PROFILE-BASED CHARACTERIZATION AND TUNING FOR SUBSURFACE SENSING AND IMAGING APPLICATIONS

M. Ashouei[1], D. Jiang[1], W. Meleis[1], D. Kaeli[1], M. El-Shenawee[2], E. Mizan[1]
Y. Wang[1], C. Rappaport[1] and C. Dimarzio[1]

Department of Electrical and Computer Engineering[1]
Northeastern University
Boston, MA 02115
meleis,kaeli,rappaport,dimarzio@ece.neu.edu

Department of Electrical Engineering[2]
University of Arkansas
Fayetteville, AR 72701
magda@uark.edu

Abstract: To address a number of the barriers present in subsurface sensing and imaging (SSI) applications, parallel computers can be effectively used. Parallel and multi-processor shared memory processing can allow experiments to begin to use more sophisticated image restoration and processing algorithms, and can allow scientists and engineers to work with higher resolution images. Working closely with SSI researchers, we have developed a number of techniques that allow us to quickly characterize an application's serial execution. Our goal is to efficiently map large applications and data sets on to both Beowulf-class clusters and shared memory multiprocessor systems. In this paper we present two example SSI applications to demonstrate the power of our techniques. We have selected to parallelize a Monte Carlo Simulation of Acousto-Photonic Imaging (API) and the Steepest Descent, Fast Multipole Method (SDFMM). We have been able to obtain a 52 times speedup over the original Matlab code on a 16-node cluster. For SDFMM, an overall speedup of 7.2 has been achieved on the 32-processor Beowulf cluster and a significant reduced runtime is achieved on the 4-processor 667MHz Alpha shared-memory system.

Keywords: Profiling, Parallelization, Clusters, SMPs, Subsurface Sensing Applications

1 INTRODUCTION

In the Center for Subsurface Sensing and Imaging Systems, researchers use a diverse set of signal processing, electromagnetics, acoustics and optics techniques and algorithms to apply to applications in the field of subsurface imaging [CenSSIS, 2002]. The goal of many of these techniques is to understand the interaction of different types of waves, separating the object of interest from the imaging medium.
One of the objectives of the Center is to solve problems using a common set of computational techniques, including Finite Difference Frequency Domain (FDFD), Finite Difference Time Domain (FDTD), Semi-Analytic Mode Matching (SAMM), and SDFMM; the computation associated with these approaches tends to be very time consuming. The size and complexity of these algorithms typically present a computational barrier to a researcher's ability to obtain results within a few days on current computing platforms.

In practice, large realistic input data sets often cannot be processed at all on a single processor. Parallel computers and parallel I/O can be used to significantly reduce the runtime of these applications. Academic and industrial research groups are actively developing techniques to parallelize scientific applications written in Matlab [Almasi and Padua, 2002],

[RTExpress, 2002], [Quinn et al, 1998]. These techniques include packages based on message passing within Matlab sessions, calls to parallel libraries, and compiler environments that directly parallelize Matlab. However, the relative effectiveness of these techniques is not currently well understood [Moler, 1995]; they differ in the amount of extra programming effort they require and their resulting performance gains [Quinn et al, 1998].

Our approach is to instead develop tools that first profile the serial application to allow us to identify the most critical portions of the execution in a program, as well as to identify program paths. One goal is to limit the amount of message passing by carefully partitioning the work to be done in the application. Our techniques are designed to address a range of applications and programming environments. We utilize a number of standardized Unix and Matlab utilities in our work. We will describe how we first profile an application, identifying the critical control and data flow present in the application. We then discuss how we identify *hot paths* and utilize this data to guide parallelization. While our present implementation performs offline optimization and parallelization, we can envision our profiling techniques to be integrated into a runtime system that monitors performance and performs optimization based on runtime dynamics.

This paper is organized as followed. In Section 2, we will describe our profile-based approach to program optimization. In Sections 3 and 4 we describe the application of this approach to both the Acousto-Photonic Imaging and the Steepest Descent Fast Multipole Method application, and report on the level of speedup obtained. In Section 5 we summarize the contributions of this paper.

2 PROFILE-GUIDED OPTIMIZATION

Profiling has been used to guide a number compilation and program restructuring algorithms. Some examples of the classes of optimizations that have been reported include:

- Code and data profiling [Kalamatianos et al, 1999], [Mowry and Luk, 2000],
- Instruction scheduling and function inlining [Conte et al, 1996], [Merten et al, 1999], and
- Distributed and parallel processing [Blume et al, 1996], [Harzallah and Sevcik, 1995], [Miller et al, 1995].

In this work we describe techniques that profile a serial implementation of the program. Our objective is to provide guidance when parallelizing the program, focusing our attention on portions of the program that dominate overall performance.

The two forms of program execution information are contained in control flow and data flow graphs. In this work we are focused on understanding program control flow, though we have also looked at data flow to determine the amount of data dependence in an application. Next we discuss program control flow as it relates to mapping a serial application onto a MPI programming model.

2.1 Function Call Graphs

The control flow of a program determines the ordering and frequency of execution of the various functions included in the application. It is very important to understand this behavior

since it quickly helps us to effectively parallelize an application.

A function call graph is a directed graph where each node represents a function/procedure in the code, and edges indicate a call to that function. Figure 1 shows a call graph for a program that contains 6 functions **A**-**F**. The edges in the graph represent that the function at the tail of the edge calls the function at the head of the edge. The label on the edge indicates the calling frequency (number of calls executed). Note that returns do not increment the edge weight.

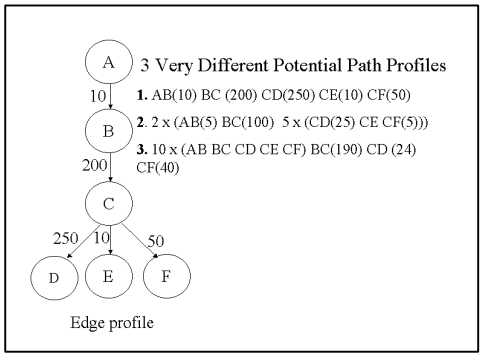


Figure 1: Function call graph. This example shows that there are a large number of potential paths through the function call graph.}

Figure 1 also shows 3 possible execution paths that could have been followed during the execution of the program that produced the edge profile. The first path profile indicates that function **A** called function **B** 10 times, and on the last call, function **B** called function **C** 200 times. On the last call to function **C**, function **D** is called 250 times, followed by function **E** being called 10 times, and finally function **F** being called 50 times.

In the second path profile, we follow a different set of paths through the program, with function **A** calling function **B** five times, followed by function **B** calling function **C** 100 times. This sequence repeats once more. On the last call to **C**, function **D** is called 25 times, function **E** is called once, function **F** is called 10 times, and the sequence is repeated 5 times. The third path profile produces yet another unique sequence of paths that are followed.

To efficiently parallelize a serial application, it is helpful to know both the execution frequency of each program component (functions in this case), as well as understand the order of execution of these components.

Path profiles have been shown to be an effective mechanism for capturing basic block path execution [Ball and Larus, 1994] [Ball and Larus, 1996]. In this work we are studying program dynamics using a coarser grain (functions). To obtain accurate path profiles of function execution, we would need to identify the call locations in the basic block graph. Then we could utilize the techniques described in [Ball and Larus, 1996] to capture full path profiles of function execution.

An additional problem occurs when studying applications that implement indirect function calls. From a single calling location in a program we can invoke a number of functions (e.g., indirect method calls in C++). This makes analysis with path profiling techniques more complicated. Control can be passed conditionally to more than two locations. Indirect branches were not considered in [Ball and Larus, 1996]. We are presently studying this issue, and have already described a highly accurate methodology to estimate the temporal locality of program units using a Temporal Relationship Graph [Kalamatianos et al, 1999].

Another important criterion for parallelizing an application is to focus our attention on the functions that dominate the overall execution. We will refer to these functions as *hot functions*. We can utilize standard Unix utilities, such as *gprof* [Graham, 1982], to obtain sampled function runtimes. We can then quickly identify the portions of an application where a majority of the execution time is spent.

Most time-based profiling applications utilize a timer interrupt to provide a timestamp and location where the program is currently executing. The granularity of the profile is individual functions, and compiler support is needed to provide debug and symbol information in the executable binary.

In Figure 2 we show a portion of a labeled profile directed graph generated for SDFMM. In this graph, nodes represent a program function, and directed edges represent a caller-callee relationship. Edges are labeled with the number of calls made from the tail function to the head function. Each node is labeled with the name of the function, the amount of time spent in each function, as well the time spent in child nodes of this function. Not all child nodes are shown in the graph in this example. Hot nodes (functions that consume a significant portion of the overall execution time) are shaded in gray.

3 EXAMPLE SUBSURFACE APPLICATION

Next we will present examples of profile-guided parallelization using the API and SDFMM applications.

3.1 Acousto-Photonic Imaging

Acousto-Photonic Imaging is a new frequency domain technique for non-invasive medical imaging combining Diffusive Optical Tomography (DOT) and focused ultrasound [Dimarzio and Gaudette, 1998]. The objective is to generate acoustically virtual sources to improve spatial resolution as well as to acquire the optical properties of human tissue. We are attempting to accelerate Monte Carlo simulations for the interaction of Near-InfraRed light (NIR) and ultrasound in dense turbid media with high albedo. Figure 3 shows a pseduocode description of the processing involved with API.

The original serial code is written in Matlab. We are using Matlab versions 5.3 and 6.1 in this work. Our target system is a 192-node Silicon Graphics Origin 2000 system at the Boston University Scientific and Visualization Center, which is part of the NCSA. We only utilize up to 16 nodes in this work. We begin by profiling the application using the built-in Matlab profiler and found that more than 90% of the program execution is spent in two inner loops of the program.

The next step in our approach to parallelize this Matlab application is to compile the code to C code using the Matlab compiler.

The speedup obtained from performing this step ranges form 8-28 times, and depends upon on the problem size chosen (the larger the problem size, the larger the speedup). This speedup is a result of the fact that compiled code is much faster than interpreted code, especially when the application's execution spends most of its time in loops. Table 1 shows that as problem size increases, so does the degree of speedup.

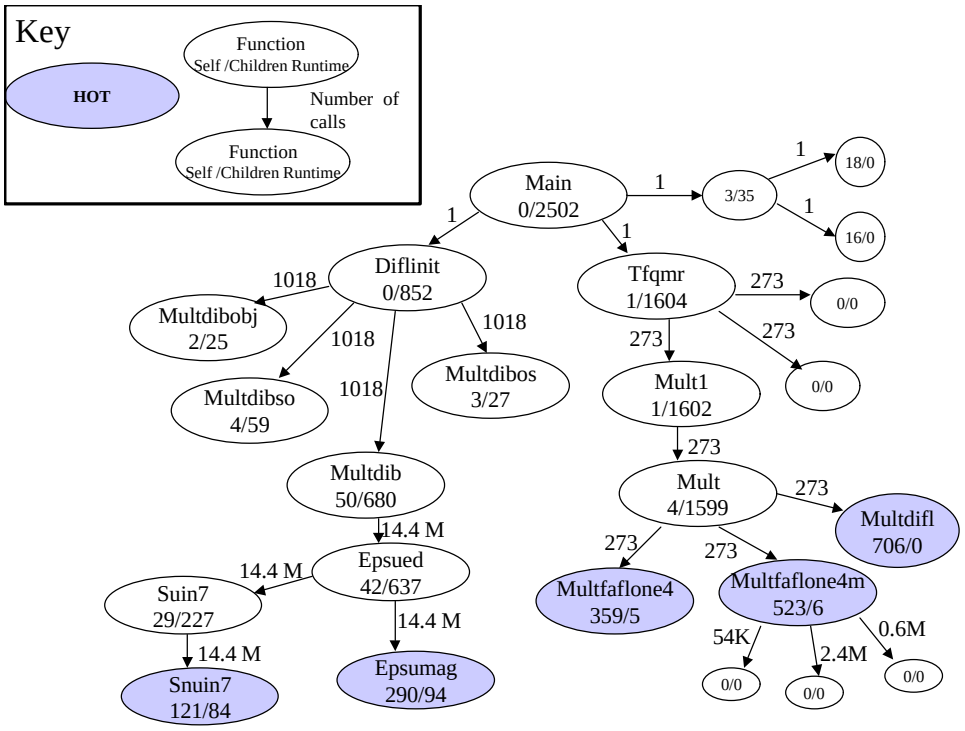


Figure 2: Profile graph for SDFMM. Example contains only a subset of the complete program graph. Each node is labeled with the name of the function, as well the amount of time spent in each function, as well as in child nodes of this function.

```
Program MonteCarloAPI
  Program Initialization
  for (i = 1 to n )
     /* perform computations */
     Call to intersect function;
     for ( j = 1 to upper-limit1)
        /* some computations contain loop-carried
          dependencies */
     end for;
     for (k = 1 to upper-limit2)
          /*  some  computations  that  contain  loop-
          carried dependencies  */
     end for;
  end for;
```

Figure 3: Pseudocode for the Monte Carlo Acousto-Photonic Imaging code evaluated in this work.

Number of photons	3000	60000	100000
Matlab code execution time (seconds)	1440	24000	28800
C code execution time (seconds)	180	960	1024
Speedup (Matlab vs. C)	8.0	25.0	28.1

Table 1: Execution time of the Matlab and C versions of the serial code for different problem sizes.

Profiling the C code with *gprof* further indicates that two inner loops consume about 50% of the total execution time.

The other time consuming portion of the application is in the Matlab *intersect* function. After converting the *intersect* Matlab code to C, we found we have reduced the function's execution time by 75%. Next, we will try to further reduce the execution time of these codes by applying parallelization and algorithm optimization to the two time consuming parts (i.e., the two loops and the *intersect* function).

The execution of the API code spends almost all of its execution time in one outer loop. Other than this loop, the program executes some initialization code, though the execution time for this code is negligible. Loop-level analysis for the outer loop has been done in order to determine dependences across iterations of the loop. The analysis shows that there is *loop-carried dependence* between different iterations of the loop. Data values that are written at some earlier point in the execution and are read at some later point in the execution.

If there is no loop-carried dependence, then the loop can be easily parallelized. It is also possible to parallelize a loop containing a loop-carried dependence as long as the dependence is not *circular*. A loop-carried dependence is called circular if there is a value which depends on previous iteration(s) of itself or there are at least two statements S_1 and S_2 which depend on one another. If the loop-carried dependence is not circular, the loop can be transformed in such a way that its different iterations can be run in parallel. However, if the loop-carried dependence is indeed circular, the loop cannot be parallelized.

The outer loop in our API code has circular loop-carried dependences, and therefore cannot be parallelized. However, there is no loop-carried dependence in the two inner loops, each of which consumes 25% of the execution time. So our goal is to parallelize the two inner loops that take 50% of the execution time.

To compute the expected speedup of our parallelization we will treat the other 50% of code execution as serial computation. If T_p is the execution time of the code after parallelizing the two inner loops, t is the execution time of the serial component in the parallel code, T_s is the execution time of the serial code, and p is the number of processors, then based on Amdahl's law we have:

$$T_p = ((T_s - t) / p) + t$$

Therefore:

$$Speedup = T_s/T_p = T_s/(((T_s - t)/p) + t)$$

Here $t = 0.5 * T_s$, therefore

$$Speedup = (2 * p) / (1 + p)$$

p (the number of processors)	1	2	4	8	16
Execution time of parallelized code	1045	970	651	587	534
Speed-up vs. C code	1	1.1	1.6	1.8	2.0
Theoretical Speedup ($2p$ / (p+1))	1	1.3	1.6	1.8	1.9
Speed-up vs. Matlab code	28.0	30.8	44.8	49.6	52.6

Table 2: Execution time after parallelizing the two inner loops.

The results of parallelizing the two inner loops, the expected speedup by parallelizing the two loops, and the speedup over the Matlab codes, are shown in Table 2.

One way to parallelize the two inner loops is to use a master-slave paradigm [Shao and Wolski, 2000]. A master processor executes the sequential portions of the code, and then distributes the necessary data to a number of slave processors to execute the parallel workload. Next, the master gathers the results from slaves and assigns them new tasks if any exist. When using a master-slave model, the master must compute both the initialization and any sequential execution during each iteration of the outer loop. Then the master disseminates the data necessary to perform the inner loops in parallel to the slaves, and then gathers data from all slaves and starts the next iteration of the outer loop. In parallelizing each of the two inner loops, we divide the computation into tasks, where each task is defined as consecutive iterations of the loop. Therefore, task generation is performed statically. Also the assignment of tasks to slave processors is done statically, based on the processor ID in the following way:

- *Assume p is the number of processors and n is the number of iteration of a loop.*
- *Furthermore, assume that n is divisible by p.*
- *Then processor i will compute iterations (i-1) * (n/p) to (i * n/p) - 1, inclusive.*

Another possible approach to parallelizing the code is to allow all processors to participate in the initialization phase as well as to execute the sequential portion of the code in each outer iteration, instead of waiting idly while the master completes them. Also each processor performs its assigned parallel execution (which is statically defined, based on processor ID, as described in the previous approach). The motivation behind this approach is to reduce communication overhead by replicating computations. The execution times shown in Table 2 are based on using this approach of parallelizing the two inner loops.

3.2 A More Efficient *intersect* Function

As mentioned earlier, the calls to the *intersect* function take 25% of the total execution time in the C version of the application. Our next goal is to make the *intersect* function run faster by both parallelization and algorithm optimization. The Matlab *intersect(A, B)* (where *A* and *B* are vectors) returns the values common to both *A* and *B*. The resulting vector is sorted in ascending order.

To find out where time is spent in the *intersect* function, we have run the following code:

```
A = random ('Normal',
0, 1, 35000,1);
B = random ('Normal',
0, 1, 35000, 1);
profile on
C = intersect(A, B);
profile report
```

The example above takes approximately 1.5 seconds, with more than 90% of the time spent in the sort function using Matlab v5.3.1. In Matlab version 6.1, the example takes 0.31 seconds, with 45-50% of the time spent in the sort function. Based on the information obtained from The MathWorks, there have been substantial changes to the built-in *sort* function from version 5.3.1 to 6.1.

To speedup *intersect*, we wrote a new function which has the same functionality and then parallelized it. Figure 4 shows the pseudocode of the algorithm for the new *intersect* that is implemented.

3.3 Steepest Descent Fast Multipole Method

The Steepest Descent Fast Multipole Method (SDFMM) was developed at the University of Illinois at Urbana Champaign to analyze large-scale three dimension (3-D) scattering problems [El-Shenawee et al, 1999], [Jandhyala, 1998], [Jandhyala et al, 1998]. We have recently modified the base algorithm to be used in modeling subsurface sensing applications. In the version of the program we are evaluating in this paper, the application looks at scattering from a penetrable spheroid object buried under a two-dimensional randomly rough ground surface [El-Shenawee et al, 2001a] [El-Shenawee, 2001b]. The spheroid represents a buried land mind, resting in a bed of soil.

```
Function Intersect(Vector A, Vector B)
   Sort(A);
   Sort(B);
   aptr = bptr = cptr = 0;
   while ((aptr < size(A)) and (bptr < size(B))
    do {
     if (A[aptr] = B[aptr])
        {
         C[cptr++] = A[aptr++];
         bptr ++;
        }
      else
        {
         while (A[aptr] < B[bptr])
         do {
               aptr ++;
               while (B[bptr] < A[aptr])
                   bptr++;
          }
        }
```

Figure 4: Pseudocode for an improved *intersect* function.

The SDFMM has computational complexity for CPU time and memory equal to only *O(N)* per iteration versus $O(N^2)$ for the Method of Moments, where *N* is the total number of the unknowns~\cite{Jandhyala98}. This level of complexity provides SDFMM with a significant advantage over several other techniques and has enabled us to utilize Monte Carlo simulation methods [El-Shenawee et al, 2001a]. While SDFMM provides us with an efficient subsurface modeling tool, to process large images at high resolution, the time to compute a single result can be many days, so we need to exploit parallel computers. In our parallel cluster work we used the MPI library for the parallel implementation of the SDFMM code [MPI, 1994], [Li et al, 2000], [Velamparambil et al, 1999], [Walker, 1994], [Walker and Dongarra, 1996].

4 PARALLELIZATION OF SDFMM

The SDFMM is used to solve the linear system of equations given by [Jandhyala, 1998]:

$$\mathbf{Z}\,\mathbf{I} = \mathbf{V} \qquad (3.1)$$

where $\mathbf{Z}$ is the impedance matrix, $\mathbf{I}$ is the vector of unknown coefficients of the electric and magnetic surface currents and $\mathbf{V}$ is associated with the incident waves on the rough ground surface. The matrix $\mathbf{Z}$, which is filled in by Method of Moments formulations, becomes sparse with SDFMM and the system of equations in (1a) can be written as:

$$\mathbf{Z'}\,\mathbf{I} + \mathbf{Z''}\,\mathbf{I} = \mathbf{V} \qquad (3.2)$$

The sparse matrix $\mathbf{Z'}$ has its non-zero elements calculated and stored using the conventional *MoM*, which are then multiplied by the vector (near field interactions), while the matrix-vector multiply $\mathbf{Z''I}$ is computed in one step without calculating or storing any elements of the matrix $\mathbf{Z''}$. This is achieved by using the FMM hybridized with the SDP integration rule.

By inspecting Figure 2, we can clearly identify the portions of the code where time is being spent. From using our profile-guided approach we are able to identify three bottlenecks in the SDFMM code that can benefit from parallelization:

(i.) the functions that calculates the elements of the sparse matrix $\mathbf{Z'}$;
(ii.) the functions that execute the matrix vector multiplication $\mathbf{Z''}\,\mathbf{I}$ in each iteration of the solver;
(iii.) and the functions that execute the fast multipole method for $\mathbf{Z''}\,\mathbf{I}$ (far field interactions).

These three bottlenecks correspond to the following functions identified in Figure 2:

- `Dflinit` for producing the sparce matrix $\mathbf{Z'}$,
- `Multdifl` for computing the matrix vector multiply, and
- `Multfaflone` and `Multfaflone4m` for computing the Fast Multipole Method (the amount of time spent in this code depends up a user-specified threshold value).

The source code has been parallelized using MPI routine, attempting to exploit the underlying available data parallelism. The key data structure in function (i) is the sparse matrix $\mathbf{Z'}$, which is stored as blocks of nonzero elements. These blocks are distributed among all processors, and no additional communication is needed.

When this routine is parallelized we achieved near-linear speedups on 32 processors. In the matrix-vector multiplication **Z" I**, the computation is parallelized by distributing **I** to all processors in each iteration. The resulting vector components produced by the multiplication are then distributed to all processors. For bottleneck (iii), the two functions involved with computing the far field interactions consist of a series of loops with complex interdependences. Each loop is separately parallelized, with *collective communication* used to distribute the results to all processors after executing each function. In addition, these two functions are executed in parallel, followed by subsequent distribution of the results to all processors. Load balancing between these two functions is achieved using a detailed performance model based on the serial execution time of each routine, the time required for collective communication operations, and the amount of communication overhead needed. The structure of the parallelized SDFMM application is shown in Figure 6.

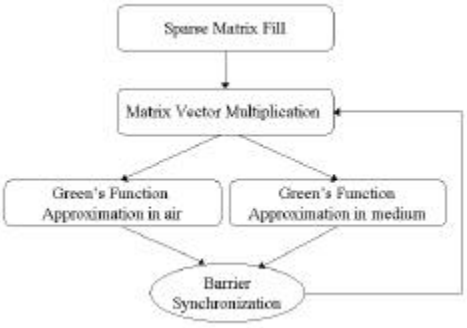


Figure 6: Structure of the parallelized SDFMM application.

4.1 Evaluation Environment

We evaluated the parallel implementation of the SDFMM computer code on a 32-node Intel Pentium-based Beowulf cluster. Thirty-one nodes of the Beowulf cluster are 350MHz Intel Pentium IIs with 256 MB of RAM and one node is a 4x450MHz Intel Pentium II Xeon shared memory processor with 2GB of RAM. The nodes are connected to a 100 BaseTX Ethernet network and they use the SuSE 6.1 operating system with Linux kernel 2.2.13, and the MPICH 1.2.1 implementation of the MPI library. We also tested the parallelized code on a 4-node shared memory Compaq Alpha-based workstation (667Mhz Alpha 21264) of 16GB total RAM. The processor uses the UNIX OSF/1 V5.1 operating system with the MPICH 1.1.2 MPI library. Our benchmark includes three small-scale cases executed on the 256MB Intel cluster, and in addition one medium-scale case that is executed on the Alpha workstation. All results obtained by executing the parallel version of the code are validated with those computed by the serial version of the code [El-Shenawee et al, 2001a], [El-Shenawee et al, 2001b].

4.2 Evaluation Parameters

The scattering problem configurations used in [El-Shenawee et al, 2001a} are employed here, but for only one rough surface realization. The rough ground (characterized by Gaussian statistics with zero mean for the height) is described by the rms height σ and the correlation length (l_c). In all cases, the relative dielectric constant of the ground soil (dry sand) and the penetrable buried object (TNT in a land mine) are $\varepsilon_r = 2.5 - j0.18$ and $\varepsilon_r = 2.0 - j0.0092$, respectively, and the ground correlation length is $l_c = 0.5\lambda_0$.

Case	Number of unknowns	σ	Object	System	Number of Processors	Seral/Parallel Time	Speedup
1	8800	0.3λ	None	Cluster	32	99/14	7.1
2	8800	0.1λ	Sphere	Cluster	32	90/14	6.2
3	8800	0.04λ	Spheroid	Cluster	32	88/12	7.2
4	60,32	0.04λ	Spheroid	Alpha SMP	32	96/37	2.5

Table 3: Runtime parameters and runtime performance data.

A Gaussian beam with horizontal polarization is employed for the incident waves [El-Shenawee et al, 2001a]. In Case 2, the buried sphere has radius of a = $0.16\lambda_0$ with burial depth equal to z = $-0.32\lambda_0$ measured from its center to the mean plane of the ground while in Case 3 and 4 the buried spheroid has dimensions a = $3\lambda_0$ and b = $0.15\lambda_0$, and is buried at z = $-0.3\lambda_0$. The ground dimensions are $3\lambda_0$ by $3\lambda_0$ in Cases 1-3, and $8\lambda_0$ by $8\lambda_0$ in Case 4. Table I summarizes the parameters and output results for Cases 1-4. The speedup of a parallelized application is defined as the ratio of the serial runtime divided by the parallel runtime. In Figure 6, the overall speedup and the speedup for the initialization routine (filling matrix **Z'**) are plotted versus the number of processors for Case 1. The speedup curves for Cases 2 and 3 (not shown) are similar, with slightly different peak values of 6.2 and 7.2, respectively. The results show a significant speedup in the initialization code that fills the sparse matrix **Z"**. This initialization speedup dramatically affects the overall speedup of the code, as shown in Figure 6.

In each case, the peak overall speedup is observed when running on 32

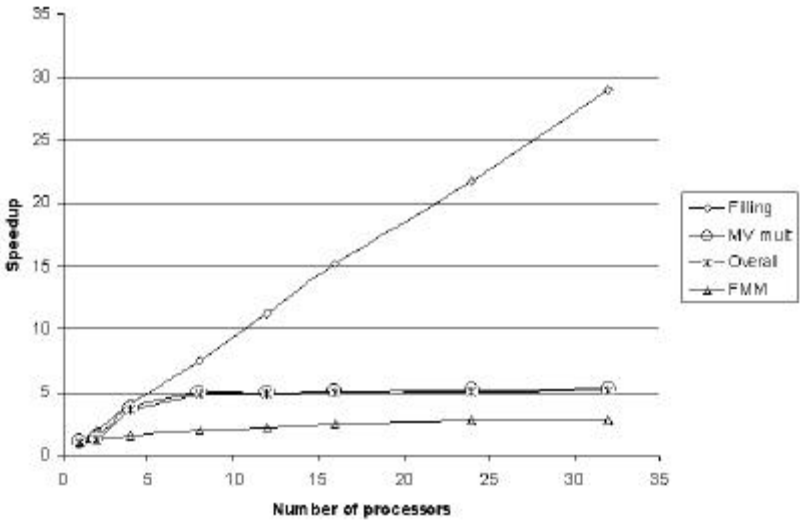


Figure 6: Speedup of the parallelized SDFMM code versus the number of processors for the three separate bottlenecks in the code on the Beowulf cluster for Case 1. The overall speedup is also shown.

processors, but most of this speedup is achieved using only 12 processors. The efficiency of an application for a given number of processors is defined as the ratio of the speedup to the number of processors. Over Cases 1-3, the average speedup on 32 processors is 6.8, giving an efficiency of 0.21. Based on the serial runtimes, 88% of the code is executed in parallel. Therefore by Amdahl's Law [Amdahl, 1967], the peak speedup achievable for the current parallelization of the code is 8.3. We conclude that communication overhead and load imbalance among the processors accounts for the reduction in speedup from 8.3 to 6.8.

A comparison between the speedups achieved in the other bottlenecks (i)-(iii) mentioned in previously is also shown in Figure 6. These results demonstrate that the overall speedup is almost the same as that achieved in the matrix-vector multiplication **Z" I**, which is the bottleneck in (ii).

In the second set of experiments, we solved the medium-scale problem of Case 4 (60,320 unknowns) on an Alpha SMP using all four processors. The overall speedup in this case is 2.5, which is close to the predicted peak speedup of 2.9. This implies that executing the parallel code on the 4-Alpha 667 MHz processors gives an impressive absolute runtime for this medium-scale case. The serial version of the code requires 950MB of memory, while the parallel version requires 1154MB of memory distributed over four processors (288, 290, 289 and 287MB each). The results of the parallel solution were identical to those of the serial implementation presented in [El-Shenawee et al, 2001a].
The results described in this section demonstrate that by exploiting fine-grained parallelism within a single surface realization (one run of the code), we have achieved significant speedups. However, when the number of rough surface realizations is much larger than the number of available processors, as with Monte Carlo simulations, larger speedups are possible. This situation occurs when we need to run Monte Carlo simulations [El-Shenawee et al, 2001a]. In this case we assign a group of these realizations (runs of the code) to be executed in parallel on each processor. Since the computations are independent and little communication is needed, this coarse-grained parallelism gives a perfect speedup that is only limited by the number of available processors. In other subsurface scattering configurations, we may need to obtain multiple views of a target buried under the same rough surface realization [El-Shenawee et al, 2001], which requires running the code only a few times. A combination of fine and coarse-grained parallelism can make efficient use of all available processors.

5 SUMMARY

In this paper we have discussed how informed profiling can guide the efficient parallelization of subsurface sensing and imaging application. We discussed how we utilized the profiling capabilities within Matlab, as well as compilation to C, to obtain a 52.6 speedup over a serial Matlab implementation of a Monte Carlo Simulation running Acousto-Photonic Imaging. For SDFMM, a speedup of 7.2 was achieved on the 32-processor Beowulf cluster and a dramatic reduction in runtime is obtained using a 4-processor Alpha shared-memory machine. The greatest potential for speedup occurs in the sparse matrix filling and far field interaction steps.

ACKNOWLEDGEMENTS

The authors would like to thank W. Chew and E. Michielssen for access to the SDFMM Fortran code. This research was sponsored by the ERC Program of the NSF under award number EEC-9986821, in part by an NSF MRI grant MRI-9871022, by an ARO MURI grant DAA 0-55-97-0013, and in part by the College of Engineering at the University of Arkansas.

REFERENCES

Almasi G. and Padua D., "MaJIC: Compiling MATLAB for Speed and Responsiveness," *Proceedings of PLDI*, to appear, 2002.

Amdahl G.M., "Validity of Single-processor Approach to Achieving Large-scale Computing Capability, *Proceedings of AFIPS Conference*, Reston, VA, pp. 483-485, 1967.

Ball T. and Larus J.R., "Efficient Path Profiling," *Proceedings of MICRO29*, pp. 46-57, 1996.

Ball T. and Larus J.R., 1994, "Optimally Profiling and Tracing *ACM Transactions on Programming Languages and Systems*, 16(4), pp. 1319-1360, July 1994.

Blume W., Doallo R., Eigenmann R., Grout J., Hoeflinger J., Lawrence T., Lee T., Padua D., Paek Y., Pttenger B., Rauchwerger L. and Tu P., "Parallel Programming with Polaris," *IEEE Computer*, 29(12), pp. 78-82, 1996.

CenSSIS, *The Center for Subsurface Sensing and Imaging Systems*, http://www.censsis.neu.edu, Northeastern University, Boston, MA, 2002.

Conte T.M., Patel B.A., Menezes K.N. and Cox J.S., 1996, "Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization," *International Journal of Parallel Programming*, 24(2), pp. 187-206, 1996.

DiMarzio C. and Gaudette T.J., 1998, "Point-Spread Functions for Acousto-Photonic Imaging," *Proceedings of the Conference on Lasers and Electro-Optics (CLEO)*, 1998.

El-Shenawee M., Jandhyala V., Michielssen E. and Chew W.C., "The Steepest Descent Fast Multipole Method (SDFMM) for Solving the Combined Field Integral Equation Pertinent to Rough Surface Scattering," *Proceedings of the IEEE AP-S International Symposium and URSI Radio Science Meeting*, Orlando, FL, pp. 534-537, July 1999.

El-Shenawee M., Rappaport C., Miller E. and Silevitch M., "3-D Subsurface Analysis of Electromagnetic Scattering from Penetrable/PEC Objects Buried Under Rough Surfaces: Use of the Steepest Descent Fast Multipole Method (SDFMM)," *IEEE Transactions on Geoscience and Remote Sensing*, 39(6), pp.1174-1182, June 2001.

El-Shenawee M., Rappaport C. and Silevitch M., "Monte Carlo Simulations of Electromagnetic Wave Scattering from Random Rough Surface with 3-D Penetrable Buried Object: Mine Detection Application Using the SDFMM," *Journal of the Optical Society of America A*, to appear 2002.

9th Heterogeneous Computing Workshop, May, 2000.

Velamparambil S.V., Schutt-Aine E., Nickel J.G., Song J.M. and Chew W.C., "Problems Using a Linux Cluster and Parallel MLFMA," *IEEE Antennas Propagation Symposium*,

Solving Large Scale Electromagnetics, pp. 635-639, July, 1999.

Walker D.W. and Dongarra J.J., "MPI: A Standard Message Passing Interface," *Supercomputing*, 12(1), pp. 56-68, 1996.

Walker D.W., "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers," *Parallel Computing*, 20(4), pp. 657-673, 1994.

Maryam Ashouei is an MS students in the Electrical and Computer Engineering Department at Northeastern University, Boston, MA. In 1998 she received her BSc in Computer Engineering from Sharif University, Tehran, Iran.

Desheng Jiang received his B.S. in Chemistry from Jiangxi Normal University in 1993, M.S in Oceanography from Texas A&M University in 1999, M.S. in Electrical Engineering from Northeastern University in 2001, respectively. He currently is working for Openwave Systems Inc. He can be reached at djiang@ece.neu.edu.

Waleed M. Meleis received the BSE degree in electrical engineering from Princeton University in 1990 and the MS and PhD degrees in computer science and engineering from the University of Michigan in 1992 and 1996, respectively. He is an associate professor in the Department of Electrical and Computer Engineering at Northeastern University, Boston. His research interests are in scheduling algorithms and bounds for modern processors and compilers.

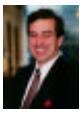
David Kaeli received his B.S. in Electrical Engineering from Rutgers University, his M.S. in Computer Engineering from Syracuse University, and his PhD in Electrical Engineering from Rutgers University. He is currently an Associate Professor on the faculty of the Department of Electrical and Computer Engineering at Northeastern University. Prior to 1993, he spent 12 years at IBM, the last 7 at IBM T.J. Watson Research in Yorktown Heights, N.Y.. In 1996 he received an NSF CAREER Award. He currently directs the Northeastern University Computer Architecture Research Laboratory (NUCAR). He is also a research thrust leader in the NSF-funded CenSSIS Engineering Research Center.

Magda El-Shenawee received the Ph.D. degree in electrical engineering from the University of Nebraska-Lincoln in 1991. In 1992, she worked as a Research Associate in the Center for Electro-Optics at the University of Nebraska. In 1997, she worked as Visiting Scholar at the University of Illinois at Urbana-Champaign and in 1999, she joined the Center for Electromagnetics Research at Northeastern University, Boston. Currently, Dr. El-Shenawee is an assistant professor in the Department of Electrical Engineering at the University of Arkansas, Fayetteville. Her research areas are rough surface scattering, computational electromagnetics, subsurface sensing of buried objects, breast cancer modeling and medical imaging, numerical methods, and micro-strip circuits. Dr. El-Shenawee is a member of Eta Kappa Nu electrical engineering honor society.

Elias Mizan obtained a BS in Computer Engineering and Informatics from the University of Patras, Greece in 2000 and an MS in Electrical and Computer Engineering from Northeastern University, Boston, MA in 2002. His research interests include computer architecture and compiler optimizations. He is currently focusing on the performance analysis

of superscalar and VLIW/EPIC microprocessors by analyzing data gatherered from hardware performance counters.

Yijian Wang is a PhD student in the Electrical and Computer Engineering Department at Northeastern University, Boston, MA. He received his BS in Computer Science from Xian Jiaotong University, China and his MS in Electrical Engineering from the Chinese Academy of Sciences. He worked at CAS on the design of an underwater digital communication system for the Chinese Navy.



Carey M. Rappaport received five degrees from the Massachusetts Institute of Technology: the SB in Mathematics, the SB, SM, and EE in Electrical Engineering in June 1982, and the PhD in Electrical Engineering in June 1987

Prof. Rappaport joined the faculty at Northeastern University in Boston, MA in 1987. He has been Professor of Electrical and Computer Engineering since July 2000. During Fall 1995, he was Visiting Professor of Electrical Engineering at the Electromagnetics Institute of the Technical University of Denmark, Lyngby, as part of the W. Fulbright International Scholar Program. He has consulted for Geo-Centers, Inc., PPG, Inc., and several municipalities on wave propagation and modeling, and microwave heating and safety. He is Principal Investigator of an ARO-sponsored Multidisciplinary University Research Initiative on Demining and Co-Principal Investigator of the NSF sponsored Center for Subsurface Sensing and Imaging Systems (CenSSIS) Engineering Research Center.

Charles DiMarzio holds a BS in Engineering Physics from the University of Maine, an MS in Physics from Worcester Polytechnic Institute, and a Ph.D. in Electrical and Computer Engineering from Northeastern University. He worked on coherent laser radars for atmospheric measurements, at Raytheon Company. He is presently an associate professor of electrical and computer engineering at Northeastern University, where he directs the Optical Science Laboratory. This laboratory is involved in optics research in such diverse fields as landmine detection, medical and biological imaging, and remote sensing.