Performance Optimization of an Integral Equation Code for Jet Engine Scattering on CRAY-C90

Mikhail Smelyanskiy
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

Edward S. Davidson
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

John L. Volakis
Radiation Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

The numerical solution of Maxwell's equations is a computationally intensive task and use of high-performance parallel computing facilities are necessary for the larger class of practical problems in scattering, propagation and antenna modeling. It is therefore necessary to carefully consider algorithm optimizations aimed at improving the code's run time performance on the computing platform employed. Although some performance improvement can be derived from compiler-level optimizations, further speed-up may involve manual effort in algorithm restructuring, data layout , and parallelization. This paper focuses on the manual optimizations used to improve the performance of a moment method code for the analysis of a cylindrically periodic structure, as is the case with a jet engine. We describe the steps taken which resulted in nearly two orders of magnitude improvement over the original version of the code. A 16-processor shared-memory CRAY-C90 vector supercomputer was employed. Our optimization took advantage of SSD its solid-state storage, enabled better loop vectorizations, parallelized the *matrix_fill* routine, and called appropriate CRAY-C90 library routines.

1. Introduction

The application of analytical and numerical techniques in conjunction with advanced architecture and software has allowed more accurate simulations of very large electrical systems. In this paper we consider the parallelization and optimization of a code for evaluating the Radar Cross-Section (RCS). This is a method of moments (MoM) code specifically applied to jet engine scattering and exploit the inherent cylindrical periodicity of the engine geometry to reduce computation. As is the case with all MoM codes, a dense system of equations results from the discretization of the jet engine blades using the usual subsectional surface basis functions [1]. This code solves the dense matrix system via LU decomposition.

Typically, the most dramatic speed-up after code optimization is achieved by concentrating on the matrix fill and solution step (LU decomposition or the inverse FFT employed in the iterative solver) of the code. We therefore concentrate on optimization techniques which emphasize performance improvements for these steps and are aimed at reducing both CPU and wall-clock time.

This paper is organized as follows. Section 2 reviews the general concepts of performance optimization and simple examples are included to demonstrate the crucial role of optimization. Concepts of parallelization are discussed in section 3. Vectorization and parallelization techniques are applied to the MoM codes of interest in section 4 and the resulting speed ups are shown, following a short introduction to vectorization. Further optimizations of the MoM code are mentioned in the conclusion.

2 Optimization Overview

The first step in optimizing a computer algorithm is to evaluate its overall performance. This stage is aimed at identifying the critical sections of the code that consume most of the execution time and various profiling tools may be used to gather related runtime statistics. Figure 1 shows an example of the output from the CRAY-C90 flowtrace profiling tool applied to the unoptimized original version of the MoM code. The routines which require most of the most execution time are

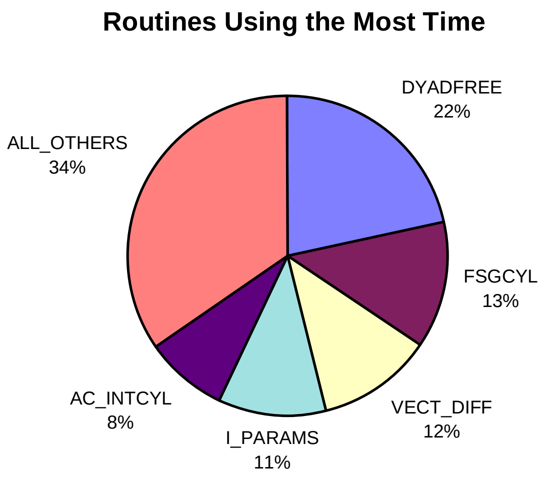

FIGURE 1: Runtime Profile of the most dominant routines

DYADFREE: calculates the free space dyadic Green's function.

FSGCYL: part of the general double surface integration routine called from the Inverse FFT routine.

VECT_DIFF: called mostly from the matrix fill routine to find the centroidal distance between source and observation triangles, to compute the edge lengths of the triangles, etc.

I_PARAMS: used to calculate the geometrical parameters for the closed-form integration routine which computes the potential integrals for uniform and linearly varying surface sources distributed on a planar polygon. This integration routine is part of the matrix fill routine.

AC_INTCYL: part of the general, double surface integration routine called from the matrix fill routine.

Having determined the most dominant sections, requiring the largest fractions of time and/or resources, effective code optimization can begin. Applying an optimization technique may involve the following steps (See Figure 2).

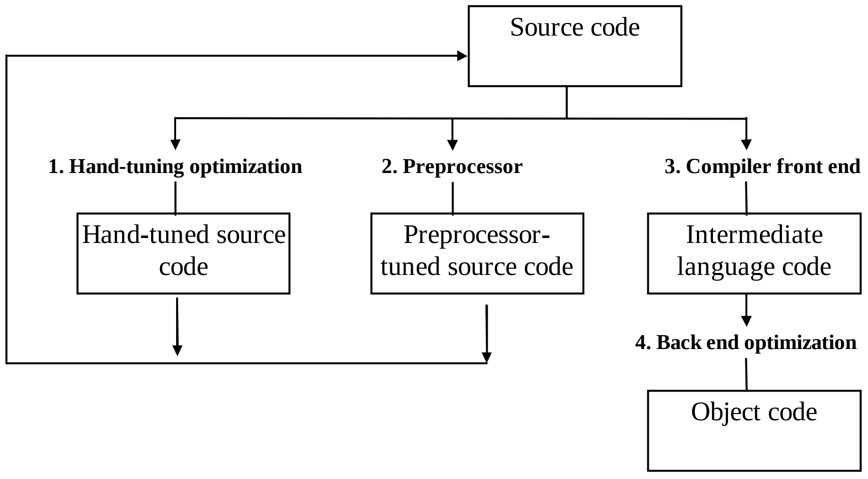

FIGURE 2: The optimization process

At the hand-tuning stage the user performs a number of optimizations at the source-code level. Examples of such optimizations include reordering programming statements or expressions and changing the memory access patterns of loops [2, 3]. Next, an optimizing preprocessor, if available, takes the source code and performs transformations enabled by user-selectable switches. Typical examples are dead code elimination, inlining, interprocedural analysis, library-call generation, etc. The output source code is also optimized to take advantage of architectural features of the host system. Then the output source code is submitted to the compiler whose front end translates it into intermediate language (IL). Its back end optimizer then translates the IL code into machine language and in this process it may apply a wide range of optimizations at the IL level, depending on the user-selectable flag settings which invoke specific sets of optimizations.

The tuning strategy should, of course, be aimed at producing the maximum performance gain with minimum effort. For the example, if we increase the performance of DYADFREE by 20% (see Fig. 1), the program will improve by 4.5% because DYADFREE takes 21.8% of the total time. If we also speed-up FSGCYL by 20%, the overall improvement will be 2.5%. Thus, sections of code that take the highest percentage of CPU time should be investigated first. It is of course important that the numerical results be checked after each optimization step to ensure the correctness and preciseness of the code after the reordering operations.

The sufficiency of the overall program performance is subsequently assessed either intuitively, or more formally, e.g. by using different performance characterization tools which provide bounds on the achievable performance of the code [4, 5]. A performance bound hierarchy model may successively include the effects of machine peak performance, high level application workloads, compiler-inserted overhead, compiler generated instruction schedule, cache effects, etc., and may be applied to loops, procedures, sections or entire codes. If later effects can be reduced or eliminated, performance may be made to approach earlier bounds which represent the potential performance of the application, if the effects of all later levels are eliminated.

Once a program has been sufficiently optimized for a single processor, the next step is to assess whether the application code can take advantage of multi-processor computing platforms. If so, the application is then parallelized and optimized for parallel execution.

2.1 Optimization Examples

It is hard to overstress the importance of optimization. The examples below demonstrate some simple techniques that can significantly improve code performance and speed-up. Details on the employed techniques themselves and further examples may be found in [2, 3].

An Array-Processing Example

Consider the two codes shown in Figure 3 which perform element-wise array multiplication [6]. These codes are clearly seen to be functionally equivalent.

```
do i=1,n
    do j=1,n
        c(i,j)=c(i,j)+a(i,j)*b(i,j)
    enddo
enddo
```

a) stride_n.f

```
do j=1,n
    do i=1,n
        c(i,j)=c(i,j)+a(i,j)*b(i,j)
    enddo
enddo
```

b) stride1.f

FIGURE 3: The Array Multiplication. (a) Original. (b) Loop interchanged

The only difference between the two examples in Figure 3 is that the array elements are referenced in a different order. All runs were made on an IBM RISC System/6000, Model 530 with a 64KB cache. The arrays were all declared REAL*8. A timing loop was inserted around the loops in the examples so that the reported time is the average of 50 million inner loop iterations. Figure 4 shows this time (in microseconds), as a function of n.

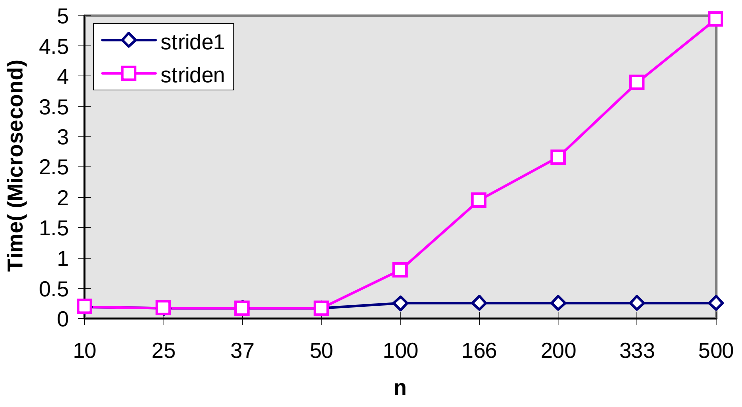

FIGURE 4: Performance on a RISC Sytem6000 Model 530 with 64 KB data cache

As seen in Figure 4, the performance differs significantly between the two codes. For small n, there is little difference in performance, but as *n* grows, **stride1** runs significantly faster then **stride_n**. In FORTRAN, arrays are stored in “column major order”, implying that the leftmost subscript changes more rapidly as memory-adjacent elements are accessed. In the **stride1** routine, successive iterations of the inner loop access array elements that are adjacent in memory. That is the array elements are accessed in the same order as stored in memory.

However, in **stride_n** successive iterations of the inner loop access array elements that are stored in memory *n* entries apart (one array column) in memory. In this case the arrays are said to be accessed with stride *n*. When a single element is read into the processor, adjacent elements (comprising one “cache line”) are automatically brought into the high-speed cache memory along with it. The user has no choice regarding this automatic procedure of cache loading. Clearly, if all entries brought into the cache are soon referenced(as in **stride1**), there is a memory access delay only for the first element in each cache line that the processor reads in. However, if other entries in this line are referenced much later (as in **stride_n**), the line with the referenced entries may get replaced in the cache before they are referenced; referencing an element that is in the cache is called a *cache hit*, otherwise the reference is a *cache miss* and suffers a delay called the *miss penalty*. The advantage of the **stride1** code is that there is roughly one miss per cache line of elements accessed, whereas almost every access to an element is a miss in **stride_n** code – unless *n* is small enough so that entire array fits in cache and remains there indefinitely. This scenario is easily seen in Figure 4, where both **stride1** and **stride_n** versions take the same time to run for $n \le 50$: 3 arrays of $50 \times 50 \times$ 8 bytes = 60 KB < Size of the Cache (64KB). Obviously, an understanding of the machine's cache structure is important in writing code routines that have the best potential for optimum performance.[1]

Example 2: (Matrix Multiplication Example)

As another example, let us consider the three codes below (see Figure 5) which contain different matrix multiplication algorithms. All three codes produce the same end mathematical results. The differences are in the loop index order and the procedure used for the execution of the matrix multiplications. The first two algorithms differ in the loop index order and the third one takes advantage of matrix block multiplication.

[1] Although CRAY has no cache, its memory is divided into multiple banks with cycle time >> 1. Problems similar to those described in *Example 1* occur if GCD(stride, number of banks) $\neq$ 1. GCD is always 1 if stride = 1.

```
do i=1,n
    do k=1,n
        do j=1,n
            c(i,j)=c(i,j)+a(i,k)*b(k,j)
        enddo
    enddo
enddo
```

a) IKJ formulation

```
do j=1,n
    do k=1,n
        do i=1,n
            c(i,j)=c(i,j)+a(i,k)*b(k,j)
        enddo
    enddo
enddo
```

b) JKI formulation

```
do ii=1,n,nb                          !  blocking loop
    do jj=1,n,nb                      !  blocking loop
        do kk=1,n,nb                  !  blocking loop
            do i=ii,ii+nb-1           !  loop within block
                do j=jj,jj+nb-1       !  loop within block
                    s=c(j,i)
                    do k=kk,kk+nb-1   ! loop within block
                        s=s+a(j,k)*b(k,i)
                    enddo
                    c(j,i)=s
            enddo
        enddo
    enddo
enddo
```

c) Block formulation

FIGURE 5: Three Formulations of Matrix Multiplication.

The performance of the three codes in Figure 5 is illustrated in Figure 6 based on runs made on a RISC System 6000 workstation with a 64 KB cache. As can be seen the *IKJ* algorithm exhibits the worst performance because successive iterations of the inner loop access elements of arrays *b* and *c* with stride *n*, resulting in multiple cache misses. Accesses to *b* and *c* are stride 1 in the JKI formulation, resulting in much fewer misses; however for large *n*, the arrays do not fit in cache and some misses still result from the fact that array elements have been replaced in the cache before they are rereferenced. Blocking is a technique for large arrays to reduce cache misses in nested array-processing loops. This is done by processing the data in blocks or strips which are small enough to fit in the cache. The principle behind blocking is that for every array element brought into the cache we wish to perform as many of the computations as possible on that element before it is forced out of the cache by other program actions. The blocking formulation of matrix multiplication algorithm with a blocking factor of 50 has much better performance (for models with a 64KB cache) than the JKI or IKJ formulations for large arrays. We note that the CPU timing for the fourth routine, **esslp2,** shown in Figure 6 refers to measurements of the engineering/scientific library subroutine (**essl**) for performing matrix multiplication. This library has been hand-coded to take maximum advantage of instruction overlap in the machine [6]; details of such proprietary libraries are, however, unavailable to users.

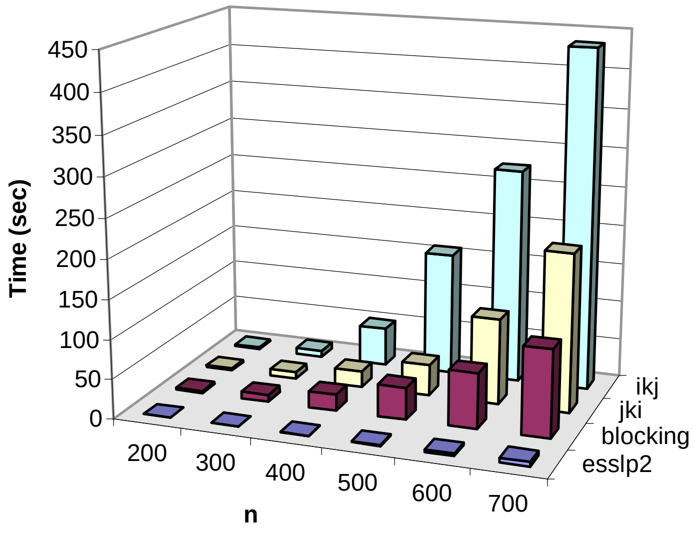

FIGURE 6: Total Run Time of Different Matrix Multiplication Formulations.

3 Basic Concepts of Parallelization

Once a program is sufficiently optimized on single-processors (relative to some goal), the next step is to find if parallelization can be used to speed up the operations/algorithms that consume the most CPU time. Many algorithms are serial on some levels and parallel on others. For instance, an iterative system solver can be parallelized at the matrix-vector product level to scale its performance, even though these solvers are inherently serial at the highest (outermost) iterative level. When parallelizing a code, one must consider the type of multi-processor platform available. ***Message passing*** and ***shared-memory*** multi processor architectures require different parallelization paradigms. In the message-passing (distributed-memory) architecture, see Figure 7(a), each processor has its own local memory space and communication among the processors is done by sending explicit messages through the interconnection network. The SP2 is typical of distributed-memory architecture. In the typical shared-memory architecture, see Figure 7(b), all processors share the same global memory space. In this case, the processors communicate implicitly simply by accessing the same memory, which allows them to share code and data.

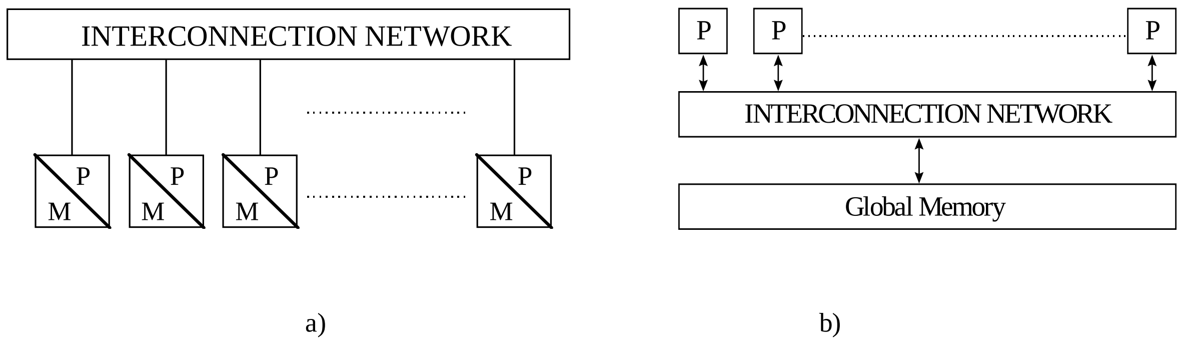

FIGURE 7: Two Parallel Processors Interaction Paradigms.. (a) Message-Passing (b) Shared Memory

Example 1 (Matrix-Vector Product Example for Shared-Memory machine):

As mentioned above, the most critical (time consuming) operation in an iterative solver is the matrix-vector product. The serial version of a code for carrying out the MATRIX-VECTOR product is given in Figure 8(a). This sequential algorithm requires n^2 multiplications and additions resulting in $O(n^2)$ runtime complexity.

```
procedure MAT_VECT(A, x, y)
begin
   do i:=1 to n
      do j:=1 to n
  S:     y(i):=y(i)+A(i,j)*x(j);
      end do
  end do
end MAT_VECT

                    a)
```

```
procedure MAT_VECT(A, x, y)
begin
  do all  i:=1 to n
    do  j:=1 to n
        y(i):=y(i)+A(i,j)*x(j);
    end do

  end do
end MAT_VECT

                    b)
```

FIGURE 8: Matrix-Vector Product. (a) Serial Algorithm. (b) Shared-Memory Implementation.

One way of expressing parallelism on a shared-memory system is through the parallelization of the iteration loop. Basically, each iteration can be divided among the processors and executed concurrently (in parallel). Full parallelism is achieved if a sufficient number of processors is available. If there are fewer than *n* processors, some or al processors may execute two or more iterations. For the serial (i.e. uniprocessor) program in Figure 8 (a), consider a particular value *i* and the execution of the statement *S* for the iterations *j=1, 2…n* of the innermost loop:

```
y(i) = y(i) + A(i, 1)*x(1)
y(i) = y(i) + A(i, 2)*x(1)
…
```

As seen, the value assigned to *y(i)* in the first iteration is used in the second and, thus, there exists a “loop-carrier dependence” from *S* to *S* preventing parallel execution of multiple *j* loop iterations. This dependence could be broken by computing *p* subtotals for *y(i)* on *p* processors and then summing these subtotals in parallel. However, this modification of the code would violate Fortran precedence rules and thus may alter the numerical precision of the results. In contrast, the *i*-loop can be executed in parallel by rewriting the code as in Figure 8(b). Suppose *n* processors are available for the execution of the doall-loop. The compiler will then assign the work in such a way that each processor executes the highlighted portion of the code for one value of *i* in Figure 8 (b). Whereas each element of y is computed sequentially, different elements are computed in parallel. For p processors where $p < n$, the compiler will distribute the *y* elements to the processors as equitably as possible. Hence, the peak performance of the doall-loop on *p* processors $(p \leq n)$ results in $O(n^2 / p)$ run time complexity for the code in Figure 8(b).

4 Moment Method Code Optimization on CRAY-C90

4.1 Moment Method Fundamentals

The method of moments (MoM) is one of the standard approaches for the solution of a surface and volume integral equations. This solution technique can be applied to a metallic, homogeneous dielectric as well as inhomogeneous [7]. The boundary integral equation is of the form

$$\oiint_{\partial S} \bar{J}(\bar{r}')G(\bar{r},\bar{r}')\partial s' = \bar{f}^{i}(\bar{r}) \tag{1}$$

where $\bar{J}(\bar{r})$ denotes the unknown current density, $G(\bar{r},\bar{r}')$ is the pertinent Green's function and $\bar{f}^{inc}(\bar{r})$ defines the excitation. Note that the parameters $\bar{r}'$ and $\bar{r}$ refer to position vectors for the integration and observation points, respectively. The latter is the location at which the integral equation is enforced. In the context of the moment method, the observation point is placed at a set of discrete points over the illuminated structure to yield a corresponding number of equations. Introducing the subsectional expansion

$$\bar{J}(\bar{r}) = \sum_{n=1}^{N} I_n \bar{f}_n(\bar{r}) \tag{2}$$

for the surface currents, where $\bar{f}_n(\bar{r})$ is the expansion basis over the *nth* pair of triangles [1], we obtain a discrete system of equations for the solution of the unknown coefficients I_n. We typically write this system as

$$[Z]_{mn}\{I_n\} = \{V_n\} \tag{3}$$

where $[Z]_{mn}$ is the $N \times N$ impedance matrix whose elements are computed from

$$Z_{mn} = \oiint_{S_m} \bar{f}_m(\bar{r}) \cdot \oiint_{S_n} \bar{f}_n(\bar{r}')\, G(\varpi\bar{r},\varpi\bar{r}')\partial s \partial s' \tag{4}$$

and

$$V_m = \oiint_{S_m} \bar{f}_m(\bar{r}) \cdot \bar{E}^{inc}(\bar{r})\partial s \tag{5}$$

where *m* and *n* are indices on the edges of the surface facets S_m (S_n) denoting the surface area over the *mth* testing (*nth* integration) triangle pair. Because of the Green's function [8],

$$G(\bar{r},\bar{r}') = A e^{-jk|\bar{r}-\bar{r}'|} / |\bar{r}-\bar{r}'| \tag{6}$$

where A is a constant and k is the wavenumber, the matrix *[Z]* is fully populated and this is a distinct feature of the moment method. The most popular and straightforward approach is to perform an LU decomposition of *[Z]* and then solve for $\{I_n\}$. As noted above, this is one of the most time consuming steps of the code and must be optimized for optimum CPU performance. For our specific application of jet engine scattering, a last step is the computation of the fields at a plane in front of the engine and the generation of the modal scattering matrix [9]. The modal scattering matrix is then used for the propagation of the fields through the inlet duct and the computation of the scattered fields due to a given excitation field.

In the following subsection we describe the optimization of the moment method code for jet engine scattering on a 16 processor, shared-memory CRAY-C90 vector supercomputer. An interesting aspect of this code is the use of a cylindrically periodic Green's function which takes advantage of the jet engine blade periodicity. As a result, the computational domain is reduced to a domain over a single blade and this yields a reduction in the number of unknowns by a factor equal to N_s, the number of blades (typically 20 to 30). However, to take advantage of blade periodicity, it is necessary to decompose the excitation field into cylindrical duct modes and to loop through all of the modes which are allowed to propagate unattenuated [10, 11, 12]. The scattered modes and subsequently the corresponding fields due to each mode is then computed by carrying out an inverse cylindrical Fourier transform. Although, this modal decomposition may be more expensive for single angle computations, this disadvantage disappears when multiple excitations are considered. Most importantly, the mode by mode excitation is particularly attractive for parallelization since the computation for each mode excitation is independent of the others.

4.2 MoM Code Structure

A flowchart of the jet engine moment method code is given in Figure 9. This flow chart shows

- the outer loop going through all cylindrical modes

- the computation of the matrix element Z_{mn}
- system solution via LU decomposition
- inverse cylindrical FFT operation for generating each row of the scattering matrix

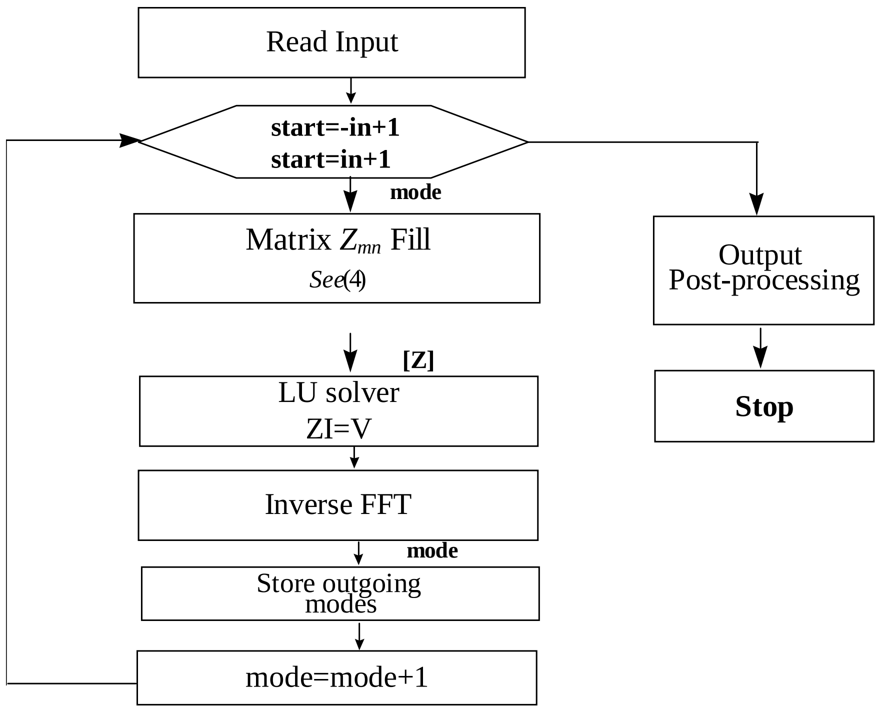

FIGURE 9: MoM Code Main Iteration Loop.

The matrix fill and Inverse FFT routine pseudo-codes are shown in Figures 10 (a) and (b) respectively. In these codes, N_s denotes the number of engine slices/blades and *const* is

```
for all slices ( s=0, $N_s$-1)
    for all source triangles ( p=1, num_source_triangles)
        for all observation triangles (q=1, num_observation_triangles)
            compute next entry of
            impedance matrix $[A]^{(s)}$
        endfor
    endfor
    $[Z]^{(ms)} = [Z]^{(ms)} + const(m, s)*[A]^{(s)}$
endfor
                                    a)
```

```
for all (angles i1=1, exrho)
    for all (point i2=1, ngas)
        for all (angles i3=1, exphi)
            for all (points i4=1, ngas)
                call integration routine
            endfor
        endfor
    endfor
endfor
                        b)
```

FIGURE 10: (a) Matrix Fill Pseudo-code. (b) Inverse FFT pseudo-code.

a complex function of the mode number (*m*) and slice number(*s*). Also, the matrices $[A]^{(0)}$, $[A]^{(1)}$, …, $[A]^{N_s-1}$ are the submatrices comprising the matrix *[Z]* and provide the interaction between the engine blades. Based on the previous discussions, the parallelization of this algorithm can be performed at several levels. For example, since the mode iterations of the main loop are entirely independent, they can be done in parallel. However, in this case each processor will be required to keep a full copy of the *[Z]* matrix and thus the storage requirements become very large. Also, load imbalance will become a significant bottleneck if number of modes is not evenly divisible by the total number of processors. We avoid these issues, we instead proceeded to individually parallelize the matrix fill, LU factorization and InverseFFT routines, thus taking advantage of the finer-grain paralellism of the code at this level.

4.3 Major Optimization Stages

Our optimization of MoM code consisted of four steps:

1. Porting of the matrix fill routine to static storage device (SSD[2])
2. Rewriting and parallelization of the matrix fill routine
3. Library call substitutions

[2] SSD-solid state storage is very high speed secondary memory of up to 2048 Mwords (16Gbytes). CRAY-C90 supports up to 4 1800-Mbytes/s channels for accessing.

4.   Vectorization and parallelization of the inverse FFT routine

Sections 4.3.1 through 4.3.5 describe the optimization procedures with Section 4.3.2 giving a concise introduction vectorization.

4.3.1 Porting Matrix Fill Routine to Static Storage Device (SSD)

As shown in Figure 9, the main iteration loop calls the *matrix_fill* routine for every mode. In the original version of the code, each impedance submatrix $[Z]^{(s)}$ had to be recomputed for every slice (Figure 10(a)). This was done to avoid in-core storage of each $[Z]^{(s)}$ submatrix, even though the submatrices were identical from one slice to the other (since the slice is a periodic cell). The CRAY-C90 however provides way to avoid both excessive storage requirements and the need for a recomputation of the submatrices is to by making use of the SSD at the first call of the *matrix_fill* routine. At each subsequent mode loop, the stored matrices are recalled from the SSD to accumulate the sum in $[Z]^{(ms)}$ matrix. The resulting pseudo-code is shown in Figure 11. This change alone resulted in an 11x   program speedup as seen below in Table 1.

```
for all (slices s=0, $N_s$-1)
     if (m==1)  then                                      ! if first mode
        for all (source triangles p=1, num_source_triangles)
            for all observation triangles q=1, num_observation_triangles
                compute next entry of
                impedance matrix $[Z]^{(s)}$
            endfor
        endfor
        store  $[Z]^{(s)}$  on SSD
      else
        read  $[Z]^{(s)}$  from SSD                             ! avoid computing
$[Z]^{(s)}$
      endif
     $[Z]^{(ms)} = [Z]^{(ms)} + const(m, s)*[Z]^{(s)}$
```

FIGURE 11: Pseudo-code to compute the impedance matrix using SSD storage.

4.3.2 Principles of Vectorization

Since a high level of vectorization is a must for efficient utilization of the CRAY-C90 processor, based on vectorization, in this section we give a brief introduction to the principles of vectorization before introducing the code speed-ups.

A vector is a set of scalar data items (elements), all of the same type. A vector instruction applies an arithmetic or logical operation to vectors by using *vector registers* as operand registers. A vector load/store instruction moves a vector between a vector register and memory with some constant memory strides. However if GCD(stride, number of memory banks) $\neq 1$, then the load/store may take twice as long, four times or more. There is also some support for irregular, nonconstant strides. However, a data layout in memory that allows stride=1 will always be fastest.

Vectorization converts scalar operations such as multiplications and additions to corresponding vector operations. That is, a vectorized code processes the multiplications/additions as a sequence of vector statements and this is particularly attractive for multiplying arrays in do-loops. A well-vectorized code can easily achieve a speed-up of 10 to 20 times, as compared to the equivalent scalar code. Vectorizing compilers perform the mechanics of vectorization; but they are much more effective if the code is written in a style that helps the compiler recognize vectors. They also report what was successfully vectorized and provide some hints about problems that prevented vectorizations. These hints are often useful for rewriting the code for more effective vectorization. Below we consider a few examples which illustrate the implementation of vectorization steps.

Examples of vectorization:

Example 3: (Recurrence)

Consider now the loop shown in Figure 12(a). The statement instances are executed in the order shown in the unrolled code. Note that the value of each *A(i),* $2 \leq i \leq n$, is dependent on the *A(i-1)* calculated by the immediately previous statement (iteration). If we were to vectorize L, then the products *A(I-1)*B(i)* in the vector statement *A(1:n)=A(0:n-1)*B(1:n)* would be computed using the values *A(0:n-1)* that existed when entering the loop. Since incorrect results would be produced, a vectorizing compiler would indicate that this loop can not be vectorized due to its (loop-carried) dependence.

```
serial loop                      vectorization not possible
L:   do i=1, n                   L(1): A(1) = A(0)*B(1)
       S:  A(i) = A(i-1)*B(i)    L(2): A(2) = A(1)*B(2)
     end do                            …
                                 L(1): A(n) = A(n-1)*B(n)
            (a)                             (b)
```

FIGURE 12: Vectorization example 4.

Example 4: (Loop Interchange)

However, consider the operation shown in Figure 13, referred to as the two-point difference scheme. One way to rewrite the loop as a set of vector statements is to interchange the loop order and this is done in Figure 13(b). Now, the inner loop may be converted to a vector statement, as shown in Figure 13(c).

```
serial loop                                  transformed loop
L:   do i=1, n                               L:   do j=1, n
       do j=1, n                                    do i=1, n
        S:  A(i, j)=(A(i, j-1) + A(i, j+1))/2        S:  A(i, j)=(A(i, j-1) + A(i, j+1))/2
       end do                                       end do
     end do                                       end do
                 (a)                                            (b)

Vectorized loop
L:   do j=1,n
       A(1:n, j)=(A(1-n, j-1)+A(1:n, j+1))/2
      end do
                  c)
```

FIGURE 13: Vectorization example 4. (a) original code (b) loop restructuring before vectorization (c) vectorized code.

Example 5: (Loop Distribution)

In the loop shown in Figure 14(a) the operation S1 depends on the result of S2 in the previous iteration. This apparent interdependence, can be removed by distributing the loop as shown in Figure 14(b). The rewritten code, although not as compact, is now amenable to vectorization as shown in Figure 14 (c).

```
serial loop
L:    do i=1, n
          S1:  D(i)=A(i-1)*D(i)
          S2:  A(i)=B(i)+C(i)
      end do
                    a)

vectorized loop
A(1:n)=B(1:n) + C(1:n)
D(1:n)=A(0:n-1)*D(1:n)
                    c)
```

```
transformed loop
L21:  do i = 1, n
          S1:  A(i)=B(i)+C(i)
      end do
L22:  do i = 1, n
          S1:  D(i)=A(i-1)*D(i)
      end do
                    b)
```

FIGURE 14: Vectorization example 5. (a) original code (b) rewritten code (c) vectorized code

For code such as examples 4 and 5 if a vectorizing compiler is unable to vectorize the loop, performing the transformation in (b) manually should allow the compiler to produce the vector code shown in (c).

4.3.3 Vectorization and Parallelization of the Inverse Transform

Runs made with the original version of the MoM code revealed very short average vector lengths. Also, very little improvement in this regard was observed when the compiler optimization flags were enabled. Such a poor utilization of vector registers is explained by the fact that the original version of the MoM code was written for RISC processors. Therefore no attempt was made to program the code in a manner suitable for vectorization (or parallelization). After careful examination of the code, both manually and using various performance tools, we discovered that the Inverse FFT (or IFFT) integration routine had the highest potential for vectorization. The original code fragment took a single source triangle and passed it sequentially through several different subroutines to perform a single surface integration over each vector function as shown in Figure 15.

```
subroutine get_vertices(vert,q)                 subroutine integrate(vert, vint)
   real vert(3,3)                                  real vert(3,3), vint(3)
   integer q                                       vint(1)=…
   vert(1,1)=…                                     vint(2)=…
   ….                                              vint(3)=…
   vert(3,3)=…                                  end integrate
end get_vertices

real vert(3,3)      ! list of vertices of the triangle
real vint(3,3)      ! result of  integration
for each source triangle q
   call get_vertices(vert,q)
   call integrate(vert, vint)
   …
endfor
```

FIGURE 15: Inverse FFT Integration Routine.

This version of Inverse FFT also yielded a poor performance on the CRAY-C90. More specifically, the average vector length was 6, with 128 being the theoretical maximum limited by the length of the processor's vector registers. Further, automatic vectorization failed to perform the interprocedural analysis to required to this code.

Our first successful phase of optimization was to replace these single element calls with a call to subroutines that accepted as inputs the entire list of integration elements rather than providing them one by one. This transformation resulted in a rather efficient vectorizable code (See Figure 16). This code permits vectorization within each low-level subroutine, thereby eliminating the need for interprocedural analysis.

```
real vert(num_triangles, 3, 3)          ! list of vertices of the triangle
real vint(num_triangles, 3, 3)          ! result of integration

call get_vertices(vert)
call integrate(vert, vint)

subroutine get_vertices(vert)                     subroutine integrate(vert, vint)
  real vert(num_triangles, 3, 3)                    real vert(num_triangles, 3, 3)
  for each triangle q                               real vint(num_triangles, 3, 3)
    vert(q, 1, 1)                                   for each triangle q
    …                                                 vint(q, 1)=…
    vert(q,  3, 3)=…                                  vint(q, 2)=...
  endfor                                              vint(q, 3)=…
end get_vertices                                    endfor
                                                  end integrate
```

FIGURE 16: Vectorizable Inverse FFT Integration Routine.

Further optimization included standard techniques such as subroutine inlining (both manually and automatically with the inline compiler option), loop splitting, scalar expansion, etc., all leading to an average vector length increase by an additional factor of 10 or so.

The next step was to parallelize the IFFT routine. Since CRAY-C90 is a parallel shared-memory machine, the loop distribution principles discussed in *Example 2* apply here. We observe that the individual calls to the integration routine (see Figure 10(b)) are entirely independent and, thus, the outermost loop can be readily distributed yielding the parallel code in Figure 17. Finally, we note that maximum parallelization is achieved by maintaining good load balance, and this is readily done provided that the outer loop trip count of divisible by the total number of processors.

```
do all angles  i1=1, exrho
    for all point i2=1, ngas
        for all angles i3=1, exphi
            for all points  i4=1, ngas
                call integration routine
            endfor
        endfor
    endfor
end do all
```

FIGURE 17: Parallel Inverse FFT Integration Routine.

4.3.4 Parallelization of MatrixFill Routine

Having achieved sufficient performance on one processor, the next optimization step is to parallelize the *matrix_fill* routine. As noted above the code uses a faceted surface model employing triangular elements for the characterization of the jet engine blade geometry (or slice) resulting in the discrete system given by (1).

Lets us begin with the serial version of *matrix_fill* as given in Figure 18(a). The code loops over the surface patches and computes the partial integrals for each of the integrals associated with currents flowing through each of the three edges that make up the patch. These three evaluations provide contributions to the matrix entries and are assembled later at their appropriate locations in the matrix *[Z]*. To parallelize the code we need to distribute the outermost loop source triangles to different processors as shown in Figure 18(b). Since, the three edges for both the source and observation triangles will not generally be assigned to the same processor, the GUARDING code block is needed to guarantee that only one processors updates a *[Z]* matrix entry at any one time.

```
for all observation triangles p
   for all source triangles q
     …
     call integration routine
     for all edges m(1), m(2), m(3) of p, v
        for all edges n(1), n(2), n(3) of q, w
          …
          Z(m(v), n(w)) = Z(m(v), n(w)) +const(p, q, v, w)
          …
       endfor
     endfor
   endfor
endfor
```

a)

```
do across all observation triangles p
   for all source triangles q
     …
     call integration routine
     for all edges m(1), m(2), m(3) of p, v
        for all edges n(1), n(2), n(3) of q, w
          …
GUARD mutex region
          Z(m(v), n(w)) = Z(m(v), n(w)) +const(p, q, v, w)
END GUARD
          …
        endfor
     endfo
   endfor
end do across
```

b)

FIGURE 18: Matrix Fill. (a) Serial version. (b)Parallel version.

The synchronization overhead to execute the guarding code is negligible compared with the time needed to go through other statements in the loop. Since, in general, the number of source triangles is much larger than the number of processors available, the above parallel version of *matrix_fill* should be quite efficient.

4.3.5 Library Call Substitutions

Where possible, the CRAY-C90 library routines were substituted into the original code. Most of these routines are carefully hand-tuned for both vector and parallel execution and it is therefore a good practice, in general to make use of the system library routines whenever possible. Of primary interest to us is the LU factorization routine which ranks second in the list of the most time-consuming operations in the MoM code. The original code employed the LINPACK routine ZGEFA and this was replaced by the equivalent CRAY-C90 CGETRF vector parallel routine. The speedup resulting from substitution of the CGTERF is shown in Figure 19, where all times are wall-clock times.

It is clearly seen that substantial speedup was achieved. More specifically, the CRAY-C90 library LU routine reduced the CPU time by 30% percent on a single processor, and from 35 sec down to 5 sec on the 16-processor.

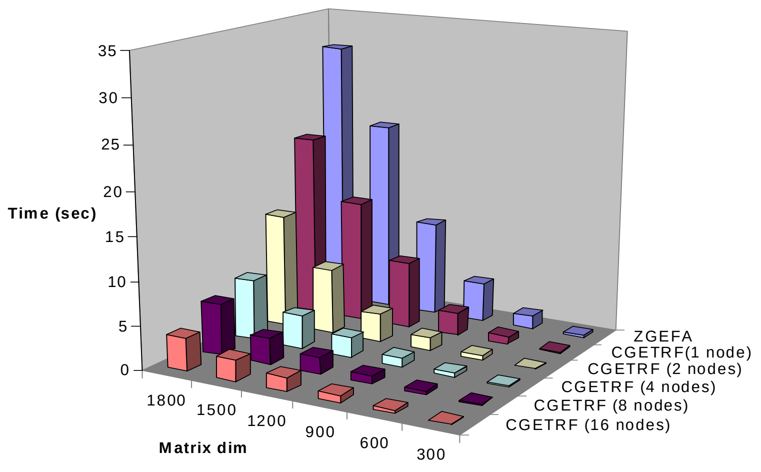

FIGURE 19: Scaling of CRAY-C90 matrix Factorization Routine

4.4 Performance of Individual Optimization Stages

The overall improvements after each optimization stage of the code, as described above, are summarized in the Table 1. The underlying geometry was a 21 engine blade geometry, 3 wavelengths in diameter.

		COMPILER OPTIMIZATION ONLY	SSD OPTIMIZATION	INVERSE TRANSFORM VECTORIZATION	FILL AUTOTASKING AND LIBRARY CALLS
FILL	CPU TIME	222	12.5	12.5	12.5
	WALL CLOCK	322	18	18	4
INVERSE TRANSFORM	CPU TIME	6.5	6.5	4	4
	WALL CLOCK	9	9	2.25	2.25
LU	CPU TIME	1.7	1.7	1.7	1.4
	WALL CLOCK	2.5	2.5	2.5	0.4
TOTAL	CPU TIME	235	21	19	18.3
	WALL CLOCK	340	30	23	6.68
	SPEED-UP		11.3	1.3	3.4
			50 TIMES TOTAL SPEEDUP		

TABLE 1: MoM Speedup after each of the Four Code Optimization Stages

The first three columns of the Table 1 present the run-time improvements of the serial code, whereas the fourth column indicate speed-up of the paralellized code which was run on the number of available processors. The first column shows the code run time in the absence of any hand-tuning with only compiler limited vectorization, inlining, etc. This time is comparable to the run time of the original code on a high performance workstation with compiler optimizations.

5 Conclusion/Further Optimizations

In this paper we described various optimization techniques that can be routinely used to improve the performance of standard moment method codes for electromagnetic applications as well as more specialized codes. We detailed several vectorization techniques which proved effective in tuning the performance of the MoM code on a CRAY-C90. We showed parallel loop distribution for *matrix_fill* and parallelization of the LU factorization. Although, *matrix_fill* scales as n^2 whereas LU factorization scales as n^3**,** the *matrix_fill* routine remains the most dominant part of the code for large values of n due to the large constant in front of n^2. In fact the crossover between $O(n^2)$ and $O(n^3)$ has not been reached [13].

Although, we successfully employed SSD storage to avoid redundant *matrix_fill* for every mode iteration, this approach is not likely to succeed outside the CRAY-C90 environment. We are currently working on porting the *matrix_fill* section of the code to a distributed memory architecture. In this case, each processor will be responsible for computing and storing only its assigned portion of the matrix entries. We hope that by carefully decomposing the mesh and mapping the triangular elements to different processors, we will be able to reduce the overall communication needs and thus achieve a scalable distributed memory code.

Acknowledgment

The results and code performance given in this paper were carried out at the Wright Laboratories Aeronautical Systems Major Shared Resource Center (ASC MSRC) computing facilities.

References

[1] S.M. Rao, D.R. Wilton, and A.W. Glisson. "Electromagnetic scattering by surfaces of arbitrary shape" IEEE Trans. Antennas Propagat., vol. AP-30, no3, pp. 409-418, May 1982.

[2] H.P Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers. Frontier Series*, ACM Press, New York, NY, 1991

[3] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, MA, 1997.

[4] Eric L. Boyd, Waqar Azeem, Hsien-Hsin Lee, Tien-Pao Shih, Shih-Hao Hung, and Edward S. Davidson. "A Hierarchical approach to modeling and improving the performance of scientific applications on the KSR1," Proceedings of the International Conference on Parallel Processing, August 94.

[5] William Mangione-Smith, Tien-Pao Shih, Santosh G. Abraham, and Edward S. Davidson. "Approaching a Machine-Application Bound in Delivered Performance on Scientific Code," Special Issue of IEEE Proceedings on Computer Performance Analysis, ugust 93.

[6] IBM Corporation, *Optimization and Tuning Guide for the Fortran and XL C Compilers*.

[7] R. F. Harrington. *Field Computation by Moment Methods*, Macmilian, New, York, 1968. (Reprinted by Krieger Publishing Company, Malabar, Florida, 1983).

[8] Johnson J.H Wang. *Generalized Moment Methods in Electromagnetics.* John Wiley & Sons, Inc., New York, 1991.

[9] H. T. Anastassiu, J. L. Volakis, and D. C. Ross. "The mode matching technique for electromagnetic scattering by cylindrical weveguides with canonical terminations," J. of Electromagnetic Waves and Applications, 9(11/12):1363-1391, Nov./Dec. 1995.

[10] D. C. Ross, J. L. Volakis, and H. T. Anastassiu, "Hybrid finite element-modal analysis of jet engine inlet scattering," *IEEE* Trans. Antennas and Propagation, 43(3):277-285, March 1995.

[11] D.c. Ross, J. L. Volakis, and H. T. Anastassiu, "Overlapping modal and geometric symmetries for computing jet engine inlet scattering," IEEE Trans. Antennas Propagation, 43(10):1159-1163, Oct. 1995.

[12] H. T. Anastassiu, J. L. Volakis, and D. C. Ross, and D. Andersh, "Electromagnetic scattering from simple jet engine models," IEEE Trans. Antennas and Propagation, 44(3):420-421, March 1996.

[13] Tom Cwik, Daniel S. Katz, "Scalable Solutions to Integral-Equatio and Finite-Element Simulations," IEEE Transactions on Antennas Propagation, vol. 45, no.3, March 1997