Efficient propagation of moment tensors in multipole-accelerated parallel BEM applications

A. A. Mammoli and M. S. Ingber

Department of Mechanical Engineering
University of New Mexico, USA

Abstract

Fast multipole methods (FMM) have recently been applied to the simulation of Stokes flows with the boundary element method (BEM). Further work led to the simulation of Stokes flow for complex, multiply connected geometries. The use of massive parallelism is imperative for the treatment of realistic problems, because of the very large number of degrees of freedom that result from the accurate geometric description of such systems.

The parallelization of standard BEM applications that use iterative solvers such as GMRES to find an approximate solution to the linear systems is straightforward. The coefficient matrix is divided row-wise manner, and matrix coefficients are generated and used on the same processor. However, using the same matrix distribution with FMM leads to inefficiencies because moment tensors generated on a particular processor are used on other processors, which can lead to excessive communication.

The minimization of the interprocessor communication is the objective of this work. The amount of communication between processors can be reduced in a number of ways. First, the most effective compromise between the size of the near field and the far field must be made. Second, the optimum depth of the Taylor series expansion used in the FMM calculation must be found. Third, by numbering domain subdivisions appropriately and by restructuring the far-field part of the matrix multiplication, it is possible to reduce the inter-processor communication to a level where near-optimum scalability can be achieved. The third method is described here. Numerical results which show the scalability and load balancing of the new formulation are presented.
1 Introduction

The study of low Reynolds number flow of linear multiphase fluids is an ideal application of the BEM. As can be seen in Fig. 1, the geometry becomes very complex even with a small number of multiply connected boundaries. In addition, the geometry continually evolves as the flow progresses, necessitating remeshing at every timestep. While this would lead to prohibitive meshing operation counts with domain methods such as the finite element method (FEM), meshing is relatively simple with the BEM. The main reason that the BEM has not been applied to treat this type of problem is that the solution of a fully dense matrix of dimension $N$, where $N$ is the number of degrees of freedom of the system, is required. The operation count, under ideal condition, scales as $O(N^2)$. Using highly efficient algorithms, some dynamic problems were nevertheless analyzed with conventional parallel BEM implementations [2, 5].

![Figure 1: BEM mesh for a system of 20 spherical particles in a cylindrical container. The mesh is automatically refined in regions of near-contact.](image)

The potential to treat problems large enough to make efficient use of FMM was realized with the advent of large massively parallel computers. This is because even though the operation count can be reduced from $O(N^2)$ to $O(N \log N)$ for Stokes flow problems, the number of degrees of freedom $N$ required to achieve substantial improvements over conventional methods is on the order of several thousand.

A completed double-layer boundary integral formulation is used here [5, 6]. For the mobility problem, this formulation leads to well-conditioned systems of equations. In this case, the most efficient method to solve large matrices is by using an iterative solver, such as the generalized minimal residual method (GMRES) or the bi-conjugate gradient method (Bi-CG). These solvers are based only on
the multiplication of a trial solution vector by the matrix. The algorithm used to perform the matrix-vector operation is arbitrary.

Using classical BEM, the entire coefficient matrix needs to be stored. Each row in the matrix is then multiplied with the trial solution vector to obtain the entry of a residual vector. The most natural method of parallelizing such an operation is to split the matrix row-wise, so that an approximately equal number of rows is assigned to each processor. The coefficients for these rows are evaluated and used locally, with no need for communication, except for gathering the entire vector that results from the partial application of the linear operator on the trial solution vector.

The complexity of the parallel matrix-vector multiplication increases considerably with the application of FMM. With FMM, direct evaluation of the boundary integral over a cluster of boundary elements is substituted by multiplication of the kernel function and its derivatives with the corresponding moment tensors for the cluster. Earlier implementations of the method were based on row-wise splitting of the matrix multiplication [3]. This procedure required a global reduction operation in order to communicate the moment tensors for all clusters to all processors. With a small number of clusters, this was not a major concern, and near-perfect scalability was obtained [7]. Unfortunately, for large problems with complex geometries, this communication overhead was such that scalability became poor with as few as 80 processors [3]. The objective of this study is to investigate alternative matrix multiplication strategies that minimize inter-processor communication and memory requirements, to such an extent that the algorithm is suitable for use on distributed memory computers with potentially thousands of processors.

2 Calculation of cluster moment tensors

Detailed descriptions of the FMM implementation are available elsewhere [1, 4], however a brief summary of the key points will be given here in the interest of clarity. For a given collocation node \( x_i \), the integral of a function \( K(x,y) \) over boundary element \( \Gamma_j \) can be expressed as

\[
\int_{\Gamma_j} K(y,x) d\Gamma = \sum_{q=0}^{\infty} \frac{\partial^q}{\partial x_{k_1} \partial x_{k_2} \ldots \partial x_{k_q}} K(y_j,x) \times \\
\int_{\Gamma_j} \frac{(-1)^q}{q!} r_{k_1} r_{k_2} \ldots r_{k_q} d\Gamma \\
= \sum_{q=0}^{n} C_{k_1 k_2 \ldots k_q} \frac{\partial^q}{\partial x_{k_1} \partial x_{k_2} \ldots \partial x_{k_q}} K(y_j,x) + E_n(e)
\]

where \( y_j \) is a point in the neighborhood of \( \Gamma_j \), \( r = y - y_j \) and \( e \) is contained within a radius of \( y_j \) defined by \( \Gamma_j \). The indices \( k_1 k_2 \ldots k_q \) represent all permutations of \( q \) indices. Note that this is simply the integral of the Taylor expansion
of the kernel function $K(x_i, y)$ around point $y_j$. The terms $C_{k_1k_2...k_q}$ are the moment tensors of the geometry of $\Gamma_j$ about $y_j$.

Using a simple shifting operation, the moment tensors about point $y_j$ may be rewritten about another point, $p_l$. By shifting the moment tensors of a number of neighboring elements to the same point, their collective effect is 'clustered'. By shifting the moment tensors of such clusters to another common point $p_{l-1}$, the collective effect of a cluster of clusters is grouped into a single moment tensor. This operation can be repeated for successively larger clusters. The boundary integral for all boundary elements contained within a cluster is calculated by a single moment-derivative multiplication.

As shown in Fig. 2, a parallelepiped is selected that encompasses the entire geometry. It is then split into two 'children parallelepipeds', each containing part of the boundary. Splitting occurs so that the resulting parallelepipeds will have an aspect ratio as close as possible to unity. The children parallelepipeds are in turn split into two parts, and so on recursively. Splitting stops when the number of boundary elements in a parallelepiped is below a specified threshold. Note that by using this splitting scheme, the aspect ratio of any parallelepiped, except possibly for a small initial number, will be less than 2, independently of the geometry.

![Figure 2: Recursive subdivision of a three-dimensional multiply connected domain (left) and cluster list for the node highlighted by a black dot (right).](image)

Moment tensors are evaluated for each boundary element, then shifted to the smallest parallelepiped that contains it. These parallelepipeds then become the lowest level clusters, and represent the contribution of the elements they contain to the boundary integral. The moment tensors for the lowest level clusters are shifted to their parent clusters, and so on until the moment tensors for the top level parallelepiped, which contains the entire geometry, are evaluated.

For a given collocation node, the complete boundary integral is evaluated by
quadrature in the near field, and by cluster-derivative multiplication in the far field. The clusters for this collocation node, as shown in Fig. 2, are the largest that satisfy a particular error criterion, which depends on the integral formulation used.

3 Cluster distribution

In earlier implementations of the parallel multipole BEM code, the matrix-trial vector multiplication was split row-wise. Each processor was allocated a number of collocation nodes, and the boundary integral for each of these nodes was evaluated. Each collocation node was associated with a list of near-field elements, over which integration occurred as normal, and with a list of far-field clusters, where integration occurred by moment-derivative multiplication. For each matrix multiplication the entire vector resulting from a matrix-vector multiplication was re-assembled with an MPI_ALLGATHER operation.

Because the collocation nodes allocated to a particular processor are not necessarily contiguous, the moment tensors required to complete the boundary integration could be those associated with any of the $2^L$ clusters, where $L$ is the maximum recursive subdivision level. It was therefore necessary that the coefficients for all moment clusters be locally available to all processors. The moment tensors could be generated on a processor and then made available to all other processors using an MPI_ALLREDUCE operation.

With complex, multiply connected 3-D geometries, the number of clusters became significantly larger, leading to noticeable communication overhead. In addition, with large numbers of clusters, processor memory became a concern, since coefficients for large numbers of clusters needed to be stored on all processors. Since potentially hundreds or thousands of processors are required to solve problems of practical interest, a new approach was clearly necessary. The reduction of the overall number of moment tensors, and of the size of moment tensors, is a necessary first step. This work is described elsewhere [3], and will not be repeated here. The main improvement, described in this work, consists of a reformulation of the matrix-vector multiplication algorithm.

The numbering scheme for the clusters is a direct result of the recursive method used to generate the clusters. A natural subdivision of clusters, to be allocated to individual processors, follows from this numbering scheme. This can best be illustrated using Fig. 3.

The recursive cluster generation results in an inverted binary tree. At the root of the tree is the main cluster, which contains the entire geometry. The second cluster is at the second highest level towards the left. The third cluster is at the third highest level, to the left. This numbering pattern, down one level to the left, is repeated until a cluster that no longer needs to be split is found. At this point, numbering proceeds to the right, if there is another cluster at the same level. If this cluster can be split, downward motion commences again, otherwise numbering proceeds upwards.

Thus numbered, the clusters are subdivided between processors in number-
Figure 3: Binary tree representation of cluster numbering structure. The clusters highlighted by a small dot are those that belong to the fourth processor of twelve.

wise contiguous groups. One such group, allocated to the fourth processor of twelve, is highlighted in Fig. 3. Moment tensor generation occurs from the lowest level up. When moment tensors are generated for a cluster, these are shifted to the parent cluster. If the parent cluster is not on the same processor, an MPLSEND operation is initiated towards the processor that contains the parent cluster, and calculation of the next moment tensors for clusters of the same level proceeds. At the next higher level, cluster moment tensors are generated by addition of the moment tensors of their children. If a child cluster is on a different processor, an MPI-RECEIVE operation is initiated. Generally, the corresponding MPLSEND has already been initiated during the lower level scan. There are at most 4 point-to-point communications per level for a given cluster subdivision. This is a considerable improvement over the previous method, where all cluster tensors were communicated to all processors with an MPI_ALLREDUCE operation. With the new algorithm, the far field boundary integral evaluation is performed for all nodes, but only for the local cluster moment tensors. The effect of all cluster moment tensors is recovered when the matrix-vector multiplication result is gathered.

4 Load balancing

An even distribution of cluster moment tensors between processors is not ideal. This is because, in general, the total number of times that lower level clusters appear in the cluster lists associated with each collocation node is higher than for higher level clusters. Because the numbering scheme will generally result in uneven cluster level distributions between processors, the total number of moment tensor-derivative multiplications will not be even, leading to unbalanced loads, as shown in Fig. 4.

For this reason, a load leveling procedure is implemented here. Initially, the clusters are subdivided evenly between processors. After the cluster lists are generated, it is possible to evaluate the total number of moment tensor-derivative multiplications that will occur on each processor, without actually performing the mult-
An iterative scheme then proceeds to redistribute the clusters between processors, in such a way that contiguous subdivisions are maintained. The resulting load distribution is also shown in Fig. 4. Because at the end of each matrix multiplication the result must be gathered, the multiplication time is equal to the calculation time of the processor with the largest load.

![Figure 4: Number of cluster moment tensor-derivative multiplications on each processor for a 24 processor parallel run, for an even cluster distribution (left) and a balanced cluster distribution (right).](image)

5 Scaling

The objective of this work is to minimize inter-processor communication in the matrix-vector multiplication routine, in order to allow the efficient use of massive parallelism. The matrix multiplication time for a problem with the geometry shown in Fig. 1 is compared in Fig. 5 for the row-wise matrix parallelization and for the cluster-wise parallelization. The row-wise parallelization clearly suffered from a large communication overhead, as shown by the curved line in the log-log plot of multiplication time vs. number of processors. On the other hand, the almost perfectly straight line for the cluster-wise parallelization indicates that this overhead is almost completely removed.

Tests where the point-to-point cluster communication were turned off resulted in indistinguishable matrix-vector multiplication times. The small remaining communication overhead can therefore be attributed to the MPI_ALLGATHER operation used to gather the global matrix-vector result. Although this communication overhead is considerably smaller than the communication overhead due to global communication of moment tensors, it may become significant for very large problems or very high numbers of processors, and it should be addressed in such cases.
6 Concluding remarks

It has been shown that by appropriately numbering and distributing cluster moment tensors between processors, it is possible to achieve near-perfect scaling, with a very simple cluster subdivision scheme and some load balancing. The use of multipole acceleration for the BEM treatment of very large problems characterized by complex and evolving geometries is therefore feasible, because very large numbers of processors can be used efficiently.

Further optimization of the scaling performance can be obtained by modifying the method with which the result of the matrix-trial solution vector multiplication is to be made available to all processors. An attractive possibility is the exploitation of the small periods of idle time that necessarily result from any load imbalance which still exists, as can be seen in Fig. 4. When a processor has completed its allocated part of the matrix-vector multiplication, it sends a message indicating ready status to all other processors. The first processor that is able to receive the message will then initiate a send-receive operation, and the partial matrix-vector results can be combined. A new ready signal is then sent out for the next available processor, and so on until all partial results are compounded. Thus, by the time the last processor has completed its part of the matrix-vector operation, the complete matrix-vector result will be available, eliminating the need for the MPI_ALLGATHER operation.

While the matrix-vector multiplication is at the core of the multipole accelerated BEM, many other factors influence the performance of the algorithm. For example, preconditioners can significantly reduce the number of iterations required to obtain a solution. Other important aspects that require careful parallelization are the generation of near field and far field lists, meshing and particle position tracking. However, the scalability of the matrix-vector multiplication is a sine qua non.
non for the feasibility of large particle-level simulations of multiphase flows using multipole accelerated BEM. The results shown here are promising.

Acknowledgment

This research was supported in part by Sandia National Laboratory contracts number BD1625 and BF2833. The authors are grateful for this support. The parallel computations were performed on the IBM SP/2 at the Albuquerque High Performance Computing Center.

References


