# A NEW PARALLEL KERNEL-INDEPENDENT FAST MULTIPOLE METHOD *

LEXING YING†, GEORGE BIROS†, DENIS ZORIN†, AND HARPER LANGSTON†

**Abstract.** We present a new adaptive fast multipole algorithm and its parallel implementation. The algorithm is kernel-independent in the sense that the evaluation of pairwise interactions does not rely on any analytic expansions, but only utilizes kernel evaluations. The new method provides the enabling technology for many important problems in computational science and engineering. Examples include viscous flows, fracture mechanics and screened Coulombic interactions. Our MPI-based parallel implementation logically separates the computation and communication phases to avoid synchronization in the upward and downward computation passes, and thus allows us to fully exploit computation and communication overlapping. We measure isogranular and fixed-size scalability for a variety of kernels on the Pittsburgh Supercomputing Center's TCS-1 Alphaserver on up to 3000 processors. We have solved viscous flow problems with up to 2.1 billion unknowns and we have achieved 1.6 Tflops/s peak performance and 1.13 Tflops/s sustained performance.

**Key words.** Fast multipole methods, adaptive algorithms, massively parallel computing, boundary integral equations, N-body problems, viscous flows

## 1. Introduction.

Many problems in mathematical physics involve computation of pairwise interactions in a system of particles where the interactions are often related to the fundamental solutions (kernels) of elliptic partial differential equations (PDEs). The Fast Multipole Method (FMM) has been very successful in accelerating such computations.

Most of the work on FMM methods however, has been restricted to the Laplacian kernel. (Important work has also been done on the Helmholtz and Maxwell equations. In this paper we do not consider oscillatory kernels.) Nevertheless, the fundamental ideas of FMM are applicable to many constant-coefficient elliptic PDE kernels. Examples include screened Coulombic interactions for molecular dynamics, velocity-pressure formulations for viscous flows, simulations of linearly elastic materials, and fluid-structure interaction problems. Unfortunately, these applications have received much less attention, despite their scientific and technological ramifications.

We believe one explanation for this is the dependence of the FMM implementation on analytic expansions of the kernel, i.e. such expansions need to be carried out differently for different kernels. This makes the implementation of efficient and accurate FMM accelerators somewhat tedious [6], [8], [19], [26]. Another problem could be related to the difficulties in devising work-efficient multipole-to-local translation schemes in three dimensions. For example, it took more than ten years to obtain such a scheme for the Laplace-based FMM scheme [9]. To our knowledge a general purpose and efficient FMM method appears to be an open problem.

In this paper, we present a new kernel-independent FMM-like algorithm, which requires only kernel evaluations. The crucial idea in our algorithm is to replace the analytic expansions with *equivalent densities*. The construction and translation of such equivalent densities are replaced by solving a series of local exterior/interior integral equation problems on spheres or cubes. In addition, the multipole-to-local translations are accelerated using local FFTs,

*This work was supported by the National Science Foundation's Knowledge and Distributed Intelligence (KDI) program through grant DMS-9980069. Computing on the TCS-1 HP Alphaserver were provided by the Pittsburgh Supercomputing center under award ASC-020017P.

†Courant Institute, New York University, New York 10012. Email: {lexing,biros,dzorin,harper}@cs.nyu.edu

resulting in performances that are on par with the fastest known adaptive FMM implementations [4]. Our algorithm has exactly the same structure as the original FMM, and thus, is highly parallelizable. Indeed, in our implementation we use standard methods in the parallel tree-code literature; and, by carefully overlapping the computation and communication phases, we have been able to scale our algorithm to thousands of processors, and for several different kernels: the Laplacian kernel, the modified Laplacian kernel (screened interactions) and the Stokes kernel (incompressible fluids and solids).

The idea of using a set of equivalent sources to represent the far field was first introduced by Anderson [1]: the far field is represented as the solution of an exterior Dirichlet problem on a ball surrounding the particles by means of the exact Green's function (Poisson formula) for the Laplacian. The method is somewhat easier than FMM to implement, but requires the analytic form of the Green's function for each kernel, and its stability does require analytic expansions, which may not be explicitly available in the general case. In contrast, our method does not use the Poisson formula—it only requires its existence. A more detailed review of other kernel-independent methods, along with a convergence proof, error analysis, and numerical results on the accuracy and complexity of the sequential algorithm, can be found in [25].

*Related work on parallel tree-codes.* The first successful distributed-memory parallel implementations for non-uniform particle distributions were obtained for the Barnes-Hut algorithm by Warren and Salmon [23]. Key ideas in this paper were the local essential trees (LETs), which provide a framework for parallelization of Barnes-Hut algorithm and can be extended to the FMM. The hashed octree data structures were first introduced in [24] along with space-filling curves used for partitioning and load balancing, and further increased efficiency and scalability of tree-codes. A similar approach for shared memory machines, and one of the first scalable FMM implementations, is found in [21], in which a cost-zones partitioning is used with orthogonal recursive bisection. A comparison between FMM algorithms, hybrids, and the Barnes-Hut method can be found in [3]. The main conclusion is that for higher accuracies, FMM is the fastest method. Another nice comparison between different platforms and algorithms can be found in Hu and Johnsson [11], in which the authors report results on up to 100 million particles on uniform particle distributions on a CM-5.

Recent papers on distributed-memory implementations include FMM for electromagnetics [10]; Helmholtz-type problems using optimal M2L translations [16]; and molecular dynamics FMM implementations that scale to 24 millions of particles on thousands of processors [14, 15]. Efficient data-structures and discussions on the theory of partitioning and complexity can be found in [20] and [22].

Other approaches for particle interactions include particle-mesh algorithms like those used in NAMD-2 [18] which employs FFTs for Ewald summation on regular grids. Such approaches could be extended to more general kernels, but they are restricted to approximately uniform particle distributions. Parallel Stokes solvers were presented in [17], but without FMM or Barnes-Hut acceleration.

*Organization of the paper.* In Section 2 we briefly describe our kernel-independent method. Section 3 explains the parallel algorithm. Section 4 presents the scalability results for three different kernels.

*Acknowledgments.* We are grateful to the support of R. Costa, D. O'Neal, S. Sanielevici, and C. Vizino of the Pittsburgh Supercomputing Center. We also thank Omar Ghattas for his insights and valuable discussions related to this project.

**2. Description of the kernel-independent method.** Given $N$ source densities $\{\phi_i\}$ located at $N$ points $\{\mathbf{y}_i\}$ in $\mathbf{R}^d$ $(d = 2, 3)$, we want to compute the potential $\{u_i\}$ induced by a kernel $G$ at $N$ points $\{\mathbf{x}_i\}$ using the following relation[1]:

$$u_i = u(\mathbf{x}_i) = \sum_{j=1}^{N} G(\mathbf{x}_i, \mathbf{y}_j)\phi(\mathbf{y}_j) = \sum_{j=1}^{N} G_{ij}\phi_j,\ i = 1, \ldots, N.$$

Direct implementation of this summation gives an $\mathcal{O}(N^2)$ algorithm. For a large class of kernels, FMM computes the same interactions in $\mathcal{O}(N)$ time. FMM is an approximate algorithm, in the sense that the summation is not computed exactly. The constant in the complexity estimate is related to the accuracy of the approximation.

Our algorithm is designed to generalize FMM to second-order constant coefficient non-oscillatory elliptic partial differential equations (PDEs). Examples of such systems are given in Appendix A, where we also list the corresponding fundamental solution kernels. Such kernels satisfy the underlying PDE everywhere except at the singularity location (pole), and are smooth away from the singularity. All problems under consideration admit a unique solution for the interior/exterior Dirichlet problems. These are basic properties which we use in order to develop our FMM approximation.

Our algorithm has the same structure as the original FMM method. The main difference in our method is how the densities are represented efficiently and how the M2M, M2L, and L2L transformations are computed. Below we summarize the notation we use in the description of the method; most of the quantities are defined in Section 2.1.

| | |
|---|---|
| $B$ | a box in the computation tree |
| $\mathcal{N}^B$ | the near range of the box $B$ in $\mathbb{R}^d$ |
| $\mathcal{F}^B$ | the far range of the box $B$ in $\mathbb{R}^d$ |
| $I_s^B$ | the index set of source points (or densities) in $B$ |
| $I_t^B$ | the index set of target points (or potentials) in $B$ |
| $\mathbf{y}^{B,u}$ | the upward equivalent surface of $B$ |
| $\phi^{B,u}$ | the upward equivalent density of $B$ |
| $\mathbf{x}^{B,u}$ | the upward check surface of $B$ |
| $u^{B,u}$ | the upward check potential of $B$ |
| $\mathbf{y}^{B,d}$ | the downward equivalent surface of $B$ |
| $\phi^{B,d}$ | the downward equivalent density of $B$ |
| $\mathbf{x}^{B,d}$ | the downward check surface of $B$ |
| $u^{B,d}$ | the downward check potential of $B$ |
| $p$ | the degree of discretization for equivalent densities |
| $s$ | the maximum number of source (or target) points allowed in a leaf box |
| $N$ | the total number of source and target points |
| $L$ | the depth of the computation tree |
| $M$ | the total number of boxes in the computation tree |

**2.1. Density translation.** Given a set of $N$ points, we define the computational domain to be a box large enough to contain all points. Then we construct the hierarchical octree so that each box contains no more than a prescribed number of points $s$. We assume that some points are labeled as sources and others as targets. Given a box[2] $B$ in the computation tree, we use $I_s^B$ and $I_t^B$ to denote the index sets of the source and target points in $B$ respectively. Sometimes, we also use $I_s^R$ and $I_t^R$ to denote these index sets in a region $R$. The source densities $\{\phi_i, i \in I_s^B\}$ at the source locations $\{\mathbf{y}_i, i \in I_s^B\}$ are given, and we want to evaluate the potentials $\{u_i, i \in I_t^B\}$ at the target locations $\{\mathbf{x}_i, i \in I_t^B\}$. If $B$ is a box centered at $\mathbf{c}$ and has side length $2r$, then the box centered at $\mathbf{c}$ with side length $6r$ is called the *near range*

[1] We use $\mathbf{x}$ to refer to target locations and $\mathbf{y}$ to refer to source locations, but in general $\{\mathbf{x}_i\}$ and $\{\mathbf{y}_i\}$ can be the same set of points.

[2] A box is a cube in 3D.

of $B$ and is denoted by $\mathcal{N}^B$. $\mathbb{R}^d \backslash \mathcal{N}^B$ is called the *far range* and is denoted by $\mathcal{F}^B$. Note that in our definition, $B$ is a part of $\mathcal{N}^B$.

*Equivalent densities.* We represent the potential in $\mathcal{F}^B$, induced by the source densities $\{\phi_i, i \in I_s^B\}$ in $B$ as the potential from a density distribution $\phi^{B,u}$, supported at prescribed locations $\mathbf{y}^{B,u}$ in $\mathcal{N}^B$ (Figure 2.1). We call $\phi^{B,u}$ the *upward equivalent density* and $\mathbf{y}^{B,u}$ the *upward equivalent surface*. Results from potential theory put two restrictions on the positions of $\mathbf{y}^{B,u}$ (see [12], chapter 6). Firstly, to guarantee the smoothness of the potential produced by $\phi^{B,u}$, its support $\mathbf{y}^{B,u}$ should not overlap with $\mathcal{F}^B$. Secondly, to guarantee that $\phi^{B,u}$ is "rich" enough to represent the potential produced by any source distribution in $B$, $\mathbf{y}^{B,u}$ needs to enclose $B$. Therefore, in order to ensure the existence of $\phi^{B,u}$, $\mathbf{y}^{B,u}$ is required to lie between $B$ and $\mathcal{F}^B$ and is usually chosen on a sphere or a cube. Since the potentials induced by the source densities and the upward equivalent density both satisfy the underlying second-order linear elliptic PDE, due to the uniqueness result of the exterior Dirichlet problem of this PDE, the two potentials are guaranteed to be equal in $\mathcal{F}^B$ if we can match them at the boundary of $\mathcal{F}^B$ or any surface between $\mathcal{F}^B$ and $\mathbf{y}^{B,u}$. This surface is called *upward check surface* (denoted by $\mathbf{x}^{B,u}$) and the matched potential is called *upward check potential* (denoted by $u^{B,u}$). We usually choose this surface to be a a sphere or cube. The properties of $\phi^{B,u}$ can be summarized as: $\forall \mathbf{x} \in \mathbf{x}^{B,u}$

$$\int_{\mathbf{y}^{B,u}} G(\mathbf{x},\mathbf{y})\phi^{B,u}(\mathbf{y})\,\mathrm{d}\mathbf{y} = \sum_{i \in I_s^B} G(\mathbf{x},\mathbf{y}_i)\phi_i. \tag{2.1}$$

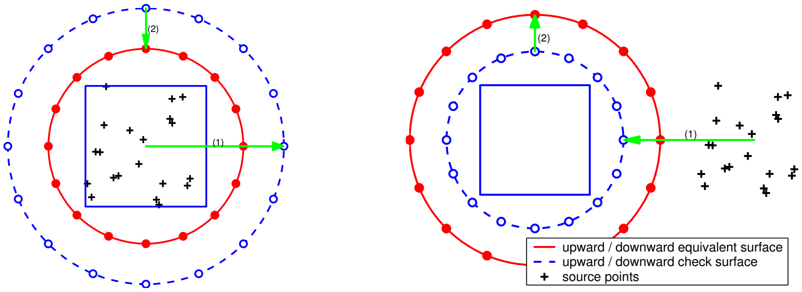

(2)
(1)
(2)
(1)
upward / downward equivalent surface
upward / downward check surface
source points

FIG. 2.1. Left*: Given the potential generated by the source densities inside a box located at the points marked with "$+$", we represent it by using the upward equivalent density located at the upward equivalent surface. The equivalent surface is shown as the solid circle enclosing the box. The upward check potentials induced by the sources and the upward equivalent density are matched at the upward check surface (the dashed circle).* Right*: To represent the potential in the box generated by the source in the far range, we use the downward equivalent density located at the downward equivalent surface. The downward equivalent potentials induced by both sources are matched at the upward check surface. In both plots, the discretization points of the equivalent and check surfaces are equally spaced and marked with "$\bullet$" and "$\circ$" respectively. For both upward or downward steps, the computation of the equivalent density includes two steps shown by arrows in each plot: (1) the evaluation of the check potential using the original source, and (2) the inversion of the integral equation to obtain the equivalent density.*

Similarly, we represent the potential in $B$ from the source densities in $\mathcal{F}^B$ as the potential induced by a density distribution $\phi^{B,d}$ supported at prescribed location $\mathbf{y}^{B,d}$ in $\mathcal{N}^B$ (Figure 2.1). We call $\phi^{B,d}$ the *downward equivalent density* and $\mathbf{y}^{B,d}$ the *downward equivalent surface*. To ensure the existence of $\phi^{B,d}$, $\mathbf{y}^{B,d}$ needs to lie between $\mathcal{F}^B$ and $B$. Since the potentials induced by both densities satisfy the underlying second order elliptic PDE, using the uniqueness result of the interior Dirichlet problem of this PDE, we only need to match the potentials on a surface (denoted by $\mathbf{x}^{B,d}$) between $B$ and $\mathbf{y}^{B,d}$. We call this surface *downward check surface* (denoted by $u^{B,d}$) and the matched potential *downward check* *potential*. Usually we choose $\mathbf{y}^{B,d}$ and $\mathbf{x}^{B,d}$ both to be spheres or cubes. The following equation summarizes our definitions: $\forall \mathbf{x} \in \mathbf{x}^{B,d}$

$$\int_{\mathbf{y}^{B,d}} G(\mathbf{x},\mathbf{y})\phi^{B,d}(\mathbf{y})\,\mathrm{d}\mathbf{y} = \sum_{i\in I_s^{\mathcal{F}^B}} G(\mathbf{x},\mathbf{y}_i)\phi_i. \tag{2.2}$$

*M2M translation.* For every leaf box $B$ in the computation tree, the computation of the upward equivalent density from the source densities follows equation (2.1) to solve for $\phi^{B,u}$ (Figure 2.2). The procedure of M2M translation is similar. To translate the upward equivalent density from a box $A$ to its parent box $B$, we solve the following equation for $\phi^{B,u}$, $\forall \mathbf{x} \in \mathbf{x}^{B,u}$

$$\int_{\mathbf{y}^{B,u}} G(\mathbf{x},\mathbf{y})\phi^{B,u}(\mathbf{y})\,\mathrm{d}\mathbf{y} = \int_{\mathbf{y}^{A,u}} G(\mathbf{x},\mathbf{y})\phi^{A,u}(\mathbf{y})\,\mathrm{d}\mathbf{y}, \tag{2.3}$$

where $U$ denotes the M2M translation operator. To ensure the existence of $\phi^{B,u}$ for $B$, $\mathbf{y}^{B,u}$ must enclose $\mathbf{y}^{A,u}$ for any of its children $A$.

*M2L translation.* Once the upward equivalent density has been computed for each box, M2L translation computes the downward equivalent density (Figure 2.2). Supposing $A$ is a box in $\mathcal{F}^B$, the M2L translation is similar to equation (2.2), and we solve for $\phi^{B,d}$ using: $\forall \mathbf{x} \in \mathbf{x}^{B,d}$

$$\int_{\mathbf{y}^{B,d}} G(\mathbf{x},\mathbf{y})\phi^{B,d}(\mathbf{y})\,\mathrm{d}\mathbf{y} = \int_{\mathbf{y}^{A,u}} G(\mathbf{x},\mathbf{y})\phi^{A,u}(\mathbf{y})\,\mathrm{d}\mathbf{y}, \tag{2.4}$$

where $T$ denotes the M2L translation operator. To ensure the existence of $\phi^{B,d}$, $\mathbf{y}^{B,d}$ must be disjoint from $\mathbf{y}^{A,u}$ for all $A$ in $\mathcal{F}^B$.

*L2L translation.* The L2L translation computes the downward equivalent density of a box $B$ from that of its parent $A$ (Figure 2.2). The procedure is again similar to equation (2.2). To compute $\phi^{B,d}$, we solve: $\forall \mathbf{x} \in \mathbf{x}^{B,d}$

$$\int_{\mathbf{y}^{B,d}} G(\mathbf{x},\mathbf{y})\phi^{B,d}(\mathbf{y})\,\mathrm{d}\mathbf{y} = \int_{\mathbf{y}^{A,d}} G(\mathbf{x},\mathbf{y})\phi^{A,d}(\mathbf{y})\,\mathrm{d}\mathbf{y}, \tag{2.5}$$

where $D$ denotes the L2L translation operator. To ensure the existence of $\phi^{B,d}$, $\mathbf{y}^{B,d}$ must lie in $\mathbf{y}^{A,d}$.

*Summary.* We use two density representations and three translations used to convert between these equivalent densities. The two equivalent densities correspond to the multipole and local representations in FMM, while the three translations replace the three transformations in FMM. In order to guarantee the existence of the equivalent densities, some restrictions are required for the choice of the equivalent and check surfaces. We summarize them as follows: for each box $B$:

- $\mathbf{y}^{B,u}$ and $\mathbf{x}^{B,u}$ lie between $B$ and $\mathcal{F}^B$, $\mathbf{x}^{B,u}$ encloses $\mathbf{y}^{B,u}$;
- $\mathbf{y}^{B,d}$ and $\mathbf{x}^{B,d}$ lie between $B$ and $\mathcal{F}^B$, $\mathbf{y}^{B,d}$ encloses $\mathbf{x}^{B,d}$;
- $\mathbf{y}^{B,u}$ encloses $\mathbf{y}^{A,u}$ for every of its children $A$;
- $\mathbf{y}^{B,u}$ is disjoint from $\mathbf{y}^{A,d}$ for all $A$ in $\mathcal{F}^B$;
- $\mathbf{y}^{B,d}$ lies in $\mathbf{y}^{A,d}$, where $A$ is $B$'s parent.

**3. The Parallel Algorithm.** In this section, we present our MPI-based parallel algorithm. The applications in which we are interested involve the solution of boundary integral equations. For these problems the particle positions and densities are associated to discretizations of integral equations, and at each time step the interaction computation (matrix vector

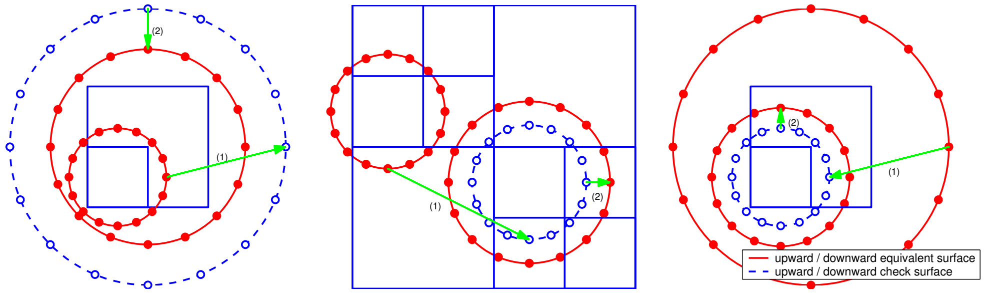

(2)
(1)
(1)
(2)
(2)
(1)
upward / downward equivalent surface
upward / downward check surface

FIG. 2.2. *The three translations.* Left*: M2M translation. To compute the upward equivalent density of the large square, we evaluate the (upward check) potential at the dashed circle using its child box's upward equivalent density at the small solid circle (arrow 1), and invert the integral equation to get its upward equivalent density at the large solid circle (arrow 2).* Middle*: M2L translation transforms the upward equivalent density of the left box to the downward equivalent density of the right box. We first evaluate the downward check potential at the dashed circle using the upward equivalent density (arrow 1), and then invert the equation to obtain the downward equivalent density at the downward equivalent surface (arrow 2). The upward density is defined on the small solid circle and the downward density is defined on the large solid circle.* Right*: L2L translation transforms the downward equivalent density of the large box to its child, the small box. In all three figures, the discretization points for the equivalent surface are marked with* ● *and the ones for check surface are marked with* ○*.*

multiplication within a Krylov method) is carried out multiple times. Therefore, our parallel implementation is designed to achieve maximum efficiency in the multiplication phase.

To parallelize an FMM algorithm, we need to meet certain communication and synchronization requirements: (1) *Tree-related communication* is required to maintain a consistent global tree. (2) *Synchronization* at upward and downward pass since there is data dependence between the equivalent densities of a parent and its child, and these two boxes can belong to different processors. (3) *Communication related to M2L translation* is necessary since one box needs the upward equivalent density or source information of another box owned by a different processor. In our implementation we have logically separated the computation and communication. During the upward and downward passes, a processor performs its own computation ignoring the existence of other processors. Between them, a single step combines the upward equivalent density computed by all processors and takes care of the communication. The advantage of this approach is that no synchronization is required at the computation passes. A disadvantage is the redundant computation at the nodes which are close to the root of the global computation tree. However since the number of these nodes is small, this has negligible influence on the overall computation. Below we discuss the components of our implementation in detail.

**3.1. Data partitioning and tree generation.** Our partitioning scheme is fairly straightforward. We take advantage of the fact that our input is a set of surface patches on which the particles are generated. We first gather all input surface patches on a single processor, and assign to each patch a weight which in the simplest case is equal to the number of particles in that patch. Second, we partition the clusters into groups with equal weights and assign each group to one processor. To do this we use Morton curve partitioning. Alternatively, we could use Morton curve partitioning directly on the particles but we have found the first approach faster. No additional load balancing information is used besides the number of particles. Work estimates from a previous time step could be used to obtain more balanced partitioning.

An essential part in the FMM algorithm is the generation of the octree. However, since our applications require tens to hundreds of interaction computations, we have adopted a simple but suboptimal tree construction algorithm. An important idea in parallel tree-based algorithms is the *Local Essential Tree* (LET) [23], which is the global tree subset that a processor needs to evaluate the interaction on particles it owns. In an adaptive FMM algorithm, in order to calculate the interaction at a box $B$, we need the information from the boxes in the following four lists ([4],[7]): (1) $U$ list $L_U^B$ which contains $B$ itself and the leaf boxes which are adjacent to $B$ if $B$ is leaf, and it is empty when $B$ is non-leaf; (2) $V$ list $L_V^B$ which contains the children of the neighbors of $B$'s parent, which are not adjacent to $B$; (3) $W$ list $L_W^B$ which contains all the descendants of $B$'s neighbors whose parents are adjacent to $B$ but who are not adjacent to $B$ themselves if $B$ is leaf, and it is empty if $B$ is a non-leaf; and (4) $X$ list $L_X^B$, which contains all boxes $A$ such that $B \in L_W^A$. Therefore, for a certain processor $P$, its LET first contains the boxes which contains points belonging to $P$ and second the boxes in the $U$, $V$, $W$, and $X$ lists of these boxes. For a box $B$ of the first kind, we say $P$ *contributes* to $B$, or equivalently, $P$ is a *contributor* of $B$. If $B$ is of the second kind, we say $P$ *uses* $B$ or $P$ is a *user* of $B$.

In the tree generation, besides LET, we maintain a compact representation of the global tree by using an array in every processor, which we call the global tree array [3]. Each entry in the global tree array corresponds to a box in the global tree, and this array is ordered according to a level-by-level traversal of the tree. The only information stored is the global number of particles in the box and the indices of its children boxes in the array. Our algorithm constructs the local tree and this array structure level by level. All processors begin at level 0 with the same box which is large enough to contain the global particle set. At every level $l$, each processor puts its local number of points in boxes at level $l$ as well as into its local copy of the global tree array. Then, an MPI_Allreduce is used over all local copies of the global tree array to sum up the local number of points for each box at level $l$. After this collective communication, each local array contains the global number of points in each box in level $l$. By comparing each box's global number of points with $s$ (the maximum number of points allowed in each leaf box), each processor can decide whether a box in level $l$ should be further subdivided. Based on this decision, a processor can construct the $l+1$ level of its local tree and the array representation of the $l+1$ level of the global tree. After the construction of the local tree, the computation of the local FMM lists is straightforward by using the global tree represented in the array.

**3.2. Interaction calculation.** Before the interaction calculation, we first partition the global tree array, so that for each box $B$ the owner processor coordinates the communication related to $B$. If only one processor contributes to $B$, then it is the owner of $B$. If multiple processors contribute to $B$, then it can be owned by any processor, and the owner is chosen to balance the communication load. This can be done as follows. For each processor $P$, first we use the global tree array to decide the boxes for which $P$ is the only contributor, and mark them as "taken". Second, we use MPI_Allreduce to combine the information, so that every processor $P$ knows all boxes already taken by some processor. Third, every processor $P$ uses the same sequential algorithm to assign unmarked boxes to processors in order to balance communication load. In the end, all processors have the owner information for any box in the global computation tree.

The interaction calculation part of our algorithm is logically separated into three stages. The first stage is a computation step which performs the upward computation. Each processor $P$ builds the upward equivalent densities for the LET nodes to which it contributes (ignoring the existence of the other processors).

The second stage has two components. First, for each leaf box, we need to collect its source positions and source density (also known as ghost information) from its contributors and make them available for its users. The *gather/scatter* procedure for doing this is given

[3] This representation only contains topological structure of the tree. In practice, for a 200M points data set with $s$ chosen to be 60, the size of the array is less than 16M.

**Algorithm 1** Communication of source positions and densities for a single processor $P$

```
STEP 1 GATHER
  for each box $B$ in the LET do
    if $P$ contributes to $B$ then
      $P$ sends its local source position/density information of $B$ to the owner of $B$
    end if
    if $P$ owns $B$ then
      $P$ receives the local information of $B$ from all the contributors of $B$
      $P$ combines them into the global position/density information of $B$
    end if
  end for
STEP 2 SCATTER
  for each box $B$ in the LET do
    if $P$ owns $B$ then
      $P$ sends the global position/density information of $B$ to all users of $B$
    end if
    if $P$ uses $B$ then
      $P$ receives the global position/density information of $B$ from the owner of $B$
    end if
  end for
```

in Algorithm 1. Second, for each box (leaf or non-leaf), we need to sum up the upward equivalent densities produced by its contributors and make it available for its users. The procedure for this is similar to Algorithm 1 with two modifications: (1) we iterate over all boxes in the LET instead of just the leaf boxes, and (2) the owner of a box *sums up* the received upward equivalent densities to obtain the global upward equivalent densities for that box.

The third stage performs the downward computation. Here, for each LET node $B$, to which it contributes, $P$ transforms the source density or upward equivalent density of the boxes in $U$, $V$, $W$ and $X$ lists into local equivalent density or target potential at node $B$ (ignoring the existence of the other processors again). In our implementation the upwards traversal is overlapped with the ghost communication; and the equivalent densities communication is overlapped with the dense and $X$-list computations.

**4. Scalability results.** Our FMM code is used in the context of fluid-structure interaction calculations (e.g. Figure 4.1). In this section we provide experimental evidence on the scalability of our implementation of the particle interaction evaluation. We present fixed-size

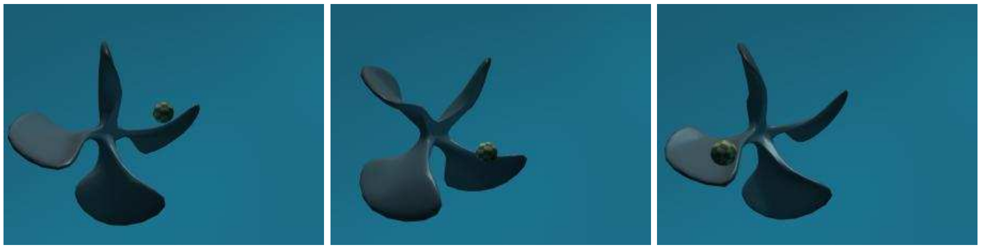


FIG. 4.1. *Three frames from an animation showing a fluid-structure interaction problem solution: the motion of a sphere under the influence of gravity and viscous forces exerted by a Stokes fluid which is stirred by a clockwise rotating propeller. The solution of this problem requires a time stepping procedure on an integro-differential system of equations describing rigid-body dynamics. The Stokes equations for the fluid are solved using a boundary integral formulation and our FMM method is used to accelerate the matrix vector multiplications—which correspond to particle interaction evaluations. At each time step we solve a linear system that requires tens of interaction calculations. In this particular example we used 64 processors. An animation can be found in* `http://cat.nyu.edu/~harper/stokes/animations/index.html`.

and isogranular scalability analysis. For fixed-size scalability analysis, we increase the number of processors for a fixed problem size. This analysis exposes the grain size[4] for which we can expect reasonable speedups. For isogranular scalability analysis we keep the grain size fixed and we increase the number of processors (and thus the problem size). Such analysis reveals communication problems related to the size and frequency of the messages as well as global reductions and problems with algorithmic scalability. In our case, however, we expect good algorithmic scalability since FMM is an $\mathcal{O}(N)$ algorithm under reasonable assumptions on the particle distribution [13].

Before we describe our numerical experiments, we site two main conclusions from our work on the sequential performance of our method [25]: First, the most expensive parts of the FMM algorithm are the M2L interactions and the dense interactions, both in the downward traversal of the tree, especially in three dimensions. Second, our method achieves algorithmic speed-ups which are on par with the fastest known implementations of the FMM for the Laplacian [4] and modified Laplacian kernels [8].

The problem setup is the following: The input is a set of surfaces, which we then sample to get the particle positions. We build the hierarchical tree structure and then we perform several interaction calculations. In this article we always report results for a single interaction calculation, averaged over several iterations. We assume the sets of source and target points to be identical. We use two sets of density distributions in the cube with range $[-1, 1]$ in each dimension. The first set is produced by sampling 512 spheres centered at an $8 \times 8 \times 8$ Cartesian grid in the unit cube. For relatively low sampling rates, up to 10 million particles, we obtain a uniform particle distribution. For higher sampling rates the distribution per processor becomes non-uniform since the sampling over a single sphere is non-uniform. Our second particle set is a non-uniform distribution of particles clustered at the eight corners of the unit cube. In all density distributions, the densities are chosen randomly from $[0, 1]$, and the relative error in all experiments is $10^{-5}$.

Our algorithm has been implemented in C++. We used the fast exponential, square root and reciprocal libraries in the CXML routines, FFTW [5] for the M2L translations, and PETSc [2] for profiling and for its Krylov iterative solvers. All our tests were performed on the Pittsburgh Supercomputing Center's TCS-1 terascale computing HP Alphaserver Cluster comprising of 750 SMP ES45 nodes. Each node is equipped with four Alpha EV-68 processors at 1 GHz and 4 Gbytes of memory. The peak performance is approximately 6 Tflops/s, and the peak performance for the top-500 LINPACK benchmark is approximately 4 Tflops/s. The nodes are connected by the Quadrics interconnect which delivers over 500 MB/s of message-passing bandwidth per node and has a bisection bandwidth of 187 GB/s. In all our tests we have used 4 processors per node.

*Description of results.* We report wall-clock timings, Gflops/s rates and parallel efficiency measurements for several problem sizes and kernels. Fixed size scalability results for 3.2M particles are reported in Table 4.1, in which we provide timings for the interaction calculation and for the tree construction, including communication. We also report aggregate Gflops/s rates.

In Figure 4.2, we report the aggregate CPU cycles (across processors) per point for the interaction calculation. These numbers are computed as $\frac{P \times C \times T(P)}{N}$, where $P$ is the number of processors; $C$ is the clock rate which in our case is 1GHz; $T(P)$ is the wall-clock time on $P$ processors; and $N$ is the number of particles. This metric is used to measure parallel scalability and can be used to compare among different machines, clock-rates and problems sizes. However, it hides architecture-dependent characteristics like cache performance and memory bus speed.

[4] By grain size we mean the number of particles per processor.

Laplacian kernel

| | Interaction computation | | | | | | | Tree |
|---|---|---|---|---|---|---|---|---|
| | Time (sec) | | | | | GFlops/s | | Time (sec) |
| *P* | *Total* | *Ratio* | *Comm* | *Up* | *Down* | *Avg* | *Peak* | *Gen/Comm* |
| 1 | 392.75 | 1.00 | 0.00 | 58.41 | 334.34 | 0.28 | 0.31 | 13.97 |
| 4 | 103.67 | 1.00 | 0.87 | 14.63 | 88.30 | 1.06 | 1.25 | 3.81 |
| 8 | 51.33 | 1.00 | 0.54 | 7.42 | 43.48 | 2.15 | 2.46 | 2.27 |
| 16 | 25.49 | 1.10 | 0.55 | 3.69 | 21.99 | 4.30 | 4.96 | 1.47 |
| 64 | 6.74 | 1.10 | 0.33 | 0.96 | 6.10 | 16.53 | 19.00 | 0.68 |
| 256 | 1.67 | 1.20 | 0.15 | 0.24 | 1.55 | 64.53 | 77.49 | 0.51 |
| 512 | 1.10 | 1.20 | 0.40 | 0.12 | 0.74 | 104.89 | 154.69 | 0.87 |
| 1024 | 1.13 | 1.20 | 0.81 | 0.07 | 0.45 | 108.67 | 258.55 | 1.09 |

Modified Laplacian kernel

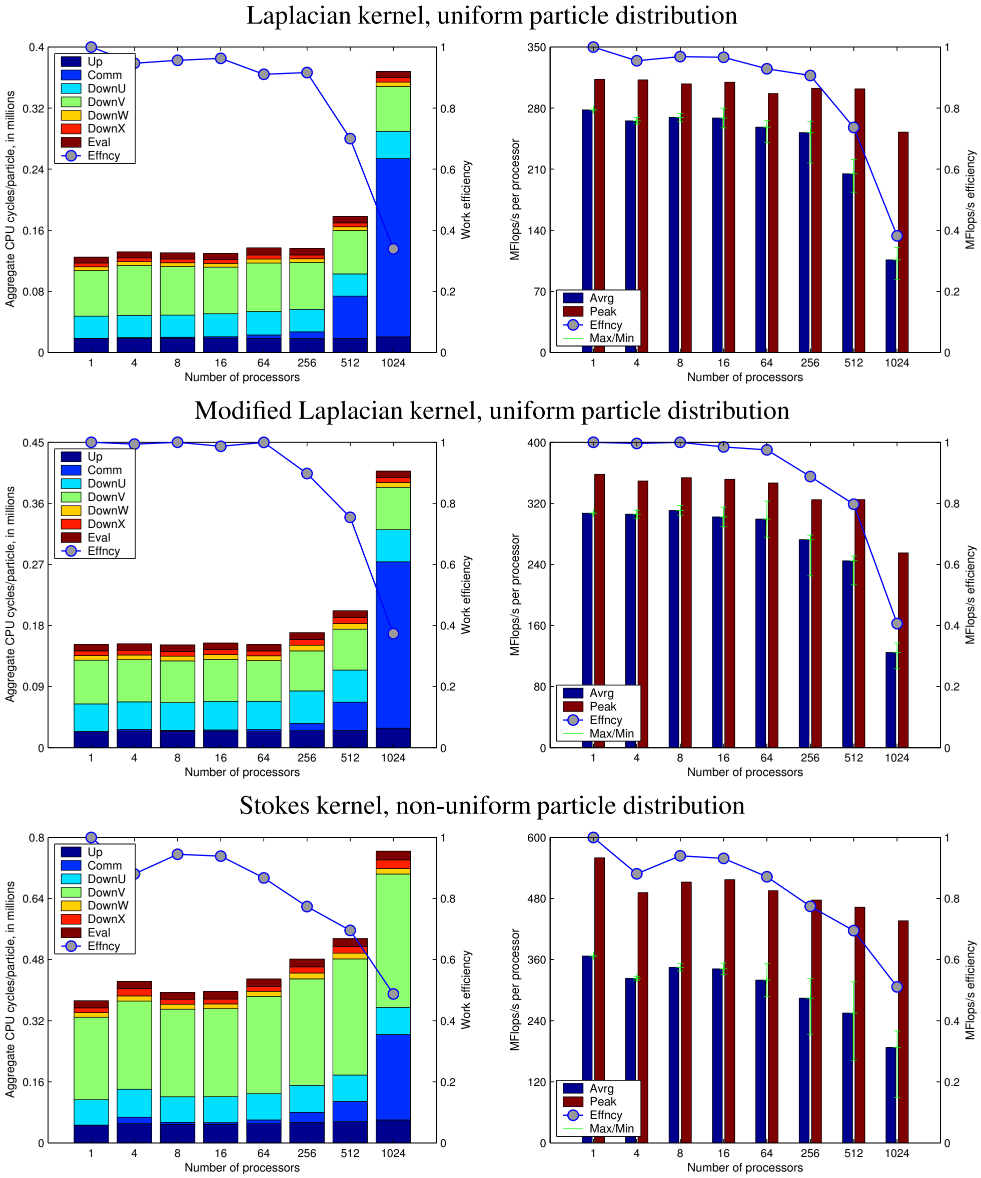
| | Interaction computation | | | | | | | Tree |
|---|---|---|---|---|---|---|---|---|
| | Time (sec) | | | | | GFlops/s | | Time (sec) |
| *P* | *Total* | *Ratio* | *Comm* | *Up* | *Down* | *Avg* | *Peak* | *Gen/Comm* |
| 1 | 478.66 | 1.00 | 0.00 | 75.09 | 403.57 | 0.31 | 0.36 | 13.88 |
| 4 | 120.43 | 1.00 | 2.48 | 19.24 | 99.41 | 1.22 | 1.40 | 3.74 |
| 8 | 59.58 | 1.00 | 0.62 | 9.50 | 49.57 | 2.49 | 2.83 | 2.19 |
| 16 | 30.32 | 1.00 | 0.46 | 4.78 | 25.82 | 4.84 | 5.62 | 1.35 |
| 64 | 7.48 | 1.10 | 0.23 | 1.21 | 6.56 | 19.16 | 22.19 | 0.55 |
| 256 | 2.08 | 1.30 | 0.22 | 0.32 | 1.96 | 69.77 | 83.16 | 0.59 |
| 512 | 1.24 | 1.20 | 0.32 | 0.16 | 1.01 | 125.29 | 166.35 | 0.55 |
| 1024 | 1.25 | 1.20 | 0.85 | 0.10 | 0.51 | 127.67 | 261.21 | 1.25 |

Stokes kernel, non-uniform particle distribution

| | Interaction computation | | | | | | | Tree |
|---|---|---|---|---|---|---|---|---|
| | Time (sec) | | | | | GFlops/s | | Time (sec) |
| *P* | *Total* | *Ratio* | *Comm* | *Up* | *Down* | *Avg* | *Peak* | *Gen/Comm* |
| 1 | 1171.92 | 1.00 | 0.00 | 146.28 | 1025.64 | 0.37 | 0.56 | 22.95 |
| 4 | 332.69 | 1.00 | 19.17 | 41.63 | 284.49 | 1.29 | 1.97 | 6.05 |
| 8 | 155.07 | 1.00 | 3.51 | 19.97 | 135.53 | 2.76 | 4.10 | 2.94 |
| 16 | 78.02 | 1.00 | 1.51 | 10.02 | 70.39 | 5.47 | 8.28 | 1.51 |
| 64 | 21.11 | 1.20 | 0.75 | 2.70 | 19.95 | 20.47 | 31.71 | 0.80 |
| 256 | 5.92 | 1.50 | 0.53 | 0.74 | 6.18 | 72.77 | 122.11 | 0.81 |
| 512 | 3.29 | 1.70 | 0.48 | 0.40 | 3.59 | 130.60 | 237.08 | 0.73 |
| 1024 | 2.35 | 1.80 | 0.82 | 0.22 | 2.18 | 191.96 | 446.76 | 0.96 |

TABLE 4.1

***Fixed size scalability*** *results. In these examples we have used 3.2 million particles. We report wall-clock time in seconds and flop rates for a single interaction computation. We also report timings for the tree construction and communication phases. Here (*P*) is the number of processors; (*Total*) is the total time (averaged across processors) of the interaction phase; (*Ratio*) is the difference between the maximum and minimum time across processors, which is an indication of load imbalance;* Comm *is the average time spent in MPI communication; (*Up*) is the average time spent in the upward traversal of the tree; (*Down*) is the average time spent in the downward traversal of the tree; (*Avg*) is the average Gflops/s during the interaction calculation; (*Peak*) is the peak Gflops/s; and (*Gen/Comm*) is the time spent in the tree construction phase. Overall, we can observe that we obtain excellent scalability up to 512 processors. Communication costs however, become significant once we hit 512 processors or more. In these cases we use less than 6,000 particles per processor, i.e. too fine a grain size. All results in this section were obtained on the 3,000 processor TCS-1 HP Alphaserver of the Pittsburgh Supercomputing Center.*


Laplacian kernel, uniform particle distribution

Modified Laplacian kernel, uniform particle distribution

Stokes kernel, non-uniform particle distribution

FIG. 4.2. ***Fixed-size scalability results*** *for different kernels. Here 3.2 million particles were used. The left column shows aggregate CPU cycles per particle; the right column shows Mflops/s/processor for different stages of the interaction calculation phase and the flop-rate efficiency. There are five stages: (*Up*) is the upward traversal of the tree used to built equivalent densities; (*Comm*) is the communication of the ghost points and equivalent densities; (*DownU*) is the dense interaction calculations for* $L_U$ *lists; (*DownV*) is the M2L translations for* $L_V$ *lists; (*DownX*) and (*DownW*) are the calculations for* $L_X$ *and* $L_W$ *lists associated with the adaptive algorithm. In the right column, (*Avg*) is the the average, across processors, Mflops rate; (*Peak*) is the peak rate; and the (*Max/Min*) indicate the Maximum/minimum average rates across processors—an indication of load imbalance. The work efficiency is computed using the timings in Table 4.1. The Mflops/s efficiency is computed based on the rates given in the figure. As we can see, up to 512 processors the efficiency is quite good: there is only a small increase in the total work per particle.*

We also compute a floating point efficiency as an index of efficient utilization of the ma-

Laplacian kernel, uniform particle distribution

| | Interaction computation | | | | | | | Tree |
|---|---|---|---|---|---|---|---|---|
| | Time (sec) | | | | | GFlops/s | | Time (sec) |
| *P* | *Total* | *Ratio* | *Comm* | *Up* | *Down* | *Avg* | *Peak* | *Gen/Comm* |
| 1 | 30.56 | 1.00 | 0.00 | 2.87 | 27.69 | 0.27 | 0.28 | 0.49 |
| 4 | 28.59 | 1.00 | 0.23 | 3.13 | 25.29 | 1.03 | 1.16 | 0.70 |
| 16 | 25.97 | 1.10 | 1.06 | 3.77 | 22.80 | 4.17 | 4.84 | 1.42 |
| 64 | 21.76 | 1.10 | 1.38 | 3.73 | 18.26 | 17.04 | 17.78 | 3.02 |
| 256 | 22.06 | 1.10 | 1.65 | 3.48 | 19.54 | 64.36 | 73.89 | 13.48 |
| 1024 | 22.22 | 1.10 | 3.09 | 3.84 | 18.39 | 247.78 | 262.64 | 71.26 |
| 2048 | 23.54 | 1.20 | 0.96 | 4.05 | 21.34 | 488.07 | 568.89 | 964.47 |

Stokes kernel, uniform particle distribution

| | Interaction computation | | | | | | | Tree |
|---|---|---|---|---|---|---|---|---|
| | Time (sec) | | | | | GFlops/s | | Time (sec) |
| *P* | *Total* | *Ratio* | *Comm* | *Up* | *Down* | *Avg* | *Peak* | *Gen/Comm* |
| 1 | 133.26 | 1.00 | 0.00 | 7.38 | 125.88 | 0.30 | 0.49 | 0.49 |
| 4 | 166.79 | 1.00 | 0.63 | 8.62 | 157.86 | 1.03 | 2.06 | 0.64 |
| 16 | 146.72 | 1.10 | 2.00 | 12.34 | 139.38 | 4.41 | 8.46 | 1.59 |
| 64 | 106.00 | 1.10 | 2.13 | 10.74 | 99.06 | 18.86 | 33.21 | 3.54 |
| 256 | 88.06 | 1.10 | 2.43 | 10.59 | 81.94 | 81.57 | 127.91 | 14.16 |
| 1024 | 79.34 | 1.10 | 2.10 | 10.59 | 72.53 | 338.56 | 494.97 | 69.42 |
| 2048 | 76.28 | 1.10 | 2.06 | 10.58 | 69.36 | 669.99 | 986.45 | 936.78 |

Stokes kernel, non-uniform particle distribution

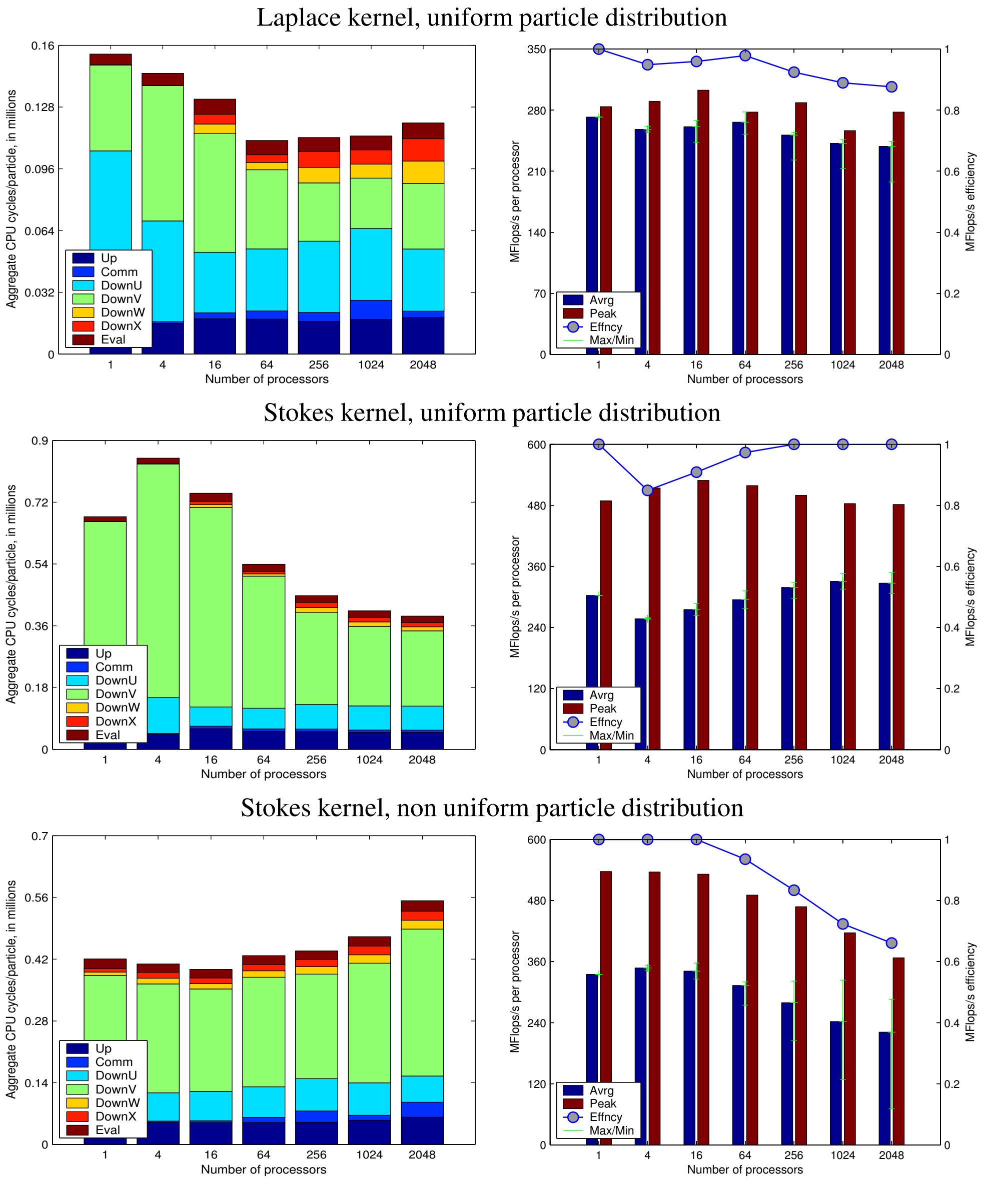
| | Interaction computation | | | | | | | Tree |
|---|---|---|---|---|---|---|---|---|
| | Time (sec) | | | | | GFlops/s | | Time (sec) |
| *P* | *Total* | *Ratio* | *Comm* | *Up* | *Down* | *Avg* | *Peak* | *Gen/Comm* |
| 1 | 82.68 | 1.00 | 0.00 | 10.39 | 72.29 | 0.34 | 0.54 | 1.17 |
| 4 | 80.43 | 1.00 | 0.72 | 9.88 | 70.01 | 1.39 | 2.14 | 1.37 |
| 16 | 78.03 | 1.10 | 1.30 | 9.75 | 70.75 | 5.46 | 8.51 | 1.47 |
| 64 | 84.16 | 1.20 | 4.02 | 10.67 | 79.55 | 20.05 | 31.40 | 2.52 |
| 256 | 86.24 | 1.50 | 8.97 | 11.17 | 92.44 | 71.49 | 119.76 | 8.21 |
| 1024 | 92.60 | 2.00 | 4.20 | 12.55 | 114.93 | 248.16 | 426.40 | 69.34 |
| 2048 | 108.64 | 2.50 | 12.32 | 17.72 | 139.36 | 453.44 | 752.03 | 988.24 |

TABLE 4.2

***Isogranular scalability results.*** *In these examples we have used 200,000 particles per processor. We report the wall-clock time and Gflops/s for the interaction computation, and timings for the tree construction and communication phases.* P, Total, Ratio, Comm, Up, Down, Avg, Peak *and* Gen/Comm *have the same meaning as in Table 4.1. Overall, we observe that the running time is slightly decreasing. This is due to algorithmic changes; the M2L translations work (*Down*) is decreasing. We observe very good scalability, i.e. low communication costs during the interaction calculation phase; the increase for 2048 processors, in the non-uniform distribution case is due to the load imbalance. For the larger problem we have a total 400 million particles, which for the Stokes case corresponds to 1.2 billion unknowns.*

chine. The work efficiency is $\frac{T(1)}{T(P)P}$ and the flop-rate efficiency is computed as $f(P)/f(1)$, where $f(P)$ is the flop-rate per processor on $P$-processors.

In Table 4.2 and Figure 4.3, we report isogranular scalability results for 200 thousand particles per processor and for the Laplace and Stokes kernels. In these experiments we do not report work efficiency because the algorithm behavior slightly changes as we increase the problem size and at first sight it appears that we obtain superlinear speedups. The particles are



Laplace kernel, uniform particle distribution

Stokes kernel, uniform particle distribution

Stokes kernel, non uniform particle distribution

Fig. 4.3. ***Isogranular scalability results*** *for Laplacian and the Stokes kernels. These charts show the aggregate CPU cycles per particle and Mflops/s/processor for the different stages of the interaction calculation. As in Figure 4.2 we report total cycles per particle and flop-rates for the whole interaction calculation and its different phases. The Mflops/s efficiency is computed based on the rates given in the figure. We are not reporting work efficiency because the algorithmic behavior of the problem changes with increasing problem size, due to the increase in nonuniformity of the distribution at finer scales. In this case the number of M2L translations decreases, resulting in a overall work decrease. In addition, M2L computations run at about 300 Mflops/s, while all other parts run at about 400+ Mflops/s. This produces apparent superlinear efficiencies, especially in the work-intensive Stokes case. Our implementation exhibits excellent scalability results on thousands of processors: As we can observe from the plots for the Laplacian kernel we maintain an 80% efficiency on up to 2048 processors. In the non-uniform distribution (last row) we observe a rather significant decrease of the efficiency down to 65%. This is due to load imbalance, and is something that we are currently working to improve. Finally notice that that for the largest problem the peak performance was 1Tflops/s.*

|  | Interaction computation |  |  |  |  |  |  | Tree |
|---|---|---|---|---|---|---|---|---|
|  | Time (sec) |  |  |  |  | GFlops/s |  | Time (sec) |
| *unknowns* | *Total* | *Ratio* | *Comm* | *Up* | *Down* | *Avg* | *Peak* | *Gen/Comm* |
| 0.300 B | 7.63 | 1.5 | 1.03 | 2.43 | 5.69 | 696.8 | 802.2 | 837.4 |
| 0.690 B | 21.59 | 2.2 | 3.23 | 4.13 | 15.29 | 789.3 | 972.1 | 1101 |
| 2.070 B | 65.97 | 1.8 | 3.06 | 9.87 | 62.10 | 1134 | 1587 | 1077 |

TABLE 4.3

***3000 Processor runs.*** *In these examples we have solved three problems 100K and 230K particles per CPU for for the Laplace and Stokes equations all for the 512 spheres input. For the larger problem we have a total 700 million particles, which for the Stokes case corresponds to 2.1 billion unknowns. Notice that for the interaction calculation we have sustained 1.13 Tflops/s which translates to 25% efficiency compared to the sustained performance for the LINPACK benchmark on the TCS-1.*

sampled for 512 spheres regularly arranged in a cube. For small numbers of particles we have uniform distributions, but for the very large problems the problem locally is non-uniform. As a result the number of M2L interactions drops and since this is the most costly part of the computation it appears that the work efficiency improves.

Finally in Table 4.3 we report results from our largest runs on 3000 processors. In this set of runs the geometry is the 512 spheres and we solve problems for the Laplace equation with 100K and 230K particles per processor and for the Stokes equations also with 230K particles per processor. For all the other experiments we have used rough 60 particles per box, while in this experiment we use 120 particles per box to slightly reduce the costs of tree construction.

*Discussion.* Examination of the performance numbers leads to the following observations: (1) The code uses about 160 thousand CPU cycles per particle for five digits of accuracy for the Laplacian kernel and about 200 thousand and 800 thousand cycles for the modified Laplacian and Stokes respectively. (2) For the fixed problem size (3.4 million particles) we obtain 80% efficiency for up to 256 processors and then the communication costs start increasing. (3) In the isogranular scalability good efficiency is maintained up to 2048 processors with peak performance of 1 Tflops/s and sustained performance of 0.7 Tflops/s [5]. (4) The communication costs during the interaction computation scale very well. (5) The tree construction and communication does not scale beyond 1024 processors. (6) Load imbalance for highly non-uniform distributions is significant. (7) In our largest runs we have obtained 1.13 Tflops/s sustained performance and 1.6 Tflops/s peak performance for 2.1 billion unknowns.

It is apparent that we get better performance for the Stokes kernel. The reason is that the scalar kernels like the Laplacian and modified Laplacian have less work per particle and less communication than the Stokes kernels. We have observed an increase in the flop-rate for the Stokes kernel albeit the higher communication costs.

We should note that our implementation of the tree construction and load balancing is not optimal; our focus was on efficient implementation of the interaction computation, which we apply several times before we update the particle positions. Tree construction and load balancing are well isolated parts in our code and known techniques can be used to improve their efficiency. We plan to incorporate more efficient algorithms in the near future. In particular, we plan to use workload information from previous time steps for load balancing. In addition we are currently changing our level-to-level tree construction in order to obtain a completely scalable algorithm.

[5] In fact, we could easily increase the flop rate by switching from the algorithmically fast, but implementationally slower FFT M2L translations to the slower direct evaluation. But the speed gains are negligible compared to the algorithmic savings.

**5. Conclusions and future work.** We have presented a new parallel fast multipole method, generalizing FMM to general elliptic kernels, along with an MPI-based scalable and platform-independent parallel implementation.

Our algorithm has several important features. (1) It achieves kernel-independence by replacing multipole and local expansions with equivalent densities. (2) It is adaptive, highly efficient and compares well with existing analytic FMM implementations. (3) Our MPI-based parallel implementation logically separates computation and communication to avoid synchronization in upward and downward pass, and to exploit maximal computation and communication overlapping. (4) We verified that the method scales up to 3000 processors and achieves very good per processor sustained performance (up to 480 Mflops/s). As a result, we were able to reach 1.13 Tflops/s sustained performance for a Stokes flow problem with 2.1 billion unknowns.

Two problems in our implementation are the tree construction algorithm and the inefficient load balancing algorithm which create problems for more than 1024 processors. We are currently working on adaptations of our algorithm for applications such as molecular dynamics, where efficient tree construction is much more important.

**Appendix A. Kernels.**

In this section, we give a summary of the elliptic PDEs studied in this paper and their related single layer kernels. In the formulas below, $\mathbf{y}$ is the location of the singularity, $\mathbf{x}$ is the location the evaluation point, $\mathbf{r} = \mathbf{x} - \mathbf{y}$ and $r = |\mathbf{r}|$.

$$-\Delta u = 0, \quad \mathbf{S}(\mathbf{x},\mathbf{y}) = \frac{1}{4\pi}\frac{1}{r} \qquad \text{Laplace.}$$

$$\alpha u - \Delta u = 0, \quad \mathbf{S}(\mathbf{x},\mathbf{y}) = \frac{1}{4\pi}\frac{1}{r}\mathrm{e}^{-\lambda r} \qquad \text{Modified Laplace.}$$

$$-\mu\Delta u + \nabla p = 0,\ \mathrm{Div}\, u = 0, \quad \mathbf{S}(\mathbf{x},\mathbf{y}) = \frac{1}{8\pi\mu}(\frac{1}{r}\mathbf{I} + \frac{\mathbf{r}\otimes\mathbf{r}}{r^3}) \qquad \text{Stokes.}$$

REFERENCES

[1] Christopher R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM Journal on Scientific and Statistical Computing*, 13(4):923–947, 1992.

[2] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 2001.

[3] Guy Blelloch and Girija Narlikar. A practical comparison of $n$-body algorithms. In *Parallel Algorithms*, Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[4] H. Cheng, Leslie Greengard, and Vladimir Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155:468–498, 1999.

[5] Matteo Frigo and Steven G. Johnson. FFTW home page. http://www.fftw.org, 2000.

[6] Yuhong Fu et al. A fast solution for three-dimensional many-particle problems of linear elasticity. *International Journal for Numerical Methods in Engineering*, 42:1215–1229, 1998.

[7] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. MIT Press, Cambridge, MA, 1988.

[8] Leslie Greengard and Jingfang Huang. A new version of the fast multipole method for screened Coulomb interactions in three dimensions. *Journal of Computational Physics*, 180:642–658, 2002.

[9] Leslie Greengard and Vladimir Rokhlin. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numerica*, pages 229–269, 1997.

[10] Bari Hariharan, Srinivas Aluru, and Balasubramaniam Shanker. A scalable parallel fast multipole method for analysis of scattering from perfect electrically conducting surfaces. In *Proceedings of Supercomputing*, The SCxy Conference series, Baltimore, Maryland, November 2002. ACM/IEEE.

[11] Yu Hu and S. Lennart Johnsson. A data-parallel implementation of $o(n)$ hierarchical N-body methods. In *Proceedings of Supercomputing*, The SCxy Conference series, Pittsburgh, Pennsylvania, November 1996. ACM/IEEE.

[12] Rainer Kress. *Linear Integral Equations*. Applied Mathematical Sciences. Springer, 1999.

[13] K. Nabors, F.T. Korsmeyer, F.T. Leighton, and Jacob K. White. Preconditioned, adaptive, multipole-accelerated interative methods for three-dimensional first-kind integral equations of potential theory. *SIAM Journal on Scientific and Statistical Computing*, 15:713–735, 1994.

[14] Aiichiro Nakano. Parallel multilevel preconditioned conjugate-gradient approach to variable-charge molecular dynamics. *Computer Physics Communications*, 104:59–69, 1997.

[15] Aiichiro Nakano et al. Scalable atomistic simulation algorithms for materials research. In *Proceedings of Supercomputing*, The SCxy Conference series, Denver, Colorado, November 2001. ACM/IEEE.

[16] John J. Ottusch, Mark A. Stalzer, John L. Visher, and Stephen M. Wandzura. Scalable electromagnetic scattering calculations on the SGI Origin 2000. In *Proceedings of Supercomputing*, The SCxy Conference series, Portland, Oregon, November 1999. ACM/IEEE.

[17] Nhan Phan-Thien, Ka Yan Lee, and David Tullock. Large scale simulation of suspensions with PVM. In *Proceedings of SC97*, The SCxy Conference series, San Jose, CA, November 1997. ACM/IEEE.

[18] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Proceedings of Supercomputing*, The SCxy Conference series, Baltimore, Maryland, November 2002. ACM/IEEE.

[19] V. Popov and Henry Power. An O(N) taylor sereis multipole boundary element method for three-dimensional elasticity problems. *Engineering Analysis with Boundary Elements*, 25:7–18, 2001.

[20] Fatih E. Sevilgen and Srinivas Aluru. A unifying data structure for hierarchical methods. In *Proceedings of Supercomputing*, The SCxy Conference series, Portland, Oregon, November 1999. ACM/IEEE.

[21] Jaswinder Pal Singh, Chris Holt, John L. Hennessy, and Anoop Gupta. A parallel adaptive fast multipole method. In *Proceedings of Supercomputing*, The SCxy Conference series, Portland, Oregon, November 1993. ACM/IEEE.

[22] Shang-Hua Teng. Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation. *SIAM Journal on Scientific Computing*, 19(2), 1998.

[23] Michael S. Warren and John K. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing*, The SCxy Conference series, Minneapolis, Minnesota, November 1992. ACM/IEEE.

[24] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of Supercomputing*, The SCxy Conference series, Portland, Oregon, November 1993. ACM/IEEE.

[25] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent fast multipole algorithm. Technical Report TR2003-839, Courant Institute, New York University, 2003. `http://www.cs.nyu.edu/csweb/Research/TechReports/TR2003-839/TR2003-839.pdf`.

[26] Kenichi Yoshida, Naoshi Nishimura, and Shoichi Kobayashi. Application of fast multipole Galerkin boundary integral equation method to elastostatic crack problems in 3D. *International Journal for Numerical Methods in Engineering*, 50(3):525–547, 2001.