2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Dynamic Load Balancing of the Adaptive Fast Multipole Method in Heterogeneous Systems

Robert E. Overman, Jan F. Prins
Dept. of Computer Science
UNC Chapel Hill, USA
{reoverma,prins}@cs.unc.edu

Laura A. Miller
Dept. of Mathematics
UNC Chapel Hill, USA
lam9@amath.unc.edu

Michael L. Minion
Lawrence Berkeley National Lab
Berkeley, CA
mlminion@lbl.gov

***Abstract*—Simulations of colliding galaxies or fluid dynamics at immersed flexible boundaries are most accurately and efficiently accomplished using the adaptive fast multipole method (AFMM) to solve an underlying n-body problem whose localized density varies with the time-dependent evolution of the system under study. Parallelization of the AFMM presents a challenging load balancing problem that must be addressed dynamically as the system evolves. We consider parallelization of the AFMM for time-dependent problems using a heterogeneous shared memory compute node consisting of multi-core processors and GPU accelerators. OpenMP task parallelism is used within the CPU cores to parallelize the construction and maintenance of the adaptive spatial decomposition tree and its traversal to compute far-field interactions at each leaf node in the tree. Concurrently, GPUs evaluate all near-field interactions using all-pairs computations. In addition to accurately resolving many physical phenomena out of reach using the uniform FMM, the more complex AFMM permits the number of bodies in leaf cells to be globally and locally varied in order to minimize the CPU and GPU time. We present a cost model and incremental adjustment strategy to load balance the AFMM on a heterogeneous system. We demonstrate using these techniques that a simulation can maintain load balance over hundreds of time steps on a heterogeneous system with 10 CPU cores and 4 GPUs with less than 2% overhead, while achieving a 98X speedup over a serial computation using a single CPU core.**

Keywords – adaptive fast multipole method; dynamic load balancing; hybrid computing; accelerators; OpenMP task parallelism; CUDA.

I. INTRODUCTION

The Fast Multipole Method (FMM) was introduced by Rokhlin and Greengard as an $O(N)$ time solution for an N-body problem [1]. The FMM has been widely adopted due to the large asymptotic advantage it offers over the $O(N^2)$ all-pairs method while simultaneously providing bounded precision in a manner more difficult to achieve using Barnes-Hut style methods. It is used in a wide variety of problems in astrophysics, molecular dynamics, fluid dynamics, and electrostatics [1, 2, 3].

A. Uniform Spatial Decomposition

The original 3D FMM [1] uses a fixed-depth octree decomposition of space. Fig. 1 shows a fixed-depth quadtree, the 2D analog of the octree. The underlying assumption for the FMM is that the distribution of bodies in the problem is relatively uniform so that all leaves in the octree hold approximately the same number of bodies. For a uniform 3D spatial decomposition the depth d of the octree is then given by $d = \lceil \log_8 (N/S) \rceil$ where N is the number of bodies in the system and S is a target number of bodies per leaf cell. Since the octree is complete and all leaf nodes have the same size, the FMM has a statically determined computational structure that simplifies parallelization [4, 5]. However when the FMM is used with a uniform spatial decomposition to solve an N-body problem for a non-uniform distribution of bodies, the actual number of bodies in each leaf node will vary and in the extreme may drive the work complexity of the algorithm to $O(N^2)$. For physical systems such as galaxies or plasmas, the local density of bodies may vary by many orders of magnitude, rendering them unsuitable for simulation using the FMM.

B. Adaptive Spatial Decomposition

The Adaptive FMM developed by Cheng, Greengard and Rokhlin [6] varies the spatial decomposition with the local density of bodies. Fig. 2 shows the 2D analog of this. The AFMM builds a variable depth octree decomposition of space in which a node is subdivided into eight children if it holds more than S bodies. In this decomposition leaf nodes may occur at any level in the octree, and the tree will in general have varying depth.

C. The Fast Multipole Method

The basic operations of the two methods are the same, regardless of the spatial decomposition. The method begins by computing a multipole expansion for the bodies in each leaf cell by use of the Particle-to-Multipole (P2M) operation. The method then proceeds upwards in the octree, combining expansions from children into a single expansion centered on the parent by application of the

Multipole-to-Multipole (M2M) operation. In the down sweep phase the method starts with a local expansion at the root and proceeds downwards in the tree, converting the parent local expansion to expansions centered on each of its children by use of the Local-to-Local (L2L) operation. In addition multipole expansions of well-separated nodes are converted and combined into the local expansion for each node using the Multipole-to-Local (M2L) operation. Upon arriving at a leaf node r, the local expansion of r gives the total *far-field* interaction experienced by bodies in r due to bodies in nodes of the tree well-separated from r. The far-field interactions are applied to the bodies in r using the Local-to-Particle (L2P) operation. It remains to incorporate the interaction with the remaining bodies in the *near-field* of *r*. This is performed using all-pairs computation between the bodies in r and all bodies in nodes that are not well-separated from r using the Particle-to-Particle (P2P) operation.

A key point about the six operations (P2M, M2M, M2L, L2L, L2P, and P2P) is that each has a predictable cost in FLOPS that can be expressed in terms of the number of bodies in a leaf node, and the number of retained terms p in the multipole expansion [7] [8].

The difference between the AFMM and the FMM is that the set of nodes involved in each of the operations is specific to the tree structure; hence parallelization of the AFMM cannot leverage the fixed computational structure of the FMM problem.

It is important to note that even an initially uniform distribution may become non-uniform over the course of many time steps, depending on the forces at work in the model. In this case the FMM can become increasingly inefficient when many

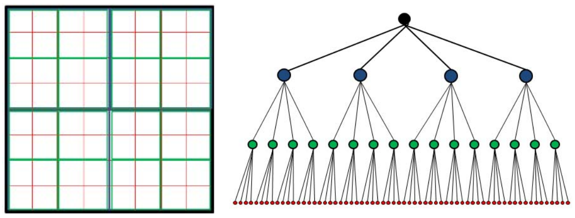
Fig. 1. 2D uniform depth decomposition of space utilizing a quadtree.

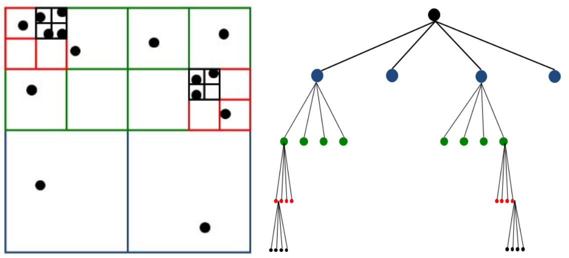
Fig. 2. Adaptive decomposition of space in 2D resulting in leaf nodes of varying depth.

bodies accumulate in specific nodes. Adaptive FMM implementations do not suffer from this problem since they may rebuild the tree so that all leaf nodes once again have a bounded size. Alternatively the tree can be adjusted incrementally in response to observed changes in the loading of nodes. Our implementation uses both methods to maintain a high degree of efficiency for the AFMM in non-uniform or evolving distributions.

II. PROBLEMS ADDRESSED

Adaptive algorithms are challenging to parallelize efficiently and in the case of time-dependent problems, dynamic load balancing may be required. Modern computing node architectures with multicore CPUs and computational accelerators present additional challenges in delegating different aspects of the algorithmic work to the devices most suited to perform them. To address these issues the following contributions are presented in this paper:

- We implement the adaptive FMM rather than the uniform FMM within a single heterogeneous multicore compute node with one or more Nvidia GPUs. Similar to [9, 7] we perform the far-field (expansion) work on the CPU which has larger caches and is better able to deal with complex control paths, while performing near-field (direct) work with very high performance on one or more GPUs. We obtain excellent performance in the far-field work by utilizing OpenMP tasking directives within the sequential AFMM algorithm and in adaptive decomposition construction and maintenance. We also obtain high performance on the near-field work by an efficient implementation of multiple all-pairs computations using one or more GPUs.

- We present a dynamic load balancing scheme for time dependent applications of the AFMM which minimizes runtime on heterogeneous systems composed of multiple CPUs and GPUs across multiple time steps. The load balancing strategy performs fine grain local modifications to the adaptive decomposition tree to minimize runtime informed by a time costing model. In addition, incremental global modifications track the evolving distribution of bodies.

Several GPU-only strategies have been developed for the FMM, including [12, 11, 10]. We have followed an approach utilizing both CPUs and GPUs because the AFMM can benefit from the considerable performance potential of multicore CPUs particularly in adaptive tree construction and maintenance. In addition OpenMP tasking is simple, and highly efficient even for highly non-uniform distributions. It is an improvement over the use of nested parallel regions that are complicated and hard to update in a time-dependent simulation [9]. While we focus on a single node implementation, we expect the method can be extended to a distributed memory cluster using techniques such as those in [13, 9].

III. Heterogeneous Design

A. Overview

The key feature of this heterogeneous design is placing the direct work on the GPUs. We carry out the far-field work (P2M, M2M, M2L, L2L and L2P) on the CPUs.

The basic parameter to balance the load is the value of S since it shifts work between far-field and near-field computation and hence between CPUs and GPUs, as shown in Fig 3. Our tree modification scheme also addresses a significant performance issue that arises with load balancing a non-adaptive FMM that we call the "Uniform Gap", shown in Fig. 4. Since the tree depth is equal everywhere, a uniform 3D spatial decomposition increases the number of leaves by a factor of 8 whenever N/S exceeds a critical value. For this reason small changes in S may yield large discontinuities in the cost of near-field and far-field work, corresponding

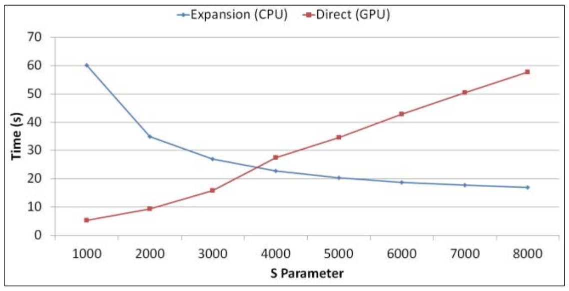

Fig. 3. Adaptive distributions result in a gradual change in the cost of the CPU and GPU work as a function of S.

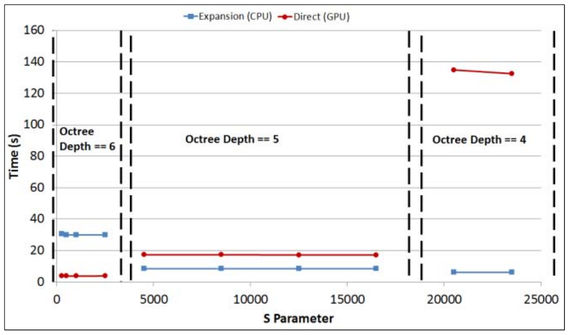

Fig. 4. Three distinct cost regimes are obtained when varying S using a uniform decomposition corresponding to different depths of the octree., making it difficult to accurately balance load using a uniform decomposition.

to removing or adding entire levels of the octree, as shown in Fig. 1. Adaptive octrees can give better results in this case by expanding only some nodes to a greater depth, compared with a uniform decomposition which would have to choose exclusively between the different levels.
Through use of the load balancing operations which we employ, we will show that we can successfully bridge the gap between tree levels and more evenly distribution work amongst the expansion and direct work phases.

B. CPU Parallelism

As mentioned earlier, one goal of this implementation was a simple yet dynamic CPU parallelism technique for highly non-uniform octrees. CPU parallelism is accomplished through the use of OpenMP tasking facilities and work is carried out parallel in space. The core OpenMP directives used are "#pragma omp task" to spawn a task and "#pragma omp taskwait" to wait for spawned tasks to finish. Tasks are spawned within parallel regions and scheduled onto threads in that region. Each thread is bound exclusively to a specific CPU core.

The combination of tasking facilities and recursion allow for simple spatial parallelism and load balancing (by the task scheduler) on the CPU, which will be shown in the results section. The general form of this technique is as follows:

```
RecursiveFunction(Node *node):
    //Carry out computation associated with function
    DoCompute(node)

    //Recurse down the octree
    if(node->isParent):
        for each child:          //for each child spawn a task
            #pragma omp task
            RecursiveFunction(child)
        #pragma omp taskwait   //idle this task until others done
```

Upon recursing down the tree from a parent to its children, a task is spawned for each child. The task does some work (possibly by spawning more tasks) and eventually completes. The parent task is idled until all child tasks complete.

For example, our implementation has a DownSweep(Node *node) function that performs the down sweep portion of the FMM. In this case the local expansion downshift from the parent (L2L) as well as the shifting in of multipole expansions (M2L) would first be carried out for the node. Then, if the node in question is a parent node, a task would be spawned for each of its eight children to carry out the DownSweep function on each of them. Similarly, the UpSweep portion of the FMM is a head-recursive function which carries out work after children return.

This technique allows issues of spatial load balancing to be dynamically resolved at runtime and avoids messy issues of nested parallel regions and their effects on performance as other groups encountered [9]. The simplicity of this model is impressive given the complex problem being solved. Another application is construction of the spatial decomposition by a recursive parallel partition of the body locations into the child trees on the way down, and lockless construction of the adaptive spatial decomposition on the way back up.

C. GPU Parallelism

The GPU performs the direct work of the FMM programmed in CUDA. Our implementation adapts an all-pairs n-body algorithm developed in [14]. We retain the general concept of giving each thread in a warp a unique target body on which to calculate the contribution of all point sources. However, due to the adaptive nature of this implementation we must allow for leaf nodes of varying size. For target nodes this means that we simply use as many blocks as necessary to assign one thread to each target body in a target node. In blocks that have fewer bodies than threads in the block, the extra threads are idle during the computation phase. This means that we want to avoid octrees which result in a significant number of small target nodes which have a large number of sources to interact with as this will decrease the efficiency of the GPUs.

Our implementation loads sources in parallel. Each thread in a warp fetches a source particle. The threads then sync and perform the calculations, marching serially through the loaded source bodies. This procedure repeats until all sources have been loaded and used in the calculation. The result of this method is seen visually in Fig. 5. The current target node t has k particles and interacts with m distinct source nodes, each in general containing a different number of particles.

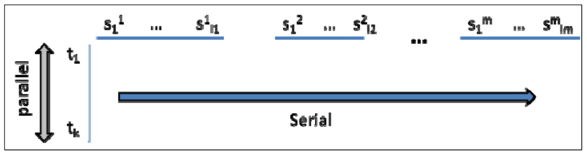

Fig. 5. P2P computations are carried out in parallel across target bodies, sequentially stepping through the source bodies in lock step. Parallel loads are performed for bodies in each target block.

In our heterogeneous model we must also efficiently distribute the direct work across the multiple GPUs on the compute node. For each target node we can calculate the number of interactions it participates in as follows:

$$\text{Interactions}(t) = \sum_{i \in D(t)} m * \text{size}(i)$$

Where $D(t)$ is the set of nodes that node t must interact with directly, $\text{size}(i)$ is the number of particles within node i and $m = \text{size}(t)$. When we are building the data to send to the GPUs, we divide up the work so that each GPU carries out approximately the same number of interactions. The implementation simply walks through the list of interaction node pairs and counts Interactions(t) for each target node. When the count meets or exceeds the total number of direct interactions divided by the number of GPUs we start counting work to send to the next GPU. Each GPU carries out the direct work for a unique set of target nodes. There is no target node whose calculations are spread out over more than one GPU. This simple division works well as shown in the results section.

D. CPU-GPU Communication

For a single FMM solve (a single time step), a single kernel is launched on each GPU to give that GPU its share of the work for the time step. While the GPU is carrying out P2P interactions, the CPU is carrying out the expansion work portion of the FMM.

There are two CPU-GPU communication functions called during a time step. Both are called from the CPU. The first function performs some initial setup work and then launches the kernels for the GPUs. This function is called from a parallel region by a single CPU thread. At the same time, the start of the FMM tree traversal is called from the same parallel region by another CPU thread. Hence CPU and GPU work begin effectively in parallel. The initial GPU call is non-blocking; after the kernels are launched, the CPU thread that invoked them can now return to pick up work spawned by the tree traversal phases.

Once all tree traversal work is done, a single CPU thread invokes the second function. This function is a blocking call. If the GPUs have finished then the calculations are immediately returned via cudaMemcpy() calls, copying the data into known locations for the CPU to utilize. If the GPU has not finished then the CPU waits until the results are ready. After the copies are done, the particle positions are updated by the CPU and then any tree optimizations are done to improve the run on the next time step.

IV. Optimization Operations

A. Overview

There are two functions used to bring the running time back to a desirable range: Collapse and PushDown.

B. Collapse Operation

The Collapse operation has the net effect of undoing the subdivision of a node into eight child nodes. It is used on parent nodes in the octree; eight leaf nodes in the tree are "removed" and the node which was previously their parent becomes a leaf node in the octree. In actuality the children are just hidden from the FMM algorithm. A flag is simply set, signaling that the old parent node is now a leaf node.

C. PushDown Operation

The PushDown operation has the opposite effect of the Collapse operation. It subdivides a given leaf node. Note that this operation has some extra expense associated with it as compared to the collapse operation. This is because pushing down may require more memory. It does happen that over time steps a node which was previously collapsed may need to be pushed down. In this case "hidden" nodes can just be reclaimed. But in general more space for nodes will be necessary and so we have some node buffer reserved in advance to minimize re-allocation.

D. Time Prediction

During the course of running the FMM, coefficients are derived for each of the major operations in the algorithm. These operations are the five core FMM expansion operations: P2M, M2M, M2L, L2L and L2P as well as the direct P2P operation. These coefficients are observational. They are not predicted values, but rather are derived from actual observed times. However, these observed coefficients will be used for prediction purposes. To derive the coefficient, for each operation, the total time spent on that operation is divided by the number of times that operation was applied. For example, on the CPU each thread keeps track of the time spent on each FMM operation and the number of times it carried out each operation. For each of these operations, the times over all threads are summed and divided by the sum of the operation count over all threads to calculate the coefficients.

Using coefficients allows for a simplification of complex details. The CPU coefficients allow us to have a single value that encompasses the collective effects of CPU speed, the number of cores, memory speed and the number of retained terms in the multipole expansions. This greatly simplifies prediction calculations.

The coefficient for the P2P operation, the only operation carried out on the GPUs, has a second use. The GPU coefficient is calculated by dividing the maximum kernel time by the total number of P2P operations carried out over all the GPUs. So this coefficient is a measure of the entire GPU system. It serves as a high level view of the efficiency of the GPUs. Unlike the operations carried out on the CPUs whose cost is a function of the fixed considerations as mentioned earlier, the P2P operation cost is a function of many low level characteristics such as the percentage of coalesced memory accesses and warp occupancy on each GPU. Every particle movement during each time step varies these low level details and changes the P2P cost in the process. Therefore this coefficient provides some insight into how efficiently the GPUs are running on the current tree.

Given these coefficients, the cost of any tree modification can be predicted. A count for the number of times each operation will be performed for the given tree is accumulated, call this count *M(Op)*. Then the following calculations are applied:

$$\text{CPU Time Prediction} = \sum_{Op \in F} C(Op) * M(Op)$$

$$\text{where} \quad F = \{P2M, M2M, M2L, L2L, L2P\}$$

$$\text{GPU Time Prediction} = C(P2P) * M(P2P)$$

$C(Op)$ is the cost coefficient for the operation Op. With these predicted times, decisions on whether or not such a tree modification would be desirable can be made without having to perform a full FMM solve on the current tree.

V. Load Balancing States

The load balancing machinery operates in one of three states: *search*, *incremental*, and *observation*. During the entire course of the simulation the load balancer is always in one of these states. Each lasts over multiple time steps. The current state of the load balancer defines how load balancing functionality is carried out and/or which actions shall be taken if undesirable run times are seen.

A. Search

The search state defines the coarsest form of optimization carried out. While in this state, after each time step a new S value is chosen and the tree is rebuilt. The particular value of S is chosen using a binary search strategy. The direction (increase or decrease) in which the new S is chosen is decided based on how tree modifications made on the previous time step influenced the compute time on the current time step. It is the slowest form of load balancing and the load balancer is only in this state at the beginning of the simulation. This state is useful because at the start of the simulation nothing is known about the distribution and it is unknown in which realm of S acceptable GPU efficiency lies. The goal of the search state is to determine an optimal global value of S without any such prior knowledge.

B. Incremental

In the incremental state, a new determination for the global S value is made without performing a full binary search. In this state the S value is incrementally adjusted in each time step. The direction of the change (increment or decrement) is again determined by how the compute time changed based on the S value chosen for the tree rebuild in the previous time step.

C. Observation

During the majority of the simulation the load balancer sits in the observation state. In this state, the solver marches through the time steps solving the FMM. At the end of each time step, if the running time is undesirable, measures are taken to enforce a desirable running time. Next we introduce the measures which can be taken.

VI. Load Balance Enforcement Mechanisms

The enforcement procedure relies on the use of two key functions. These functions utilize the concepts introduced in the "Optimization Operations" section.

A. Enforce_S

As particles move over time steps, leaf nodes containing significantly more particles than the current global S may develop. Our first line of defense against this is the Enforce_S function. It runs through the existing octree, enforcing the existing S parameter for the tree. If it discovers a parent node containing fewer than S particles, then it performs a collapse operation on that node. If it encounters a leaf node with more than S particles within it, it carries out a pushdown operation on that node.

B. FineGrainedOptimize

FineGrainedOptimize() is the source of our fine tuning. This function makes local changes to the tree regardless of the global S value. This is a single function call. It is not like the states described earlier which persist for many time steps. This function makes changes to multiple nodes at a time. If the CPU is running too long the procedure begins by performing the collapse operation on multiple nodes. If the GPU is running too long, then the pushdown operation is performed on multiple nodes. After a group of nodes is collapsed or pushed down, the procedure utilizes the time prediction described earlier to predict how that change will affect the running time on the next time step. Based on the previous change made and the resulting predicted running time, the procedure will continue to make further changes until the predicted time is minimized.

VII. Load Balancing Workflow

A. Timing

Due to the heterogeneous nature of this model the time necessary to minimize is the maximum of the CPU and GPU wall clock times; we will call this maximal time the *Compute Time*. Each time step has a compute time associated with it. More concretely we define this time as follows.

Definition: The CPU Time for a time step is the wall clock time between the first call to the upward sweep portion of the FMM and the completion of the last task spawned during the downward sweep portion of the FMM.

Definition: GPU Time is taken to be the maximum of all the GPU kernel times in the time step. There is one kernel per GPU. We time the kernels by placing a cudaEvent on the event queue of each GPU immediately prior to and after the kernel invocation. The kernel time is taken to be the value returned by cudaEventElapsedTime() when called on the events enclosing the kernel.

Definition: Compute Time for a time step is the maximum of the previously defined CPU and GPU times.

B. Full Load Balancing and State Switching

The simulation starts in the binary search state. On our test distributions this state typically persists for fewer than 15 time steps. The load balancer leaves the binary search state and moves into the incremental state when CPU and GPU times differ by 0.15s or less.

The load balancer remains in the incremental state until the computational unit which dominates the runtime cost changes. For example, if the CPU dominates the runtime, the S value is slowly incremented up until an S value is found for which the GPU is the dominant cost. Once this transitional S value is found, if the CPU and GPU times differ by more than 0.15s, then FineGrainedOptimize() is called and upon return from this function the load balancer enters the observation state. If the times are already within 0.15s then the load balancer moves directly into the observation state without calling FineGrainedOptimize(). Before moving into the observation state, the current compute time (which is the best time seen thus far) is recorded.

While the load balancer sits in the observation state, nothing is done if the compute time for the current time step is within 5% of the previously recorded best time. If the current compute time differs by more than 5%, then Enforce_S() is called. After this call the compute time for the next time step is predicted and if it is not within 5% of the best, then FineGrainedOptimize() is called and the time is again predicted. If the fine grained adjustment fails to bring the predicted time within 5% of the best time, the load balancer moves into the incremental state again on the following time step.

VIII. EXPERIMENTAL RESULTS

A. Test Systems

Test System A**:** Test system A consists of two Intel Xeon X5670 (2.93 GHz, 6 cores each) CPUs for a total of 12 cores and four Tesla C2050 (ECC on, single precision) GPUs. We use the Intel ICC 11.1 compiler via NVCC.

Test System B**:** Test System B consists of four Intel X7560 Nehalem-EX (8 cores each) for a total of 32 cores and no GPUs. This system will be used for showing CPU scaling only, for which we also use ICC 11.1.

B. Test Problem

Unless otherwise stated, all results were collected by solving the gravitational problem. Our time step size was 0.0001s and the gravitational constant used was $G = 6.7 \cdot 10^{-7}$. Every particle in this system has a mass of 1.0kg. We use a multipole precision of $p = 6$ retained terms in the spherical harmonics expansion.

We also examine a fluid dynamics simulation of immersed flexible boundaries using the method of regularized Stokeslets as described in [15]. We will explicitly state when the data at hand is derived from this problem simulation.

C. Multicore Performance

Test System B allows for a more thorough evaluation of CPU performance than System A. In Fig. 6 we show speedup due to OpenMP CPU parallelization when run on Test System B with ten million particles distributed in a Plummer distribution for a fixed S value. The resultant octree was highly non-uniform with a depth of 16. The finest level of refinement was at level 15 and the coarsest at level 2. Even with this highly adaptive octree, we achieve good load balancing of tasks and significant speedup as a result. The baseline for the speedup shown here is the serial execution time. As evidenced by these numbers we have not seen significant overheads introduced by OpenMP task creation when using this recursive traversal of the octree and the task stealing runtime of ICC 11.1.

A small superlinear speedup can be observed when using up to 16 cores. This is likely a consequence of the additional L3 cache available across multiple sockets that enable multipole expansions to be reused and give improved performance relative to the single processor case. At high thread count the speedup diminishes; we conjecture this is due to saturation of the memory system.

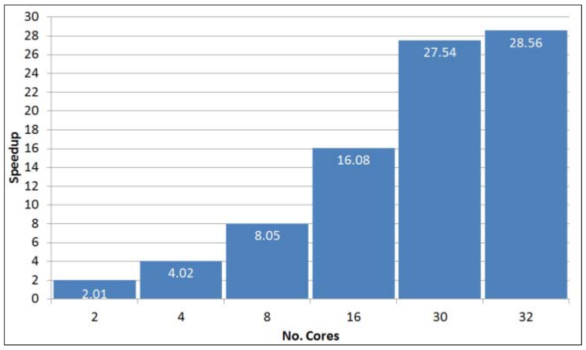

Fig. 6. CPU speedup as a function of the number of CPU cores for a Plummer distribution of 10 million particles

D. GPU Performance

Our method for work distribution amongst the available GPUs works well as seen in Table I. The data collected in this table was for a fixed workload of 10 million bodies arranged in a Plummer distribution. The S chosen was the S which minimized the total runtime for the system when utilizing 10 CPU cores and 1 GPU. The problem was carried out with this same S value while varying the number of GPUs utilized.

Table I. GPU scaling for a fixed workload

No. GPUs	Speedup
2	1.99
3	2.96
4	3.95

E. Heterogeneous Node Speedup

In order to view the overall effect of the combination of the parallelization techniques, we wanted to look at the speedup with a few different CPU and GPU combinations. As our baseline we used the time to run our implementation with a single core on Test System A. Both the expansion and direct work were run on this single core. The S chosen for this serial run was the S that minimized the time for this single core case. We then plotted speedup relative to this time for the following cases: one GPU and four CPU Cores, one GPU and ten CPU Cores, two GPUs and four CPU Cores, two GPUs and ten CPU Cores, four GPUs and four CPU Cores, and four GPUs and ten CPU Cores. A single Plummer distribution with one million particles was used as our test simulation. The results are shown in Fig. 7.

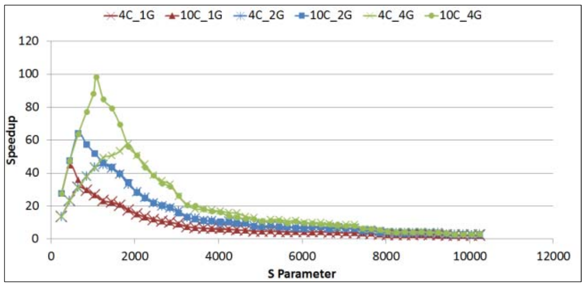

Fig. 7. Speedup shown for the entire heterogeneous system as a function of S values.

The first point to be made is that even with a relatively small N of one million and minimum runtimes around seven tenths of a second we were able to achieve just over 98x speedup with ten CPU cores and 4 GPUs compared to a single CPU core.

An interesting scenario arises when we have an underpowered CPU system as compared to the GPU system. The difference between the 10C_2G run and 4C_4G run reflect this situation. With two GPUs and ten CPU cores the implementation achieves 64x speedup compared with 57x speedup with four GPUs and 4 CPU cores. The problem with the four GPU four CPU case is that the CPUs collectively are so much less powerful than the GPUs collectively that we need to send a lot of work to the GPUs. Sending a lot of work to the GPU significantly increases the FLOPs required to solve the problem because $O(N)$ expansion work is being converted into the asymptotically inferior $O(N^2)$ direct work. Note that we have a similar situation when comparing the 10C_1G run to the 4C_2G run, but here we are required to send just enough work to the GPU such the two runs achieve similar performance.

The way forward in such an unbalanced situation is to move additional work to the GPU that can be performed more efficiently. This can include the P2M expansion formation and L2P expansion evaluation.

IX. Performance of Time Dependent Optimizations

A. Dynamic Workloads

The true test of the load balancing machinery is on workloads that change significantly over time. To examine this, we ran the gravitational problem starting from a Plummer distribution. This distribution was initially contained within 1/64th of the simulation space. The intent here was to allow particles that would otherwise have exited the system (or wrapped around with periodic boundary conditions) enough room to return back towards the center of mass.

We examine three load balancing strategies for this problem. In the first strategy, an optimal value for S is chosen at the outset (using binary search), but no adjustment of S is performed as the simulation evolves. The value of S is never changed and the tree structure is never modified. Particle positions are updated after each time step, but no other changes are made.

The second strategy performs dynamic load balancing to keep the runtime minimized. Again the binary search state is carried out at the start. After exiting the binary search state, the implementation watches for changes in the compute time. When the compute time runs more than 5% slower than the best time seen thus far, the implementation calls Enforce_S. The compute time on the time step immediately following Enforce_S() calls becomes the new best time and the simulation continues.

The third strategy is the full load balancing scheme we have described. All phases of load balancing (Search, Incremental and Observation) are fully utilized and both Enforce_S() and FineGrainedOptimized() are utilized as previously described in the workflow. The time spent on the sum of all load balancing operations and the compute time is shown in Fig. 8 for each of the 2000 time steps run. The corresponding values of S are shown in Fig. 9.

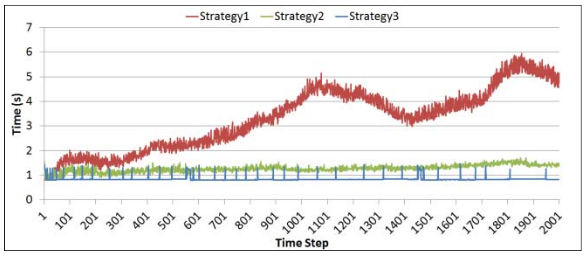

Fig. 8. Total runtime for each of 2000 time steps for each of the three described strategies.

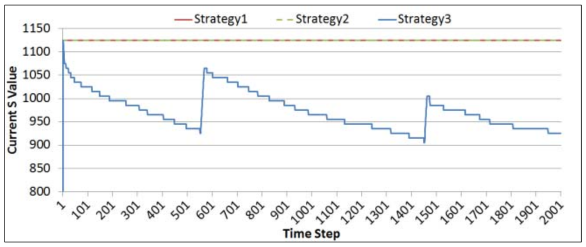

Fig. 9. The resulting S values on each time step is shown for each of the three strategies.

By looking at the trend line for strategy one, it is clear that some form of correction is necessary. Simply maintaining the original S value and reinforcing it when the compute time increases provides much improvement as evidenced by the data series for the second strategy. The average time over all time steps for strategy two was 1.29s. The spikes in strategy three are instances when load balancing occurred. The spike is a combination of the increased compute time which triggered the load balancing to take effect as well as the load balancing cost itself. The most time spent on load balancing operations in a single time step was approximately 0.52s. The average compute time per time step was 0.82s. The largest compute time spike which triggered load balancing operations to take effect was 0.12s and it was a deviation of 14% from the best time seen thus far. As a result of the spikes, 34 out of the 2000 time steps resulted in a total time greater than the average time of 1.29s per time step seen in strategy two.

A tabulated comparison of the different strategies is shown in Table II. The key point is that the simulation completes in the shortest wall clock time using our load balancing strategy, while overhead of the strategy is small at 1.88%. The non-incremental load balancing strategy 2 took roughly 1.51 times longer to finish the simulation than strategy 3. And the strategy that performed no dynamic load balancing took approximately 3.91 times longer to complete the same simulation.

Table II. Strategy Summary for Dynamic Workloads

Strategy	Total Compute	Total LB	LB as % of Compute	Relative cost per time step
1	6576.17s	1.32s	0.02%	3.91
2	2544.79s	2.78s	0.11%	1.51
3	1651.57s	30.98s	1.88%	1.00

Total compute and total load balance are the sums of compute time and load balancing time over all 2000 time steps, respectively. LB as % of compute is total LB / total compute. Relative cost per time step compares the average runtime per time step relative to strategy 3.

B. Uniform Gap and Static Workloads

Our load balancing scheme primarily works to improve the performance of workloads that vary significantly over the course of a simulation, but the FineGrainedOptimize() component can also be well suited for acceleration of very uniform, static workloads. As shown in Fig. 4, a uniform distribution may present a significant hurdle when optimizing a heterogeneous implementation like ours. The size of the "uniform gap" which the problem will present depends on many factors. CPU and GPU speed, and the cost of the implementation-specific expansion and direct work operations will have an impact on the specific times. Also the size of the jump in time from one level to the next will depend on the number of nodes in each of the levels in question as well as the number of particles in the leaf nodes, which in turn depends on the distribution and the S value (which again depends on CPU and GPU speed).

Our gravitational problem with one million bodies did not have a huge gap. The speedup offered by the use of FineGrainedOptimize() was not large, around a 1% improvement on relatively small run times. For this reason we chose to display the results of our scheme on the fluid dynamics problem mentioned earlier. We saw much more improvement here on the same machine which ran the gravitational problem due to the fact that the M2L cost for the fluid dynamics problem is about 4x the M2L cost for the gravitational problem.

Two simulations of 200 time steps each using ten million sources in a uniform distribution were carried out. One simulation utilized FineGrainedOptimize() and the other did not. Fig. 10 shows the ratio of the per-time step times of the two simulations. The time for each time step includes any optimizations made in addition to compute time. The first 15 time steps constitute the initial binary search for a good S realm. For the remainder of the time steps we achieve slightly more than a 3% advantage per time step as a result of using the fine grained optimizations.

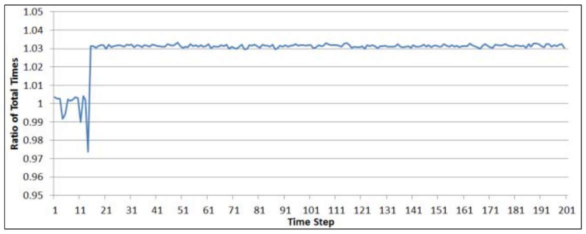

Fig. 10. For each time step we show the ratio of total time without using fine grained load balance to the total time when using fine grained load balance.

Different implementations of different problems on different machines will be able to derive varying amounts of benefits from this local modification. Certainly instances with a very small gap between the running times of the varying computational units will not be able to extract any benefit from this setup.

X. Conclusion

Accelerating a time-dependent N-body simulation using the AFMM on heterogeneous machines requires balancing the many sources of run time variation inherent to this setup. Each problem instance on a specific machine will have varying CPU and GPU runtimes as a function of S. High level modifications such as binary search work well to optimize the runtime but the complex nature of the CPU-GPU cost relationship near regions of transition between the dominant computational costs require the full load balance model to achieve peak performance. The discrete gaps in the uniform distribution costs can be handled better using the AFMM and our fine grained load balance procedure. However, even the significantly smoother case for the Plummer model benefits from fine grained load balancing to adjust the local decomposition tree structure in order to compensate for issues such as GPU efficiency, which varies greatly with S and depends on the current distribution of bodies in the leaves of the octree.

When applied to time-dependent applications, the AFMM achieves excellent parallel performance that can be maintained even while the underlying system evolves considerably.

Acknowledgments

This work was supported by an NSF FRG grant on computational bio-fluids dynamics (DMS-0854961). We thank R. Fowler and A. Porterfield and RENCI for access to computer resources provided under DOE SciDAC PERI (DE-FC02-06ER25764) and SciDAC SUPER (DE-FG02-11ER26050/DE-SC0006925).

References

[1] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations," *Journal of Computational Physics,* vol. 73, no. 2, pp. 325-348, 1987.

[2] J. A. Lupo, Z. Wang, A. M. McKenney, R. Pachter and W. Mattson, "A Large Scale Molecular Dynamics Simulation Code Using The Fast Multipole Algorithm (FMD): Performance and Application," *Journal of Molecular Graphics Modeling,* vol. 21, no. 2, pp. 89-99, 2002.

[3] L. F. Greengard and J. Huang, "A New Version of the Fast Multipole Method for Screened Coulomb Interactions in Three Dimensions," *Journal of Computational Physics,* vol. 180, no. 2, pp. 642-658, 2002.

[4] Z. Wang, J. A. Lupo, A. M. McKenney and R. Pachter, "Large Scale Molecular Dynamics Simulations with Fast Multipole Implementations," in *ACMIEEE SC 1999 Conference SC99*, 1999.

[5] J. Kurzak and B. Pettitt, "Massively Parallel Implementation of a Fast Multipole Method for Distributed Memory Machines," *Journal of Parallel and Distributed Computing,* vol. 65, no. 7, pp. 870-881, 2005.

[6] H. Cheng, L. Greengard and V. Rokhlin, "A Fast Adaptive Multipole Algorithm in Three Dimensions," *Journal of Computational Physics,* vol. 155, no. 2, pp. 468-498, 1999.

[7] 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," *Communications of the ACM,* vol. 55, no. 5, pp. 101-109, May 2012.

[8] H. Dachsel, M. Hofmann, J. Lang and G. Rünger, "Automatic Tuning of the Fast Multipole Method Based on Integrated Performance Prediction," in *IEEE 14th International Conference on High Performance Computing and Communication*, Liverpool, UK, 2012.

[9] Q. Hu, N. A. Gumerov and R. Duraiswami, "Scalable Fast Multipole Methods on Distributed Heterogeneous Architectures," in *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[10] N. A. Gumerov and R. Duraiswami, "Fast multipole methods on graphics processors," *Journal of Computationla Physics,* vol. 227, no. 18, p. 8290–8313, 2008.

[11] R. Yokota and L. A. Barba, "Treecode and Fast Multipole Method for N-Body Simulation with CUDA," in *GPU Computing Gems Emerald Edition*, Morgan Kaufmann, 2011.

[12] J. Bédorf, E. Gaburov and S. Portegies Zwart, "A sparse octree gravitational N-body code that runs entirely on the GPU processor," *Journal of Computational Physics,* vol. 231, no. 7, pp. 2825-2839, 2012.

[13] I. Lashuk, G. Biros, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying and D. Zorin, "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 p. 1*, 2009.

[14] L. Nyland, M. Harris and J. Prins, "Fast N-body Simulation with CUDA," *GPU Gems,* vol. 3, no. 1, pp. 677-696, 2007.

[15] R. Cortez, L. Fauci and A. Medovikov, "The method of regularized Stokeslets in three dimensions: Analysis, validation, and application to helical swimming," *Physics of Fluids,* vol. 17, no. 3, 2005.