incomplete due to one child box on another node. Hence its parents (B and A) in the tree up to the minimal level are all incomplete.

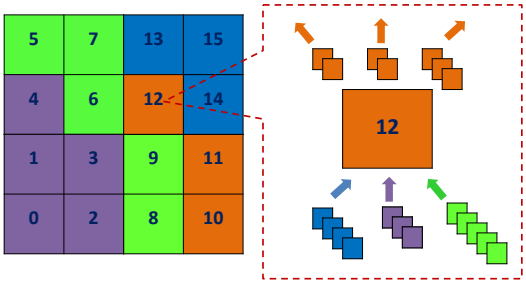

Figure 2. Non-uniform domain distribution. Boxes with different colors represent different partitions (on different nodes). Here the box labeled as 12 needs box data from purple, green and blue regions. It also has to distribute its data to those regions.

such overlap domains and distribute data efficiently but also guarantee that contributions from such repeated source data points are only translated once among all nodes. The second issue is incomplete translations, i.e. complete stencils may require translation coefficients from many boxes from other nodes. Thirdly, when source or receiver data points are distributed non-uniformly, the number of boxes assigned to each node may be quite different and the shape of each node's domain and its boundary regions may become irregular. Such divisions require inter-node communication of the missing or incomplete spatial box data of neighborhoods and translations among nodes at all the octree levels (Fig. 2). Given the fact that no data from empty boxes are kept, it is challenging to efficiently determine which boxes to import or export (translation) data to at all the levels.

In the literature, distributed FMM and tree code algorithms commonly use the *local essential tree* (LET [10], [11]) to manage communication among all the computing nodes. Implementation details for import or export data via LETs are not explicitly described in the well known distributed FMM papers, such as [9], [10], [12], [13]. Recently, [1] developed a distributed FMM algorithms for heterogeneous clusters. However, their algorithm repeated part of translations among nodes and required a global coefficient exchange of all the spatial boxes at the octree's maximal level. Such a scheme works well for small and middle size clusters but is not scalable to large size clusters. In another very recent work [14], homogeneous isotropic turbulence in a cube containing 2^{33} particles was successfully calculated, using a highly parallel FMM using 4096 GPUs, but the global communication of all the LETs is the performance bottleneck. In this paper, our purpose is to provide new data structures and algorithms with implementation details to address the multiple node data management issues.

A. Present contribution

Starting from [1], we design a new scalable heterogeneous FMM algorithm, which fully distributes all the translations among nodes and substantially decreases communication costs. This is a consequence of the new data structures which separate the computation and communication and avoid synchronization during GPU computations. The data structures are similar to the LET concept but use a master-slave model and further have a parallel construction algorithm, the granularity of which is at the level of spatial boxes (which allows finer work distribution than at the single node level). Each node divides its assigned domain into small spatial boxes via octrees and classifies each box into one of five categories in parallel. Based on the box type, each node determines which boxes need to import and export their data so that it would have the complete translation data after one communication step with the master. This can be computed on the GPU at negligible cost and this algorithm can handle non-uniform distributions with irregular partition shapes (Fig. 2). Our distributed algorithm improves timing results of [1] substantially and can be applied to large size clusters based on the better strong and weak scalability demonstrated. On our local Chimera cluster, we can perform a N-body sum of Coulomb potential for 1 billion particles on 32 nodes in 12.2 seconds (with a truncation number $p = 8$).

II. The Baseline FMM Algorithm

In the FMM, the far-field term of Eq. 2 is evaluated using the factored approximate representations of the kernel function, which come from local and multipole expansions over spherical basis functions and are truncated to retain p^2 terms. This truncation number p is a function of the specified tolerance $\varepsilon = \varepsilon(p)$. Larger values of p result in better accuracy, but also increase computational time as $\log \epsilon^{-1}$. The WSPD is recursively performed to subdivide the cube into subcubes via an octree until the maximal level l_{max}, or the tree depth, is achieved. The level l_{max} is chosen such that the computational costs of the local direct sum and far field translations can be balanced, i.e. roughly equal. The baseline FMM algorithm consists of four main parts: the initial expansion, the upward pass, the downward pass and the final summation.

1) **Initial expansion (P2M)**:
   a) At the finest level l_{max}, all source data points are expanded at their box centers to obtain the far-field multipole or M-expansions $\{\mathcal{C}_n^m\}$ over p^2 spherical basis functions. The truncation number p is determined by the required accuracy.
   b) M-expansions from all source points in the same box are consolidated into a single expansion.
2) **Upward pass (M2M)**: For levels from l_{max} to 2, M-expansions for each child box are consolidated via multipole-to-multipole translations to their parent source box.
3) **Downward pass (M2L, L2L)**: For levels from 2 to l_{max}, local or L-expansions are created at each receiver box

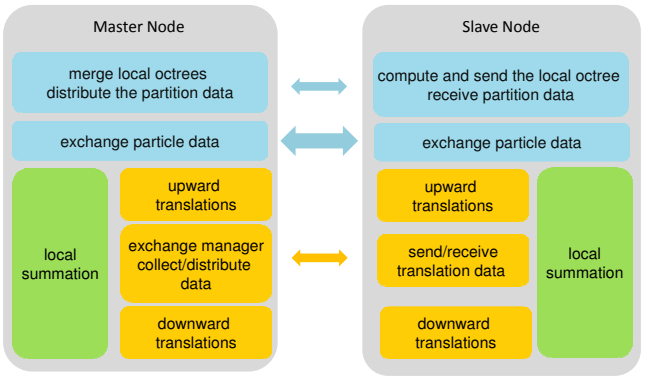

Figure 3. An overview of the distributed FMM algorithm. Blue parts are the overheads. Yellow and green parts represent the kernel evaluation performed concurrently on the GPU (it processes the green part) and the CPU (it computes the yellow parts).

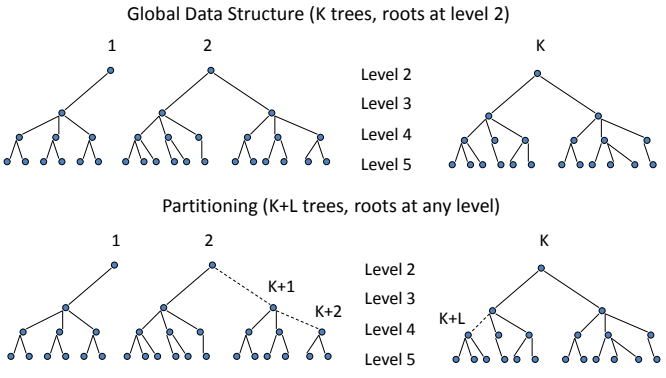

Figure 4. Global data structure with K trees with roots at level 2 and partitioned data structure with roots at any level.

a) Translate M-expansions from the source boxes at the same level belonging to the receiver box's parent neighborhood but not the neighborhood of that receiver itself, to L-expansions via multipole-to-local translations and consolidate the expansions.
b) Translate the L-expansion from the parent receiver box center to its child box centers and consolidate expansions.

4) **Final summation (L2P)**: Evaluate the L-expansions for all receiver points at the finest level l_{max} and performs a local direct sum of nearby source points.

The evaluations of direct sums of nearby source points are independent of the far-field expansions and translations, and can be performed separately. The costs of near-field direct sum and the far-field translations must be balanced to achieve optimal performance. Several different bases and translation methods have been proposed for the Laplace kernel. We used the expansions and methods described in [15], [16] and do not repeat details. Real valued basis functions that allow computations to be performed recursively with minimal use of special functions, or large renormalization coefficients are used. L- and M-expansions are truncated to retain p^2 terms given the required accuracy. Translations are performed using the RCR decomposition.

III. Distributed Heterogeneous FMM Algorithm

Our distributed algorithm (Fig. 3) has five main steps :

1) *Source and receiver data partition*: a partition of the whole space that balances the workload among the computing nodes based on a cost estimation; it also handles overlapped neighborhoods.
2) *Upward evaluation*: each node performs initial M-expansions of source points, upward translations and computes its own export/import box data.
3) *Multiple node communication*: we build *a data exchange manager*, which collects, merges and then distributes data from and to all nodes.
4) *Downward evaluation*: each node performs the downward pass and final L-expansions of its receiver points.
5) *Local direct sum*: each node performs this evaluation independent of the translations.

The costs of partitions in Step (1) depend on the applications. For example, in many dynamics problems, from one time step to the next, it is very likely that most particles still reside on the same node, in which case the new partition may only require a small communication with other nodes. In this paper, however, we assume the worst case that all data is assigned initially on each node randomly, which involves a large amount of particle data exchange. Based on the conclusion of [1], we perform Step (5) on the GPU, while Steps (2)–(4) are performed on the CPU in parallel.

A. Distributed data structures

The FMM data structure is based on data hierarchies for both sources and receivers; Figure 4 shows one. This can be viewed as a forest of K trees with roots at level 2 and leaves at level l_{max}. In the case of uniform data and when the number of nodes $K \leq 64$ (for the octree) each node may handle one or several trees. If the number of nodes is more than 64 and/or data distributions are substantially non-uniform, partitioning based on the work load balance should be performed by splitting the trees at a level > 2. Such a partitioning can be thought of as breaking of some edges of the initial graph. This increases the number of the trees in the forest, and each tree may have a root at an arbitrary level $l = 2, ..., l_{max}$. Each node then takes care for computations related to one or several trees according to a split of the workload. We define two special octree levels:

- **Partition level** l_{par}: At this level, the whole domain is partitioned among different nodes. Within each node,

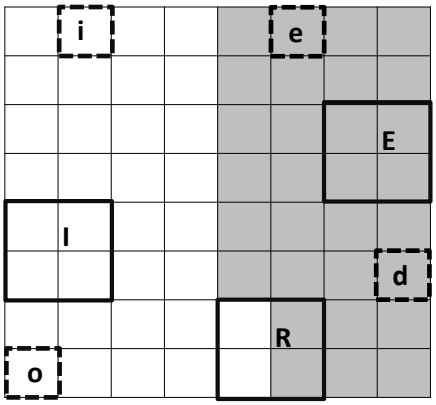

Figure 5. An example of source box types. White boxes are Partition I and gray boxes are Partition II. The partition level is 3 and the critical level is 2. Solid line boxes correspond to level 2 and dash line boxes correspond to level 3. At partition II, box $\mathbf{e}$ and $\mathbf{E}$ are *export boxes*. Boxe $\mathbf{i}$ and $\mathbf{I}$ are *import boxes*. Box $\mathbf{R}$ is a *root box*. Box $\mathbf{d}$ is a *domestic box*. Box $\mathbf{o}$ is an *other box*.

all the subtrees at this level or below are totally complete, i.e., no box at level $\geq l_{par}$ is on other nodes.

- **Critical level** $l_{crit} = \max(l_{par} - 1, 2)$: At this level, all the box coefficients are broadcasted such that all boxes at level $\leq l_{crit}$ can be treated as local boxes, i.e., all the box coefficients are complete after broadcasting.

Note that the partition level is usually quite low for 3D problems, e.g., $l_{par} = 2, 3, 4$, therefore only a small amount of data exchange is needed to broadcast the coefficients at the critical level l_{crit}. To manage the data exchange, we classify source boxes at all levels into five types. For any node, say J, these five box types are:

1) **Domestic Boxes**: The box and all its children are on J. All domestic boxes are organized in trees with roots located at level 1. All domestic boxes are located at levels from l_{max} to 2. The roots of domestic boxes at level 1 are not domestic boxes (no data is computed for such boxes).
2) **Export Boxes**: These boxes need to send data to other nodes. At l_{crit}, the M-data of export boxes may be incomplete. At level $> l_{crit}$, all export boxes are domestic boxes of J and their M-data are complete.
3) **Import Boxes**: Their data are produced by other computing nodes for importing to J. At l_{crit}, the M-data of import boxes may be incomplete. At level $> l_{crit}$, all import boxes are domestic boxes of nodes other than J and their M-data are complete there.
4) **Root Boxes**: These are boxes at critical level, which need to be both exported and imported. For level $> l_{crit}$ there is no root box.
5) **Other Boxes**: Boxes which are included in the data structure but do not belong to any of the above types, e.g. all boxes of level 1, and any other box, which for some reason is passed to the computing node (such boxes are considered to be empty and are skipped in computation, so that affects only the memory and amount of data transferred between the nodes).

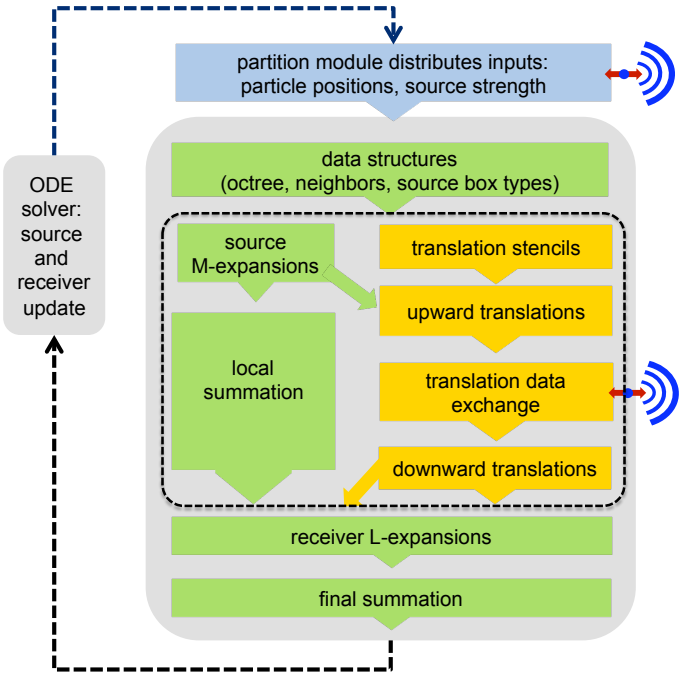

Figure 6. The distributed FMM algorithm. The blue part is the data partition module. The green parts of the algorithm is computed by using the GPU while yellow parts are computed by using the CPU. Two red double ended arrows represent the communication with other nodes. The rectangle with dash lines represents the concurrent region.

Refer Fig. 5 for an example. Note that there are no import or export boxes at levels from $l_{crit} - 1$ to 2. All boxes at these levels are either domestic boxes or other boxes after the broadcast and summation of incomplete M-data at l_{crit}. In our algorithm, we only need compute M-data and box types from level l_{max} to l_{crit} and exchange the information at l_{crit}. After that we compute the M-data for all the domestic boxes up to level 2 then produces L-data for all receiver boxes at level l_{max} handled by the computing node.

B. The distributed FMM algorithm

In our FMM algorithm, all the necessary data structures, such as octree and neighbors, and particle related information, are computed on the GPU using the efficient parallel methods presented in [1], [17]. Details of the algorithms and implementations are not reported here. Given the global partition, how the algorithm appears on a single node is illustrated in Fig. 6. Assume that all the necessary data structures for the translations, such as box's global Morton indices, the neighbor, export/import box type lists and those

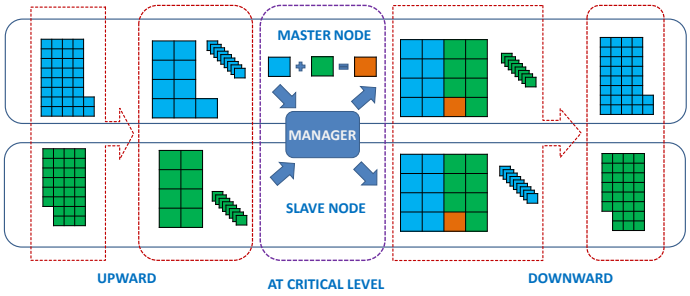

Figure 7. A simple multiple node 2D FMM algorithm illustration ($l_{crit} = 2$) : the top rounded rectangular is the MASTER NODE which is also chosen as the data exchange manager. The bottom rounded rectangular is the SLAVE NODE. From left to right is how the algorithm proceeds. Each node perform upward M-translations from l_{max} to l_{crit}. At critical level the manager collect and distribute data from and to all the nodes. Those isolated boxes in this figure are import/export boxes. After this communication with master, all the nodes perform downward M2L and L2L translations only for its own receiver boxes.

initial M-expansion data, are available, then each node J executes the following translation algorithm:

1) **Upward translation pass**:
   a) Get M-data of all domestic source boxes at l_{max} from GPU global memory.
   b) Produce M-data for all domestic source boxes at levels $l = l_{max} - 1, \ldots, \max(2, l_{crit})$.
   c) Pack export M-data, the import and export box indices of all levels. Then send them to the data exchange manager.
   d) The master node, which is also the manager, collects data. For the incomplete root box M-data from different nodes, it sums them together to get the complete M-data. Then according to each node's emport/import box indices, it packs the corresponding M-data then sends to them.
   e) Receive import M-data of all levels from the data exchange manager.
   f) If $l_{crit} > 2$, consolidate S-data for root domestic boxes at level l_{crit}. If $l_{crit} > 3$, produce M-data for all domestic source boxes at levels $l = l_{crit} - 1, \ldots, 2$.
2) **Downward translation pass**:
   a) Produce L-data for all receiver boxes at levels $l = 2, \ldots, l_{max}$.
   b) Output L-data for all receiver boxes at level l_{max}.
   c) Redistribute the L-data among its own GPUs.
   d) Each GPU finally consolidates the L-data, add the local sums to the dense sums and copy them back to the host according to the original inputting receiver's order.

Here all boxes mean all boxes handled by each node. A simple illustration of this algorithm for a 2D problem is shown in Fig. 7.

Algorithm 1 Compute source box type on the node J

```
Input: a source box index BoxIndex[i]= k at level l
Output: BoxType[i]
isOnNode←isImportExport←isExport←FALSE
if l<l_crit then
   BoxType[i]←DOMESTIC
else if l=l_crit then
   for any k's child c_i at partition level do
      if c_i is not on J then
         isImportExport←TRUE
      else
         isOnNode←TRUE
   if isOnNode=FALSE then
      BoxType[i]←IMPORT
   else
      for any k's neighbor of M2L translation n_i do
         if one of n_i's children at l_crit is not on J then
            isExport←TRUE
            // update the type of a different box
            n_i's box type ←IMPORT
else
   if k's ancestor at l_crit is not on J then
      BoxType[i]←OTHERS
   else
      for any k's neighbor of M2L translation n_i do
         if the ancestor of n_i at l_crit is not on J then
            isExport←TRUE
            // update the type of a different box
            n_i's box type ←IMPORT
synchronize all threads
if isImportExport=TRUE then
   BoxType[i]←ROOT
else if isExport=TRUE then
   BoxType[i]←EXPORT
else
   BoxType[i]←DOMESTIC
```

Since each node only performs translations for its own assigned spatial domains, i.e. octree boxes, there are no repeated translation operations, therefore, this algorithm is truly distributed. The amount information exchanged with the manager is actually small because only boxes on the partition boundary layers (2D surface in contrast to 3D space) need to be sent/received.

C. Algorithm to assign source box type

The type of a source box $\mathbf{k}$ is determined by the M2M and M2L translation because its children or neighbors might be missing due to the data partition, which have to be requested from other nodes. However, once the parent box M-data is complete, the L2L translations for its children are always complete. Hence, based on this observation, we can

summarize the key idea of Alg. 1, which computes the type of each box, as follows:

- At the critical level, we need all boxes to perform upward M2M translations. If one child is on a node other than J, its M-data is either incomplete or missing, hence we mark it an import box. We also check its neighbors required by M2L translation stencil. If any neighbor is not on J, then the M-data of these two boxes have to be exchanged.
- For any box at the partition level or deeper levels, if this box is not on J, then it is irrelevant to this node, in which case it is marked as other box. Otherwise we check all its neighbors required by M2L translations. Again if any neighbor is not on J, these two boxes' M-data have to be exchanged.

We compute all box types in parallel on the GPU. For each level from l_{max} to 2, a group of threads on the node J are spawned and each thread is assigned by one source box index at that level. After calling Alg. 1, all these threads have to be synchronized before the final box type assignment in order to guarantee no race conditions. Note that some "if-then" conditions in Alg. 1 can be replaced by `OR` operations so that thread "divergent branches" can be reduced.

D. Communication cost

We cannot avoid moving $O(N+M)$ data points if the initial data on each node are distributed randomly. However, in many applications, this cost can be avoided if the data distribution is known. Also many papers discuss initial partition (tree generation) and communications such as [11], [13], [18], which can be used.

Let the total number of source/receiver boxes (non-empty) at all levels be B_{src}/B_{recv}. Because of data partition, each node roughly has B_{src}/P source boxes and B_{recv}/P receiver boxes. There are no simple expressions for the cost of sending export boxes and receiving import boxes, and the cost of the master node finding and packing M-data for each node. However, the boundary layers are nothing but the surface of some 3D object. Thus, it is reasonable to estimate its box number as $\mu B_{recv}^{2/3}$ with some constant μ for all the nodes (as for example, [10]). Thus at l_{crit}, the total cost of communication can be estimated as:

$$T_{comm} = a_0 \frac{B_{src}}{P} + a_1 \mu \left(\frac{B_{recv}}{P} \right)^{2/3} p^2 P + a_2 8^{l_{crit}} p^2 \log P. \quad (3)$$

Each term of Eq. (3) is explained in Table I. Since l_{crit} is usually small and MPI broadcasts data efficiently, the last term can be treated as a small constant. Even though our communication cost is asymptotically proportional to $P^{1/3}$, its asymptotic constant is very small such that the overall cost can be neglected compared with kernel evaluation.

$a_0 \frac{B_{src}}{P}$	each node examines its source box types and extract import and export source box indices
$a_1 \mu \left(\frac{B_{recv}}{P} \right)^{\frac{2}{3}} p^2$	each node exchanges boundary box's M-data with the master node; the total cost is its P times
$a_2 8^{l_{crit}} p^2 \log P$	the master node broadcasts M-data at the critical level ($\log P$ is due to the MPI broadcast)

Table I
DESCRIPTION OF EQUATION (3)

IV. PERFORMANCE TEST

A. Hardware

We used a small cluster ("Chimera") at the University of Maryland (32 nodes) to perform tests. The basic node architecture was interconnected via Infiniband. Each node was composed of a dual socket quad-core Intel Xeon 5560 2.8 GHz processors, 24 GB of RAM per node, and two Tesla C1060 accelerators each with 4 GB of RAM.

B. Single heterogeneous node

On a single node, we test the present algorithm and compare with the performance reported in [1]. We run the same tests by using both spatially uniform random particle distributions and non-uniform distributions (points or the sphere surface). As expected, the present algorithm is more or less the same as [1] on a single node with the similar behavior hence we won't repeat the algorithm analysis. Test results are summarized in Tables II and III, which show comparable performance.

	Prec	p = 4	8	12	16
Time	S	0.36	0.39	0.69	1.45
(sec)	D	0.97	1.02	1.13	1.68
Error	S	$3.4\cdot(-4)$	$1.2\cdot(-6)$	$3.7\cdot(-7)$	$1.3\cdot(-7)$
	D	$3.3\cdot(-4)$	$9.4\cdot(-7)$	$4.1\cdot(-8)$	$4.6\cdot(-9)$

Table II
PERFORMANCE AND ACCURACY [1]

C. Multiple node algorithm test

In the multiple heterogeneous node tests, we varied the numbers of particles, nodes, GPUs per node, and the depth of the space partitioning to derive optimal settings, taking into account data transfer overheads and other factors.

As in [1], we repeat the time taken for the "concurrent region", where the GPU(s) computes local summation while the CPU cores compute translation. We test the weak scalability of our algorithm by fixing the number of particles per node to $N/P = 2^{23}$ and varying the number of nodes (see Table 4). In Fig. 8, we show our overhead vs. concurrent region time against the results in [1]. For

[1] The GPU computation uses either single or double precision. Here$(-m)$ means 10^{-m}. Tests are performed for potential computation for $N = 2^{20}$ on a workstation with a NVIDIA Tesla C2050 GPU with 3 GB, and two dual core Intel Xeon X5260 processors at 3.33 GHz and 8 GB RAM.

Time (s) \ N	1,048,576		2,097,152		4,194,304		8,388,608		16,777,216	
Num of GPUs	1	2	1	2	1	2	1	2	1	2
CPU wall clock	0.13	0.13	1.09	1.08	1.08	1.08	1.06	1.08	8.98	1.08
C/G concurrent region	0.55	0.28	1.09	1.08	1.24	1.08	4.24	2.14	8.98	9.23
Force+Potential total run	0.68	0.37	1.31	1.20	1.62	1.30	4.95	2.52	10.75	10.00
Potential total run	0.39	0.22	1.21	0.83	1.32	1.22	2.83	1.44	10.08	5.74
Partitioning	–	0.20	–	0.47	–	0.85	–	1.69	–	3.25

Table III
PERFORMANCE ON A SINGLE HETEROGENEOUS NODE[2]

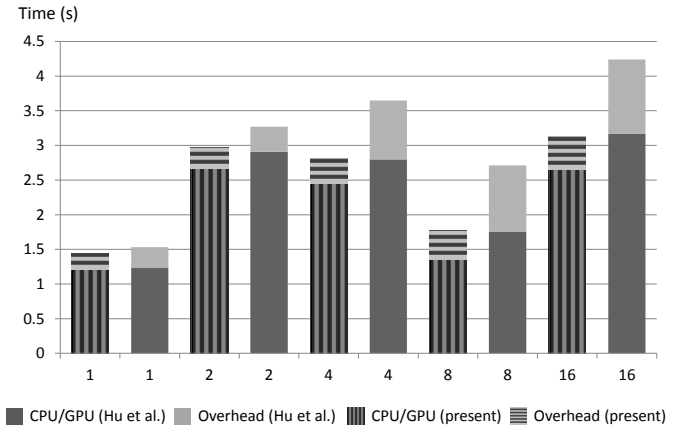

Figure 8. The CPU/GPU concurrent region time and the overhead (data transfer between the nodes and CPU/GPU sequential region) in the present implementation for 2 GPUs per node against the results in [1]. The testing case size increases proportionally to the number of nodes (8M particles per node). The time is measured for computations of the potential.

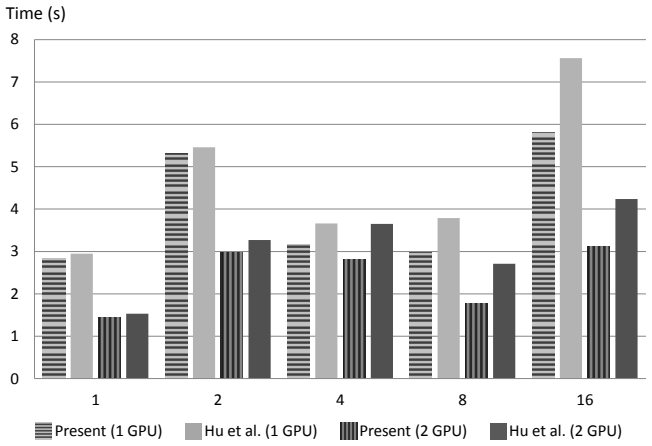

Figure 9. The time comparisons between the present algorithm and [1]. The testing case has 8M particles and run on 1, 2, 4, 8, 16 nodes using 1 or 2 GPUs on each node.

perfect parallelization/scalability, the run time in this case should be constant. In practice, we observed an oscillating pattern with slight growth of the average time. In [1], two factors were explained which affect the perfect scaling: reduction of the parallelization efficiency of the CPU part of the algorithm and the data transfer overheads, which also applies to our results. We distribute L2L-translations among nodes and avoid the unnecessary duplication of the data structure, which would become significant at large sizes. Since our import/export data of each node only relates to the boundary surfaces, we improve the deficiency of their simplified algorithm that also shows up in the data transfer overheads, which increases with $l_{\max}$. In Fig. 8, our algorithm shows almost the same parallel region time for the cases with similar particle density (the average number of particles in a spatial box at l_{max}). Moreover, the overheads of our algorithm only slightly increases in contrast to the big jump seen in [1] when l_{max} changes. Even though the number of particles on each node remains the same, the problem size increases hence results in the deeper octree and more spatial boxes to handle, which also contributes to such overhead increase (besides communication cost). As for the total run time comparisons, we summarize the improvements in Fig. 9. Generally speaking, as the problem size and octree depth increase, our algorithm shows substantial savings against [1], which implies the much improved weak scalability.

We also performed the strong scalability test, in which N is fixed and P is changing (Fig. 10). The tests were performed for $N = 2^{23}, 2^{24}$ and $P = 1, 2, 4, 8, 16$ with one and two GPUs per node. Even though, our algorithm demonstrates superior scalability compared with [1], we still observe the slight deviations from the perfect scaling for the 8M case. For 16M case, the total run time of both 1 and 2 GPU shows the well scaling because the GPU work was a limiting factor of CPU/GPU concurrent region (the dominant cost). This is consistent with the fact that the sparse MVP alone is well scalable. For 8M case, in the case of two GPUs, the CPU work was a limiting factor for the parallel region. However, we can see approximate correspondence of the times obtained for two GPUs/node to

[2]The tests are performed on a single node of Chimera. We evaluate both force and potential. We report the time profiling for force but only the total time for potential.

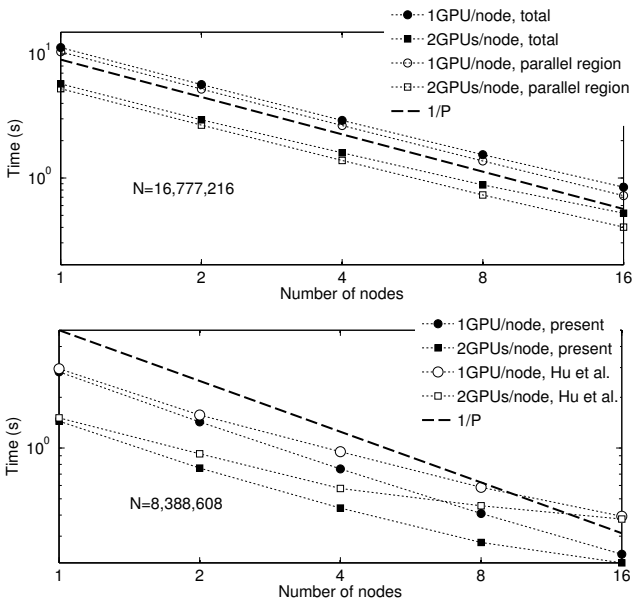

Figure 10. The results of the strong scalability test for 1 and 2 GPUs per node of the testing cluster. The thick dashed line shows perfect scalability $t = O\left(1/P\right)$. The time is measured for potential only computations. In the top figure, it shows the present algorithm's performance and the problem size is fixed to be 16M running on 1 to 16 nodes. In the bottom figure, it shows the strong scalability comparison (total run time) between the present algorithm and the [1]. The problem size is fixed to be 8M running on 1 to 16 nodes.

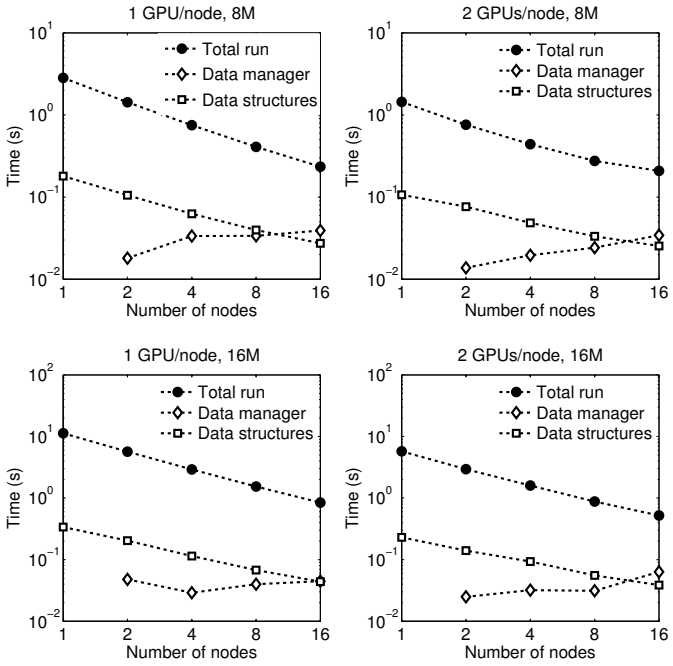

Figure 11. The data manager and data structure processing time against the total run time. The problem sizes are fixed to be 8M (top) and 16M (bottom) running on 1 to 16 nodes. Each node uses 1 (left) or 2 (right) GPUs. The time is measured for potential only computations.

the ones with one GPU/node, i.e. doubling of the number of nodes with one GPU or increasing the number of GPUs results in approximately the same timing. This shows a reasonably good balance between the CPU and GPU work in the case of 2 GPUs per node, which implies this is more or less the optimal configuration for a given problem size.

To validate the reduced cost of our communication scheme and the computation of box type, we compare the data manager processing time including M-data exchange time and the overall data structure construction time with the total running time in Fig. 11. Given the problem size and truncation number fixed, our communication increases as the number of nodes (roughly $P^{1/3}$ in Eq. 3). In our strong scalability tests, such time is in the order of 0.01 seconds while the wall clock time is in the order of 1 or 0.1 seconds (contribute $1\% \sim 15\%$ of overall time), even though GPUs are not fully occupied in some cases. This implies such cost can be neglected in larger problems, in which the kernel evaluations keep all GPUs fully loaded. Our implementation incorporate the box type computation with other data structures, such as octree and translation neighbors, hence it makes more sense to report the total data structure cost. From Fig. 11 we observe that our data structure time decrease similarly as the wall clock time (as $1/P$) and shows good strong scalability.

We did similar tests on the cluster to match their billion case performance. In Fig. 12, we only show the best results of the optimal settings when the GPU(s) reach their peak performance up to $N = M = 2^{28}$. For larger problems a suboptimal performance is obtained because it requires more nodes by using the optimal particle size; however, such cases are still of practical importance, as solving billion-size problems in terms of seconds per time step is quite useful and shows the substantial improvements by compared the latest best results on heterogeneous clusters reported in [1]. Fig. 12 presents the results of the run employing 32 nodes with one or two GPUs per node. Note that the overhead increases also linearly as the problem size and is a small part of the overall time. The largest case computed in the present study is $N = 2^{30}$ for 32 two-GPU nodes. For this case, the CPU/GPU concurrent region time was 10.3 s and the total run time 12.2 s (in contrast 12.5 s and 21.6 s in [1]). We believe that the achieved total run times are among the best ever reported for the FMM for the sizes of the problems considered (e.g, comparing with [1], [9], [12], [19], [20]).

Using the same method of performance counting as [1], we can estimate the single heterogeneous node performance for two GPUs as 1243 GFlops and the 32 nodes cluster with two GPUs each as 39.7 TFlops. To evaluate the performance count, we use the same method as [1] by looking at the actual number of operations performed on the GPU during

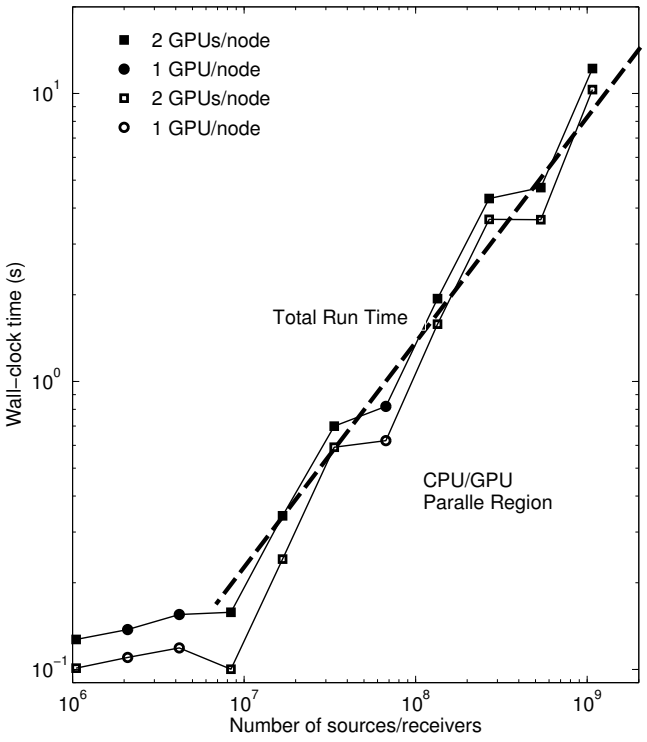

Figure 12. The wall clock time for the heterogeneous FMM running on 32 nodes and using 1 or 2 GPUs per node (potential only computations). The number of GPUs per node is optimized according to the load balance between CPU and GPU(s) in the concurrent region. The thick dashed lines show the linear scaling. Solid and hollow symbols are for the total and concurrent region time respectively.

the sparse MVP. Given the number of operations per direct evaluation and summation of the potential or potential+force contribution, which value 27 is the commonly accepted, our present algorithm provides 612 GFlops (933 GFlops peak performance reported by NVIDIA). For the contribution of the CPU part of the algorithm in the concurrent region, we count all M2M, M2L, and L2L translations, evaluated the number of operations per translation, and used the measured times for the GPU/CPU parallel region at $l_{\max} = 5$. That provided as a estimate of 28 GFlops per node for 8 CPU cores. Alternatively, following [9], [12], [21], [22], we could also use a fixed flop count for the original problem and computed the performance that would have been needed to achieve the same results via a "brute-force" computation. For $N = 2^{30}$ sources/receivers and a computation time of 12.2s (total run), which we observed for 32 nodes with two GPUs per node, the brute-force algorithm would achieve a performance of **2.21 Exaflops**.

V. Conclusion

In the light of recent heterogenous FMM algorithms developments, our presented algorithmic improvements substantially extend the scalability and efficiency of the FMM algorithm for large problems on heterogenous clusters. Our algorithm provides both fast data structure computations and efficient CPU/GPU work split, which can best utilize the computation resources and achieve the state of the art performance. With the present algorithm, dynamic problems in the order of millions or billions from fluid mechanical applications, molecular dynamics or stellar dynamics can be efficiently solved within a few seconds per time step on a single workstation or small heterogeneous cluster.

We demonstrate the weak and strong scalability improvement compared with algorithms presented in [1]. Both our single node algorithm and distributed algorithm using novel data structures are shown to outperform those results in all cases. A FMM run for $2^{30} \gtrsim 1$ billion particles is successfully performed with this algorithm on a midsize cluster (32 nodes with 64 GPUs) in 12.2 seconds.

The data structures developed here can handle non-uniform distributions and achieve workload balance. In fact, our algorithm splits the global octree among all the nodes and processes each subtree independently. Such a split can be treated as an isolated module which is free to use different methods based on different applications, to estimate workload. Moreover, since each node constructs its own subtree independently, the limitation of the depth of octee constructed by GPU only applies to the local tree, which implies such algorithm can handle deeper global octrees. Our approach using import and export box concepts only exchange necessary box data hence substantially reduces the communication overheads. We develop parallel algorithms to determine the import and export boxes in which the granularity is spatial boxes. Their parallel GPU implementations are shown to have very small overhead and good scalability.

Acknowledgements

Work partially supported by Fantalgo, LLC; by AFOSR under MURI Grant W911NF0410176 (PI Dr. J. G. Leishman, monitor Dr. D. Smith); We acknowledge NSF award 0403313 and NVIDIA for the Chimera cluster at the CUDA Center of Excellence at UMIACS.

References

[1] Q. Hu, N. A. Gumerov, and R. Duraiswami, "Scalable fast multipole methods on distributed heterogeneous architectures," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 36:1–36:12.

[2] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, August 2007, ch. 31, pp. 677–695.

Time (s) \ N (P)	8,388,608 (1)		16,777,216 (2)		33,554,432 (4)		67,108,864 (8)		134,217,728 (16)	
Num of GPUs	1	2	1	2	1	2	1	2	1	2
CPU wall	1.07	1.07	4.71	0.60	2.53	2.44	1.39	1.35	0.83	0.81
CPU/GPU	2.38	1.20	4.71	2.66	2.53	2.44	2.43	1.35	5.22	2.65
Overhead	0.45	0.24	0.59	0.30	0.64	0.37	0.54	0.42	0.58	0.47
Total run	2.83	1.44	5.30	2.96	3.17	2.81	2.97	1.77	5.80	3.12

Table IV
PERFORMANCE FOR P HETEROGENEOUS NODES WITH $N/P = 2^{23}$ (POTENTIAL ONLY).

[3] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *J. Comput. Phys.*, vol. 73, pp. 325–348, December 1987.

[4] P. B. Callahan and S. R. Kosaraju, "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *J. ACM*, vol. 42, pp. 67–90, January 1995.

[5] L. Greengard and W. D. Gropp, "A parallel version of the fast multipole method," *Computers Mathematics with Applications*, vol. 20, no. 7, pp. 63–71, 1990.

[6] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta, "A parallel adaptive fast multipole method," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 54–65.

[7] J. P. Singh, J. L. Hennessy, and A. Gupta, "Implications of hierarchical n-body methods for multiprocessor architectures," *ACM Trans. Comput. Syst.*, vol. 13, pp. 141–202, May 1995.

[8] S.-H. Teng, "Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation," *SIAM J. Sci. Comput.*, vol. 19, pp. 635–656, March 1998.

[9] R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, and K. Yasuoka, "Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence," *Computer Physics Communications*, vol. 180, no. 11, pp. 2066–2078, 2009.

[10] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 58:1–58:12.

[11] L. Ying, G. Biros, D. Zorin, and H. Langston, "A new parallel kernel-independent fast multipole method," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 14–30.

[12] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, "42 tflops hierarchical n-body simulations on GPUs with applications in both astrophysics and turbulence," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 1–12.

[13] F. A. Cruz, M. G. Knepley, and L. A. Barba, "PetFMM dynamically load-balancing parallel fast multipole library," *Int. J. Numer. Meth. Engng.*, vol. 85, no. 4, pp. 403–428, 2010.

[14] R. Yokota, T. Narumi, L. A. Barba, and K. Yasuoka, "Petascale turbulence simulation using a highly parallel fast multipole method," *ArXiv e-prints*, Jun. 2011.

[15] N. A. Gumerov and R. Duraiswami, "Comparison of the efficiency of translation operators used in the fast multipole method for the 3D Laplace equation," University of Maryland Department of Computer Science and Institute for Advanced Computer Studies, Tech. Rep. CS-TR-4701, UMIACS-TR-2005-09, April 2005.

[16] N. A. Gumerov and R. Duraiswami, "Fast multipole methods on graphics processors," *J. Comput. Phys.*, vol. 227, no. 18, pp. 8290–8313, 2008.

[17] Q. Hu, M. Syal, N. A. Gumerov, R. Duraiswami, and J. G. Leishman, "Toward improved aeromechanics simulations using recent advancements in scientific computing," in *Proceedings 67th Annual Forum of the American Helicopter Society*, May 3–5 2011.

[18] S.-H. Teng, "Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation," *SIAM J. Sci. Comput.*, vol. 19, pp. 635–656, March 1998.

[19] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc, "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures," *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, pp. 1–12, 2010.

[20] A. Chandramowlishwarany, K. Madduri, and R. Vuduc, "Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12.

[21] M. S. Warren, T. C. Germann, P. Lomdahl, D. Beazley, and J. K. Salmon, "Avalon: An alpha/linux cluster achieves 10 gflops for $150k," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, 1998, pp. 1–10.

[22] A. Kawai, T. Fukushige, and J. Makino, "$7.0/mflops astrophysical n-body simulation with treecode on grape-5," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '99. New York, NY, USA: ACM, 1999, pp. 1–6.