1 AN EXAMINATION OF THE PERFORMANCE OF TWO ELECTROMAGNETIC SIMULATIONS ON A BEOWULF-CLASS COMPUTER

Abstract: *This paper uses two electromagnetic simulations to examine some performance and compiler issues on a Beowulf-class computer. This type of computer, built from mass-market, commodity, of-the-shelf components, has limited communications performance and therefore also has a limited regime of codes for which it is suitable. This paper first shows that these codes fall within this regime, and then examines performance data, including run-time, scaling, compiler choices, and the use of some hand-tuned optimizations, comparing results from a Beowulf with those from a Cray T3D, and a T3E.*
Keywords: Beowulf, cluster, pile of PCs, parallel computation, electromagnetics, finite-difference time-domain, physical optics, radiation integral.

Daniel S. Katz , Tom Cwik, Thomas Sterling

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA
E-Mail: {Daniel.S.Katz, cwik}@jpl.nasa.gov, tron@cacr.caltech.edu

1.1 INTRODUCTION

A typical small Beowulf system, such as the machine at the Jet Propulsion Laboratory (JPL) may consist of 16 nodes interconnected by 100Base-T Fast Ethernet. Each node may include a single Intel Pentium Pro 200 MHz microprocessor, 128 MBytes of DRAM, 2.5 GBytes of IDE disk, and PCI bus backplane, and an assortment of other devices. At least one node will have a video card, monitor, keyboard, CD-ROM, floppy drive, and so forth. But the technology is evolving so fast and price performance and price feature curves are changing so fast that no two Beowulfs ever look exactly alike. Of course, this is also because the pieces are almost always acquired from a mix of vendors and distributors. The power of *de facto* standards for interoperability of subsystems has generated an open market that provides a wealth of choices for customizing one's own version of Beowulf, or just maximizing cost advantage as prices fluctuate among sources. Such a system will run the Linux (Husain et al., 1996) operating system freely available over the net or in low-cost and convenient CD-ROM distributions. In addition, publicly available parallel processing libraries such as MPI (Snir et al., 1996) and PVM (Giest et al., 1994) are used to harness the power of parallelism for large application programs. A Beowulf system such as described here, taking advantage of appropriate discounts, costs about $30K including all incidental components such as low cost packaging.

At this time, there is no clearly typical medium to large a Beowulf system, since as the number of processors grows, the choice of communications network is no longer as clear. (If the machine can use a crossbar that can support the entire machine, the choice is simply to use that crossbar switch.) Many choices exist of various topologies of small and large switches and hubs, and combinations thereof.

Naegling, the Beowulf-class system at the California Institute of Technology, which currently has 140 nodes, has had a number of communications networks. The first was a tree of 8- and 16-port hubs. At the top of the tree was a standard 100 Mbit/s 16-port crossbar, with full backplane bandwidth. Each port of this was connected to a hub. Each hub had 100 Mbit/s ports connected to 8 or 16 computers; however, the backplane bandwidth of each hub was also 100 Mbit/s. The second topology used additional 16-port crossbars at the low level of the tree, where 15 ports of each crossbar were connected to computers, and the last port was connected to a high-level crossbar. A third network (which is the current topology) involves 2 80-port switches, connected by 4 Gbit/s links. Each switch is intended to have 100 Mbit/s ports and full backplane bandwidth. More details about how this network performs will be discussed in the two results sections.

The Beowulf approach represents a new business model for acquiring computational capabilities. It complements rather than competes with the more conventional vendor-centric systems-supplier approach. Beowulf is not for everyone. Any site that would include a Beowulf cluster should have a systems administrator already involved in supporting the network of workstations and PCs that inhabit the workers' desks. Beowulf is a parallel computer, and as such, the site must be willing to run parallel programs, either developed in-house or acquired from others. Beowulf is a loosely coupled, distributed memory system, running message-passing parallel programs that do not assume a shared memory space across processors. Its long latencies require a favorable balance of computation to communication and code written to balance the workload across processing nodes. Within the constrained regime in which Beowulf is appropriate, it should provide the best performance to cost and often comparable performance per node to vendor offerings (Katz, et. al., 1998). This paper will examine two electromagnetic simulation codes which fit within this regime.

1.2 PHYSICAL OPTICS SIMULATION

The first code described in this paper (Imbriale and Cwik, 1994) is used to design and analyze reflector antennas and telescope systems. It is based simply on a discrete approximation of the radiation integral (Imbriale and Hodges, 1991). This calculation replaces the actual reflector surface with a triangularly faceted representation so that the reflector resembles a geodesic dome. The Physical Optics (PO) current is assumed to be constant in magnitude and phase over each facet so the radiation integral is reduced to a simple summation. This program has proven to be surprisingly robust and useful for the analysis of arbitrary reflectors, particularly when the near-field is desired and the surface derivatives are not known.

Because of its simplicity, the algorithm has proven to be extremely easy to adapt to the parallel computing architecture of a modest number of large-grain computing elements. The code was initially parallelized on the Intel Paragon, and has since been ported to the Cray T3D, T3E, and Beowulf architectures.

For generality, the code considers a dual-reflector calculation, as illustrated in Figure 1, which can be thought of as three sequential operations: (1) computing the currents on the first (sub-) reflector using the standard PO approximation; (2) computing the currents on the second (main) reflector by utilizing the currents on the first (sub-) reflector as the field generator; and (3) computing the required observed field values by summing the fields from the currents on the second (main) reflector. The most time-consuming part of the calculation is the computation of currents on the second reflector due to the currents on the first, since for N triangles on the first reflector, each of the M triangles on the second reflector require an N-element sum over the first. At this time, the code has been parallelized by distributing the M triangles on the second reflector, and having all processors store all the currents on the N triangles of the first reflector (though the computation of the currents of the first reflector is done in parallel.) Also, the calculation of observed field data has been parallelized. So, the three steps listed above are all performed in parallel. There are also sequential operations involved, such as I/O and the triangulation of the reflector surfaces, some of which potentially could be performed in parallel, but this would require a serious effort, and has not been done at this time.

This code is written in FORTRAN, and has been parallelized using MPI. Communication is required in two locations of the code. At the end of the first step, after each processor has computed a portion of the currents on the first reflector, the currents must be broadcast to all the processors. While this may be done in many ways, a call to *MPI_Allgatherv* is currently used. During the third step, each processor calculates a partial value for each final observed field, by integrating over the main reflector currents local to that processor. A global sum (an *MPI_Reduce* call) is required to compute the complete result for each observed field value. Since there are normally a number of far fields computed, currently there are that number of global sums. These could be combined into a single global sum of larger length, but this has not been done at this time, since the communication takes up such a small portion of the overall run time.

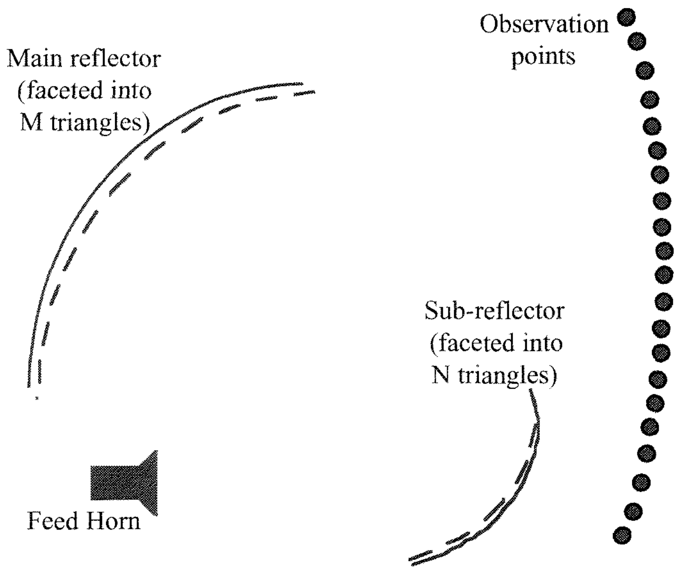
Observation points
Main reflector (faceted into M triangles)
Sub-reflector (faceted into N triangles)
Feed Horn

Figure 1. The dual reflector Physical Optics problem, showing the source, the two reflectors, and the observation points.

1.3 FDTD SIMULATION

The Finite-Difference Time-Domain (FDTD) code (Taflove, 1995) studied here is a fairly simple example of a time-stepping partial differential equation (PDE) solution over a physical problem domain which is distributed over the memory of a mesh of processors. In traditional FDTD electromagnetic codes, there are generally six field unknowns which are staggered in time and space. For the purposes of this paper, they can be thought of as residing in a spatial cell, where each cell is updated at each time step. This particular code adds two specific features. First, each field component is split into two sub-components which are stored in memory and updated separately. This is done in order to implement a boundary condition (Berenger 1994). Second, the parallelization that was done for this code tried to reduce the required communication, and therefore redundantly updates some of the sub-components on the face of each processor's domain, and communicates only four of the sub-components on each face.

This code is written in FORTRAN using MPI. The decomposition performed is two-dimensional (in x and y), while the spatial region modeled is three-dimensional. The processors are mapped into a two-dimensional Cartesian mesh using MPI's facilities for Cartesian communicators, and each processor models a physical domain that contains a subset of the entire physical domain in x and y, and the entire domain in z. Because of this, at each time each processor swaps one face (a complete y-z plane) of four sub-components in the $\pm x$ direction, and one face (a complete x-z plane) of four other sub-components in the $\pm y$ direction. The communication is done as follows: each processor issues an *MPI_IRecv* call to each neighboring processor (usually 4, except on the edges of the processor mesh); each processor fills and sends a buffer in each appropriate direction, suing an *MPI_SSend* call; and finally, each processor does a number of *MPI_Wait* operations, followed by unpacking the received data. This combination of calls should produce no additional buffering of data, since the program is already doing some in an effort to reduce the number of messages.

1.4 PO RESULTS

Timing results for the PO code in this paper are presented by breaking down the overall timing into three parts. Part I is input I/O and triangulation of the main reflector surface, some of which is done in parallel. No communication occurs in part I. Part II is triangulation of the sub-reflector surface (sequential), evaluation of the currents on the sub-reflector (parallel), and evaluation of the currents on the main reflector (parallel). A single *MPI_Allgatherv* occurs in part II. Part III is evaluation of the observed fields (parallel) and I/O (on only one processor). A number of 3 word global sums occur in part III, one for each observation point. In the test cases used here, there are 122 observation points. The Beowulf results are from the 16 node system, using the GNU g77 compiler.

Two different compilers were compared (Gnu g77 and Absoft f77) on the Beowulf system. One set of indicative results from these runs are shown in Table 1. For this code, the Absoft compiler produced code that was approximately 30% faster, and this compiler was used hereafter.

It should be mentioned that the computation of the radiation integral in two places (in parts II and III) originally had code of the form:

```
CEJK = CDEXP(-AJ*AKR),
```

where `AJ = (0.d0,1.d0)`. This can be changed to:

```
CEJK = DCMPLX(DCOS(AKR),-DSIN(AKR)).
```

On the T3D, these two changes led to improved results (the run-times were reduced by 35 to 40%,) which are shown in this paper. When these changes were applied to the Beowulf code using the second compiler, no significant performance change was observed, leading to the conclusion that one of the optimizations performed by this compiler was similar to this hand-optimization.

Number of Processors	*Compiler 1*			*Compiler 2*		
	I	*II*	*III*	*I*	*II*	*III*
1	0.0850	64.3	1.64	0.0482	46.4	0.932
4	0.0515	16.2	0.431	0.0303	11.6	0.237
16	0.0437	4.18	0.110	0.0308	2.93	0.0652

Table 1. The effect of a two Beowulf compilers (gnu g77 and Absoft f77), shown by timing results (in minutes) for PO code, for M=40,000, N=4,900.

It may be observed from Tables 2, 3, and 4 that the Beowulf code performs slightly better than the T3D code, both in terms of absolute performance as well as scaling from 1 to 64 processors. (Tables 2 and 3 contain results obtained on Hyglac, and Table 4 contains results obtained on Naegling.) This performance difference can be explained by the faster CPU on the Beowulf versus the T3D, and the very simple and limited communication not enabling the T3D's faster network to influence the results. The scaling difference is more a function of I/O, which is both more direct and more simple on the Beowulf, and thus faster. By reducing this part of the sequential time, scaling performance is improved. Another way to look at this is to compare the results in the three tables. Clearly, scaling is better in the larger test case, in which I/O is a smaller percentage of overall time. It is also clear that the communications network used on Naegling is behaving as designed for the PO code running on 4, 16, or 64 processors. Since the majority of communication is single word global sums, this basically demonstrates that the network has reasonable latency.

Number of Processors	*Beowulf*			*T3D*		
	I	*II*	*III*	*I*	*II*	*III*
1	3.19	230	56.0	14.5	249	56.4
4	1.85	57.7	14.2	8.94	62.5	14.7
16	1.52	14.6	3.86	8.97	16.6	4.13

Table 2. Timing results (in seconds) for PO code, for M=40,000, N=400.

Number of Processors	*Beowulf*			*T3D*		
	I	*II*	*III*	*I*	*II*	*III*
1	0.0482	46.4	0.932	0.254	48.7	0.941
4	0.0303	11.6	0.237	0.149	12.2	0.240
16	0.0308	2.93	0.0652	0.138	3.09	0.0749

Table 3. Timing results (in minutes) for PO code, for M=40,000, N=4,900.

Number of Processors	*Beowulf*			*T3D*		
	I	*II*	*III*	*I*	*II*	*III*
4	0.0950	94.6	0.845	0.546	101	0.965
16	0.0992	23.9	0.794	0.463	25.6	0.355
64	0.0950	6.38	0.541	0.520	6.93	0.116

Table 4. Timing results (in minutes) for PO code, for M=160,000, N=10,000.

Tables 5, 6, and 7 show comparisons of complete run time for the 3 test problems sizes, for the Beowulf, T3D, and T3E-600 systems. These demonstrate good performance on the two Beowulf-class machines when compared with the T3D in terms of overall performance, as well as when compared with the T3E-600 in terms of price-performance. For all three test cases, the Beowulf scaling is better than the T3D scaling, but the results are fairly close for the largest test case, where the Beowulf being used is Naegling. This can be explained in large part by I/O requirements and timings on the various machines. The I/O is close to constant for all test cases over all machine sizes, so in some way it acts as serial code that hurts scaling performance. The I/O is the fastest on Hyglac, and slowest on the T3D. This is due to the number of nodes being used on the Beowulf machines, since disks are NFS-mounted, and the more nodes there are, the slower the performance is using NFS. The T3D forces all I/O to travel through its Y-MP front end, which causes it to be very slow. Scaling on the T3D is generally as good as the small Beowulf, and faster than the large Beowulf, again due mostly to I/O. It may be observed that the speed-up of the second test case on the T3E is superlinear in going from 1 to 4 processors. This is probably caused by a change in the ratio of some of the size of some of the local arrays to the cache size dropping below 1.

Number of Processors	*Beowulf (Hyglac)*	*Cray T3D*	*Cray T3E-600*
1	289	320	107
4	73.8	86.1	29.6
16	20.0	29.2	8.36

Table 5. Timing results (in seconds) for complete PO code, for M=40,000, N=400.

Number of Processors	*Beowulf (Hyglac)*	*Cray T3D*	*Cray T3E-600*
1	47.4	49.4	18.4
4	11.9	12.6	4.43
16	3.03	3.30	1.14

Table 6. Timing results (in minutes) for complete PO code, for M=40,000, N=4,900.

Number of Processors	*Beowulf (Naegling)*	*Cray T3D*	*Cray T3E-600*
4	95.5	102	35.1
16	24.8	26.4	8.84
64	7.02	7.57	2.30

Table 7. Timing results (in minutes) for complete PO code, for M=160,000, N=10,000.

A hardware monitoring tool was used on the T3E to measure the number of floating point operations in the M=40,000, N=4,900 test case as 1.32×10^{11} floating point operations. This gives a rate of 46, 44, and 120 MFLOP/s on one processor of the Beowulf, T3D, and T3E-600 respectively. These are fairly good (23, 29, and 20% of peak, respectively) for RISC processors running FORTRAN code.

1.5 FDTD RESULTS

All FDTD results that are shown in this section use a fixed size local (per processor) grid, of 69×69×76 cells. The overall grid sizes therefore range from 69×69×76 to 552×552×76 (on 1 to 64 processors). (All Beowulf results are from Naegling.) This is the largest local problem size that may be solved on the T3D, and while the other machines have more local memory and could solve larger problems, it seems more fair to use the same amount of local work for these comparisons. In general, the FDTD method requires 10 to 20 points per wavelength for accurate solutions, and a boundary region of 10 to 20 cells in each direction is also needed. These grid sizes therefore correspond to scattering targets ranging in size from 5×5×5 to 53×53×5 wavelengths.

Both available compilers were used on the Beowulf version of the FDTD code. While the results are not tabulated in this paper, the Gnu g77 compiler produced code which ran faster than the code produced by the Absoft f77 compiler. However, the results were just a few percent different, rather than on the scale of the differences shown by the PO code. All results shown here are from the Gnu g77 compiler.

Table 8 shows results on various machines and various numbers of processors in units of CPU seconds per simulated time step. Complete simulations might require hundreds to hundreds of thousands time steps, and the results can be scaled accordingly, if complete simulation times are desired. Results are shown broken into computation and communication times, where communication includes send, receive, and buffer copy times.

Number of Processors	*Beowulf*	*Cray T3D*	*Cray T3E-600*
1	2.44 - 0.0	2.71 - 0.0	0.851 - 0.0
4	2.46 - 0.12	2.79 - 0.026	0.859 - 0.019
16	2.46 - 0.28	2.79 - 0.024	0.859 - 0.051
64	2.46 - 0.51	2.74 - 0.076	0.859 - 0.052

Table 8. Timing results (in computation - communication CPU seconds per time step) for FDTD code, for fixed problem size per processor of 69×69×76 cells.

It is clear that the Beowulf and T3D computation times are comparable, while the T3E times are about 3 times faster. This is reasonable, given the relative clock rates (200, 150, and 300 MHz) and peak performances (200, 150, 600 MFLOP/s) of the CPUs. As with the PO code, the T3D attains the highest fraction of peak performance, the higher clock rate of the Beowulf gives it a slightly better performance than the T3D, and the T3E obtains about the same fraction of peak performance as the Beowulf. As this code has much more communication that the PO code, there is a clear difference of an order of magnitude between the communication times on the Beowulf and the T3D and T3E. However, since this is still a relatively small amount of communication as compared with the amount of computation, it doesn't really effect the overall results.

The communications portion of the results from the Beowulf runs deserve further discussion. The choice of communication network to use on Naegling was always difficult. For any Beowulf-class machine, the most general choice of network is a large switch that provides good latency and bandwidth between any pair of ports, while providing full backplane bandwidth between all the ports. Observing the current marketplace, is appears that this is hard to build (at a reasonable cost) for large numbers of ports. Today, it is not clear that anyone has succeeded for a machine of Naegling's size (100-150 nodes at various times).

The current network on Naegling breaks the ports into groups of 20. The hardware for the first two groups were recently upgraded, and the hardware for the remaining groups will be upgraded in coming days. These changes in hardware have created a situation where the communication times are not constant from one day to the next, and additionally, for large runs, there can be a variation from one run to the next on the same day. Multiple runs on 4 and 16 nodes produced communication times that varied only within a few percent, while three successive 64-node runs that were most recently performed produced communication times of 0.67, 0.55, and 0.31 (seconds per time step.) These numbers were averaged to obtain the 0.51 used in Table 8, though it could be argued that once the hardware problems are solved, 0.31 (or lower) should be produced consistently. (Note: the current problems appear to be in bandwidth, only; latencies both are acceptable and have not been observed to vary more than a few percent.)

1.6 CONCLUSIONS

This paper has shown that for both parallel calculation of the radiation integral and parallel finite-difference time-domain calculations, a Beowulf-class computer provides slightly better performance that a Cray T3D, at a much lower cost. The limited amount of communication in the physical optics code defines it as being in the heart of the regime in which Beowulf-class computing is appropriate, and thus it makes a good test code for an examination of code performance and scaling, as well as an examination of compiler options and other optimizations. The FDTD code contains more communication, but the amount is still fairly small when compared with the amount of computation, and this code is a good example of domain decomposition PDE solvers. (The timing results from this code show trends that are very similar to the results of other domain decomposition PDE solvers that have been examined at JPL.)

An interesting observation is that for Beowulf-class computing, using commodity hardware, the user also must be concerned with commodity software, including compilers. As compared with the T3D, where Cray supplies and updates the best compiler it has available, the Beowulf system has many compilers available from various vendors, and it is not clear that any one always produces better code than the others. In addition to the compilers used in this paper, at least one other exists (to which the authors did not have good access.) The various compilers also accept various extensions to FORTRAN, which may make compilation of any given code difficult or impossible without re-writing on some of it, unless of course the code was written strictly in standard FORTRAN 77 (or FORTRAN 90), which seems to be extremely uncommon.

It is also interesting to notice that the use hand-optimizations produces indeterminate results in the final run times, again depending on which compiler and which machine is used. Specific compiler optimization flags have not been discussed in this paper, but the set of flags that was used in each case were those that produced the fastest running code, and in most but not all cases, various compiler flag options produced greater variation in run times that any hand optimizations. The implication of this is that the user should try to be certain there are no gross inefficiencies in the code to be compiled, and that it is more important to choose the correct compiler and compiler flags. This is not a good situation.

The choice of communication network for a large Beowulf is certainly not obvious. Current products in the marketplace have demonstrated scalable latencies, but not scalable bandwidths. However, this may be changing, as seems to be demonstrated by the new portions of Naegling's network.

Overall, this paper has validated the choice of a Beowulf-class computer for both the physical optics application (and other similar low-communication applications) as well as for the finite-difference time-domain application (and other domain decomposition PDE solvers). It has examined performance of these codes in terms of comparison with the Cray T3D and T3E, scaling, and compiler issues, and pointed out some "features" of which users of Beowulf-systems should be aware.

ACKNOWLEDGMENTS

The authors would like to acknowledge helpful conversations with John Salmon and Jan Lindheim at Caltech, as well as the contribution of Bill Imbriale at JPL in developing the original POPO code studied here. The FDTD code studied here includes large contributions made by Allen Taflove at Northwestern University.

The work described was performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration. The Cray T3D supercomputer used in this investigation was provided by funding from the NASA Offices of Earth Science, Aeronautics, and Space Science. Part of the research reported here was performed using the Beowulf system operated by the Center for Advanced Computing Research at Caltech; access to this facility was provided by Caltech. Access to the Cray T3E-600 was provided by the Earth and Space Science (ESS) component of the NASA High Performance Computing and Communication (HPCC) program.

REFERENCES

Berenger, J-P. (1994). A perfectly matched layer for the absorption of electromagnetic waves. *J. Comp. Physics* 114:185-200.

Giest, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM: A Users' Guide and Tutorial for Networked and Parallel Computing*, The MIT Press, Cambridge, Mass.

Husain, K., Parker, T., et al. (1996). *Red Hat Linux Unleashed*, Sams Publishing, Indianapolis, Ind.

Imbriale, W. A. and Cwik, T. (1994). A simple physical optics algorithm perfect for parallel computing architecture. In *10th Annual Review of Progress in Appl. Comp. Electromag.*:434-441, Monterey, Cal.

Imbriale, W. A. and Hodges, R. (1991). Linear phase approximation in the triangular facet near-field physical optics computer program. *Appl. Comp. Electromag. Soc. J.*, 6:74-85.

Katz, D. S., Cwik, T., Kwan, B. H., Lou, J. Z., Springer, P. L., Sterling, T. L., and Wang, P. (1998). An assessment of a Beowulf system for a wide class of analysis and design software. To appear in *Advances in Engineering Software* 29.

Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W. and Dongarra, J. (1996). *MPI: The Complete Reference*. The MIT Press, Cambridge, Mass.

Taflove, A. (1995). *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, Artech House, Norwood, Mass.