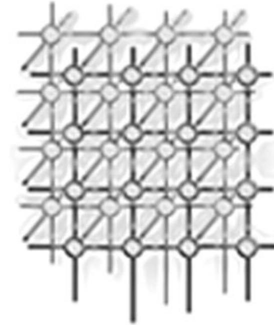
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **15**:607–621 (DOI: 10.1002/cpe.718)



A scalable HPF implementation of a finite-volume computational electromagnetics application on a CRAY T3E parallel system[‡]

Yi Pan[1,*,†], Joseph J. S. Shang[2] and Minyi Guo[3]

[1] *Department of Computer Science, Georgia State University, Atlanta, GA 30303, U.S.A.*
[2] *Air Vehicle Directorate, Air Force Research Laboratory, WPAFB, OH 45433-7521, U.S.A.*
[3] *Department of Computer Software, The University of Aizu, Aizu-Wakamatsu City, Fukushima, 965-8580 Japan*

SUMMARY

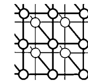
The time-dependent Maxwell equations are one of the most important approaches to describing dynamic or wide-band frequency electromagnetic phenomena. A sequential finite-volume, characteristic-based procedure for solving the time-dependent, three-dimensional Maxwell equations has been successfully implemented in Fortran before. Due to its need for a large memory space and high demand on CPU time, it is impossible to test the code for a large array. Hence, it is essential to implement the code on a parallel computing system. In this paper, we discuss an efficient and scalable parallelization of the sequential Fortran time-dependent Maxwell equations solver using High Performance Fortran (HPF). The background to the project, the theory behind the efficiency being achieved, the parallelization methodologies employed and the experimental results obtained on the Cray T3E massively parallel computing system will be described in detail. Experimental runs show that the execution time is reduced drastically through parallel computing. The code is scalable up to 98 processors on the Cray T3E and has a performance similar to that of an MPI implementation. Based on the experimentation carried out in this research, we believe that a high-level parallel programming language such as HPF is a fast, viable and economical approach to parallelizing many existing sequential codes which exhibit a lot of parallelism. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: parallel computers; execution time; efficiency; scalability; loop parallelization; Cray T3E; high performance Fortran

*Correspondence to: Yi Pan, Department of Computer Science, Georgia State University, Atlanta, GA 30303, U.S.A.
†E-mail: pan@cs.gsu.edu
‡A preliminary version of this paper appeared in ICA3PP 2000, Hong Kong, December 2000.

Contract/grant sponsor: Air Force Office of Scientific Research; contract/grant number: F49620-93-C-0063
Contract/grant sponsor: National Science Foundation; contract/grant number: CCR-9211621, OSR-9350540, CCR-9503882, ECS-0010047
Contract/grant sponsor: Air Force Avionics Laboratory, Wright Laboratory; contract/grant number: F33615-C-2218
Contract/grant sponsor: Japan Society for the Promotion of Science; contract/grant number: 14580386

Copyright © 2003 John Wiley & Sons, Ltd.
Received 11 October 2001
Revised 8 August 2002



1. INTRODUCTION

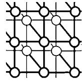
Computational electromagnetics (CEM) in the time domain is the most general numerical approach for describing dynamic or wide-band frequency electromagnetic phenomena. Computational simulations are derived from discretized approximations to the time-dependent Maxwell equations [1–10]. High numerical efficiency of CEM simulation procedures can be attained either by algorithmic improvements to solve the Maxwell equations or by using scalable parallel distributed memory computer systems. Since a massive volume of data-processing is involved in solving the Maxwell equations, distributed memory computer systems are viable means to solve the memory shortage problem on workstations or vector computer systems. The other advantage is the reduced time when parallel processing is employed to solve the Maxwell equations. Hence, parallelization of existing sequential Fortran code for solving the Maxwell equations is an important effort towards developing an efficient and accurate CEM code to analyze refraction and diffraction phenomena for aircraft signature technology.

Until now, three versions of the solver have been available. One is the sequential Fortran version called MAX3D developed by the second author of this paper for vector machines such as the CRAY C90. The second is the MPI (Message-passing Interface) [11,12] version implemented by Dr Marcus Wagner of IBM [13]. The third one is the Power Fortran Accelerator (PA) [14] version developed by the first author when he was an Air Force Office for Scientific Research (AFOSR) Faculty Research Fellow at the Wright Patterson Air Force Base. The description of the programs and their performance results are summarized in [15]. Using the message-passing paradigm has several disadvantages: the cost of producing a message-passing code is usually very high, the length of the code grows significantly and it is much less readable and less maintainable than the sequential version. For these reasons, it is widely agreed that a higher level programming paradigm is essential if parallel systems are to be widely adopted. Previous results have shown that the performance of the PA code on SGI Origin 2000 is even better than the MPI counterpart besides the advantages already stated [15]. This suggests that a higher level programming paradigm such as the HPF is a viable and alternative way to parallelize CEM codes.

However, the results achieved so far are only a preliminary step in achieving a scalable version of the MAX3D code. The major problem with the PA version is that it is not portable across different parallel computer systems since Power Fortran is only available on SGI Power Challenger and Origin 2000.

The primary research objectives of this research was to develop a parallel code for the MAX3D using High Performance Fortran (HPF). HPF is more powerful than Power Fortran and is available in almost every major parallel computer system. Our goals are to improve the scalability of the Power Fortran version of the MAX3D code further, investigate the possibility of using HPF to implement parallel CEM applications and make the parallel code portable across several parallel platforms. These objectives have been accomplished through implementation of the parallel code for the MAX3D via PGHPF [16] (an HPF compiler developed by the Portland Group). Experimental runs have shown that the execution time is reduced drastically through the use of HPF. The time spent for the implementation is about two months. Compared with MPI implementation, it is much shorter. Since HPF is a general parallel language and not targeted to any particular machine, its performance is usually not as good as specialized parallel languages such as Power Fortran on the SGI machines. Yet, the HPF code we have produced is still scalable up to 98 processors on the Cray T3E. Based on the experimentation carried out in this research, we show that HPF is a viable and alternative way to

Copyright © 2003 John Wiley & Sons, Ltd.



parallelize many existing sequential codes which exhibit a lot of parallelism besides MPI. In many cases, using HPF to implement a sequential Fortran code is more economical than using MPI. HPF also solves the problem of limited memory size on a single processor and provides a fast way to port the code on parallel computer systems.

In the following sections, we will describe the numerical formulation of the problem and the related subroutines in the sequential code. We then present the parallelization schemes used, the results achieved during the research and outline the impact of the achieved research results.

2. THE MAXWELL EQUATIONS

The behavior of electromagnetic phenomena is modeled by the set of four equations collectively known as Maxwell's equations. In differential vector form for a Cartesian reference frame, these equations are written as follows (see [8,9] for more details).

Faraday's law:

$$\frac{\partial B}{\partial t} + \nabla \times E = 0 \tag{1}$$

Ampère's law:

$$\frac{\partial D}{\partial t} - \nabla \times H = -J \tag{2}$$

Gauss's law:

$$\nabla \cdot B = 0, \quad B = \mu H \tag{3}$$

$$\nabla \cdot D = 0, \quad D = \epsilon E \tag{4}$$

where ϵ and μ are the electric permittivity and the magnetic permeability which relate the electric displacement to the electric field intensity and magnetic flux density to the magnetic field intensity, respectively.

Using a coordinate transformation from the Cartesian coordinates, the time-dependent Maxwell equations can be cast in a general body-conformal curvilinear frame of reference. The resultant governing equations in flux-vector form acquire the following form [17]:

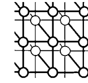
$$\frac{\partial U}{\partial t} + \frac{\partial F_\xi}{\partial \xi} + \frac{\partial F_\eta}{\partial \eta} + \frac{\partial F_\zeta}{\partial \zeta} = -J \tag{5}$$

where U is the transformed dependent variable now scaled by the local cell volume, V. F_ξ, F_η and F_ζ are the contravariant components of the flux vectors of the Cartesian coordinates.

$$U = \{B_x V, B_y V, B_z V, D_x V, D_y V, D_z V\}^{\mathrm{T}} \tag{6}$$

$$\begin{aligned} F_\xi &= \xi_x F_x + \xi_y F_y + \xi_z F_z \\ F_\eta &= \eta_x F_x + \eta_y F_y + \eta_z F_z \\ F_\zeta &= \zeta_x F_x + \zeta_y F_y + \zeta_z F_z \end{aligned} \tag{7}$$

Copyright © 2003 John Wiley & Sons, Ltd.



where the flux-vector components of the Cartesian frame are:

$$
\begin{aligned}
F_x &= \{0, -D_z/\epsilon, D_y/\epsilon, 0, B_z/\mu, -B_y/\mu\}^{\mathrm{T}} \\
F_y &= \{D_z/\epsilon, 0, -D_x/\epsilon, -B_z/\mu, 0, B_x/\mu\}^{\mathrm{T}} \\
F_z &= \{-D_y/\epsilon, D_x/\epsilon, 0, B_y/\mu, -B_x/\mu, 0\}^{\mathrm{T}}
\end{aligned} \tag{8}
$$

and ξ_x, η_x, ζ_x, ξ_y, η_y, ζ_y, ξ_z, η_z, as well as ζ_z are the nine metrics of the coordinate transformation.

The characteristic-based finite-volume approximation is achieved by splitting the flux vector according to the signs of the eigenvalues of the coefficient matrix in each spatial direction. The flux vector at any cell interface is represented by the superposition of two components: F_ξ^+, F_ξ^-, F_η^+, F_η^-, F_ζ^+ and F_ζ^- according to the direction of the wave motion [17]. At the cell surfaces, the split flux vectors are calculated by the reconstructed dependent variables on either side of the interface according to the Kappa scheme [18]:

$$
\begin{aligned}
F_{\xi,i+\frac{1}{2}} &= F_\xi^+(U^{\mathrm{L}}_{i+\frac{1}{2}}) + F_\xi^-(U^{\mathrm{R}}_{i+\frac{1}{2}}) \\
F_{\eta,j+\frac{1}{2}} &= F_\eta^+(U^{\mathrm{L}}_{j+\frac{1}{2}}) + F_\eta^-(U^{\mathrm{R}}_{j+\frac{1}{2}}) \\
F_{\zeta,k+\frac{1}{2}} &= F_\zeta^+(U^{\mathrm{L}}_{k+\frac{1}{2}}) + F_\zeta^-(U^{\mathrm{R}}_{k+\frac{1}{2}})
\end{aligned} \tag{9}
$$

where U^{L} and U^{R} denote the reconstructed dependent variables at the left- and right-hand side of the cell interface.

A single-step, two-stage Runge–Kutta scheme is adopted for the temporal integration process: the resultant numerical procedure is capable of generating numerical solutions accurate in space from first to third order and accurate in time to second order.

The initial and boundary conditions for a perfectly electrical conducting (PEC) sphere can be summarized as follows. The incident wave consists of a linearly polarized harmonic field propagating in the negative z-axis direction [9,10]. The far-field boundary condition for the truncated numerical domain is easily implemented by using the split flux vectors from the characteristic formulation.

$$
F_\xi^-(\xi_0, \eta, \zeta) = 0 \tag{10}
$$

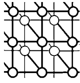
The appropriate boundary conditions at the surfaces of a PEC scatterer are [6]:

$$
\begin{aligned}
\vec{n} \times (E_1 - E_2) &= 0 \\
\vec{n} \times (H_1 - H_2) &= J_s \\
\vec{n} \cdot (B_1 - B_2) &= 0 \\
\vec{n} \cdot (D_1 - D_2) &= \rho_s
\end{aligned} \tag{11}
$$

where the subscripts 1 and 2 denote fields in two adjacent regions. Following previous efforts, two extrapolated numerical boundary conditions are introduced to replace the equations that contain unknowns, J_s and ρ_s. The detailed formulation can be found in [6].

The code MAX3D contains many big loops. The main body of the code which is the part for solving sequences and Maxwell's equations is shown here. Since NEND in the code is usually very large (in our test case, it equalled 1562), loop 10 takes most of the time.

Copyright © 2003 John Wiley & Sons, Ltd.



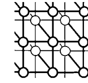

```
      .....
C     START THE SOLVING SEQUENCES
      DO 10 N=1,NEND
         T=FLOAT(N-1)*DT
         CALL IC(N,NEND)
         DO 12 M=1,2
            CALL FXI
            CALL GETA
            CALL HZETA
            CALL SUM(M,N,NEND)
 12      CONTINUE
         IF((N.GT.NES).OR.(N.LT.NBS)) GO TO 10
C     PERFORM FOURIER'S TRANSFORM
         WT=W*(T+DT)
         DO 14 L=1,6
         DO 14 K=1,KL
         DO 14 J=1,JL
            FU(J,K,L)=FU(J,K,L)+U1(IB,J,K,L)*CEXP(RI*WT)
 14      CONTINUE
 10   CONTINUE
      .....
```

After testing and performance profiling of the sequential code MAX3D, we found that the subroutines FXI, GETA, HZETA and SUM occupy the most time for the sequential code and deserve special attention. Subroutines FXI, GETA, and HZETA generate electromagnetic coefficients for the flux vectors and SUM computes the new time level solution. These subroutines are related to Equations (6)–(8). In our study, we concentrate on the most time-consuming subroutines such as FXI, GETA, HZETA and SUM, and try to parallelize the loops within these subroutines efficiently.

3. METHODOLOGIES

High Performance Fortran (HPF) is an informal standard for extensions to Fortran to assist its implementation on parallel architectures, particularly for data-parallel computation [19]. Among other things, it includes directives for expressing data distribution across multiple memories, extra facilities for expressing data parallel and concurrent execution and a mechanism for interfacing HPF with other languages and programming models. HPF is available on almost all popular parallel computer systems, such as Cray T3E, SP2 and SGI Origin 2000, and is much easier to use and faster to port than MPI. HPF allows HPF directives to be added to the sequential Fortran code and to apply the capabilities of a multiprocessor system to the execution of a single job. It splits the job into concurrently executing pieces, thereby decreasing the wall-clock runtime of the job. Although HPF uses a lot of complicated analysis tools to analyze the user program and can produce a corresponding parallel version without user intervention, the quality of the parallel code produced in such a way is usually very poor. In our study, we found that using the PGHPF compiler's option `-Mautopar` to parallelize the MAX3D code automatically does not give effective parallel results. The compiler is simply not intelligent enough to make a smart decision. Another problem with automatic parallelization is that it can create a large number of temporary arrays. Hence, automatic parallelization has to be combined with hand

Copyright © 2003 John Wiley & Sons, Ltd.



tuning to produce more efficient code. For example, when we parallelize the MAX3D code using automatic parallelization on SGI Origin 2000, the parallel time is worse than the sequential time due to data dependence and the heavy communication overhead in the code. In our research, only hand parallelization (inserting directives manually versus automatic code generation by software) is used to avoid memory waste and increase efficiency.

The HPF model can be seen as a collection of distinguishable, but complementary, programming styles. The models and methods we used in the implementation are data- and work-sharing. In data-sharing, data, such as an array, are spreaded over the memory of all of the available processors so that each processor operates primarily on its own part. In work-sharing, the loop iterations are distributed within loops among the system's processors with the goal of executing them in parallel. For instance, the INDEPENDENT directive can be used to divide the iterations of a DO loop among processors or the compiler can divide the work by choosing an implicit array syntax. One natural and powerful strategy involves distributing an array, data-sharing and the iterations of a DO loop that operate on that array, work-sharing, over your available processors. Processors executing a DO loop in parallel to help realize the power of the CRAY T3E and other parallel computing systems. In the following sections, the details of data allocation and parallelization schemes are discussed.

In order for a parallel code to run efficiently, it is also essential that the sequential code uses the cache effectively. The overall optimization steps adopted in our research are:

(1) locate the 'hot spot' of the sequential code via profiling tools;
(2) study these subroutines identified in step 1 carefully for performance improvement;
(3) improve the cache locality behavior of these subroutines through a series of systematic loop transformations;
(4) study the listing file produced by the HPF compiler and locate the overhead caused by inefficient parallel loops; and
(5) insert HPF compiler directives to direct the compiler to parallelize the code so that most of the overhead is eliminated and the most time-consuming loops are parallelized.

Note that some of the goals may be in conflict. Hence, these steps may need to be repeated several times to achieve an overall good performance.

3.1. Distributed data allocation

In a parallel program on a distributed memory architecture, data can be shared (distributed) or private (replicated). Arrays can also be distributed to speedup the code. Arrays are distributed across all processors via the `!HPF$` DISTRIBUTE directive in HPF. The DISTRIBUTE directive names the variables that are to be shared data objects and specifies how they are to be distributed across the PEs. If the data are not specified as shared, they are private (the default). In a program that declares a private data item, each processor gets a copy of storage for that item. In many programming models, this datum is called replicated. When an array is shared, its elements are distributed across the available PEs.

The DISTRIBUTE directive can specify array distribution within the directive by following the array name with distribution information contained in parentheses. The distribution information may include the keywords BLOCK or CYCLIC, which tell the compiler how to distribute the array elements among the available PEs. The BLOCK distribution distributes a block of consecutive memory locations to a processor, while the CYCLIC distribution distributes an array in a cyclic manner.

The DISTRIBUTE directive allows data to be distributed over processors in a variety of patterns. The ALIGN directive is used to specify that certain data objects are to be mapped in the same way as certain other data objects in order to reduce inter-processor communication. Operations between aligned data objects are likely to be more efficient than operations between data objects that are not known to be aligned. In our MAX3D implementation, U0, U1, U2, F and G are all aligned and have the same distribution, as shown later. Hence, we can parallelize the code across dimension K efficiently when working on these large arrays. The ONTO clause specifies the processor arrangement declared in a `PROCESSORS` directive. In our case, the arrangement is `PROCS`.

```
!HPF$ PROCESSORS PROCS(NUMBER_OF_PROCESSORS())
!HPF$ ALIGN U1(I,J,K,L) WITH U0(I,J,K,L)
!HPF$ ALIGN U2(I,J,K,L) WITH U0(I,J,K,L)
!HPF$ ALIGN F(I,J,K,L) WITH U0(I,J,K,L)
!HPF$ ALIGN G(I,J,K,L) WITH U0(I,J,K,L)
!HPF$ DISTRIBUTE U0(*,*,BLOCK,*) ONTO PROCS
```

Similarly, since we need to parallelize HZETA on dimension J, the H array is distributed on dimension J as follows:

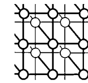
```
!HPF$ DISTRIBUTE H(*,BLOCK,*,*) ONTO PROCS
```

Since different phases require different data distributions, it would be nice to dynamically distribute an array during execution. HPF provides such a mechanism—the `DYNAMIC` and `REDISTRIBUTE` directives. Although various redistribution methods have been studied and a lot of improvements have been made, the redistribution of large arrays still takes a huge amount of time in general [20]. In our research, we have also tried to use these two directives to distribute arrays dynamically. Since redistribution of large arrays such as U0 and H takes a lot of time, the overhead incurred is larger than the saving in time. Hence, we did not use this redistribution scheme in our final implementation. Another distribution method is `CYCLIC`. This distribution distributes an array of elements cyclically to different processors. We found this method was not efficient for the MAX3D code.

3.2. Loop parallelization

The major part of the parallelization process for the MAX3D code is loop parallelization since the codes contain several big loops. The parallelism model used focuses on the Fortran DO loop. The compiler executes different iterations of the DO loop in parallel on multiple processors.

The most flexible and efficient data partition of a computational domain would be the three-dimensional domain decomposition in each coordinate of the transformed space (ξ, η, ζ) since this will cause the least amount of data communication. The global domain has IL, JL and KL cells in the ξ, η and ζ coordinates respectively. After careful study, however, we found that it is hard to parallelize all three dimensions at the same time due to data dependency. Hence, we decided to parallelize one loop at a time and different loops are parallelized in different subroutines (whichever array dimension leads to least data dependencies). Experiments indicate that the parallel code using this approach runs quite efficiently as shown later.



In HPF, we may use a FORALL statement to parallelize a nested loop [21]. However, a FORALL statement automatically parallelizes all the loops within a nested loop and users have no control over the way in which loops are parallelized [22]. Hence, users cannot specify which loop to parallelize within a nested loop. Due to data dependence, this sometimes results in poor performance. In some cases, a DO loop cannot be simply transformed to a FORALL statement, because there are different semantics between DO and FORALL statements [19]. The other compiler directive for multiprocessing in HPF is `!HPF$ INDEPENDENT`. This directive directs the compiler to generate a special code to run iterations of a DO loop in parallel. The loop is also called a shared loop. In a shared loop, the iterations are divided among the available tasks. A shared loop can be specified by the INDEPENDENT directive. A private loop, by contrast, is executed only by the task that invokes it; no work is shared among tasks. Private loops are not preceded by an INDEPENDENT directive. For example, the following directive specifies that loop K in subroutine FXI should be parallelized and I, IM, etc., are private variables.

```
!HPF$ INDEPENDENT, NEW(I, IM, IP, J, SSXI, RSSXI, ....)
      DO 1 K=1,KLM
      DO 1 J=1,JLM
      DO 2 I=1,ILM
C     Initial value set up
  2   CONTINUE
      DO 3 I=2,ILM
      IM=I-1
      IP=I+1
C     RECONSTRUCT THE DATA AT THE CELL INTERFACE, KAPA
      UP1(I)=U1(I,J,K,1)+0.25*RP*((1.0-RK)*(U1(I,J,K,1)-U1(IM,J,K,1))
     1                          +(1.0+RK)*(U1(IP,J,K,1)-U1(I,J,K,1)))
      ......
```

Shared loops specify the behavior of all tasks collectively but they only implicitly define the behavior of individual tasks. Shared loops do not guarantee the order in which iterations will be executed. The lack of a defined order allows the system to execute iterations concurrently.

Inside a shared loop, each task executes its assigned loop iterations as if each task were its own serial region. The assignment of loop iterations is accomplished by aligning the iterations according to the distribution of the array named in the ON clause of the INDEPENDENT directive.

There is an implicit barrier synchronization at the end of a shared loop, at which all tasks wait until the last one has completed. The shared loop ends after the DO loop that immediately follows the INDEPENDENT directive. After the shared loop has finished executing, the tasks continue to execute in parallel.

For multiprocessing to work correctly, the iterations of the loop must not depend on each other; each iteration must stand alone and produce the same answer regardless of when any other iteration of the loop is executed. Not all DO loops have this property and loops without it cannot be correctly executed in parallel. However, many of the loops encountered in practice fit this model. Furthermore, many loops that cannot be run in parallel in their original form can be rewritten to run wholly or partially.

Many loops that have data dependencies can be rewritten so that some or all of the loop can be run in parallel. The essential idea is to locate the statement(s) in the loop that cannot be made parallel and try to find another way to express it that does not depend on any other iteration of the loop. Through loop fusion and loop interchange, we eliminate a lot of data dependencies in the code.

Copyright © 2003 John Wiley & Sons, Ltd.

Some data dependencies cannot be eliminated due to the nature of the computations involved in the code. Our strategy is to parallelize the loops which have no data dependencies. For example, loop K in subroutine HZETA has data dependencies. However, loop J does not. Hence, we parallelized loop J instead of loop K in subroutine HZETA. The code segment of the subroutine HZETA is as follows. The data dependencies exist in the statements `S1, S2`, and `S3` for loop K:

```
      SUBROUTINE HZETA
      PARAMETER(IL=73,JL=61,KL=97)
      ......
!HPF$ INDEPENDENT,  NEW(K, KM, KP, I, SSZT, RSSZT, ....)
      DO 2 J=1,JLM
      DO 2 K=1,KLM
      KM=K-1
      KP=K+1
      DO 2 I=1,ILM
C S1
      UP1(I,K)=U1(I,J,K,1)+0.25*RP*((1.0-RK)*(U1(I,J,K,1)-U1(I,J,KM,1))
     1                            +(1.0+RK)*(U1(I,J,KP,1)-U1(I,J,K,1)))
      ......
C S2
      UC1=UI1(I,KM)*UP1(I,K)+UI2(I,KM)*UP2(I,K)+UI3(I,KM)*UP3(I,K)
      ......
C S3
      H(I,J,K-1,1)=SSZT*(HP(I,K-1,1)+HM(I,K,1))
      ......
 2    CONTINUE
      ......
      RETURN
      END
```

Similarly, we parallelized loop K in subroutines FXI and GETA. This is also reflected in their data distributions: array F, G and H are distributed on different dimensions as shown previously.

4. PERFORMANCE RESULTS

During this research, we used a CRAY T3E computer system at the Ohio Supercomputer Center, which is a powerful scalable parallel system with 128 processing elements. Its peak performance can reach 76.8 GFLOPS. Each processor is a DECchip 64-bit super-scalar RISC processor. It has four-way instruction issue with two floating-point operations per clock. Each processor has on-chip 8 kbyte direct-mapped L1 instruction cache and an on-chip 8 kbyte direct-mapped L1 data cache. It also has an on-chip 96 kbyte three-way-set-associative L2 unified cache. Each processor has a local memory of 16 Mwords (or 128 Mbyte). The clock speed of the processor is 300 MHz and its peak performance is 600 MFLOPS. Although the speed is quite fast compared with other parallel systems such as the SGI Origin 2000, the cache and local memory sizes of the T3E are much smaller (each processor in the SGI Origin 2000 contains a 4 Mbyte secondary cache). This limits the power of the T3E for programs using a lot of memory space such as the MAX3D code. Actually, the T3E's smaller cache size effects sequential programs as well as parallel programs.

Copyright © 2003 John Wiley & Sons, Ltd.

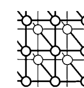


Table I. Execution times of the major subroutines.

No. of processors	FXI	GETA	HZETA	SUM
4	0.592 36	0.895 35	0.8809	2.8894
8	0.314 00	0.456 46	0.443 89	1.4939
16	0.169 63	0.252 90	0.226 04	0.836 92
32	0.099 733	0.150 42	0.118 62	0.507 16
48	0.074 417	0.108 90	0.115 01	0.376 80
64	0.053 583	0.076 501	0.064 894	0.256 80
90	0.053 734	0.076 40	0.064 764	0.255 11
98	0.029 48	0.041 19	0.064 06	0.1443
128	0.029 48	0.041 08	0.063 84	0.1430

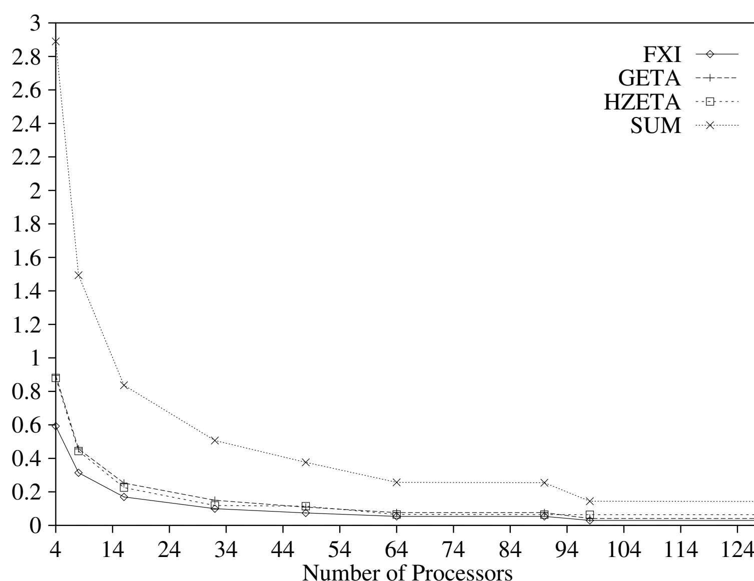


Figure 1. Execution times of the major subroutines.

Several experiments were carried out to tune our parallel code and to adjust our strategy as to how to distribute the various arrays and which loops and subroutines to parallelize. The HPF compiler used is PGHPF [16] with optimization level 3. In this section, some experimental results are reported for the final parallel code produced. All the times in the following presentation are in seconds.

Table I lists the the timing information with hand parallelization for the major subroutines in the code with array size ($73 \times 61 \times 97$). The corresponding times are also presented in Figure 1. Since these subroutines are repeated many times, the time spent there contributes heavily to the total time of the

Copyright © 2003 John Wiley & Sons, Ltd.

Table II. Total parallel execution times for problem size ($73 \times 61 \times 97$) on T3E.

No. of processors	Execution time
4	16 331
8	8713
16	4853
32	2919
64	1659
71	1680
90	1688
98	1127
128	1148

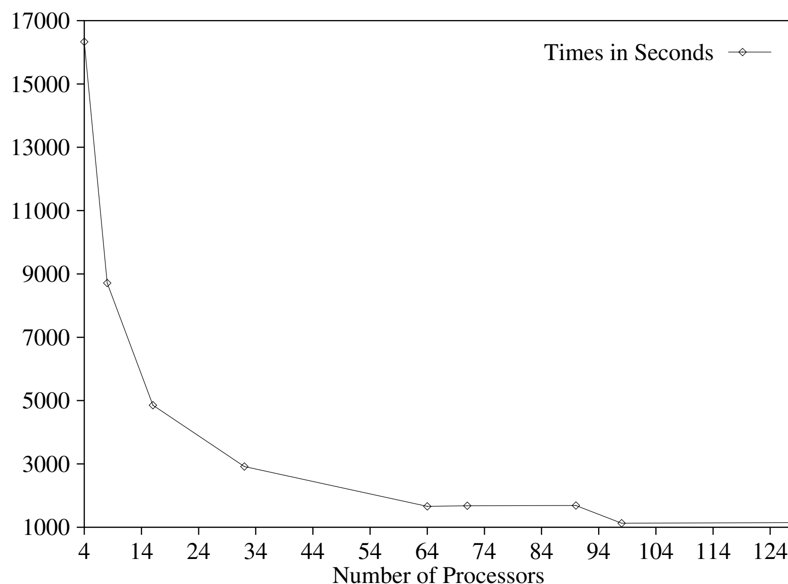
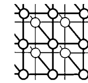


Figure 2. Execution times of the HPF code.

code. As shown in Figure 1, all the subroutines are scalable up to 98 processors. When using 128 processors, no further reduction in time is achieved. This phenomenon will be explained later. It is also clear that the subroutine SUM takes the most time compared to other subroutines. Due to memory size, the program cannot be executed on a T3E with fewer than four processors.

Table II shows the total execution times for the code after loop transformations and hand parallelization. Because of the limited memory space available on each processor in the T3E system,

Copyright © 2003 John Wiley & Sons, Ltd.



the parallel program could be run on a T3E with one or two processors. Hence, Table II only shows the execution times using at least four processors. Due to the requirement of the HPF compiler we used, many huge arrays had to be initialized before they were used and, hence, a lot of overhead is introduced. Therefore, the execution time on the T3E with four processors is longer than that for the Origin 2000 with four processors. However, the HPF code on the T3E is more scalable than the PA code on the Origin 2000 [23]. From Figure 2, we can also see that the execution time reaches its minimum when 98 processors are used. Clearly, the parallel code not only reduces the total execution time drastically but it is also scalable up to 98 processors on the T3E system.

We can also see that the time is not reduced any further when we increase the number of processors from 64 to 71 or 90. This is natural since we parallelize the code on dimension K which has a size of 97; 71 or 90 processors are not enough to partition the array on dimension K. Due to array padding, the actual size on dimension K is 98 and hence has a perfect partition over 98 processors. Further increase in the number of processors will not be effective unless we can distribute over more than one array dimension.

Because the timing information is not available for the T3E using only one processor (due to memory space limitation, we cannot run the code on one processor), we cannot calculate the speedup exactly. In order to estimate the speedup, we assume that the program is executed optimally on the T3E with four processors. This assumption is quite reasonable since experiments indicate that the MAX3D code runs efficiently with a small number of processors. With this assumption, we can estimate the execution time on the T3E with one processor by multiplying the execution time on the T3E with four processors by four. Table III shows the speedups of the HPF code after loop transformations and hand parallelization using this assumption. The corresponding data are also shown in Figure 3. It is clear that the code is still quite scalable when the number of processors used is 98.

For comparison purposes, we also include the speedups of the same code running on an IBM SP2 machine using MPI in Figure 3. The MPI code uses a three-dimensional decomposition scheme [13] and hence the amount of data exchange is smaller than with the HPF code which basically uses one-dimensional decomposition (parallelizing one loop only). Because of the different architectures and memory sizes, it is hard to compare the execution times of the T3E and IBM SP2. However, we can still use the speedup performance to compare the scalability of the two implementations. It is clear that the speedups are similar when the number of processors used is less than 64. As the number of processors increases, the MPI code shows better scalability than the HPF code. This phenomenon can be explained easily. Here, we are distributing only one array dimension of size 98. Therefore, for any number of processors between 49 and 97, one or more processors stores two elements in the distributed dimension, resulting in load imbalancing. In contrast, the MPI code uses a three-dimensional decomposition, so the number of elements per processor, and hence the computational load per processor, can vary more smoothly with the number of processors (it is better ‘load balanced’), giving a smoother speedup for MPI.

The experiments conducted in this research are limited to a fixed problem size ($73 \times 61 \times 97$). When the problem size changes, the scalability and efficiency may also change. Theoretically, when the problem size is bigger, the ratio of computation over communication also increases. This implies that the communication overhead becomes relatively smaller. It is our conjecture that the parallel code will be more scalable and efficient when computing larger problems. However, more experiments are needed to confirm our conjecture.

Copyright © 2003 John Wiley & Sons, Ltd.

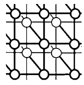


Table III. Speedups for problem size (73 × 61 × 97).

No. of processors	Speedup
1	1
2	2
4	4
8	7.49
16	13.46
32	22.38
64	39.38
71	38.88
90	38.70
98	57.96
128	56.90

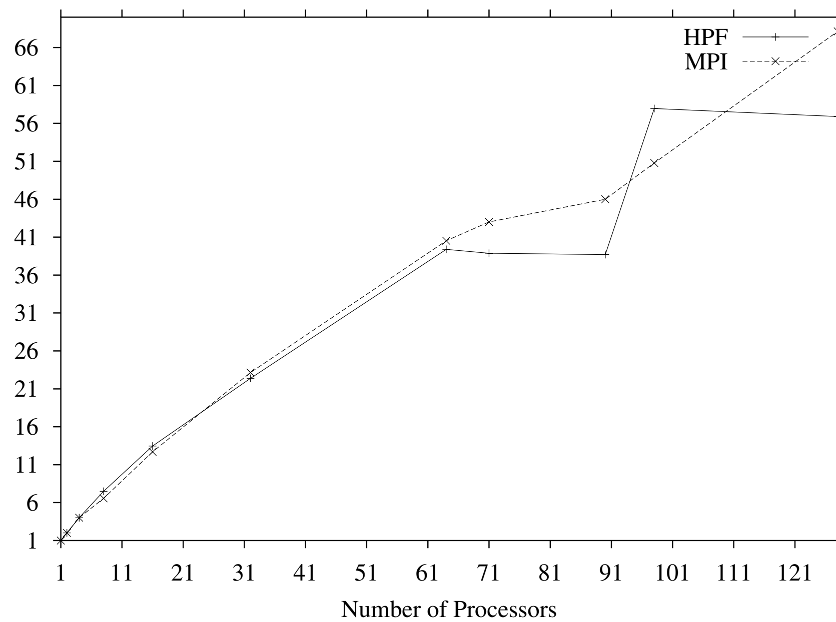
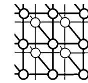


Figure 3. Comparison of speedups using HPF on T3E and MPI on SP2.

Copyright © 2003 John Wiley & Sons, Ltd.



5. CONCLUSIONS

As we all know, a low-level message-passing programming language such as MPI has several disadvantages compared with a high-level parallel programming language: the cost of producing a message-passing code is usually much higher, the length of the code grows significantly and it is much less readable and maintainable than the one produced using a high-level parallel programming language such as HPF. For these reasons, it is widely agreed that a higher level programming paradigm is essential if parallel systems are to be widely adopted. HPF has reached a critical stage in its history. Having struggled while the compiler technology evolved into a usable state, the parallel computing community has now found it possible to write portable, high-performance implementations for selected applications in HPF. HPF is becoming a standard for high-level parallel programming and is available on almost every major parallel computer system. This research indicates that large-scale data-parallel applications such as CEM simulations can use HPF to achieve a reasonable performance.

This research addresses the portability and scalability problems of the MAX3D code through using HPF. Our results achieved during the research have demonstrated that the sequential MAX3D code can be parallelized quickly by simple directives insertion and yet the code produced this way can still be executed efficiently. Furthermore, the code is still quite scalable. While the MPI code is more scalable than the HPF code due to different data distribution schemes (3D versus 1D) when using a larger number of processors, the labor cost of producing the MPI code is much larger than the HPF code. As new versions of commercial HPF compilers emerge in the market, and the HPF technology becomes more mature, we expect that the performance of the HPF MAX3D code will be improved substantially in the near future with little change in the code. We believe that high-level parallel programming languages such as HPF is the future for fast implementation of parallel codes.

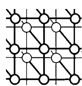
ACKNOWLEDGEMENTS

This research was supported by the Air Force Office of Scientific Research under grant F49620-93-C-0063. Additional support has been provided by the National Science Foundation under grants CCR-9211621, OSR-9350540, CCR-9503882, and ECS-0010047, the Air Force Avionics Laboratory, Wright Laboratory, under grant F33615-C-2218, and the Japan Society for the Promotion of Science Grant-in-Aid Basic Research (C) No. 14580386. We would also like to thank Mark Young of Portland Group for his help in using PGI's High Performance Fortran compiler pghpf. Computational resources for the work presented here were provided by the Ohio Supercomputing Center, Columbus, Ohio. Many thanks also go to the three reviewers and Professor Mark Baker for their constructive comments and suggestions which improved our paper greatly.

REFERENCES

1. Shang JS. A fractional-step method for solving 3-D time-domain Maxwell equations. *31th Aerospace Sciences Meeting and Exhibit*. American Institute of Aeronautics and Astronautics: New York, 1993.
2. Shang JS. Characteristics based methods for the time-domain Maxwell equations. *29th Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics: New York, 1991.
3. Shang JS, Calahan DA, Vikstrom B. Performance of a finite volume CEM code on multicomputers. *Computing Systems in Engineering* 1995; **6**(3):241–250.
4. Shang JS, Fithen RM. A comparative study of characteristic-based algorithms for the Maxwell equations. *Journal of Computational Physics* 1996; **125**:378–394.
5. Shang JS, Gaitonde D. Scattered electromagnetic field of a re-entry vehicle. *Journal of Spacecraft and Rockets* 1995; **32**(2):294–301.

Copyright © 2003 John Wiley & Sons, Ltd.



6. Shang JS, Gaitonde D. Characteristic-based, time-dependent Maxwell equation solvers on a general curvilinear frame. *American Institute of Aeronautics and Astronautics Journal* 1995; **33**(3):491–498.
7. Shang JS, Gaitonde D, Wurtzler K. Scattering simulations of computational electromagnetics. *27th AIAA Plasmadynamics and Lasers Conference*. American Institute of Aeronautics and Astronautics: New York, 1996.
8. Shang JS, Gaitonde D. High-order finite-volume schemes in wave propagation phenomena. *27th AIAA Plasmadynamics and Lasers Conference*. American Institute of Aeronautics and Astronautics: New York, 1996.
9. Shang JS, Gaitonde D. On high resolution schemes for time-dependent Maxwell equations. *34th Aerospace Sciences Meeting and Exhibit*. American Institute of Aeronautics and Astronautics: New York, 1996.
10. Shang JS, Scherr SJ. Time-domain electromagnetic scattering simulations on multicomputers. *26th AIAA Plasmadynamics and Lasers Conference*. American Institute of Aeronautics and Astronautics: New York, 1995.
11. Snir M *et al.*. *MPI: The Complete Reference*. MIT Press: Cambridge, MA, 1996.
12. Gropp W, Lusk E, Skjellum A. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. MIT Press: Cambridge, MA, 1994.
13. Shang JS, Wagner M, Pan Y, Blake DC. Strategies for adopting FVTD on multicomputers. *IEEE Computing in Science and Engineering (formerly known as IEEE Computational Science and Engineering)* 2000; **2**(1):10–21.
14. Hogue C. *MIPSpro (TM) Power Fortran 77 Programmer's Guide*. Silicon Graphics. Inc., 1996.
15. Pan Y. Improvement of cache utilization and parallel efficiency of a time-dependent Maxwell equation solver on the SGI Origin 2000. *Final Report for AFOSR Summer Faculty Research Program, Air Force Office of Scientific Research*, Bolling Air Force Base, DC, August 1997.
16. Miles D, Schuster V, Young M. The PGHPF high performance Fortran compiler: Status and future directions. *2nd Annual HPF User Group Meeting*. Springer: Berlin, 1998; 25–26.
17. Shang JS, Gaitonde D. Characteristic-based, time-dependent Maxwell equations solvers on a general curvilinear frame. *American Institute of Aeronautics and Astronautics Journal* 1995; **33**(3):491–498.
18. Anderson WK, Thomas JL, Van Leer B. A comparison of finite volume flux splittings for the Euler equations. *American Institute of Aeronautics and Astronautics Journal* 1986; **24**(9):1453–1460.
19. Koelbel CH. *The High Performance Fortran Handbook*. MIT Press: Cambridge, MA, 1994.
20. Guo M, Yamashita Y, Nakata I. An efficient data distribution technique for HPF compliers on distributed memory parallel computers. *Transactions of Information Processing Society of Japan* 1998; **39**(6):1718–1728.
21. Wolfe M. *High Performance Compilers for Parallel Computing*. Addison-Wesley: Reading, MA, 1996.
22. Guo M. A denotational semantic model of an HPF-like language. *Proceedings of The First International Conference on Parallel and Distributed Computing, Applications and Technologies*, University of Hong Kong, May 2000; 1–8.
23. Pan Y. Parallel implementation of computational electromagnetics simulation using high performance Fortran. *Final Report for AFOSR Summer Research Extension Program*, Air Force Office of Scientific Research, Bolling Air Force Base, DC, November 1998.

Copyright © 2003 John Wiley & Sons, Ltd.