Parallel Solution of the Volume Integral Equation of Electromagnetic Scattering

Jussi Rahola[1]

Center for Scientific Computing
P.O. Box 405, FIN-02101 Espoo, Finland

Abstract. Dense linear systems arising from volume integral equations can be effectively solved with iterative solvers. In this article we show how these iterative methods can be parallelized. The performance of the serial code is discussed on a vector system and on a RISC processor. The performance depends very much on the memory architecture. The parallel version of the code is written using MPI. We obtain good parallel speedup when the elements of the coefficient matrix are recomputed during each iteration. The speedup is moderate when a special FFT algorithm is used to compute the matrix-vector product.

1 Introduction

Apart from the usual technical applications, electromagnetic scattering can also be applied to study the scattering of light by dust particles in the atmosphere, in the solar system and in interstellar dust clouds [9]. These dust particles are often irregular, inhomogeneous and maybe also anisotropic. To model such scatterers, a volume integral equation formulation is needed. We have been studying the efficient solution of dense linear systems arising from the volume integral equation. Iterative solution methods and special methods for computing the matrix-vector product have made it possible to solve systems with hundreds of thousands of unknowns.

Parallel computing is becoming more and more important in achieving high performance in a cost-effective way. In this article we describe the parallel solution of the systems of linear equations. First, the integral equation formalism and its discretization are introduced. Iterative solution methods and different methods to compute the matrix-vector product are considered in Sect. 3. Then in Sect. 4 the performance of the code is measured on two example architectures. Section 5 describes the parallel implementation and its performance.

2 Integral Equations

The volume integral equation of electromagnetic scattering is given by

$$\mathbf{E}(\mathbf{r}) = \mathbf{E}_{\text{inc}}(\mathbf{r}) + k^3 \int_V (m(\mathbf{r}')^2 - 1)\mathbf{G}(\mathbf{r}, \mathbf{r}') \cdot \mathbf{E}(\mathbf{r}')\, d^3\mathbf{r}', \qquad (1)$$

where $\mathbf{E}(\mathbf{r})$ is the electric field inside the particle, $\mathbf{E}_{\text{inc}}(\mathbf{r})$ is the incident field, k is the wave number, m is the complex refractive index, $\mathbf{G}$ is the dyadic Green's function

$$\mathbf{G}(\mathbf{r},\mathbf{r}') = \left(1 + \frac{\nabla\nabla}{k^2}\right) g(|\mathbf{r}-\mathbf{r}'|) \tag{2}$$

and

$$g(r) = \frac{e^{ikr}}{4\pi kr}. \tag{3}$$

This integral equation is a strongly singular one, where special care must be taken in handling the singularity. To concentrate on the solution methods for the linear systems, we chose a simple discretization for the integral equation and computed the singular terms analytically. We also considered a closely related technique for scattering calculations, the discrete-dipole approximation, where each computational cell is replaced by a dipole [9].

The simplest discretization of the integral equations uses cubic cells and assumes that the electric field is constant inside each cube. By requiring that the integral equation (1) be satisfied at the centers $\mathbf{r}_i$ of the N cubes and by using simple one-point integration, we end up with the equation [12, 8]

$$\left(1 - (m(\mathbf{r}_i)^2 - 1)\left(k^3 M - \frac{1}{3}\right)\right)\mathbf{E}_i = \mathbf{E}_i^0 + \frac{k^3 V_1}{4\pi}\sum_{\substack{j=1\\ j\neq i}}^{N}(m(\mathbf{r}_j)^2 - 1)\mathbf{T}_{ij}\cdot\mathbf{E}_j, \tag{4}$$

where $i = 1,\ldots,N$, V_1 is the volume of the computational box, M is given by

$$M = \frac{2}{3k^3}\left(\left(1 - ikb(3/4\pi)^{1/3}\right)e^{ikb(3/4\pi)^{1/3}} - 1\right), \tag{5}$$

b is the length of side of the computational box and $\mathbf{T}_{ij}$ is given by

$$\begin{aligned}\mathbf{T}_{ij} &= \frac{e^{i\rho_{ij}}}{\rho_{ij}^3}(\rho_{ij}^2 + i\rho_{ij} - 1)\mathbf{1} + \frac{e^{i\rho_{ij}}}{\rho_{ij}^3}(-\rho_{ij}^2 - 3i\rho_{ij} + 3)\hat{\mathbf{r}}_{ij}\hat{\mathbf{r}}_{ij}, \\ \rho_{ij} &= k|\mathbf{r}_i - \mathbf{r}_j|.\end{aligned} \tag{6}$$

Here $\hat{\mathbf{r}}_{ij} = (\mathbf{r}_i - \mathbf{r}_j)/|\mathbf{r}_i - \mathbf{r}_j|$.

3 Iterative Methods and Matrix-vector Products

The computational challenge in our scattering computations is the solution of a large dense system of linear equations with complex coefficients. For homogeneous and isotropic scatterers, the coefficient matrix is complex symmetric. We have studied the use of iterative methods in this setting [14] and found that the complex symmetric version of QMR [3] performs very well. We also tested the block version of QMR [1] but the computational benefit from this method is limited for the refractive indices used in our simulations.

Most of the time the iterative methods converge so quickly that there is no need for preconditioning. It also seems that the possible benefit from preconditioning is limited. We tried for example the approximate inverse preconditioning but it did not sufficiently reduce the number of iterations to warrant for the extra work. For high values of the refractive index the convergence slows down and then preconditioning becomes an important issue.

An iterative solver (namely a Krylov space method) consists of four basic operations: a vector update (saxpy), a dot product, the matrix-vector product and the application of the preconditioner. When we are solving a dense system of linear equation without preconditioning, the matrix-vector product takes by far the most CPU time in the computations.

In our production code, the matrix-vector product can be computed using four methods:

1. The matrix elements can be recomputed each time they are referenced. This method needs very little storage.
2. The matrix can be precomputed and stored. The matrix-vector product is computed efficiently using a BLAS routine. This method quickly exceeds the memory size of the machine.
3. Using the symmetry of the matrix, the matrix can be stored in packed form that takes one third of the space required for the full matrix.
4. A special FFT algorithm can be used.

Let the scatterer consist of N computational cells. The system of linear equations has $3N$ equations because the x, y and z components of the electric fields are considered. Correspondingly, the matrix has the following block structure:

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}. \tag{7}$$

All the 9 $N \times N$ blocks are symmetric: $A_{ij}^T = A_{ij}$. Furthermore, $A_{ij}^T = A_{ji}$. Thus the whole information is stored in the upper triangular parts of the blocks A_{11}, A_{22}, A_{33}, A_{12}, A_{13} and A_{23}.

When the computational domain is enlarged to a homogeneous cube, the matrix-vector product reduces to a 3D-convolution that can efficiently be computed with the FFT [4]. The use of FFT is plausible when the scatterer can be modeled with densely packed cubic cells.

In the FFT algorithm, the computational domain is enlarged to a cubic lattice containing the scatterer. The computational cells of the scatterer are assumed to sit on the regular cubic lattice. When the scatterer consists of N computational cells, let the enlarged lattice contain N_c cells. The cubic lattice has to be doubled in each direction to get a 3D convolution.

The matrix-vector product $\mathbf{y} = A\mathbf{x}$ can be computed by taking the FFT of the first row of the coefficient matrix, multiplying this by the FFT of the vector $\mathbf{x}$, and then recovering $\mathbf{y}$ by the inverse FFT. Note that here both $\mathbf{x}$ and $\mathbf{y}$ together with the first row of A are considered as values on the enlarged and

doubled cubic lattice. Also the block structure of the coefficient matrix is taken into account.

One can also exploit the symmetry in the FFT calculations: only 6 of the 9 blocks of the coefficient matrix have to be stored. Thus the storage requirements for $\mathbf{x}$ and $\mathbf{y}$ in the enlarged lattice is $24N_c^2$ and for the row of the coefficient matrix is $48N_c^2$ double precision complex numbers.

We have also considered the use of another special technique, the fast multipole method (FMM) [5] to compute the matrix-vector product. However, this has not been incorporated into the production code. The fast multipole method is a method for computing the pairwise interactions of a large number of particles by using truncated potential expansions. The centers of the expansions have to be translated during the FMM algorithm.

In the electromagnetic scattering case the translation becomes very expensive as the potential expansions involve many special functions and the translation involves a multiple summation. Using the so called diagonal translation operators [15] the potentials are represented in Fourier space where the translation is equivalent to point-wise multiplication and the potential can be recovered by numerical integration. We gave a simplified derivation of the diagonal translation operators and also derived error bounds for the FMM algorithm that take into account the truncation error of the potential expansions together with the error from the numerical integration [13]. The FMM algorithm involves fairly complicated data structures and communication patterns and thus special care has to be taken to ensure load balancing and data locality in the parallel implementation [11]. The FMM is perhaps better suited to the surface integral equation formulation of scattering where the FFT cannot be used.

4 Single Node Performance

In the rest of the article we will consider spherical particles of various sizes where the side length of the computational cell is kept at 0.3 (the wave number is assumed to be 1). In all the experiments, the refractive index is $1.6 + 0.05i$. The code uses double precision throughout.

The integral equation solver is currently implemented for a Cray C94 vector computer and for an IBM SP2 distributed memory parallel computer at the Center for Scientific Computing (CSC). The SP equipment consists of both the so-called thin and wide nodes. The thin nodes are allocated for parallel computing. In the future the code will be ported to the Cray T3E parallel computer that will be installed at CSC during the summer of 1996.

The code is written using the Fortran 90 programming language. The matrix-vector products and the Fourier transforms are computed using optimized subroutines in the Cray and IBM mathematical libraries. We are working with a single source code which contains cpp preprocessor directives. Serial and parallel versions for the Cray and IBM are created on compile time.

Table 1 shows the execution times and MFLOPS rates on a single processor of the Cray C94 for several problem sizes. Note that the execution time is given

for a single solve of the system of linear equations. In the actual production code the system of linear equations has to be solved twice, once for each incident polarization state. Table 2 shows the same information for a single thin node processor of the IBM SP2. The theoretical peak MFLOPS rates are 952 for the Cray C94 and 264 for the SP2 processor.

Table 1. The performance of the scattering code on a single processor of the Cray C94. The first column gives the number of computational cells in the scatterer, the second column gives the number of iterations for the QMR solver. Note that the number of equations is three times the number of computational cells. The CPU times t (in seconds) and MFLOPS rates r_{MF} are reported for four methods of computing the matrix-vector products A_t (1=recomputation of the matrix entries, 2=use of the full matrix, 3=use of the packed matrix, 4=FFT algorithm). A dash indicates that the problem could not be run due to insufficient memory

		$A_t = 1$		$A_t = 2$		$A_t = 3$		$A_t = 4$	
Size	Iterations	t	r_{MF}	t	r_{MF}	t	r_{MF}	t	r_{MF}
32	8	0.12	22	0.004	17	0.006	15	0.08	23
136	11	0.11	182	0.02	150	0.05	128	0.2	57
304	12	0.75	233	0.09	376	0.3	208	0.3	81
1064	16	7.4	425	1.5	664	3.5	360	1.2	132
2330	20	39	461	–	–	37	282	2.2	203

Table 2. Same as Table 1 but for the IBM SP2 thin node processor.

		$A_t = 1$		$A_t = 2$		$A_t = 3$		$A_t = 4$	
Size	Iterations	t	r_{MF}	t	r_{MF}	t	r_{MF}	t	r_{MF}
32	8	0.03	41	0.01	37	0.01	40	0.07	29
136	11	0.88	48	0.17	49	0.43	33	0.33	30
304	12	4.9	49	0.89	39	2.95	27	1.1	35
1064	16	117	34	–	–	–	–	4.2	35

Table 3 gives the performance of the code using the FFT algorithm for the Cray and for the SP2 thin and wide node processors. The MFLOPS numbers are acquired from the hpm command on the Cray and from the rs2hpm tool developed by Jussi Mäki for the IBM SP2 [10].

For large problems the performance of a wide node is almost double the performance of a thin node because of the improved memory bandwidth of the wide nodes. In the Cray implementation the arrays holding the FFT data were enlarged by one position in each dimension in order to avoid memory bank

Table 3. Performance on a single CPU of the Cray C90 and on a thin and wide node of the SP2 for large problems when the matrix-vector product is computed with the FFT algorithm.

		Cray		IBM thin node		IBM wide node	
Size	Iterations	t	r_{MF}	t	r_{MF}	t	r_{MF}
2320	20	2.2	203	17	32	11	47
5232	29	9	319	75	35	34	77
10048	38	18	381	176	42	98	76
20336	65	60	418	541	41	270	82
137376	264	1233	446	–	–	–	–

conflicts. For large problems this improved the performance of the code by a factor of three.

5 Parallelization

As almost all computational work in the scattering calculations goes into the solution of the systems of linear equations, we concentrate on the parallelization of the iterative solver. The pre- and postprocessing stages can be executed on one processor only. Because in a typical scattering calculation many orientational averages are taken, the calculation could trivially be parallelized by calculating each orientational average independently in different processors. Here we are aiming at the solution of large scattering problems for which a truly parallel solver is necessary. A parallel solver for the discrete-dipole approximation was also given in [6, 7].

The scattering code is parallelized using the MPI (Message Passing Interface) communication library. More exactly, the MPICH implementation from Argonne National Laboratory is used.

All the vectors in the parallel version of the QMR iterative solver are distributed across the processors. The vector update ($x = x + \alpha y$) can be done totally indenpendently. To compute dot products, each processor computes a partial sum, broadcasts this value to all other processors who can then compute the global sum. In the code the broadcast and the reduce operations can be accomplished with a single MPI call, `MPI_ALLREDUCE`.

Now we consider the parallelization of the four different methods to compute the matrix-vector product $\mathbf{y} = A\mathbf{x}$. Initially the vector $\mathbf{x}$ is distributed and the result vector $\mathbf{y}$ should also be. For the method 2 (use of the full matrix), blocks of rows of the matrix are assigned to each processor (see Fig. 1). The matrix elements are precomputed in parallel. To compute $\mathbf{y} = A\mathbf{x}$, the whole vector $\mathbf{x}$ is first gathered and broadcast to all processors (`MPI_ALLGATHERV`) who can then independently compute their slices of the result vector $\mathbf{y}$. These slices will conform to the distribution of other vectors in the QMR algorithm.

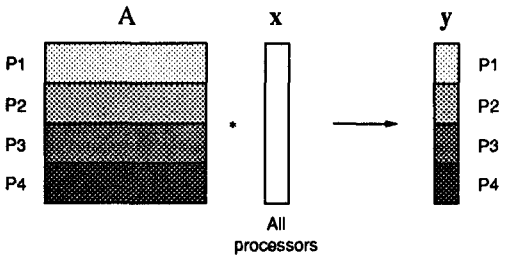

Fig. 1. Decomposition of the full coefficient matrix for the matrix-vector product among four processors. The distributed vector **x** is gathered and broadcast to all processors. After the local matrix-vector operations the result vector **y** is distributed between the processors.

In the matrix-vector product 3 (packed form) the rows each of the three packed matrix blocks are distributed cyclically in order to obtain load balancing. In the method 1 (recomputation), rows and columns of the coefficient matrix are assigned cyclically to each processor so that it uses the elements from the rows in the upper triangular parts of the matrix blocks and elements from the columns in the lower triangular parts. In the methods 1 and 3, the processors each first need the whole vector **x** whereafter each processor computes a partial sum of the result vector. In the end the vectors are summed to obtain **y** which is then scattered and distributed among the processors (`MPI_REDUCE_SCATTER`).

The method 4, matrix-vector product with the 3D Fourier transform, can be parallelized as follows. First the enlarged computational cube is evenly partitioned in the z direction between the processors, i.e. in N_{proc} slices. In the enlarged cube, elements of the original vector **x** correspond to elements in the first octant of the cube. Thus, within QMR this distribution would leave half of the processors idle. To ensure load balance, within the QMR algorithm the vectors obey another distribution. Some parts of the vectors will be mapped in the same processor for both distributions, other require communication.

Another possibility is to divide the whole computational cube into $2N_{proc}$ slices and use the distribution implied by the first octant inside the QMR algorithm. If the computational elements are very unevenly distributed within the FFT cube, there can be some load balancing problems for the QMR algorithm.

Now the matrix-vector product with the FFT is computed as follows. First local parts of the QMR vector **x** are mapped to the local parts of the enlarged computational cube, implying communication. Each processor holds initially blocks of x-y planes so that the cube is sliced in the z direction.

We parallelized the 3D FFT using the transpose-based algorithm [2]. The 1D FFT's in the x and y directions can be done independently. Then the data is transposed so that each processor holds a number of x-z planes (cube sliced in the y direction). Now the 1D FFT's in the z direction can be done in parallel. Figure 2 clarifies the situation.

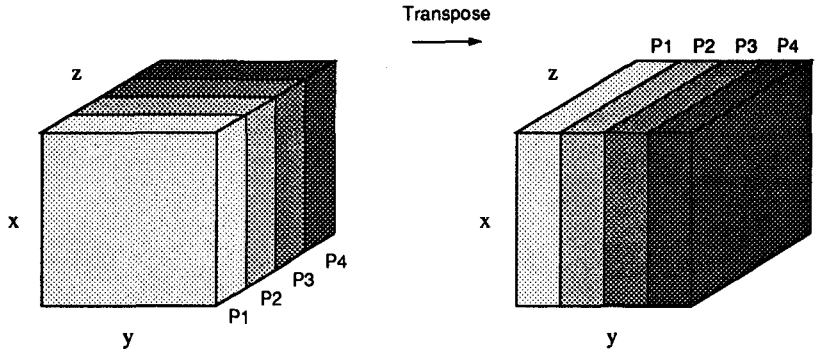

Fig. 2. Decomposition of the computation cube for the FFT method. In the left figure, the 1D FFT's can be done in parallel in the x and y directions. After a transposition, the 1D FFT's in the z direction can be done in parallel, too.

The first row of the coefficient matrix is transformed similarly in the initialization phase and left in the transposed position. It can now be multiplied by the FFT of **x**. The inverse Fourier transform is accomplished by an inverse FFT in the z direction, transposition and then by inverse FFT's in the x and y directions. The actual vector **y** is gathered from the enlarged computational box.

Our first parallel experiments are run on the SP2 at CSC. The methods 2 and 3 for computing the $A\mathbf{x}$ did not show much parallel speedup because we could not run large enough problems or use enough processors.

Tables 4 and 5 give the execution times (wall-clock time) for some of the largest configurations and for 1 to 4 processors when the matrix elements are recomputed and the FFT algorithm is used, respectively. Currently it is difficult to get more than 4 processors to a single user in CSC's production environment. We also list the execution time for the serial version. Note that there is some overhead in the parallel code on one processor that result from copying of data.

Table 4. Execution times (in elapsed seconds) for the serial code (t_{serial}) and for the parallel code on one, two and four processors. The matrix elements are recomputed when they are needed

Size	t_{serial}	$p = 1$	$p = 2$	$p = 4$
1064	123	127	64	34
2320	843	848	427	212

Our initial parallel version of the code only used the parallel matrix-vector product so that the QMR algorithm was run on a single processor. The com-

Table 5. Same as Table 4 except the matrix-vector product is computed using the FFT algorithm.

Size	t_{serial}	$p = 1$	$p = 2$	$p = 4$
2320	17	23	17	12
5232	76	95	61	33
10048	178	228	142	93
20336	558	-	376	237

putational results were quite similar, indicating that here one could indeed only concentrate on the matrix-vector product. However, when large systems are solved on hundreds of processors, the QMR vector operations take a considerable amount of time and have thus to be parallelized. Also, the parallelization reduces the memory requirements of the master processor.

6 Conclusion

We have shown how to solve the systems of linear equations arising from integral equations of electromagnetic scattering in parallel. The iterative methods QMR and the matrix-vector product is parallelized using the MPI library. The most important case is when the matrix-vector product can be computed with a 3D FFT. We showed performance of the serial code on a Cray C90 and on thin and wide nodes on an IBM SP2. The Cray gets much closer to its peak performance due to better memory bandwidth. On the SP, the performance also depends on the memory bandwidth, as shown by the differences of the two node types.

If the elements of the coefficient matrix are recomputed each time they are needed, the code needs very little memory and parallelizes very well. On the other hand, if the matrix-vector product is computed using the FFT algorithm, the speedup is initially not as great.

The final target system for the scattering code is the Cray T3E. Some of the message-passing calls might have to be replaced by shared-memory calls of the T3E to get high performance. The use of a vendor-tuned parallel FFT library code will also be examined.

The parallelization of the solver is fairly straightforward. One can concentrate on the computationally important routines and thus only a small amount of code has to be changed for the parallel implementation. The assortment of high-level communication subroutines in MPI proved to be very helpful in coding the program. We used the routines for gathering a distributed vector to all processors, computing the componentwise sum of vectors in different processors and simultaneously scattering the vector to the processors, together with the routine to add up partial sums and to broadcast the result to all processors. Many of the MPI routines can work with vectors of variable size which is very useful when the dimensions of the problem are not divisible with the number of processors.

References

1. W. E. Boyse and A. A. Seidl. A block QMR method for computing multiple simultaneous solutions to complex symmetric systems. *SIAM J. Sci. Comput.* **17**, 263–274, 1996.
2. Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley, Reading, Massachusetts, 1995.
3. R. W. Freund. Conjugate gradient-type methods for linear systems with complex symmetric coefficient matrices. *SIAM J. Sci. Stat. Comput.* **13**, 425–448, 1992.
4. J. J. Goodman, B. T. Draine, and P. J. Flatau. Application of fast-Fourier-transform techniques to the discrete-dipole approximation. *Optics Letters* **16**, 1198–1200, 1991.
5. L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.* **73**, 325–348, 1987.
6. A. Hoekstra. *Computer Simulations of Elastic Light Scattering, Implementation and Applications.* PhD thesis, University of Amsterdam, 1994.
7. A. G. Hoekstra and P. M. A. Sloot. Coupled dipole simulations of elastic light scattering on parallel systems. *Int. J. Modern Phys. C* **6**, 663–679, 1995.
8. A. Lakhtakia and G. W. Mulholland. On two numerical techniques for light scattering by dielectric agglomerated structures. *J. Res. Natl. Inst. Stand. Technol.* **98**, No. 6, 699–716, 1993.
9. K. Lumme and J. Rahola. Light scattering by porous dust particles in the discrete-dipole approximation. *Astrophys. J.* **425**, 653–667, 1994.
10. J. Mäki. Power2 hardware performance monitor tools. http://www.csc.fi/~jmaki/-rs2hpm.html.
11. J. Pal Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole and radiosity. *J. Parallel Distr. Comput.* **27**, 118–141, 1995.
12. J. Rahola. Solution of dense systems of linear equations in electromagnetic scattering calculations. Licenciate's thesis, Helsinki University of Technology, 1994.
13. J. Rahola. Diagonal forms of the translation operators in the fast multipole algorithm for scattering problems. *BIT* **36**, 333–358, 1996.
14. J. Rahola. Solution of dense systems of linear equations in the discrete-dipole approximation. *SIAM J. Sci. Comput.* **17**, 78–89, 1996.
15. V. Rokhlin. Diagonal forms of translation operators for the Helmholtz equation in three dimensions. *Applied and Computational Harmonic Analysis* **1**, 82–93, 1993.