CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–17
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

PGAS-FMM: Implementing a distributed fast multipole method using the X10 programming language

Josh Milthorpe[1*], Alistair P. Rendell[1] and Thomas Huber[2]

[1]*Research School of Computer Science, Australian National University*
[2]*Research School of Chemistry, Australian National University*

SUMMARY

The fast multipole method (FMM) is a complex, multi-stage algorithm over a distributed tree data structure, with multiple levels of parallelism and inherent data locality. X10 is a modern partitioned global address space language with support for asynchronous activities. The parallel tasks comprising FMM may be expressed in X10 using a scalable pattern of activities. This paper demonstrates the use of X10 to implement FMM for simulation of electrostatic interactions between ions in a cyclotron resonance mass spectrometer.

X10's task-parallel model is used to express parallelism using a pattern of activities mapping directly onto the tree. X10's work stealing runtime handles load balancing fine-grained parallel activities, avoiding the need for explicit work sharing. The use of global references and active messages to create and synchronize parallel activities over a distributed tree structure is also demonstrated.

In contrast to previous simulations of ion trajectories in cyclotron resonance mass spectrometers, our code enables both simulation of realistic particle numbers and guaranteed error bounds. Single-node performance is comparable with the fastest published FMM implementations, and critical expansion operators are faster for high accuracy calculations. A comparison of parallel and sequential codes shows the overhead of activity management and work stealing in this application is low. Scalability is evaluated for 8k cores on a Blue Gene/Q system and 512 cores on a Nehalem/InfiniBand cluster. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: X10; Partitioned Global Address Space (PGAS); active messages; parallel programming models; scientific computing; fast multipole method

1. INTRODUCTION

Tree codes are an important class of parallel algorithm for physical simulations, as they allow the approximate evaluation of long-range interactions in greatly reduced time compared to naïve pairwise calculation. Amongst tree codes, the fast multipole method (FMM)[1] is of interest for large-scale molecular dynamics simulation, due to its low computational complexity of $\mathcal{O}(N)$ where N is the number of particles. FMM gives rigorous error bounds for the calculation of potential and forces, in contrast to particle-mesh methods, where only an error estimate may be available[2].

*Correspondence to: Research School of Computer Science, Building 108, Australian National University, Canberra ACT 0200 Australia. E-mail: josh.milthorpe@anu.edu.au

Contract/grant sponsor: NCI National Facility at ANU

Contract/grant sponsor: Australian Research Council and IBM; Australian Research Council; contract/grant number: Linkage grant LP0989872; Future Fellowship FT0991709

This well-defined error behavior makes FMM a suitable choice for applications where provable accuracy is required. Also in contrast to particle-mesh methods, FMM supports both periodic and non-periodic systems.

The motivation for this work is the desire to simulate the interactions between charged particles in a mass spectrometer. The remainder of this section outlines the requirements of our application, identifies important considerations for parallel implementation of tree codes in general and FMM in particular, and introduces the X10 programming language used for our implementation.

1.1. Application requirements

We aim to simulate the behavior of a packet of ions in a Fourier-transform ion cyclotron resonance mass spectrometer (FTICR-MS) over an experimentally meaningful timescale. A typical packet contains $10^4 - 10^6$ ions in circular motion with cycle times on the order of microseconds. For adequate frequency resolution, a measurement period of tens of milliseconds is required. However, to avoid significant error in integrating the ion trajectories requires a simulation timestep of tens of nanoseconds[3, 4]. Thus a full simulation requires around a million timesteps. The circular motion is generated by a Penning trap, which consists of a uniform magnetic field and an approximately quadrupolar electric trapping field. The apparatus is cubic in shape, which is highly convenient for simulation in an octree of cubic boxes. As ions can hit the wall of the Penning trap and effectively "disappear", periodic boundary conditions should not be imposed if the simulation is to accurately reproduce experiment. Moreover particles that venture outside of the domain defined by the octree should simply be removed from the simulation. In the code developed here the ions are modeled as point charges moving under the influence of electric and magnetic fields, interacting with each other only through non-bonded electrostatic forces. The non-periodic nature of the experimental environment plus the dominance of the computation time by evaluation of long-range electrostatic interactions makes FMM the ideal method to use for this problem.

1.2. Common features of tree codes

Tree codes enable rapid computation of N-body interactions by approximating distant interactions using aggregate expansions. Each box[†] in the tree holds an aggregate representation of all particles within that box; the representation for each parent box is created by combining the representations for all its child boxes. This suggests the use of a divide-and-conquer programming model, such as fork-join. There is a natural locality defined by the tree division of space; boxes that are near each other in the tree interact more strongly or frequently than more distant boxes.

When using a shared memory space, the most natural representation of the tree uses pointers between parent and child boxes. If the memory is shared but has non-uniform access characteristics then exploiting locality becomes critical to achieving good performance[5]. However, if the tree is distributed across multiple memory spaces, pointers in one space are not meaningful in another without some sort of translation. This consideration led Warren and Salmon[6] to propose the *hashed octree*, in which the boxes are stored in a hash table indexed by Morton encoding. This has formed the basis of most distributed tree codes for the last two decades. However, modern partitioned global address space (PGAS) languages allow for the possibility of distributed pointer-based tree structures, which we explore in §3.3.

Traversing the tree involves executing a given function over every box in a specified order. A *pre-order* traversal operates on parent boxes first, followed by their children; a *post-order* traversal operates on child boxes first. To allow portions of the tree to be treated approximately for long-range interactions, each box holds an aggregate representation of the sub-tree rooted at that box. Aggregate representations are carried downwards in a pre-order traversal, and upwards in a post-order traversal.

[†]In this section, the terms *box* and *particle* are used instead of the more general terms *node* and *source / target point*. We use these terms for easier comparison with the discussion of FMM and molecular dynamics application in following sections.

DOI: 10.1002/cpe

1.3. Specific features of the fast multipole method (FMM)

In the 3D FMM[1], the simulation space is divided into an octree of cubic boxes. Interactions between particles in nearby boxes are evaluated directly, whereas distant interactions are evaluated by means of series expansions around box centers or equivalent densities.

There are many different choices of series expansion for the fast multipole method, including spherical harmonics[7], plane wave expansions[1], equivalent charges[8] and Cartesian Taylor expansions[9]. The different expansions and operations have different computational and memory complexity with regard to the order of expansion (and consequent accuracy). Our code, PGAS-FMM, uses spherical harmonics with rotation-based translation and transformation operations, as these have the lowest asymptotic complexity[9]. The expansions and operations are as described by White and Head-Gordon[10] with rotation matrix relations as given by Dachsel[11].

FMM is characterized by irregular, but highly localized data access patterns. The algorithm comprises several steps:

- generation of multipole expansions from particles in each leaf box (*P2M*);
- a post-order traversal (upward pass) combining multipole expansions at higher levels in the tree (*M2M*);
- transformation from multipole to local expansions (*M2L*);
- a pre-order traversal (downward pass) translating parent local expansions to child boxes (*L2L*);
- evaluation of far-field interactions using local expansions for leaf boxes (*L2P*); and
- direct evaluation of near-field interactions (*P2P*).

The fast multipole method allows *a priori* calculation of strict error bounds[7]. The tradeoff between accuracy and computation time is controlled by parameters to the algorithm: the number of terms p in the multipole and local expansions and the maximum depth of the tree D_{max} or the number of particles per lowest-level box q.

Load balancing is an issue in FMM for two reasons: the difference in number of interactions between boxes on the edge of the simulation space compared with those for interior boxes, and differences in the work lists for near- and far-field interactions[12]. The first issue does not arise in molecular dynamics simulations with periodic boundary conditions, as with the periodic FMM every box can be considered an 'inside' box (with 26 neighbors). However for our simulation it is of real importance as periodic boundary conditions do not apply. The second issue occurs because the main components of the near- and far-field interactions - the *P2P* and *M2L* steps - are proportional to the number of particles within neighboring boxes and the number of boxes in well-separated boxes respectively. As these values are not directly correlated, the balance of work for each component will be different for each box. One approach to load balancing uses a separate domain decomposition for the near- and far-field computations[12]. An alternative is to use a single domain decomposition based on a combined work estimate for both components[13]. We describe such a combined work estimate in §3.2.

1.4. X10 programming language

The X10 programming language[14] is a modern asynchronous partitioned global address space (APGAS) language, which incorporates a number of concepts that address the concerns listed above. The sequential core of the language is similar to and largely interoperable with Java. The main differences with Java are the explicit representation of data locality in the form of *places*, and the expression of task parallelism through asynchronous *activities*. Every object has a host place, and may be accessed remotely via *global references* which may be passed between places. Active messages are used to combine data transfer, distributed task creation and synchronization. Efficient complex arithmetic is implemented through *structs*, which are headerless datatypes supporting inlined user-definable operations[15].

Load balancing within a place is supported by a work stealing runtime[16]. The programmer exposes parallelism by creating lightweight thread objects which are termed *activities*. The runtime maintains a pool of worker threads, each of which processes a double-ended queue (deque) of activities. Each worker adds and removes activities at the head of its own queue. When a worker

DOI: 10.1002/cpe

becomes idle it attempts to steal work from the tail of another thread's queue. The runtime uses cooperative scheduling where an activity runs to completion unless it executes a blocking statement or explicit yield[17].

X10 can be compiled to generate either C++ or Java code, with dependencies on runtime libraries[17]. The generated C++ code is compiled with a platform C++ compiler to form an executable. The Java code is compiled to bytecode and run on a JVM.

2. RELATED WORK

Early simulations of ion trajectories in FTICR mass spectrometry focused on single-ion behavior, neglecting Coulomb interactions. Recent simulations model Coulomb interactions using either virtual particles[18] or particle-in-cell approximations[19, 20, 21]. There appears to have been little consideration of the accuracy of electrostatic force evaluation. In contrast, our use of FMM allows the calculation of strict bounds on the truncation error for both force and potential calculations.

exaFMM[9] is an implementation of FMM that uses Cartesian Taylor expansions and is optimized for low accuracy (≤ 3 decimal digits) calculations. To our knowledge it is currently the fastest published FMM implementation for low accuracy when running on a single Intel node. Single-precision arithmetic is used, and the number of terms, p, in expansions is a compile-time constant, which enables template meta-programming with specialization for low p. Although Cartesian Taylor expansions have a higher asymptotic complexity ($\mathcal{O}(p^6)$) compared to spherical harmonics ($\mathcal{O}(p^3)$), they are always faster for low p due to smaller pre factors[9]. exaFMM has been parallelized using data-driven execution and work stealing within a single node[22].

A series of papers describing octree construction using the DENDRO package[23, 24, 13] shows how FMM tree construction may be scaled to very large numbers of particles and processes. This article focuses on the evaluation of potential and forces and does not treat tree construction in detail; we direct interested readers to the above papers for a more thorough treatment.

3. IMPLEMENTATION

3.1. Finding parallelism

Typical HPC clusters provide multiple levels of parallelism, including between distributed nodes and between cores within a single shared memory node. The most widely used approach to exploiting this parallelism is a hybrid programming model, in which message passing (MPI) is used between nodes, and a shared-memory programming model such as OpenMP is used within each node. In contrast, X10 supports a unified programming model which exploits parallelism at both levels.

The fast multipole method is a complex, multi-stage algorithm which allows for parallelism at multiple levels. The most basic parallelism is between computation of near- and far-field interactions, which are independent other than the need to synchronize updates to forces on particles. Within each stage of the algorithm at a given level, evaluation of each box may proceed independently, although there are dependencies between stages and levels. For example, in the downward pass, the local expansion of the parent box is an input to the computation of the local expansion of each child box.

Our code uses activities of box-level granularity, and overlaps computation with communication where possible. At a high level, the fetching of particle data for near-field computation overlaps with all bar the final phase of computation, as follows:

```
1 // at each place
2 finish {
3   async fetchParticleData();
4   upwardPass();
5   multipolesToLocal();
6 }
7 downwardPass();
```


DOI: 10.1002/cpe

In the above code, the **async** statement on line 3 starts an activity to fetch particle data from neighboring places. Each place constructs a local essential tree and fetches particles from only those boxes which are necessary to compute near-field interactions at the current place. Fetching particle data proceeds in parallel with the `upwardPass()` and `multipolesToLocal()` phases. However the finish block (lines 2–6) ensures that fetching is completed before the `downwardPass()`, which includes near-field computation.

3.2. Load balancing

Exposing parallelism through parallel activities does not by itself ensure the efficient use of available processing resources. Load balancing between processing elements is necessary to ensure full utilization of resources. We consider both load balancing between distributed places, and load balancing between processing elements within a place.

3.2.1. Load balancing between places We chose to implement a single static domain decomposition of the FMM tree based on work estimates for each box at the lowest level. A full work estimate for FMM would contain many terms, accounting for computation and communication in each stage of the algorithm. However, a simple two-part estimate is possible based on two simplifying assumptions: first, that FMM is heavily compute-bound on current architectures (although this assumption may become false as early as 2020)[25]; and second, that only the P2P and M2L steps contribute significantly to the total runtime. Given these assumptions, the cost estimate for work due to a box B containing n particles is:

$$cost(B) = cost_u(B) + cost_v(B) \tag{1}$$

$$cost_u(B) = C_{P2P} \cdot n \cdot size(U(B)) \cdot q \tag{2}$$

$$cost_v(B) = C_{M2L} \cdot size(V(B)) \tag{3}$$

where q is the average number of particles per lowest level box, C_{P2P} is the cost of computing a single direct interaction and C_{M2L} is the cost of computing a single multipole-to-local transformation. C_{P2P} and C_{M2L} are estimated at runtime as part of the 'setup' for the method by executing a small number of P2P and M2L kernels on the target architecture. The U-list $U(B)$ is the list of all neighboring boxes. P2P (particle-to-particle) interactions are computed for all n particles in box B with every particle in all boxes in the U-list. The V-list $V(B)$ is the list of all well-separated boxes for which the parent boxes are not also well separated. M2L (multipole-to-local) transformations are computed for all boxes in the V-list.

The cost estimate above is used to divide the lowest-level boxes between places. Boxes are sorted using Morton ordering and portions of the Morton curve assigned to each place so that the cost of each portion is roughly equal.

3.2.2. Load balancing within a place We divide each tree traversal into one activity for each box. Therefore the number of activities in a traversal of a tree of D_{max} levels is

$$8^{D_{max}} + 8^{D_{max}-1} + ... + 8^2$$

(The topmost level of boxes is not included in the traversal as there are no far-field evaluations at this level.) For example, for a complete tree of 5 levels, the number of activities for a traversal is 37440.

The work required for traversal varies substantially between boxes; it is therefore necessary to load balance activities between the worker threads within a place. This is performed by the X10 work stealing runtime; once a thread becomes idle it attempts to steal work from the queue of another thread. Work stealing leads to good locality when applied to tree traversals, as the earliest created (and stolen) activities are those at the higher levels of the tree. The lower level activities created and processed by each thread will therefore tend to belong to the same subtree, which means that they can reuse cached data pertaining to that subtree.

DOI: 10.1002/cpe

As each activity is relatively long-lived (>100k cycles), the overhead of activity management and load balancing is expected to be small. We present performance results confirming this expectation in §4.2.

3.3. Distributed tree structure using global references

The tree is comprised of lowest-level boxes containing particles, represented by the class `LeafOctant`, and boxes at higher levels, represented by the class `ParentOctant`. Both classes inherit from `Octant`, defined as follows:

```
1  public abstract class Octant {
2    public id:OctantId;
3    public var parent:Octant;
4    public val multipoleExp:MultipoleExpansion;
5    public val localExp:LocalExpansion;
6    abstract traverse[T,U](parentRes:T,
7      preFunc:(a:T)=>T,
8      postFunc:(c:List[U])=>U
9    ):U;
10   ...
11 }
```

The PGAS programming model in X10 allows for the construction of distributed pointer-based data structures. X10 provides a special type `GlobalRef`, which is a global reference to an object at one place that may be passed to any other place. Rather than using `GlobalRef` directly for child or parent references, we use it to implement a proxy class, `GhostOctant`, which sends an active message to the host place of a remote box. If a child box is held at a different place to its parent, a `GhostOctant` is created for the child and linked from the parent box.

The following code performs the traversal of a subtree for a parent octant:

```
12 class ParentOctant extends Octant {
13   public val children:Rail[Octant];
14   traverse[T,U](parentRes:T,
15     preFunc:(a:T)=>T,
16     postFunc:(c:List[U])=>U
17   ):U {
18     val myRes = preFunc(parentRes);
19     val childRes = new Rail[T](numChildren);
20     finish for([i] in children) {
21       async childRes[i] =
22         children[i].traverse(myRes, preFunc, postFunc);
23     }
24     val x = postFunc(childRes);
25     // store for use by ghost
26     atomic this.result = x;
27     return result;
28   }
29   traverse[T,U](postFunc:(c:List[U])=>U):U {
30     val childRes = new Rail[T](numChildren);
31     finish for([i] in children) {
32       async childRes[i] =
33         children[i].traverse(postFunc);
34     }
35     val x = postFunc(childRes);
36     return result;
37   }
38 }
```

On lines 15 and 16, `preFunc` and `postFunc` are arbitrary *closures*, that is, functions provided together with their variable environments. In the fast multipole method they are used to translate or transform multipole expansions. Lines 20–23 traverse each child in parallel using the result of `preFunc` for the parent (a pre-order traversal). Line 24 executes `postFunc` for the parent, which typically involves a reduction over the result of `postFunc` for all children (a post-order traversal).

DOI: 10.1002/cpe

Line 26 atomically sets the result field for the parent octant. The atomic statement is necessary to allow progress for any activities that are currently waiting on the result field. Lines 29–37 are a simpler version of traversal for the case where only a post-order traversal is required.

During tree traversal, the `GhostOctant` creates an active message to the child's host place to get the aggregate representation of the child. If a pre-order traversal is required, the active message initiates computation at the host place as follows:

```
31  class GhostOctant extends Octant {
32    private var target:GlobalRef[Octant];
33    traverse[T,U](parentRes:T,
34      preFunc:(a:T)=>T,
35      postFunc:(c:List[U])=>U
36    ):U {
37      at(target.home) {
38        val t = target();
39        return t.traverse(parentRes, preFunc, postFunc);
40      }
41    }
42    ...
```

Lines 37–40 generate an active message to the home place of the target octant to compute and return the aggregate representation of the octant.

If only post-order traversal is required, then the various subtrees at each place can be evaluated in parallel, without waiting for evaluation of parent octants held at other places. In this case, a `GhostOctant` can simply return the value of the host octant that has previously been computed at the host place, as follows:

```
43    ...
44    traverse[U](postFunc:(c:List[U])=>U):U {
45      at(target.home) {
46        val t = target();
47        when(t.result != null);
48        return t.result;
49      }
50    }
51  }
```

The statement **`when`**`(...);` on line 47 is a conditional atomic statement, i.e. it blocks the current activity until the condition is true, during which time other activities can perform useful work. The thread may proceed once the condition has been set to true by an atomic block in another activity initiated separately at the host place.

3.4. Global collective operations

Certain communication patterns in our current implementation of FMM use global tree-structured collective operations. For example, the load-balancing algorithm requires a count of particles held in each box. The local counts at each place are combined using a global all-reduce operation over all places. During far-field evaluation, a global barrier is used to ensure that all multipole expansions have been transferred before commencing the multipole-to-local (M2L) transformations. Barrier, all-reduce and other collective operations are provided by the `x10.util.Team` API, and are hardware accelerated where possible[17]. Exploratory benchmarks of these collective operations suggested that they are not a limiting factor for the small to moderate numbers of places considered here. For very large place counts this is likely, however, to become a bottleneck. At that point one alternative approach would be to communicate the multipole and particle data using active messages with only local synchronization between neighbors[26].

DOI: 10.1002/cpe

4. EVALUATION

There are many published implementations of FMM, which differ in both algorithm and technology (programming language, libraries) used to implement them. In evaluating the performance of our PGAS-FMM, we compare it to a state-of-the-art implementation, and determine whether performance differences are due to algorithm or technology. Specifically we consider i) the overall performance of the code when running on a single core and the efficiency of each major component of the algorithm; ii) the overhead of X10 activity management; iii) scaling with number of threads on a shared-memory system of each component; iv) multi-place scaling on two typical HPC architectures with different CPU and network characteristics; and v) performance of FMM within a complete FTICR simulation.

Evaluation was performed on three classes of parallel machines:

- a typical desktop machine, containing a Sandy Bridge quad-core Core i7-2600 with 8GB DDR3-1333 memory;
- a loosely coupled cluster machine: the *Vayu* Oracle/Sun Constellation cluster installed at the NCI National Facility at the Australian National University. Each node of *Vayu* is a Sun X6275 blade, containing two quad-core 2.93GHz Intel Nehalem CPUs, 24GB DDR3-1333 memory and on-board QDR InfiniBand; and
- a tightly integrated system with a custom interconnect: the *Watson 2Q* Blue Gene/Q system at IBM Watson Research Center. Each Blue Gene/Q compute node contains 16 PowerPC A2 1.6GHz compute cores (with an additional core for operating system services) and 16GB DDR-3 memory[27].

For all reported results, the Native (C++ backend) version of X10 2.3 was used. One multithreaded X10 place was created per socket, with `X10_NTHREADS` (the number of worker threads) equal to the number of cores. Thus `X10_NTHREADS`=4 was used for the quad-core CPUs of *Vayu*, and `X10_NTHREADS`=16 was used for the 16-core CPUs of *Watson 2Q*. On *Vayu*, Open MPI version 1.4.3 was used with the options `-bind-to-socket -cpus-per-proc 4` to run one process per quad-core CPU.

4.1. Single-threaded performance

We first compare the single-threaded performance of our code against exaFMM[28], which is an FMM implementation optimized for low accuracy calculations.

Figure 1 compares the performance of PGAS-FMM with that of exaFMM on a single Sandy Bridge core (Core i7-2600). We used both codes to calculate forces and potential to low accuracy (RMS force error $\approx 1 \times 10^{-2}$) for a range of particle numbers between 10,000 and 1,000,000. All simulations used uniform lattice distributions over $[-1, 1]^3$ with total charge $\sum q_i = 1$ and equal particle charges of $\frac{1}{n}$. For low accuracy, our code is between 9 and 25 times slower than exaFMM across this range. This difference is similar to that observed by Yokota[9] when the performance of exaFMM was compared with four other major open-source FMM or tree code implementations for a similar benchmark.

Table I. Component timings in seconds of PGAS-FMM on Core i7-2600 (1 thread) for varying numbers of particles and accuracies (low: $p = 3, \epsilon = 10^{-2}$, high $p = 6, \epsilon = 10^{-4}$).

		$n = 10^5$		$n = 10^6$	
component		low	high	low	high
tree construction		0.038	0.038	0.34	0.35
near		0.714	0.718	9.37	9.28
far	upward	0.118	0.150	1.15	1.44
	M2L	0.286	0.775	2.75	7.31
	downward	0.052	0.097	0.52	0.96
total		1.21	1.78	14.13	19.35

DOI: 10.1002/cpe

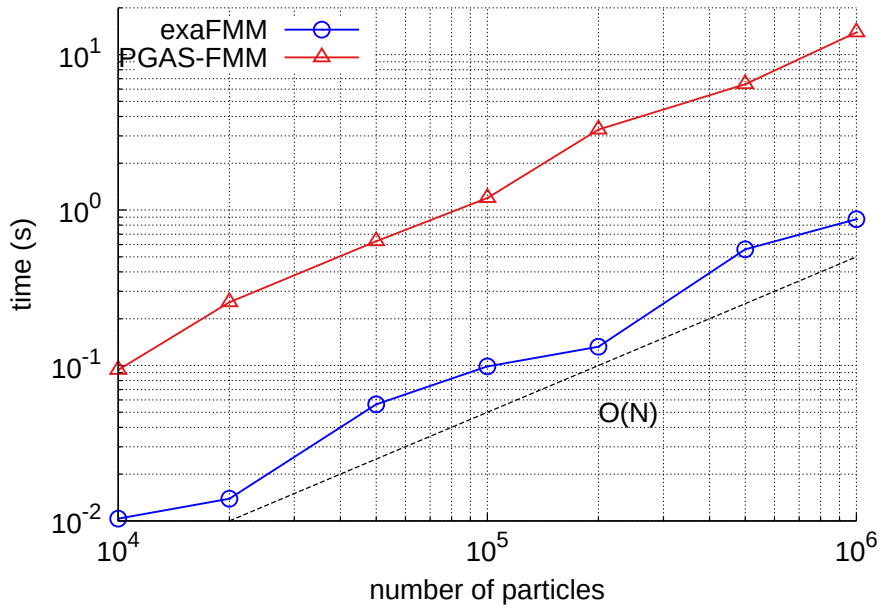


Figure 1. Comparison between PGAS-FMM and exaFMM on Core i7-2600 (1 thread): scaling with number of particles, low accuracy ($p = 3, \epsilon = 10^{-2}$).

We now focus on the execution time of PGAS-FMM, and consider the major components of the algorithm separately. Table I shows the breakdown of computation time between the major components: tree construction, far-field evaluation (upward, multipole-to-local transformations and downward pass) and near-field evaluation. Tree construction accounts for approximately 4-10% of the total time, with a roughly even split between near- and far-field evaluation. This is somewhat slower than what might be expected for a single thread, in part because the code assumes that the particles are distributed between places and require load balancing and redistribution before evaluation can occur. For low accuracy the near-field computation time is dominant representing 58% of the total for the "small" 10^5 particles simulation and 66% for the "large" 10^6 particles simulation. However, as the accuracy is increased the near and far field computation times become roughly equal.

As X10 is a new language with less well-understood performance characteristics than traditional languages such as C++ or Fortran, it is reasonable to ask how good the absolute performance of PGAS-FMM is, and what fraction of the observed performance difference between PGAS-FMM and exaFMM is due to algorithmic differences compared to inefficiencies in the language implementation. These issues are considered in detail in the following subsection for the near- and far-field components separately.

4.1.1. Near-field calculation The U-list (near field) interactions for each box are computed in a loop over neighboring boxes. All atom data for neighboring boxes have previously been retrieved and stored in Morton order, i.e. in a dense memory layout. This promotes better locality and cache re-use. For each box, the maximum number of boxes in the U-list is 27. There is a substantial overlap between U-lists for different boxes that are nearby in Morton order. Hence there is good temporal locality in the use of particle data in the U-list calculation. Given an average number of ions $q = 50$ per lowest level box, the working set size is approximately 43kB, which easily fits within the L2 cache on all the systems we used.

The calculation of a single near-field interaction (energy and forces) between two particles requires 9 adds, 9 multiplies, 1 division and 1 square root, for a total of 20 floating point operations. Using PAPI, we measured between 20 and 23 FP instructions per near-field interaction, and between

DOI: 10.1002/cpe

40 and 46 cycles per interaction for 60000 particles with $D_{max} = 4, ws = 1$. We measured identical execution time and FP instructions for an equivalent sequential C++ near-field kernel code, therefore the X10 language implementation does not add any overhead for this kernel.

The most expensive part of this calculation is the inverse square root. This requires a divide operation and a square root with a typical latency of around 40 cycles[29]. exaFMM and other codes use an approximate inverse square root operation to reduce this cost. For example, exaFMM uses single-precision Intel AVX and SSE reciprocal square root operations with a latency of 6 cycles. Using the Kernel Independent Fast Multipole Method and an approximate square root function, Chandramowlishwaran et al.[5] give a figure of 19 instructions for potential evaluation only, which takes roughly 17 cycles to execute. The generic GROMACS[30] non-bonded kernel for Coulomb interactions (nb_kernel100.c) contains 27 FLOPS per interaction, and also uses an approximate inverse square root.

4.1.2. Far-field calculation The major algorithmic consideration in the far-field calculation is the type of expansions and transformation operators used. By far the largest contribution to calculation time is the multipole-to-local (M2L) transformation, which converts a multipole expansion for a well separated box to a local expansion for a target box. Yokota[9, fig. 1] compared the performance of M2L operator using Cartesian Taylor expansions and transformations with a rotation-based M2L operator over spherical harmonic expansions. The average time for a single M2L transformation was measured for a complete FMM calculation for $n = 10^6$ particles. We replicated this experiment using the spherical harmonic expansions and operators implemented in PGAS-FMM, and report the results in figure 2. Whereas Yokota found a crossover point at $p \approx 12$ ($\epsilon \approx 10^{-7}$), we find that the crossover occurs much earlier at $p = 7$ ($\epsilon \approx 10^{-5}$).

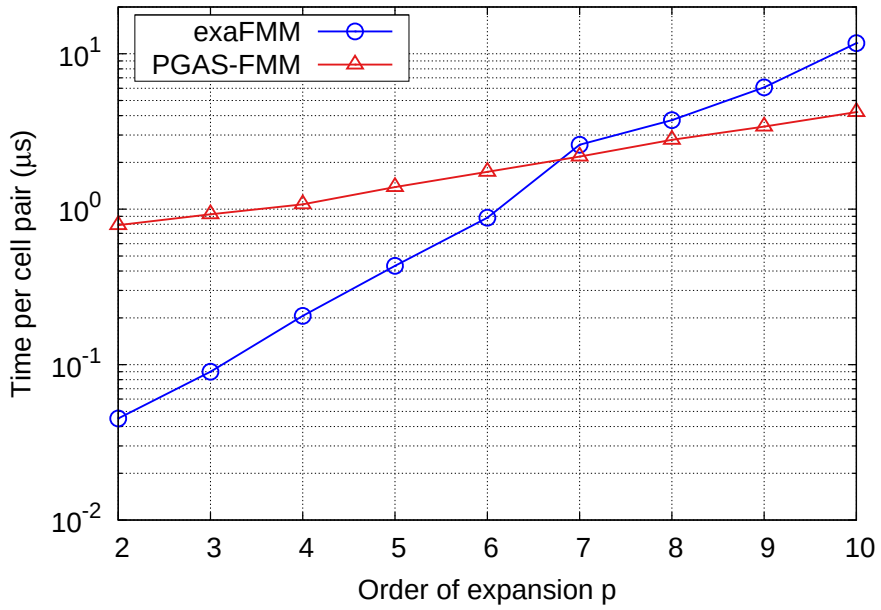


Figure 2. Time for M2L transformation on Core i7-2600 (8 threads) for different orders of expansion p ($n = 10^6$).

The M2L operations on the Cartesian Taylor expansions used in exaFMM scale as $\mathcal{O}(p^6)$, whereas the rotation-based operators in PGAS-FMM scale as $\mathcal{O}(p^3)$ but with a larger prefactor. It is therefore expected that PGAS-FMM would exhibit relatively better performance for higher-accuracy calculations with greater p. Figure 3 shows computation time for the same systems used in figure 1, this time with a slightly higher accuracy (RMS force error $\approx 1 \times 10^{-3}$). The exaFMM C++ code is now only 1.5-4.1 times faster than our X10 code ($p = 6$). The difference in scaling

DOI: 10.1002/cpe

suggests that algorithmic differences between the two codes are likely to be more important for far-field evaluation than differences due to the use of X10 vs. C++.

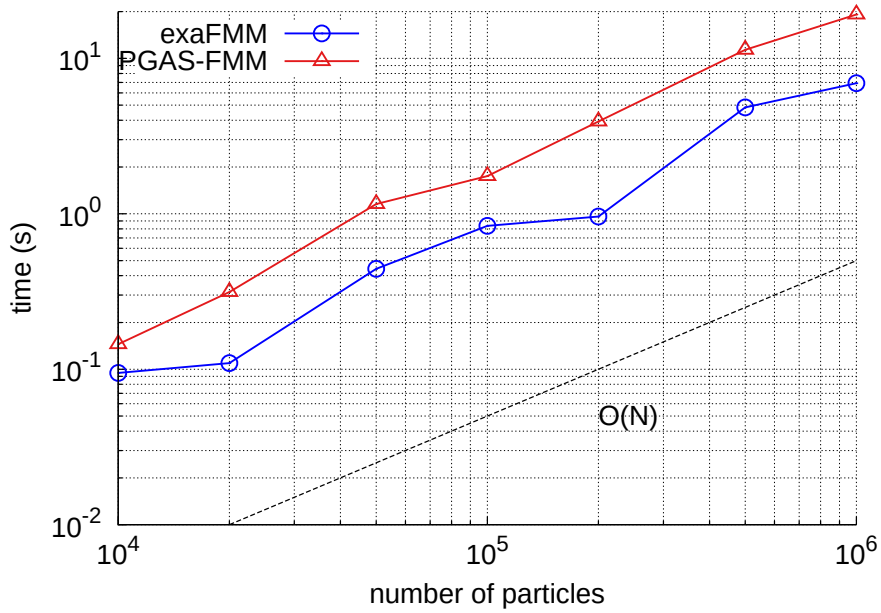


Figure 3. Comparison between PGAS-FMM and exaFMM on Core i7-2600 (1 thread): scaling with number of particles, higher accuracy ($p = 6$).

Having compared the overall performance of our code to that of a highly-optimized FMM implementation, we now consider its floating-point performance with respect to theoretical peak FLOPs on our desktop system. Darve et al.[31] report between 15% and 50% of peak FLOP/s achieved for a BLAS formulation of the M2L kernel with $p = 32$ running on a dual quad-core Intel Xeon E5345. Our M2L operations average 3782 cycles and 3938 FP instructions with $p = 6$ (force error $\leq 10^{-3}$) on an Intel Core i7-2600. This is a FP intensity of 1.04 FLOPs/cycle, which is 26% of peak FLOP/s. Our code achieves between 14% and 22% of peak FLOP/s for the entire FMM excluding the near-field calculation. This might be improved through appropriate use of SSE and/or AVX instructions.

4.2. Overhead of activity management

For efficient parallel execution it is necessary to divide FMM computation evenly between processing elements. Our code divides the work into a large number of activities to take advantage of dynamic load-balancing by X10's work stealing runtime. To assess the overhead of activity management we removed all `async` statements from the upward, M2L and downward components of the code, effectively converting it into a sequential program. We ran both sequential and parallel versions of our code on a single thread to compare component timings. Table II shows the difference in time between sequential and parallel versions of each component, and the proportional slowdown due to activity management overhead. The slowdown is less than 4% for each component suggesting that the overhead of activity management is relatively low.

4.3. Multithreaded scaling

Figure 4(a) shows multithreaded scaling on a single quad-core Sandy Bridge node of the different components of FMM computation time for the largest system used in §4.1 (10^6 particles) with p = 6 (RMS force error $\approx 1 \times 10^{-3}$).

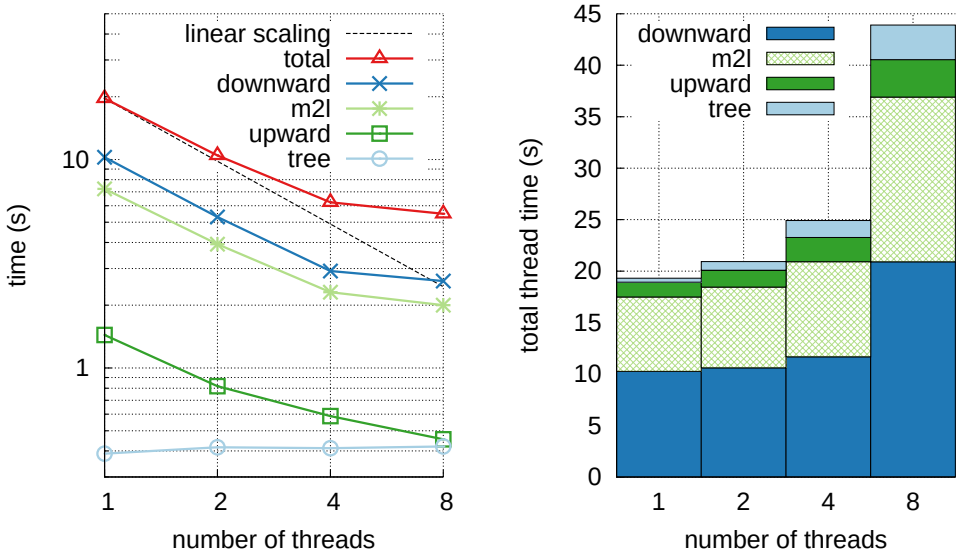
Figure 4(b) presents the same data in terms of multithreaded efficiency, showing the total thread time (elapsed time $\times$ number of threads). A component that exhibits perfect linear scaling shows

DOI: 10.1002/cpe

Table II. Slowdown due to X10 activity management overhead for PGAS-FMM on Core i7-2600 (1 thread) for low accuracy calculation ($n = 10^6$, $p = 3$).

component	time (s) sequential	parallel	slowdown
upward	1.12	1.16	1.03
M2L	2.65	2.67	1.009
downward	9.79	9.81	1.002

constant total thread time, while an increase in total thread time represents a loss of parallel efficiency.



(a) Scaling (b) Efficiency

Figure 4. Multithreaded component scaling and efficiency of PGAS-FMM on Core i7-2600 (1-8 threads, $n = 10^6$, $p = 6$).

On a single thread, the largest components are the 'downward pass', which includes direct calculations, and the multipole-to-local transformations for the V-list. These components show reduction in computation time from 1 to 4 threads, with a slight increase in total thread time reflecting imperfect load balancing due to differences in task size. There is a further reduction in computation time for 8 threads as hyper-threading is used to schedule eight threads on four physical cores. The latter does however increase the total thread time as threads compete for resources. Further experiments (not shown) found no additional performance improvement above 8 threads.

4.4. Distributed scaling

Figure 5 shows strong scaling for FMM force calculation on a uniform lattice distribution of 1,000,000 particles on *Vayu*. The maximum tree depth for this problem size is 5 (32,768 boxes at the lowest level), and $p = 6$ terms were used in expansions for a force error of approximately 10^{-3}. Total computation time is shown along with the time for each of the major components for 1 to 256 places. The lowest total time (0.24s) is observed on 128 places (512 cores), compared to 7.6s on a single place (4 cores). Parallel efficiency reduces gradually due to less than linear scaling of the upward pass, which includes the time to send multipole expansions to neighboring places.

DOI: 10.1002/cpe

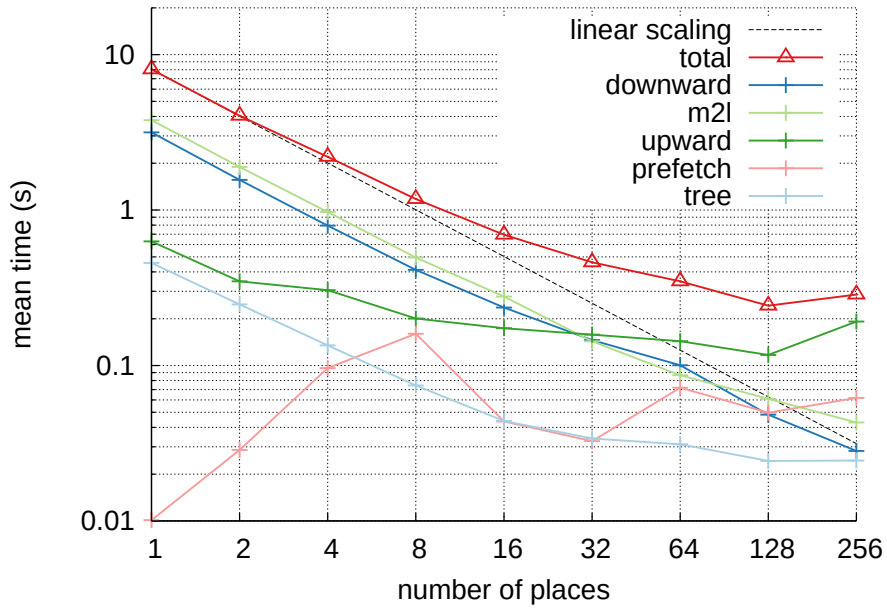


Figure 5. Strong scaling of FMM force calculation on *Vayu* (4 cores per place, $n = 10^6$, $p = 6$).

Strong scaling experiments were also conducted on the *Watson 2Q* Blue Gene/Q system. Blue Gene/Q represents a different system balance to the *Vayu* Nehalem/IB cluster, with a greater relative performance of the communication subsystem compared to floating point computation[27]. It also has substantially greater levels of parallelism; a single BG/Q compute node may execute up to 64 hardware threads (on 16 4-way SMP cores). The results are shown in figure 6.

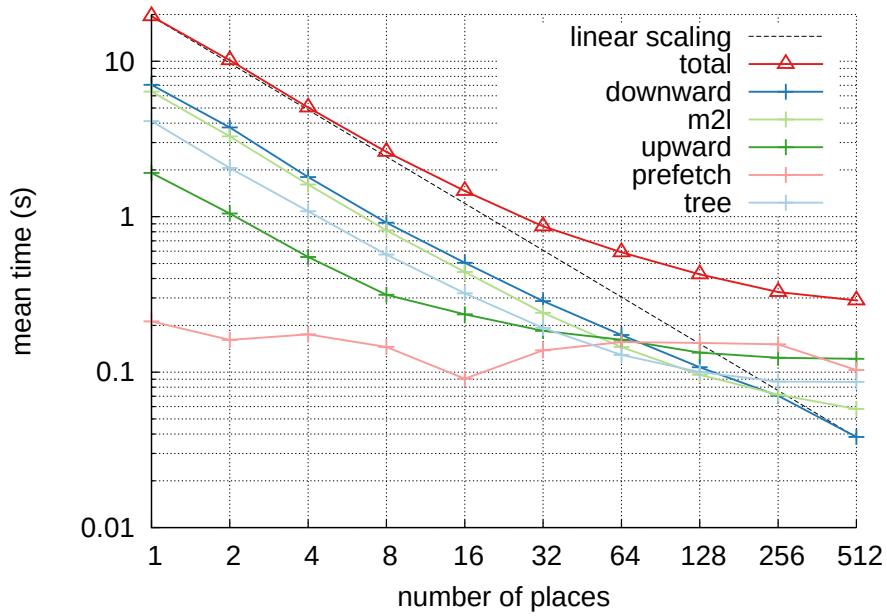


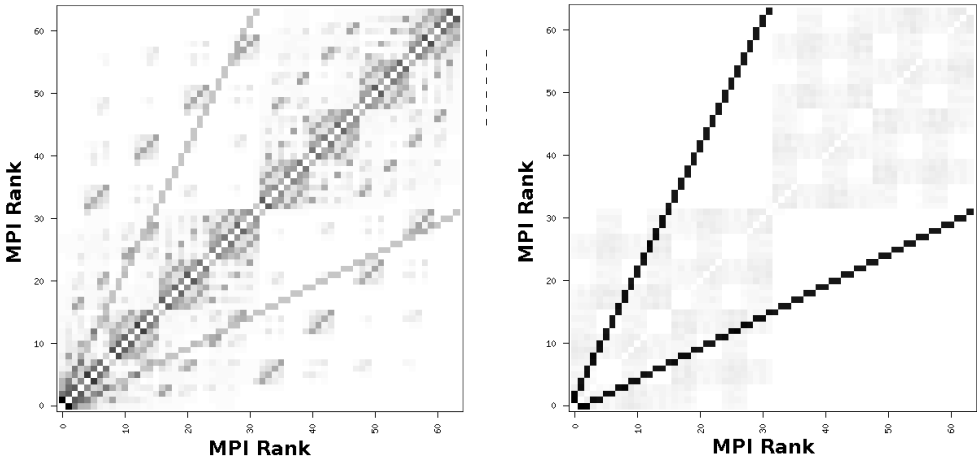
Figure 6. Strong scaling of FMM force calculation on *Watson 2Q* (16 cores per place, $n = 10^6$, $p = 6$).

Per core, *Watson 2Q* is significantly slower with a single place (16 cores) taking 15.3s to run the benchmark, which is about twice as long as a single place on *Vayu* (4 cores). The computation overall

DOI: 10.1002/cpe

scales better on *Watson 2Q* than it does on *Vayu*, which reflects the relatively higher performance of communications over the torus network, which makes communication-intensive components like the upward pass relatively cheaper on BG/Q. Also, key collective operations used in tree construction and the upward pass (see §3.4) are hardware accelerated. Total time reduces from 19.5s on 1 place (16 cores) to 0.28s on 512 places (8192 cores).

Figure 7(a) is a heatmap of the MPI pairwise communications with 64 processes on *Vayu* for a single FMM force calculation, profiled using IPM. Darker areas on the map indicate larger volumes of communications between processes. The heatmap demonstrates that the communication pattern is relatively localized, with large amounts of data exchanged between neighboring processes and very little between more distant processes. A noticeable feature of the communication topology is the two strong off-diagonal lines. These represent global tree-structured collective communications (broadcast and all-reduce), which are used in tree construction.[‡] Figure 7(b) shows the communication topology for tree construction alone. Comparing the two figures, it is apparent that the communication pattern of FMM evaluation excluding tree construction is fractal-structured and mostly on-diagonal (between neighboring processes).



(a) Complete FMM force calculation (b) Tree construction only

Figure 7. MPI communication map between 64 processes for FMM force calculation on *Vayu*.

4.5. Application: FTICR mass spectrometer

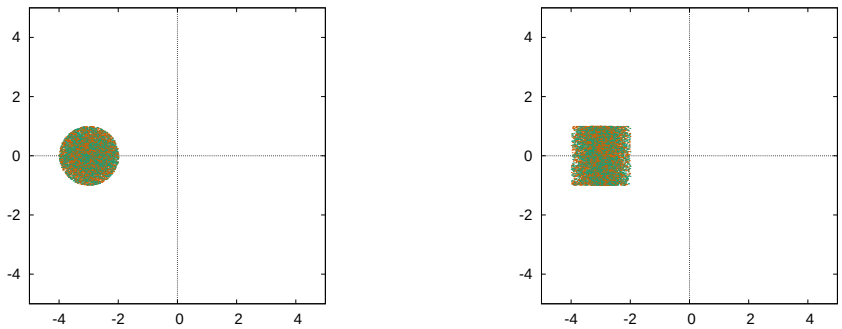
Finally, we present performance results for the simulation of ion cloud behavior in a FTICR mass spectrometer (see §1.1).

Figure 8 shows the ion cloud evolution in the first 2.5ms for a simulation of 10,000 ions. The Penning trap is a 5mm cube with a 4.7T magnetic field and a 1V trapping field. Initially there is a single ion cylindrical ion cloud composed of two species of similar mass/charge ratio: glutamine ($m/q = 147.07698$) and lysine ($m/q = 147.11336$). The simulation consists of timesteps of length 25ns, using FMM evaluation of electrostatic interactions with $p = 8$. After 2.5ms, the clouds have separated and are beginning to form a comet shape. This has been observed previously in other simulations[19] and attributed to the ion-ion interactions.

We ran simulations with varying numbers of particles and timesteps for ion clouds of amino acids of similar mass/charge ratios. Table III shows the computation time per timestep, FMM evaluation and tree construction times for varying number of particles and maximum tree depth.

[‡]The MPI collective functions `MPI_Bcast` and `MPI_Allreduce` exhibit similar communication patterns.

DOI: 10.1002/cpe



(a) time 0: X-Y projection

(b) time 0: X-Z projection

(c) time 2.5ms: X-Y projection

(d) time 2.5ms: X-Z projection

Figure 8. FTICR-MS ion cloud evolution in the first 2.5ms of simulation: packet of 5K lysine and 5K glutamine.

Table III. Amino acids in ANU mass spectrometer.
Timings for one place (4 cores) to 128 places (512 cores) on *Vayu*, ($p = 6$).

		time per cycle (s)		
			FMM	
N	places	**total**	**evaluate**	**construct tree**
10^5	1	3.05	2.94	0.085
10^5	16	2.40	1.11	0.93
10^5	32	2.23	0.74	1.24
10^5	64	2.72	0.53	1.98
10^5	128	4.08	0.44	3.48

5. CONCLUDING REMARKS

Our implementation of the fast multipole method demonstrates the use of the X10 programming language and the asynchronous partitioned global address space model to express an efficient pattern of parallel tasks in a complex, multi-phase application. The work stealing runtime provides dynamic load balancing between worker threads within a place and was shown to have a low overhead due to activity management.

On the multi-socket nodes of the *Vayu* cluster, we performed experiments using one X10 place per socket. We speculate that on typical architectures, it will be most efficient to run one X10 place per uniform memory area as we have done.

Our results demonstrate that the fast multipole method using spherical harmonics with rotations is faster than using Cartesian Taylor expansions for target RMS force error $\leq 10^{-3}$. This suggests

DOI: 10.1002/cpe

that a general purpose FMM may implement multiple different expansion types, and switch between them depending on which is fastest for the target accuracy.

The application code described in this paper is available at `http://cs.anu.edu.au/~Josh.Milthorpe/anuchem.html` and is free software under the Eclipse Public License.

ACKNOWLEDGEMENTS

We would like to thank David Grove at IBM Watson Research Center and Andrey Blizniuk from the NCI National Facility at the ANU. Andrew Haigh implemented rotation-based operators for the FMM code. This work was supported by the Australian Research Council and IBM through Linkage grant LP0989872, and by the NCI National Facility at the ANU. Thomas Huber thanks the Australian Research Council for a Future Fellowship (FT0991709).

REFERENCES

1. Greengard L, Rokhlin V. A new version of the Fast Multipole Method for the Laplace equation in three dimensions. *Acta Numerica* 1997; **6**:229.
2. Deserno M, Holm C. How to mesh up Ewald sums. II. an accurate error estimate for the particle-particle-particle–mesh algorithm. *Journal of Chemical Physics* Nov 1998; **109**(18):7694–7701, doi:doi:10.1063/1.477415.
3. Birdsall C, Langdon A. *Plasma Physics via Computer Simulation*. McGraw-Hill, New York, 1985.
4. Patacchini L, Hutchinson I. Explicit time-reversible orbit integration in particle in cell codes with static homogeneous magnetic field. *Journal of Computational Physics* 2009; **228**:2604, doi:10.1016/j.jcp.2008.12.021.
5. Chandramowlishwaran A, Madduri K, Vuduc R. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*, 2010, doi:10.1109/SC.2010.19.
6. Warren M, Salmon J. A parallel hashed oct-tree n-body algorithm. *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, 1993; 12–21.
7. White C, Head-Gordon M. Derivation and efficient implementation of the fast multipole method. *Journal of Chemical Physics* 1994; **101**(8):6593–6605.
8. Ying L, Biros G, Zorin D. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics* Jan 2004; **196**:591, doi:10.1016/j.jcp.2003.11.021.
9. Yokota R. An FMM based on dual tree traversal for many-core architectures Sep 2012. arXiv:1209/3516v3.
10. White C, Head-Gordon M. Rotating around the quartic angular momentum barrier in fast multipole method calculations. *Journal of Chemical Physics* Nov 1996; **105**(12), doi:10.1063/1.472369.
11. Dachsel H. Fast and accurate determination of the Wigner rotation matrices in the fast multipole method. *Journal of Chemical Physics* Apr 2006; **124**(14):144 115, doi:10.1063/1.2194548.
12. Kurzak J, Pettitt BM. Massively parallel implementation of a fast multipole method for distributed memory machines. *Journal of Parallel and Distributed Computing* 2005; **65**:870–881, doi:10.1016/j.jpdc.2005.02.001.
13. Lashuk I, Chandramowlishwaran A, Langston H, Nguyen TA, Sampath R, Shringarpure A, Vuduc R, Ying L, Zorin D, Biros G. A massively parallel adaptive fast-multipole method on heterogeneous architectures. *Proceedings of the 2009 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC09)*, 2009.
14. Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioğlu K, von Praun C, Sarkar V. X10: an object-oriented approach to non-uniform cluster computing. *Proceedings of OOPSLA '05*, ACM, 2005; 519–538, doi: 10.1145/1094811.1094852.
15. Milthorpe J, Ganesh V, Rendell A, Grove D. X10 as a parallel language for scientific computation: practice and experience. *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011)*, 2011; 1080–1088.
16. Tardieu O, Wang H, Lin H. A work-stealing scheduler for X10's task parallelism with suspension. *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012)*, 2012, doi:10.1145/2370036.2145850.
17. Grove D, Tardieu O, Cunningham D, Herta B, Peshansky I, Saraswat V. A performance model for X10 applications. *Proceedings of the ACM SIGPLAN 2011 X10 Workshop*, 2011.
18. Fujiwara M, Happo N, Tanaka K. Influence of ion-ion Coulomb interactions on FT-ICR mass spectra at a high magnetic field: A many-particle simulation using a special-purpose computer. *Journal of the Mass Spectrometry Society of Japan* 2010; **58**:169–173.
19. Nikolaev E, Heeren R, Popov A, Pozdneev A, Chingin K. Realistic modeling of ion cloud motion in a Fourier transform ion cyclotron resonance cell by use of a particle-in-cell approach. *Rapid Communications in Mass Spectrometry* 2007; **21**:3527, doi:10.1002/rcm.3234.
20. Leach F, Kharchenko A, Heeren R, Nikolaev E, Amster I. Comparison of particle-in-cell simulations with experimentally observed frequency shifts between ions of the same mass-to-charge in Fourier transform ion cyclotron resonance mass spectrometry. *Journal of the American Society for Mass Spectrometry* Oct 2009; **21**:203–208, doi:10.1016/j.jasms.2009.10.001.
21. Vladimirov G, Hendrickson C, Blakney G, Marshall A, Heeren R, Nikolaev E. Fourier transform ion cyclotron resonance mass resolution and dynamic range limits calculated by computer modeling of ion cloud motion. *Journal of the American Society for Mass Spectrometry* Dec 2011; **23**:375, doi:10.1007/s13361-011-0268-8.
22. Ltaief H, Yokota R. Data-driven execution of fast multipole methods. *CoRR* 2012; arXiv:abs/1203.0889.

23. Sundar H, Sampath RS, Adavani SS, Davatzikos C, Biros G. Low-constant parallel algorithms for finite element simulations using linear octrees. *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, ACM: New York, NY, USA, 2007; 25:125:12, doi:10.1145/1362622.1362656.
24. Sundar H, Sampath RS, Biros G. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing* Jan 2008; **30**(5):2675–2708, doi:10.1137/070681727.
25. Chandramowlishwaran A, Choi J, Madduri K, Vuduc R. Towards a communication optimal fast multipole method and its implications at exascale. *Proceedings of the 24th ACM symposium on Parallelism in Algorithms and Architectures (SPAA '12)*, 2012; 182–184, doi:10.1145/2312005.2312039.
26. Milthorpe J, Rendell A. Efficient update of ghost regions using active messages. *Proceedings of the 19th IEEE International Conference on High Performance Computing (HiPC)*, 2012.
27. Haring R, Ohmacht M, Fox T, Gschwind M, Satterfield D, Sugavanam K, Coteus P, Heidelberger P, Blumrich M, Wisniewski R, *et al.*. The IBM Blue Gene/Q compute chip. *Micro, IEEE* 2012; **32**(2):48–60.
28. exafmm. URL `https://bitbucket.org/rioyokota/exafmm-dev`, accessed: Dec 12, 2012.
29. Intel 64 and IA-32 architectures optimization reference manual. *Technical Report 248966-025*, Intel Corporation June 2011.
30. Hess B, Kutzner C, van der Spoel D, Lindahl E. GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation* Mar 2008; **4**(3):435–447.
31. Darve E, Cecka C, Takahashi T. The fast multipole method on parallel clusters, multicore processors, and graphics processing units. *Comptes Rendus Mécanique* Mar 2011; **339**:185–193, doi:10.1016/j.crme.2010.12.005.