

A Tuned and Scalable Fast Multipole Method as a Preeminent Algorithm for Exascale Systems

Rio Yokota, L. A. Barba

Mechanical Engineering Department, Boston University, Boston MA 02215

Abstract

Among the algorithms that are likely to play a major role in future exascale computing, the fast multipole method (FMM) appears as a rising star. Our previous recent work showed scaling of an FMM on GPU clusters, with problem sizes in the order of billions of unknowns. That work led to an extremely parallel FMM, scaling to thousands of GPUs or tens of thousands of CPUs. This paper reports on a campaign of performance tuning and scalability studies using multi-core CPUs, on the Kraken supercomputer. All kernels in the FMM were parallelized using OpenMP, and a test using 10^7 particles randomly distributed in a cube showed 78% efficiency on 8 threads. Tuning of the particle-to-particle kernel using SIMD instructions resulted in $4\times$ speed-up of the overall algorithm on single-core tests with $10^3 - 10^7$ particles. Parallel scalability was studied in both strong and weak scaling. The strong scaling test used 10^8 particles and resulted in 93% parallel efficiency on 2048 processes for the non-SIMD code and 54% for the SIMD-optimized code (which was still $2\times$ faster). The weak scaling test used 10^6 particles per process, and resulted in 72% efficiency on 32,768 processes, with the largest calculation taking about 40 seconds to evaluate more than 32 billion unknowns. This work builds up evidence for our view that FMM is poised to play a leading role in exascale computing, and we end the paper with a discussion of the features that make it a particularly favorable algorithm for the emerging heterogeneous and massively parallel architectural landscape.

1 Introduction

Achieving computing at the exascale means accelerating today's applications by one thousand times. Clearly, this cannot be accomplished by hardware alone, at least not in the short time frame expected for reaching this performance milestone. Thus, a lively discussion has begun in the last two or three of years about programming models, software components and tools, and algorithms that will facilitate exascale computing.

The hardware path to exascale systems, although not entirely certain, points to computing systems involving nodes with increasing number of cores, and consisting of compute engines with significant structural differences (for example, having different instruction set architectures), i.e., heterogeneous systems. Among heterogeneous systems, those involving GPUs are currently at the forefront and poised to continue gaining ground, until and unless a totally new and unexpected technology comes along. The latest Top500 list of the world's supercomputers (as of June 2011) has three GPU-based systems ranked among the top five. Furthermore, on the Green500

list (Feng and Cameron, 2009) of the world's most energy efficient supercomputers (as of June 2011), three out of the top five are also GPU-based systems (not the same systems leading the Top500 list, however). Even if a totally new technology were to come around the corner on our path to the exascale, the lessons learned from adapting our scientific algorithms to the challenges posed by GPUs and many-core systems will better prepare us. Features that are inevitable in emerging hardware models, as recognized in Bergman et al. (2008), are: many (hundreds) of cores in a compute engine and programmer-visible parallelism, data-movement bottlenecks, and cheaper compute than memory transfers.

Computational methods and software will have to evolve to function in this new ultra-parallel ecosystem. One of the greatest challenges for scalability (as well as programmability) is the growing imbalance between compute capacity and interconnect bandwidth. This situation demands our investment in algorithmic research, co-developed with exascale applications so that the new architectures can indeed be exploited for scientific discovery at the highest performance.

On the other hand, it is clear that achieving exascale computing in the short time frame anticipated (the next decade) will require much more than hardware advances. Developments in algorithms must play a leading role. If history is to be our guide, algorithm developments have often provided great leaps in capability, comparable or higher than those offered by progress in hardware. Thus, we can be fairly optimistic that, if we maintain a substantial research effort into both scientific algorithms and their implementation, the chances of success are higher.

It is interesting to note that many of the most successful algorithms of today are hierarchical in nature, such as the case of multi-grid methods. Hierarchical algorithms often result in the ideal $\mathcal{O}(N)$ scaling, with N representing the problem size. A quintessential $\mathcal{O}(N)$ algorithm, which we focus on here, is the fast multipole method, FMM. This algorithm was introduced by Greengard and Rokhlin (1987) for performing fast calculations of N -body type problems, such as are encountered in gravitational or electrostatic calculations. These problems naturally have a computational complexity of $\mathcal{O}(N^2)$ for N bodies (masses or charges), but the FMM provides a solution in $\mathcal{O}(N)$ operations, to within a controllable accuracy. It is important to note that the FMM can be used not only as an N -body solver, as it was originally designed to be, it can also be used to solve any elliptic PDE (e.g., Ethridge and Greengard, 2001; Cheng et al., 2006; Langston et al., 2011).

There are several groups actively contributing to the field of hierarchical N -body algorithms, and their high-performance implementation in both massively parallel CPU systems and GPU architecture. These efforts have often been divided along two lines: the $\mathcal{O}(N \log N)$ treecodes, with a strong following in the astrophysics community, and the $\mathcal{O}(N)$ fast multipole method, and its variants. A notable player in the field is the kernel-independent version of the FMM, a variant which does not make use of multipole expansions of the sources in a cluster but rather an equivalent density on a surface enclosing the cluster. The group working on this algorithm, known as *kifmm* (Ying et al., 2004), has provided a continued stream of evidence of the high performance levels that can be attained with this family of algorithms. Most recently, they were awarded the coveted ACM Gordon Bell prize at the Supercomputing conference (Rahimian et al., 2010), with a sustained performance of 0.7 Petaflop/s—this is merely $3.7\times$ less than the peak HPC Linpack benchmark of the same date, but achieved with a full application run (involving flow of deformable red blood cells). Thus, we are convinced that hierarchical algorithms, and the FMM in

particular, are especially poised to be leading players in the exascale computing future.

In our own recent efforts, we have achieved excellent scaling on GPU clusters, with problem sizes that are in the order of billions of unknowns (Yokota et al., 2011b,a). This work has led to an extremely parallel FMM, which is capable of scaling to thousands of GPUs or tens of thousands of CPUs. In the present study, we demonstrate the scalability of our new FMM on a large CPU-based system (Kraken, at the National Institute for Computational Science, University of Tennessee). We also perform single-node performance tuning by using SIMD instructions and OpenMP. Our efforts are on a par with the most high-performing computations using FMM, yet are fully independent. Thus, we contribute to the mounting evidence of the decisive role of FMM and related algorithms in the quest for exascale. In view of this, we conclude our paper with a brief discussion of the features of the FMM that make it a particularly favorable algorithm for the emerging heterogeneous, massively parallel architectural landscape. We hope that this will enrich the ongoing discussions about the challenges and opportunities to reach the milestone of exascale computing within a decade.

Given the magnitude of the challenge, we maintain that working in collaboration and under a model of openness should offer the best opportunities. Consistent with this view, all codes developed as part of this research effort are open source, and released at the time of publication. The full implementation of the FMM used to produce the results of this paper, as well as some of our previous publications, is available via the website at <http://www.bu.edu/exafmm/>.

2 The fast multipole method, and its place in high-performance computing

2.1 Brief overview of the algorithm

Evaluating the interactions among N sources of a force potential that has long-range effects in principle requires $\mathcal{O}(N^2)$ operations. The canonical examples of this situation are gravitational and electrostatic problems. The fast multipole method (FMM) of Greengard and Rokhlin (1987) is able to reduce the computational complexity of this problem to $\mathcal{O}(N)$ operations, and thus can have tremendous impact on the time to solution in applications.

The FMM begins with a hierarchical subdivision of space, allowing a systematic categorization of spatial regions as either *near* or *far* from one another. Different analytical approximations to the ker-

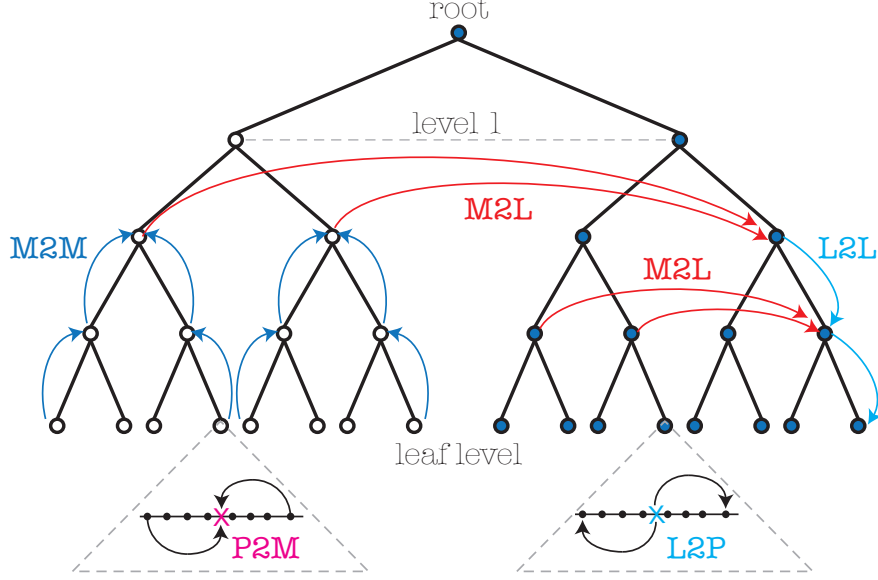


Figure 1: Illustration of FMM algorithm components, with the *upward sweep* depicted on the left side of the tree, and the *downward sweep* depicted on the right side of the tree. The multipole expansions are created at the leaf level in the P2M operation, they are translated upwards to the center of the parent cells in the multipole-to-multipole (M2M) translation, then transformed to a local expansion in the M2L operation for the siblings at all levels deeper than level 1. The local expansions are translated downward to children cells in the L2L operation and finally, the local expansions are added at the leaf level and evaluated in the L2P operation.

nel that represents the interaction are used for pairs of points that are in near or far regions, correspondingly. These approximations are then transferred between different scales, up and down, using the subdivision hierarchy, eventually resulting in the assembly of a complete interaction force field at all of the points to a given approximation. A view of the complete algorithm is given in Figure 1, where the main computational steps are associated to a tree graph representing the hierarchical subdivision of a one-dimensional domain space. The multi-resolution nature of the hierarchical space division, with its associated tree, is illustrated in Figure 2 for the 2D case.

The FMM algorithm defines the following mathematical tools. The Multipole Expansion (ME) is a series expansion, truncated after p terms, which represents the influence of a cluster of sources on far-away regions of space, and is valid at distances large with respect to the cluster radius. The Local Expansion (LE) is also a series expansion of p terms, valid only *inside* a given subdomain, and used to efficiently evaluate the contribution of a group of MEs on that subdomain. In other words, the MEs and LEs are series (e.g, Taylor series) that converge in different subdomains. The center of the series for an ME is the center of the cluster of source particles, and it only converges outside the cluster of particles. In

the case of an LE, the series is centered near an evaluation point and converges locally. All the distinct operations that are required in the algorithm are illustrated in the flow diagram of Figure 3.

The utilization of an aggregate representation for a cluster of particles, via the ME, effectively permits a decoupling of the influence of the source particles from the evaluation points. This is a key idea, resulting in the factorization of the computations of MEs that are centered at the same point. This factorization allows pre-computation of terms that can be re-used many times, increasing the efficiency of the overall computation. Similarly, the LE is used to decouple the influence of an ME from the evaluation points. A group of MEs can be factorized into a single LE so that one single evaluation can be used at multiple points locally. The combined effect of using MEs and LEs is a computational complexity of $\mathcal{O}(N)$ for a full evaluation of the force field.

2.2 Our ongoing work on parallel FMM

Previously, our research group has led the development of an open-source parallel software, called PetFMM, which aims to offer a PETSc-like package for FMM algorithms. It is characterized by dynamic load-balancing based on re-casting the tree-based algorithm, illustrated in Figure 1, into an undirected

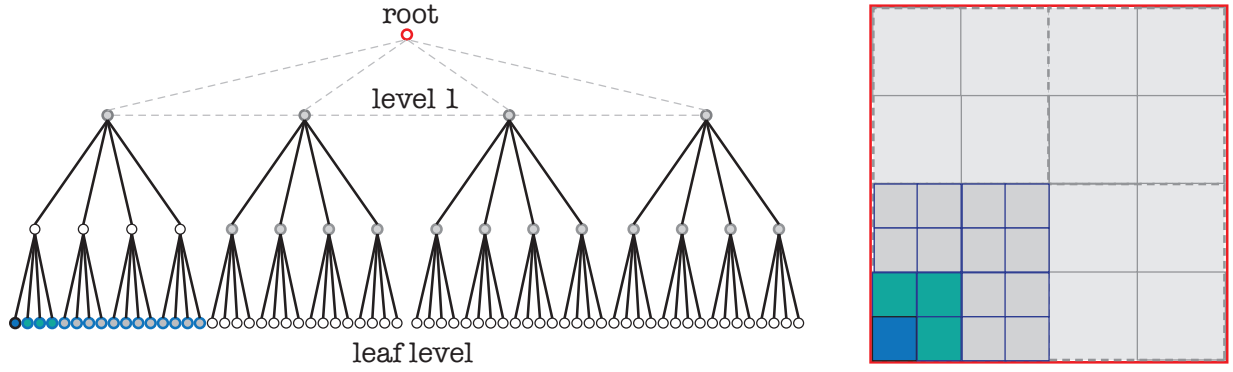


Figure 2: A three-level hierarchical decomposition of a 2D domain. Tree nodes corresponding to domain boxes are shown in corresponding color.

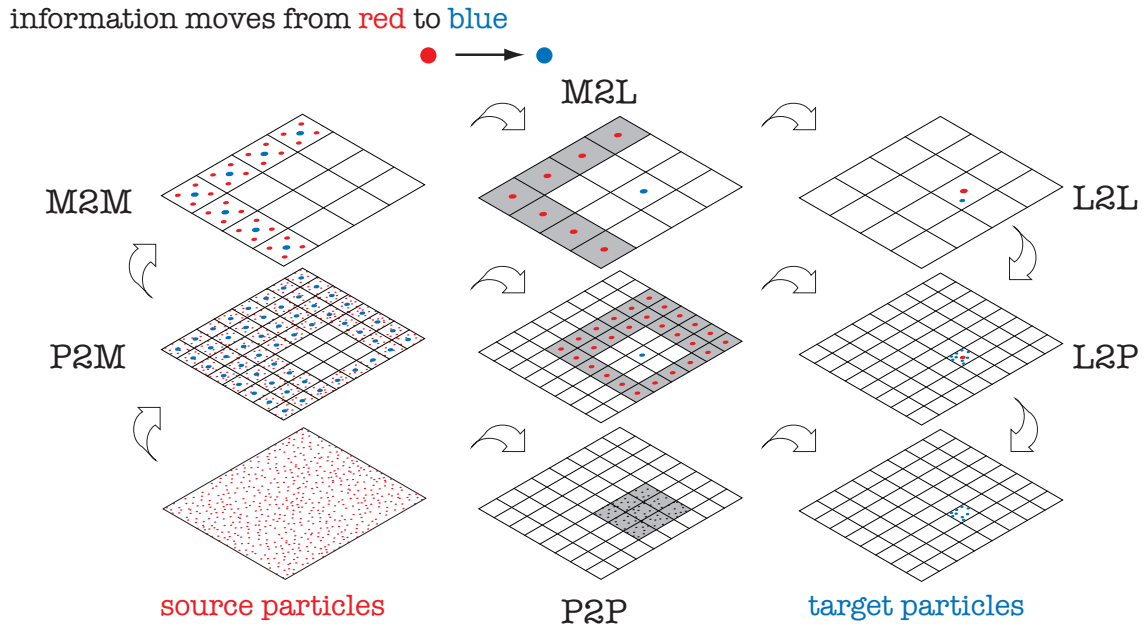


Figure 3: Flow of the FMM calculation, showing all of the operations required in the algorithms: P2M: transformation of points into MES (points-to-multipole); M2M: translation of MES (multipole-to-multipole); M2L: transformation of an ME into an LE (multipole-to-local); L2L: translation of an LE (local-to-local); L2P: evaluation of LEs at point locations (local-to-point).

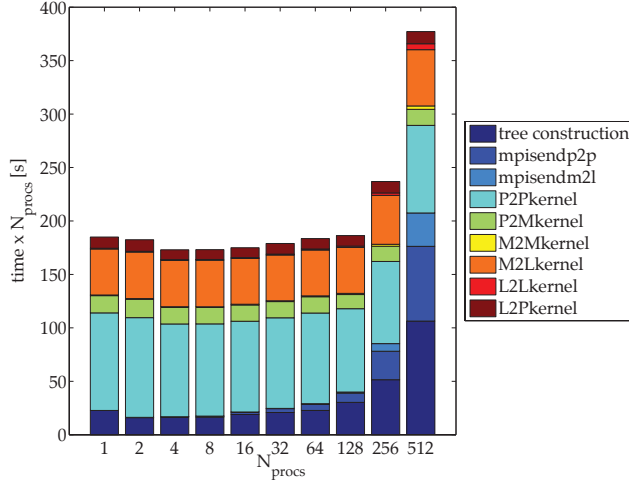


Figure 4: Multi-GPU strong scaling, and timing breakdown of the different FMM kernels, tree construction (done on the CPU) and communications. Test problem: 10^8 points placed at random in a cube; FMM with order $p = 10$ and spherical harmonic rotation before each translation at $\mathcal{O}(p^3)$ cost. Parallel efficiency is slightly above 1 for 4–32 processes, possibly due to cache advantage on the tree construction. Parallel efficiency is 78% on 256 and 48% on 512 GPUs. From Yokota et al. (2011a).

graph created using a computational model of the FMM (Cruz et al., 2010). The spatial hierarchy is divided into many subdomains, more than the number of processes available, and an auxiliary optimization problem for the partition is constructed such that it minimizes both load imbalance and communication. The problem is modeled by a weighted graph, and solved using the partitioner ParMetis (Karypis and Kumar, 1998). Strong scaling results showing the parallel performance of PetFMM were given in Cruz et al. (2010), with a test evaluating the field due to 10 million particles using up to 256 processors of a CPU cluster.

Recently, we have produced a multi-GPU FMM implementation, and performed new scalability studies. The parallel partitioning in this code is based on work-only load balancing, by equally distributing the Morton-indexed boxes at the leaf level; this is the standard and most commonly used technique for distributing N -body codes, first introduced by Warren and Salmon (1993). A host of detailed performance improvements were applied to accomplish good scalability on up to 512 GPUs in our first study (Yokota et al., 2011b). We should note that achieving parallel scalability on a cluster of GPUs is a greater challenge than scaling on a CPU-only cluster. The high compute capability of GPUs accelerates the computation and exposes the time required

for communications and other overheads, which is exacerbated by the fact that, for multi-GPU calculations, the GPUs cannot communicate with each other directly. With current technology, it is necessary to send the data back to the host machine and communicate between nodes via MPI. Yet, our *strong scaling* tests for the FMM on multi-GPUS demonstrated high parallel efficiency despite these challenges; see Figure 4. On the Degima cluster¹, parallel efficiency remains close to 1 all the way up to 128 GPUs, then drops to 78% at 256, and to 48% at 512 processes. The efficiency drop was ascribed to the configuration of the interconnect in the machine, and the number of GPUs per node: up to 128 GPUs, only one process was running on each node; for the case with 256 GPUs, two processes were running on each node; and for the case with 512 GPUs, four processes were running per node. This situation limits the effective bandwidth per process as bandwidth must be shared within the node. To alleviate this problem, we implemented a hierarchical all-to-all communication with MPI_Comm_split, splitting the MPI communicator into sub-sets of size 4. With 4 processes per node, we are effectively splitting the communicator into an inter-node and an intra-node communicator. Thus, we first perform an intra-node MPI_Alltoallv, then an inter-node MPI_Alltoallv. We found that this could speed-up the communication nearly 4 times at 512 processes. This is an example of applying a *hierarchical parallel model* to effectively obtain performance from a heterogeneous system.

2.3 High-performance FMM computing

Hierarchical N -body algorithms, including both the $\mathcal{O}(N \log N)$ treecode and the $\mathcal{O}(N)$ FMM, have been visible players in the field of high-performance computing for many years. In fact, the top place in performance of the Gordon Bell award² has been obtained several times with these algorithms. In 1992, the first place was awarded to Warren and Salmon (1992) for a simulation of 9 million gravitating stars with a parallel treecode (sustaining 5 Gflop/s); some of the techniques used in that work remain important to this day. The same authors (with others) achieved the first place again in 1997 with a simulation of the dynamics of 322 million self-gravitating particles, at a sustained 430 Gflop/s (Warren et al., 1997). That year, the award in the

¹Degima is the GPU cluster at the Nagasaki Advanced Computing Center, which at the time had two NVIDIA GTX295 cards per node (two GPUs per card), and mixed QDR/SDR Infiniband network.

²<http://awards.acm.org/bell/>

price/performance category also went to this team, who reported \$50/Mflop.

More recently, we have seen treecodes and FMM figure prominently with Gordon Bell awards in the last two years. In 2009, the work by Hamada et al. (2009) was recipient of the award in the price/performance category, achieving 124 Mflop/\$ (more than 6 thousand times “cheaper” than the 1997 winner) with a GPU cluster. And last year, the first place in the performance category was achieved with an FMM-based Stokes solver of blood flow which sustained 0.7 Pflop/s when solving for 90 billion unknowns (Rahimian et al., 2010). To achieve this performance, a fast multipole method that already had undergone development to attain high parallel scaling (Lashuk et al., 2009) was further optimized by means of explicit SSE vectorization and OpenMP multi-threading (Chandramowlishwaran et al., 2010). As a result, a full application code was obtained that was merely $3.7\times$ slower than the Linpack high-performance computing benchmark that is used to build the Top500 list³.

Given that the FMM algorithm can achieve performance *in applications* that is close to the Linpack benchmark, and that moreover it has excellent compatibility with massively parallel GPU hardware, we estimate that it will feature prominently in the path to exascale. In the present contribution, we perform optimizations of our FMM for single-core performance, and carry out scalability studies at large scale on CPU-based systems. Added to our ongoing work on GPU implementation, we keep the momentum going in the direction of exascale FMM for the next decade.

3 Intra-node performance optimization

This section is dedicated to reporting a campaign of code optimization for single-node performance, including both SIMD vectorization and multithreading within a node using OpenMP. All runs were performed on the Kraken Cray XT5 system of the National Institute for Computational Sciences (NICS) at the University of Tennessee, via TeraGrid access.

Each node of the Kraken system has two 2.6 GHz six-core AMD Opteron processors (Istanbul) and 16 GB of memory. The system is connected via a Cray SeaStar2+ router, and it has 9,408 compute nodes for a total of 112,896 compute cores.

³<http://www.netlib.org/benchmark/hpl/>

3.1 Multithreading with OpenMP

All kernels in the FMM were parallelized using OpenMP (the outermost loop for each kernel was explicitly parallelized by using the thread id). Results are presented in Figure 5, consisting of the breakdown of the calculation time of the FMM for different numbers of threads. The calculation time is multiplied by the number of threads for ease of comparison, i.e., bars of equal height would indicate perfect scaling. These tests were performed on a single node of Kraken using 1, 2, 4, and 8 cores, and the test problem was a set of $N = 10^7$ particles randomly distributed in a cube. In the legend on the right hand side, the P2P, P2M, M2M, M2L, L2L, L2P labels correspond to the individual stages of the FMM as shown in Figure 3. The P2P kernel (the direct particle-particle interaction for the near field) takes up most of the calculation time, with the M2L (multipole-to-local translation between two distant cells) also taking a significant portion. These two kernels are balanced in an FMM calculation by the number of levels in the tree, and the proportion of time taken by each will change as N changes; in this case, with $N = 10^7$, the situation shown in Figure 5 offers the best timing and hence it is balanced. The results in Figure 5 show that the two main kernels scale quite well up to 8 threads. Some of the other kernels do not scale as well, which requires further investigation; however, this is not of primary importance since we do not need to scale to a large number of threads with the OpenMP model at the moment.

The order of the FMM expansions is set to $p = 3$ for the tests in Figure 5, which will give 3 – 4 significant digits of accuracy. We simply choose this value because it is common in astrophysics applications, one of the main application areas for the FMM, where high accuracy is not required. Using a larger p would result in more work for the P2M, M2M, L2L, L2P, and especially the M2L kernel, which would require in turn using more particles per leaf cell to balance the calculation load of P2P and M2L kernels. This would affect the parallel scalability of the P2P kernel favorably, because of more data parallelism within each cell. Increasing p would also increase the fine-grained parallelism of the M2L kernel because of more expansion coefficients to calculate, but would decrease the coarse-grained parallelism because the tree would become shallower for a given N .

3.2 Vectorization with inline assembly

Recently, other researchers have dedicated substantial effort to single-node optimization and multi-core parallelization of an FMM-type algorithm. Chan-

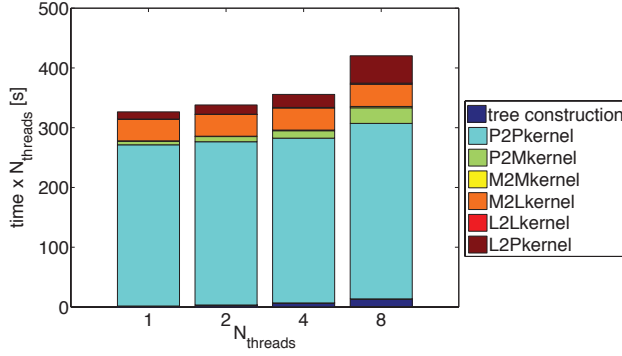


Figure 5: openMP strong scaling, and timing breakdown of the different kernels, tree construction and communications. Calculation time is multiplied by the number of threads. Test problem: $N = 10^7$ points placed at random in a cube; FMM with order $p = 3$. Parallel efficiency is 78% on 8 threads.

dramowlishwaran et al. (2010) reported the first extensive study of its kind, which included not only SIMD vectorization and intra-node parallelization, but also various other optimizations of the `kifmm` algorithm. They reported a total performance improvement of $25\times$ for both optimizations and 8-core parallelization on Intel Nehalem CPUs (plus differing performance improvements in other, less advanced, platforms). The speed-up from optimizations only on the overall algorithm is not specifically stated, but assuming perfect scaling on the 8 Nehalem cores, we can infer a $3\times$ speed-up by the optimizations (including manual SIMD vectorization) in double precision.

We have performed a similar optimization campaign using inline assembly and achieve a $4\times$ overall acceleration for a single-thread execution using single precision. Results are shown in Figure 6, consisting of the calculation time of the FMM with and without SIMD vectorization, for $N = 10^3 - 10^7$ particles randomly distributed in a cube. At the moment, the optimization was performed for the P2P kernel only. The P2P kernel itself accelerates $16\times$ compared to an unoptimized double-precision kernel, but since the acceleration is limited to a certain part of the algorithm the overall acceleration is only $4\times$. We expect a further increase in performance when the other kernels are optimized using the same method. Nevertheless, even with the present implementation we are able to calculate $N = 10^7$ particles in less than 100 seconds on a single core on Kraken (as shown in Figure 6).

It is worth noting that in citing the independent work on single-node optimization of the `kifmm` al-

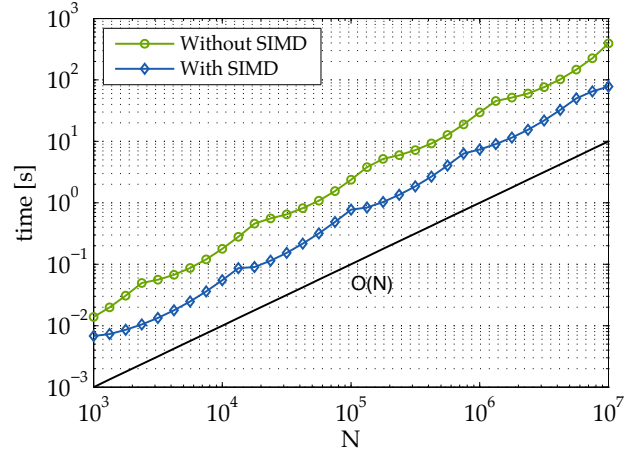


Figure 6: Calculation time w/o SIMD Test problem: $N = 10^3 - 10^7$ points placed at random in a cube; FMM with order $p = 3$. Only the P2P kernel is accelerated with SIMD instructions. The overall speed up of the FMM is approximately 4 times.

gorithm by Chandramowlishwaran et al. (2010), we are not encouraging that our work be compared toe to toe in regards to performance. The `kifmm` algorithm is in many ways different from our FMM, although it solves the same type of problems and shares a fundamental structure (e.g., hierarchical domain partition and associated tree structure). Quite the contrary, the point is that two completely independent efforts with hierarchical N -body algorithms are demonstrating the potential for applications based on these algorithms to benefit from excellent performance on multi-core systems.

4 Parallel scalability on Kraken

In this section, we report the results of parallel scalability tests on Kraken using the same FMM code we described in the previous section. Strong scaling is shown between 1 and 2048 processes and weak scaling up to 32,768 processes, with excellent parallel efficiency. For the strong scaling tests, we used 1 thread per process, and for the weak scaling tests we used 8 threads per process. The largest calculation is performed on a problem size of 32 billion particles, taking less than 40 seconds to complete.

The most recent results by researchers working with the `kifmm` algorithm and code base were reported in Rahimian et al. (2010), a work awarded the 2010 ACM Gordon Bell prize in the performance category. They report 84% parallel efficiency for a strong scaling test between 48 and 3072 processes

of the Jaguar supercomputer⁴. The study was carried out on an application of the `kifmm` algorithm to Stokes flow (red blood cells), and the reported strong scaling test used a problem of size $N = 10^8$ unknowns. This is the same problem size that we used in our strong scaling test (below), but it must be noted that we use are using a different (Laplace) kernel. Moreover, the current literature points to many sophisticated features in the `kifmm` code (including: bottom-up parallel tree construction scheme, all-reduce-on-hypercube communication, FFT-based translation methods for the cell-cell interactions, among others), which are not utilized in our present software. Thus, it is not appropriate to make a performance comparison between these two very different variations of the FMM and scaling studies. The point we would like to make here is that the FMM is a highly “strong-scalable” algorithm, and therefore a truly suitable algorithm for the exascale era. The fact that two completely independent efforts both show similar excellent scalability is indeed promising.

We start by presenting our strong scaling results without SIMD vectorization. Figure 7 shows the breakdown of each stage of the FMM for a test consisting of $N = 10^8$ points placed at random in a cube, with the order of multipole expansions set to $p = 3$. The calculation time is multiplied by the number of MPI processes (N_{procs}) for better visibility at large N_{procs} . The exponent of the number of processes is shown in the horizontal axis, e.g., the bar labeled with “11” represents $N_{procs} = 2^{11} = 2048$. The legend is similar to that of Figure 5 except we include the MPI communication times “`mpisendp2p`” and “`mpisendm2l`”, which are the communication times for the P2P kernel and M2L kernel, respectively. All the other kernels do not require communication. The parallel efficiency at 2048 processes is in this case approximately 93%.

The results shown in Figure 7 were obtained with the FMM code *without* single-node performance optimizations. The results for the same strong-scaling test using the FMM *with* the SIMD kernels are shown in Figure 8. As we have mentioned in the previous section, only the P2P kernel is optimized in our present code. Therefore, the tree is shallower and there are more particles in the leaf cell to keep the P2P and M2L balanced. Since the calculation time of the P2P kernel decreases significantly, the other parts start to dominate. The parallel efficiency is 54% at 2048 processes, with about half of the total run time corresponding to the sum of tree construction time and time for communication between nodes and —

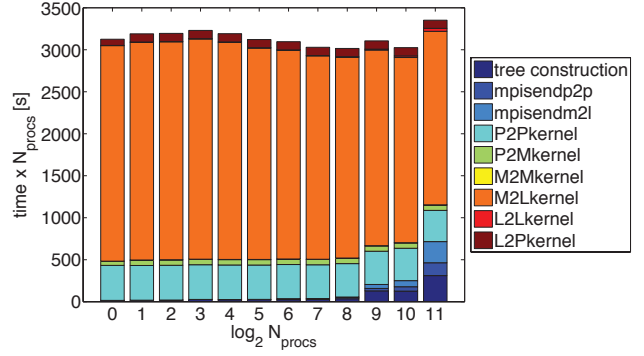


Figure 7: MPI strong scaling from 1 to 2048 processes, and timing breakdown of the different kernels, tree construction and communications. Test problem: $N = 10^8$ points placed at random in a cube; FMM with order $p = 3$. Calculation time is multiplied by the number of processes. Parallel efficiency is 93% on 2048 processes.

we must stress, however, that this is a strong-scaling test between 1 and 2048 processes (not offset scaling, as shown in some other works), and that communications are not overlapped.

By comparing Figures 7 and 8, we see that the calculation is approximately 4 times faster for the optimized version when the number of processes is small. However, since the optimized version does not scale as well, the speed-up is about 3 times at 1024 and only doubles at 2048 processes. These trends are similar to what we observed in the strong-scaling tests on the GPU, where the acceleration of the P2P and M2L kernels causes the communication and tree construction to become the bottleneck.

Figure 9 shows weak scaling results on a test using $N = 10^6$ particles per process and $p = 3$. We achieve a parallel efficiency of 72% on 32,768 processes. The largest problem size that was calculated consisted in **32 billion unknowns**, being solved in less than 40 seconds. It can be seen in Figure 9 that the calculation times per process of both P2P and M2L kernels remain somewhat constant up to 32k processes.

5 Suitability of the FMM for achieving exascale and future directions

The results shown here, added to our previous work and comparable recent work by other researchers, point to certain features of the FMM that make it a particularly favorable algorithm for the emerging heterogenous, many-core architectural landscape. In this section, we will make this view more clear, addressing specific exascale application characteristics

⁴Jaguar is a Cray XT5 system, very similar to Kraken.

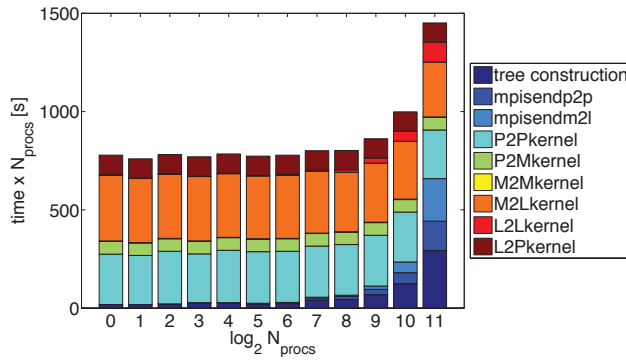


Figure 8: MPI strong scaling with SIMD from 1 to 2048 processes, and timing breakdown of the different kernels, tree construction and communications. Test problem: $N = 10^8$ points placed at random in a cube; FMM with order $p = 3$. Calculation time is multiplied by the number of processes. Parallel efficiency is 54% on 2048 processes.

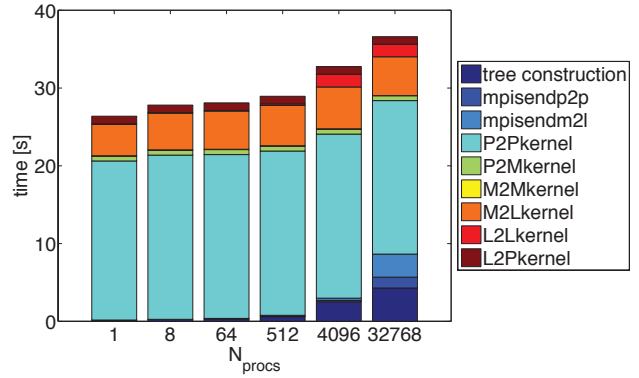


Figure 9: MPI weak scaling with SIMD from 1 to 32768 processes, and timing breakdown of the different kernels, tree construction and communications. Test problem: $N = 10^6$ points per process placed at random in a cube; FMM with order $p = 3$. Parallel efficiency is 72% on 32768 processes.

identified in [Bergman et al. \(2008\)](#). The expectation for exascale is that in the order of a million terascale processors are required. (Note that the results presented on Figure 4 mean that we already have several million simultaneous threads running, if we consider the 32 non-divergent threads in a SIMD warp to be independent threads.) In recent feasibility studies, hypothetical exascale systems are proposed which would consist of 1024-core nodes, with each core clocking at 1 GHz, and a total of 2^{28} or 2^{30} such cores ([Gahvari and Gropp, 2010](#); [Bhatele et al., 2011](#)). Analysis of representative algorithms can then be performed to determine the problem sizes required to reach 1 exaflop/s, and the constraints in terms of system communication capacity that would make this performance feasible. In [Bhatele et al. \(2011\)](#), three classes of algorithms are considered by this analysis: pure short-range molecular dynamics, tree-based N -body cosmology, and unstructured-grid finite element solvers. [Gahvari and Gropp \(2010\)](#) similarly consider multigrid algorithms and fast Fourier transform. The conclusion of these studies is that the feasibility region for molecular dynamics (which is pleasantly parallel) and tree-based simulations is considerably less restricted than for other algorithms, having viable bandwidth requirements (in the order of 1 – 2 GB/s). These analysis-based studies add to the experimental demonstrations of both our and other groups' ongoing work with FMM to indicate great potential for this family of algorithms. Below, we explain some of the reasons for this potential.

Load balancing. Customarily, the FMM is load-balanced via space-filling curves, either Morton or

Hilbert. The latter is an improvement because the length of each segment of the space-filling curve is more uniform, i.e., it never jumps from one space location to another far away. These techniques balance only the computational work, without attempting to optimize communications. We have demonstrated an approach that optimizes both work and communications via a graph partitioning scheme in [Cruz et al. \(2010\)](#); however, this approach is not tested in large numbers of processors and extending it (by means of recursive graphs, e.g.) is an open research problem. In general, we believe that research into hierarchical load balancing techniques should be a high priority in our quest for exascale. The underlying tree-based structure of the FMM points to an opportunity in this case.

Spatial and temporal locality. The FMM is an algorithm that has intrinsic geometric locality, as the global N -body interactions are converted to a series of hierarchical local interactions associated with a tree data structure. This is illustrated in Figures 1 and 2. However, the access patterns could potentially be non-local. An established technique is to work with sorted particle indices, which can then be accessed by a start/offset combination (this also saves storage because not every index needs to be saved). Temporal locality is especially important on the GPU, and is moreover programmer-managed. We have implemented a technique that improves locality at a coarse level and is appropriate for GPUs, namely, an efficient queuing of GPU tasks before execution. The queuing is performed on the CPU, and buffers the input and output of data making mem-

ory access contiguous. On the other hand, locality at a fine level is accomplished by means of memory coalescing; this is natural on the FMM due to the indexing technique used. In conclusion, the FMM is *not* a “locality-sensitive application” (Bergman et al., 2008). This is in contrast to stencil-type calculations, for example, where non-contiguous memory access is unavoidable.

Global data communications and synchronization. The overheads associated with large data transfers and global barrier synchronization are a significant impediment to scalability for many algorithms. In the FMM, the two most time-consuming operations are the P2P (particle-to-particle) and M2L (multipole-to-local) kernels. The first is purely local, while the second effectively has what we can call *hierarchical synchronization*. The M2L operations happen simultaneously at every level of the hierarchical tree, without synchronization required *between* levels. This is in contrast, for example, to the situation in the multi-grid algorithm, which requires each level to finish before moving on to the next level. Among hierarchical algorithms, the FMM appears to have especially favorable characteristics for “exascaling”.

Future directions

Based on the performance study reported in this paper, and also the features of the FMM as mentioned above, we plan to continue improving our FMM code and addressing some immediate tasks. First of all, strong scaling results in Figure 8 show that communication and tree construction become a bottleneck for continuing to achieve strong scaling beyond 2000 processes. Therefore, improvement of the tree construction phase is necessary, as well as overlapping of the local computation and communication. The tree construction can be accelerated by taking the particle distribution into account. For example, many applications in fluid dynamics and molecular dynamics have a uniform distribution of particles. For these cases, the tree structure is very easily formed by assigning a Morton index for a prescribed maximum level. It will be useful to have the capability to switch between uniform and adaptive tree construction, depending on the application. The efficient overlapping of communications and computations is the obvious next step. We are currently working on these improvements, and others, which will be reported in separate publications.

These and other new features will be available in the ExaFMM code that we are releasing on the occasion of the Supercomputing conference in November 2011; for more information and access to the codes,

visit <http://www.bu.edu/exafmm>.

6 Conclusions

The present contribution aims to build on the mounting evidence of the suitability of the fast multipole method, FMM, and related algorithms, to reach exascale. The challenges to reach that milestone, which the community expects should be achieved by 2018, are varied and quite severe. They include serious technical hurdles, such as reducing the power consumption for computing by a significant factor, and dealing with frequent failures in systems involving millions of processors. While those challenges are recognized as the top concerns, the need for research into the appropriate algorithms and software for exascale is equally recognized as important. We maintain that FMM is among the algorithms that will play a visible role in the exascale ecosystem.

We have conducted a campaign of single-node optimization and massively parallel scalability studies of our FMM. Intra-node performance optimization is obtained both with multi-threading using OpenMP, and with vectorization using inline assembly for the most costly kernel. OpenMP parallelization of all kernels achieved 78% efficiency in strong scaling with 8 cores, while inline assembly vectorization of only the P2P kernel provided a $4\times$ speed-up of the overall algorithm.

Parallel scalability was studied both in strong and weak scaling. Strong scaling between 1 and 2048 processes obtained 93% parallel efficiency with the non-vectorized code, and 54% efficiency with the SIMD-capable code. The SIMD instructions sped up the calculation by $4\times$ when using less than a thousand processes, and by $2\times$ when using 2048 processes. In this last case, we start to see the inter-node communications and tree construction take a significant portion on the total runtime. However, even the vectorized code strong scales excellently up to one thousand processes, which is a situation rarely seen with other popular algorithms. Weak scaling tests resulted in 72% parallel efficiency on 32,768 processes for a test problem with a million points per process, and the largest calculation involved more than 32 billion points with a time to solution of under 40 seconds.

Further work is ongoing to improve parallel efficiency in strong scaling by overlapping of communications and computations and enhancing tree construction. All such improvements are available on real time in the code repository, and the code is open for unrestricted use under the MIT License.

Acknowledgements

This research was supported in part by the National Science Foundation (NSF) through TeraGrid resources provided by the National Institute for Computational Sciences (NICS) at University of Tennessee, under grant number TG-ASC100042. Additional support from NSF via grant OCI-0946441, from Office of Naval Research (ONR) via award #N00014-11-1-0356, and from Boston University College of Engineering. We also acknowledge guest access to the GPU cluster at the Nagasaki Advanced Computing Center and to TSUBAME 2.0 at the Tokyo Institute of Technology. LAB is also grateful of the support from NVIDIA via an Academic Partnership award.

Author Biographies

Rio Yokota obtained his PhD from Keio University, Japan, in 2009 and went on to work as a postdoctoral researcher with Prof. Lorena Barba at the University of Bristol and then Boston University. He has worked on the implementation of fast N -body algorithms on special-purpose machines such as MD-GRAPE-3, and then on GPUs after CUDA was released, and on vortex methods for fluids simulation. In 2009, he co-authored a paper awarded the ACM Gordon Bell prize in the price/performance category, using GPUs. He joins the King Abdullah University of Science and Technology (KAUST) as a research scientist in September 2011.

Lorena Barba is an Assistant Professor of Mechanical Engineering at Boston University since 2008. She obtained her PhD in Aeronautics from the California Institute of Technology in 2004, and then joined Bristol University in the United Kingdom as a Lecturer in Applied Mathematics. Her research interests include computational fluid dynamics, especially particle methods for fluid simulation; fundamental and applied aspects of fluid dynamics, especially flows dominated by vorticity dynamics; the fast multipole method and applications; and scientific computing on GPU architecture. She was a recipient of the Engineering and Physical Sciences Research Council (EPSRC) First Grant in 2007, and was awarded an NVIDIA Academic Partnership grant in 2011.

References

- Bergman, K., S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, 2008: Exascale computing study: Technology challenges in achieving exascale systems. Tech. rep., DARPA IPTO.
- Bhatele, A., P. Jetley, H. Gahvari, L. Wesolowski, W. Gropp, and L. Kale, 2011: Architectural constraints to attain 1 exaflop/s for three scientific application classes. In *Parallel Distributed Processing Symposium (IPDPS)*, 2011 *IEEE International*, pp. 80–91.
- Chandramowliswaran, A., S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc, 2010: Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, IEEE, pp. 1–12.
- Cheng, H., J. Huang, and T. J. Leitnerman, 2006: An adaptive fast solver for the modified Helmholtz equation in two dimensions. *J. Comput. Phys.*, **211**(2), 616–637.
- Cruz, F. A., M. G. Knepley, and L. A. Barba, 2010: PetFMM—a dynamically load-balancing parallel fast multipole library. *Int. J. Num. Meth. Engineering*, **85**(4), 403–428.
- Ethridge, F. and L. Greengard, 2001: A new fast-multipole accelerated Poisson solver in two dimensions. *SIAM J. Sci. Comput.*, **23**(3), 741–760.
- Feng, W.-c. and K. W. Cameron, 2009: The Green500 List. <http://www.green500.org/>.
- Gahvari, H. and W. D. Gropp, 2010: An introductory exascale feasibility study for FFTs and multigrid. In *Int. Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, Atlanta, GA, pp. 1–9.
- Greengard, L. and V. Rokhlin, 1987: A fast algorithm for particle simulations. *J. Comput. Phys.*, **73**(2), 325–348.
- Hamada, T., T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, 2009: 42 TFlops hierarchical N -body simulations on GPUs with applications in both astrophysics and turbulence. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ACM, ISBN 978-1-60558-744-8, pp. 1–12.
- Karypis, G. and V. Kumar, 1998: A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Par. Dist. Comp.*, **48**, 71–85.
- Langston, H., L. F. Greengard, and D. Zorin, 2011: A free-space adaptive FMM-based PDE solver in three dimensions. *Comm. Applied Math. Comput. Science*, **6**(1), 79–122.
- Lashuk, I., A. Chandramowliswaran, H. Langston, T. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, 2009: 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*, Portland, Oregon, pp. 1–12.

- Rahimian, A., I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros, 2010: Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society, SC '10, ISBN 978-1-4244-7559-9, pp. 1–11.
- Warren, M., J. Salmon, D. Becker, M. Goda, T. Sterling, and W. Winckelmans, 1997: Pentium Pro inside: I. A treecode at 430 Gigafllops on ASCI Red, II. Price/performance of \$50/Mflop on Loki and Hyglac. In *Supercomputing, ACM/IEEE 1997 Conference*, p. 61.
- Warren, M. S. and J. K. Salmon, 1992: Astrophysical N -body simulations using hierarchical tree data structures. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, IEEE Computer Society Press, Los Alamitos, CA, USA, ISBN 0-8186-2630-5, pp. 570–576.
- , 1993: A parallel hashed oct-tree N -body algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ACM, New York, pp. 12–21.
- Ying, L., G. Biros, and D. Zorin, 2004: A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *J. Comput. Phys.*, **196**(2), 591–626.
- Yokota, R., L. A. Barba, T. Narumi, and K. Yasuoka, 2011a: Petascale turbulence simulation using a highly parallel fast multipole method. Submitted, preprint on [arXiv:1106.5273](https://arxiv.org/abs/1106.5273).
- Yokota, R., J. P. Bardhan, M. G. Knepley, L. A. Barba, and T. Hamada, 2011b: Biomolecular electrostatics using a fast multipole BEM on up to 512 GPUs and a billion unknowns. *Comput. Phys. Comm.*, **182**(6), 1271–1283.