ACES JOURNAL, VOL. 13, NO. 2, JULY 1998, SI: CEM & HPC

Optimizing the Parallel Implementation of a Finite Difference Time Domain Code on a Multi-user Network of Workstations

J.V. Mullan, C.J. Gillan and V.F. Fusco

The High Frequency Electronics Laboratory
Dept. Electrical and Electronic Engineering
The Queen's University of Belfast
Ashby Building, Stranmillis Road, Belfast
N. Ireland BT9 5AH, UK

Abstract

The implementation of a parallel, three dimensional, finite difference time domain (FDTD) computer program is considered and applied to a test scattering problem on a multi-user network of desktop workstations. The computation has primarily been done on a local area network (LAN) using six identical HP 9000/715 workstations (i.e. a homogeneous environment) with the Parallel Virtual Machine (PVM) software being employed as the communications harness.

In this paper the sequential and parallel FDTD approaches are reviewed. We investigate the factors which cause a reduction in efficiency in the latter, such as host allocation and load balancing. We propose a task migration process, which is efficient for the FDTD algorithm, as a partial solution. The advantages of this approach are discussed and further developments based on available computational resources are suggested.

Introduction

The finite difference time domain method (FDTD), as formulated by Yee[1], is now a well established, numerical and storage intensive, approach to solving Maxwell's equations for the electromagnetic field. The FDTD method has been employed, for example, in the design of microwave circuits[2], in radar cross section prediction[3] and in antenna design[4]. Presently, the method is being used extensively in investigations of biological interaction with electromagnetic fields[5]. The modelling of the fields radiated by a mobile telephone, when it is close to the human head, is a very active area of research at this time. Not least due to the complexity of human anatomy and physiology, the latter interaction necessitates that a parallel algorithm be used in order to solve the FDTD equations within a reasonable amount of time with available resources.

The *non-sequential* FDTD method has been investigated on a variety of hardware including transputers[6, 7], vector computers[8], massively parallel processors[8, 9] and networks of workstations[10, 11]. Excluding vector computation, the general strategy has been to apply a straightforward geometric decomposition of the total volume, an approach that is also employed in other areas of technical computing, for example molecular dynamics[12]. In order to achieve optimum geometric decomposition, the topology of the connections between the processors working concurrently on the problem must map the geometry of the data-domain that the problem is defined upon.

In this paper we follow the approach of Chew and Fusco, this time operating on a network of HP9000/715 workstations with the aid of the Parallel Virtual Machine (PVM) software[13]. The test problem simulates

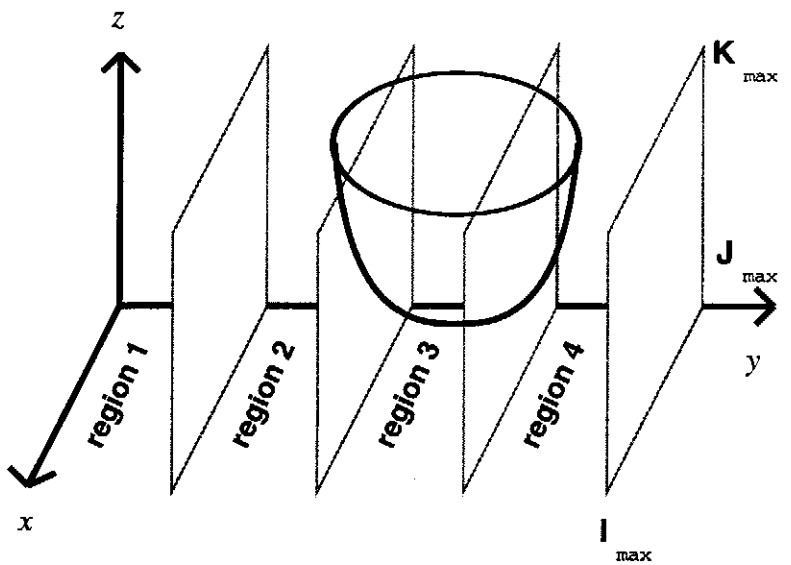

Figure 1: Segmentation of solution region into N subspaces by partitioning the grid along one dimension. In this case, the y-axis is partitioned and $N = 4$.

the scattering of a 2.5 GHz sinusoidal plane wave by a lossless dielectric sphere ($\epsilon_r = 4.0$), where the rest of the computation space consists of air. Timestepping continues until steady state conditions are achieved in each Yee cell; owing to symmetry only one quarter of the problem needs to be evaluated. This simple test case was used so that the efficiency of this method could be assessed and optimized before being used on more complex and useful calculations.

The sequential and parallel FDTD Algorithms

Other papers in this volume present the FDTD algorithm in considerable detail. However, for the sake of completeness, we give the salient details for a sequential implementation of the method here in order to facilitate an explanation of the way in which this is enhanced to achieve a parallel implementation.

The finite difference approach to solving Maxwell's equations for the electromagnetic field, as a function of position and time, approximates the infinite space-time continuum by a discrete three dimensional spatial grid of finite extent on which the electromagnetic field components are updated at successive, discrete time steps, that is on a temporal grid. The partial derivatives occurring in Maxwell's equations are approximated by difference equations defined with respect to the spatial and temporal grid. Yee's[1] approach is to treat Maxwell's curl equations as a pair of coupled, first order, partial differential equations and uses a specific griding arrangement, motivated by simple electromagnetic principles, which makes the difference approximations accurate to second order. Although this it is by no means the only solution strategy, or choice of grid structure, the Yee method quickly became a de-facto standard in computational electromagnetics[8].

The Yee algorithm is expressed as a loop over a finite number of timesteps where the work done at each timestep is as follows:

1. Update the E-field components, a task which uses only previously computed E and H-field components. The update, at each point on the spatial grid, is therefore *independent* of updates at other points on the spatial grid, for this timestep, and requires only field values from adjacent Yee cells due to the

form of the finite difference equations.

2. Augment time by one half of a time step.

3. Update the H-field components, a task which uses the E-field components computed in 1 above as well as only the previously computed H-field components. Again, the update, at each point on the spatial grid, is *independent* of updates at other points on the spatial grid, for this timestep, and requires only field values from adjacent Yee cells due to the form of the finite difference equations.

4. Apply outer radiation boundary conditions to the surfaces of the finite volume in order to compensate for truncation of the infnite spatial continuum. The boundary condition is applied in a point by point fashion with the computations at each point being independent but requiring results from step 1 above.

Clearly, the Yee algorithm exhibits a high degree of concurrency, a feature that cannot be fully exploited on a single processor computer even with vectorization. Theoretically, a multi-processor computer with enough processors could update the E or H field accross the entire spatial grid in the time taken to update at one spatial point, providing of course that the computer had enough memory to simultaneously hold all field values in core (see, for example, the work by Davidson *etal* [9] on the implementation on the FDTD method using 8192 processors on a Connection Machine, model CM-2).

In practice, geometric decomposition is used to partition the grid into subspaces each of which is assigned to one processor. Due to the form of the finite difference equations, it is necessary to exchange data between processors, in a distributed or non-uniform memory access environment, at the end of each time step; this is needed in order to update field values along the interfaces. This corresponds, physically, to a travelling wave propagating across the total volume. In figure 1 we show the way in which we have partitioned the total volume among four processors; in general there may be N such processors. This is consistent with the linear topology of the 10 Base-2 thinwire ethernet (10 Mbits/second) which connects together the HP workstations in our laboratory.

In an attempt to obtain (static) load balancing and hence synchronization of the tasks we make sure that each processor is given an equal number of points to handle, meaning that the number of points along the Y-axis ($J_{\max} + 1$) must be an exact multiple of the number of processors. This is a reasonable assumption for a *parallel virtual machine* consisting of a homogeneous network of *free* workstations. By *free*, we mean that no one is logged onto the workstation or a negligible amount of work is being performed, in other words the workstation is lightly loaded. The programs which update fields in the first and last regions (subspaces 1 and 4, respectively in figure 1) require boundary conditions to be obeyed but are otherwise identical to the program used in all other regions. The extra time consumed in applying the boundary conditions is insignificant compared to the time spent updating fields within the volume and does not therefore introduce a load imbalance into the virtual machine. These programs have been named, for obvious reasons, *first*, *last* and *middle*.

During a single timestep, a host must:

1. transmit/receive the E-field components from the previous time set-up (evaluated at the boundary between subspaces);

2. compute H-field values in the subspace;

3. transmit/receive the H-field components (in the opposite direction to the E-field values);

4. compute the E-field values in the subspace.

This process is continued until a steady state is achieved. The transmission and reception of boundary field values, at each timestep, is illustrated schematically in figure 2, for an arbitrary *middle* host called X. The *first* and *last* hosts also transmit and receive boundary field values but only half as many as a *middle* host. If figure 2 were redrawn for the *first* host, then the lefthand side would be missing, while a similar figure for the *last* host would omit the righthand side.

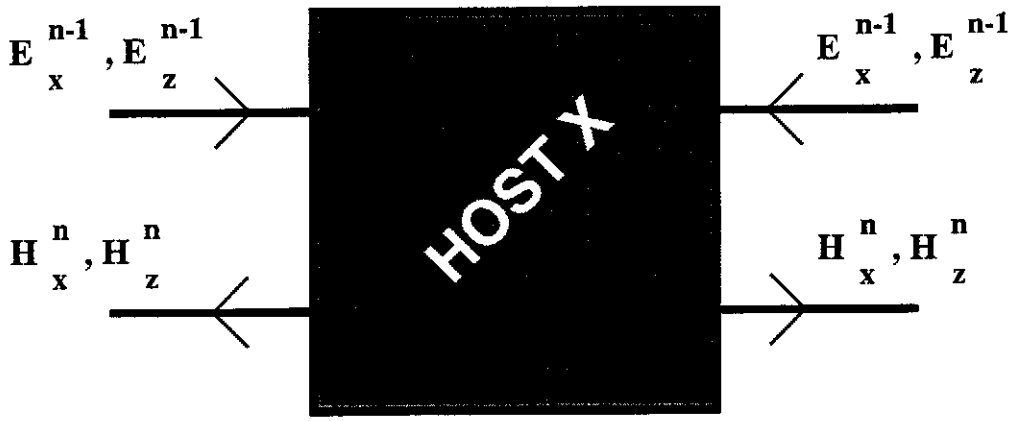

Figure 2: Communications of the electromagnetic field components for one host during the nth timestep; the geometric domain has been partitioned along the Y-axis only.

Message Passing Harnesses

Even today, many scientific programmers develop sequential codes in a top-down fashion with the Von-Neumann concepts of computer architecture in mind. Notwithstanding the fact that such codes are often restructured to take advantage of compiler optimizations exploiting hardware features such as pipelining, chaining and, perhaps, vectorization, the approach taken by scientific programmers has been largely to rely upon third party utilities/libraries to enhance their code performance; one example is the basic linear algebra system (BLAS). As long as these libraries are available on an architecture, codes dependent on the libraries are assured of high performance on that architecture. Consequently, in the era of parallel and distributed computing a number of *message passing harnesses* appeared obviating the need for scientific, particularly Fortran, programmers to learn about network programming. Message passing harnesses first appeared on transputers towards the late eighties, but with the adoption of the Internet Protocol (IP) as a defacto global standard in the early ninties harnesses quickly became available for UNIX workstations, and even UNIX mainframes, connected by IP networks.

The novice should appreciate that message passing harnesses are simply a layer of software that sit between an application program and operating system interfaces. It is the within the operating system that the physical transmission and reception of data takes place just as data is written to and read from disk. Sometimes the message passing layer is thin and provides very little abstract functionality, sometimes the converse is true. Related to the rapid growth in the use of message passing is the fact that the Berkeley Standard Unix distribution (BSD) implemented and standardized an abstract data construct known as a socket. In this model, the network hardware, the CPU hardware, their mutual interaction, the specifics of the protocol and the role of the operating system are all encapsulated within *socket* routines and thereby hidden from a socket programmer. This means that network programming becomes a matter of manipulating sockets and data buffers in real time, a task easily accomplished in the C language and frequently referred to as IP socket programming. With the standardization of the IP socket programming model, implementations of sockets have become available for all operating systems in common use. On a UNIX workstation, and on a PC, message passing harnesses are simply an interface between an application program and socket routines therefore.

In the late eighties and early ninties there was a proliferation of message passing harnesses, each providing different functionality and therefore incompatible. In the physical sciences and engineering communities PVM was seen as a de-facto standard and was, and continues to be, widely used. Many computer manufacturers

now provide implementations of PVM which are highly optimized for use on their architecture just as they do also for BLAS routines. To date, there is no ANSI standard for message passing harnesses, however beginning in 1992 around forty organizations created a working group to propose, but not to implement as such, a 'standard' for message passing; this effort was known as the Message Passing Interface forum and the standard created is known as MPI. Vendors of massively parallel processors support the MPI standard on their own particular architectures. for example SGI/Cray on the Cray T3E; a number of implementations of MPI have also appeared for workstations, for example CHIMP: the common high level interface to message passing from the Edinburgh parallel computer centre[14]. In this article, we have chosen to focus only on PVM deferring consideration of other message passing architectures for future work.

Latency in PVM

The costs inherent in using PVM for our FDTD calculation are twofold, regardless of the computing environment used. Initially, there is a fixed set up cost due to the fact that the *first* process must spawn the *last* process and several copies of the *middle* processes and prepare them for subsequent FDTD timestep updates. This cost can be amortized over subseqent timesteps such that if there are very many timesteps, the setup costs become insignificant. The second, and critical, limitation is the cost of communicating boundary field values coupled to process synchronization, because this has to be done at every timestep and is therefore an increasing cost; in the event that this cost, within one timestep, exceeds the reduction in computation time for that timestep, derived from domain decomposition, then the distributed FDTD implementation will always require more time than its sequential analogue.

In simple terms, the time consumed in updating the electromagnetic fields in a subdomain depends on the total number of points in the volume and on the CPU speed; part of this time may be spent applying boundary conditions at the surface of the volume. Correspondingly, the time consumed in transferring boundary field values depends on the surface area of the volume and on the transmission time for a message between processes. Aside from the limitations of network hardware, the PVM system itself has an associated latency just as any message passing system does; the reader should appreciate that the design of a complicated utility such as PVM requires trade-offs between reliability, portability and efficiency. A rudimentary discussion of PVM, and a trivial implementation, can be found in Robbins and Robbins[15].

PVM uses a *spoke and hub* system akin to that employed in the airline transport industry. On each processor there is a PVM daemon (*pvmd*), that is a hub, which communicates directly with PVM tasks running on that processor using either TCP/IP sockets or Unix domain style sockets. The PVM daemons, on each processor, communicate with each other using UDP/IP sockets. A message from a task on host A destined for a task on host B, travels first to the pvmd on host A, then to the pvmd on host B and finally to the task on host B. Enroute there is some buffering through send and receive queues within the pvmds; more critically, the UDP/IP protocol is an unreliable delivery protocol requiring PVM to implement message fragmentation and an acknowledgement/retry mechanism to ensure message delivery. It must be pointed out that PVM does permit direct task to task communication using TCP/IP sockets and thereby avoiding the pvmds; this option is mentioned in the PVM user manual with the corollary that it does not scale well, therefore we have not used it in our present work.

Initial Results

The effectiveness of the parallel FDTD code was gauged by defining the speed-up, S, as the ratio of the execution times for the equivalent serial code (T_{ser}) to the parallel one (T_{par}), that is:

$$S = \frac{T_{\text{ser}}}{T_{\text{par}}}.$$

In the first instance, the network was not dedicated to these tasks. One can see, from figure 3, that the speed up for a given number of processors increases with the number of grid elements, that is with a decreasing mesh size, although there are some occasions where the speed-up drops below what one would expect. An example of this behaviour is the case of five processors with the finest mesh ($\delta = 1.48$ mm). The speedup

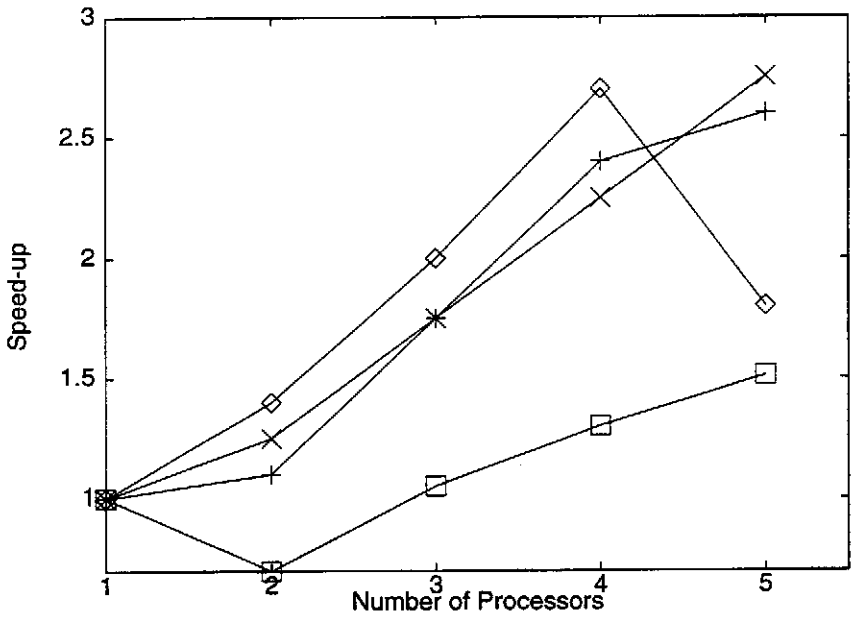

Figure 3: The speed-ups for up to five processors when the network was not dedicated to the PVM tasks. $\diamond$ points are for a mesh size of 0.00148 metres, $\times$ for a mesh size of 0.0017 metres, $+$ for a mesh size of 0.003 metres, $\square$ for a mesh of 0.00616 metres.

would appear to far from optimal with $S = 1.86$. It was speculated that another user had begun a job on one of the hosts and so the run was repeated with the all workstations reserved for the duration of the test; results are shown in figure 4. For the particular case just mentioned, a much improved speedup, $S = 3.45$, was found. The approximate factor two increase in the speedup, that is a halving of the execution time, reinforces the assumption that two jobs sharing a host was the cause of the initial poor result. The speedup of $S = 3.45$ using five processors, with dedicated use of the network, is clearly the upper bound for this particular problem. We have shown that this is an unlikely figure to achieve under typical network conditions and have quantified the dramatic reduction in efficiency that can result from having just one host otherwise occupied. This is a striking reminder of the fact that desktop platforms, that is UNIX workstations and PCs, were never really designed as multi-user platforms to replace minicomputers or even mainframes. Unfortunately, faster CPU speeds have masked this problem somewhat. Simple economics combined with the fact that the operating systems in use on the desktop use premeptive multi-tasking has led, by default, to use of desktop platforms as minicomputers and even mainframe replacements however.

Surprisingly, another drop in efficiency was found, see figure 4, when the mesh corresponding to ($\delta = 1.70$ mm) was distributed over five processors. Following further checks, it was found that although six processors were configured, the PVM application takes one host to use as a group server when the programs include dynamic groups. Furthermore, the default host selection is through a *round robin* process which places a spawned task onto the next workstation in the host file. It was found that this method did not guarantee that each task was placed on a separate host giving rise to uncertainty in the obtainable speedup on a given run. Our method used the `PvmTaskDefault` option on the *pvmfspawn()* call, a feature that can be replaced by the `PvmTaskHost` option requiring a specific host to be given. The *pvmfconfig* and *pvmftasks()* routines provide the facility to interrogate the virtual machine and so discover, for example, which host is acting as the group server and which hosts are not yet used for PVM tasks. We could use these routines to make the code adapt to the tasks already present on the virtual machine, however we have not investigated this solution because it is encapsulated, by default, into the dynamic loadbalancing solution that we suggest in the next section.

It is noticable in figures 3 and 4 that the speedups for the three finest meshes, 0.00148, 0.0017 and 0.003 metres, are clustered together and well separated from the data for the most coarse mesh, 0.00616. We

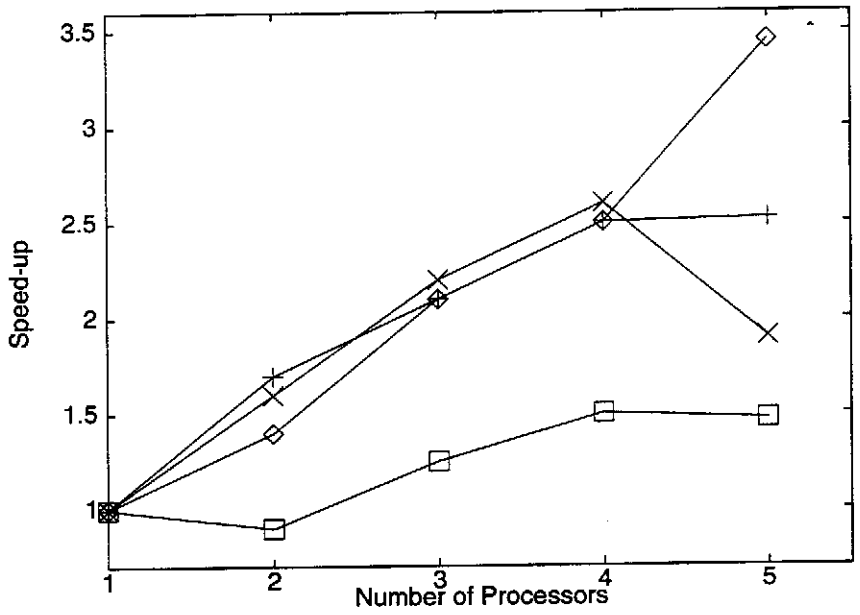

Figure 4: The speed-ups for up to five processors when the network was dedicated to only the PVM tasks. As in figure 3, $\diamond$ points are for a mesh size of 0.00148 metres, $\times$ for a mesh size of 0.0017 metres, $+$ for a mesh size of 0.003 metres, $\square$ for a mesh of 0.00616 metres.

performed a calculation for the intermediate mesh size of 0.00403 metres and found that the speedup values clustered with the 0.00616 metre case. The overall trend is, as mentioned above, that a finer mesh benefits more substantially from concurrent computations however the increase is not linear. We believe that this reflects performance characteristics of the data buffering that is part of the any message passing process, coupled perhaps to the paging of virtual memory by the operating system. An investigation of the buffering issue could be achieved by using small RISC assembler routines to write those sections of the code concerned with moving blocks of field data around; unfortunately, we have have not had time, or resources to do this as yet.

For problems which require a large number of geometric grid points, the memory requirements can often be a limiting factor in performing an FDTD computation; under these conditions optimum CPU performance is a only secondary issue relative to actually solving the problem at all. On a single processor, or even on a massively parallel processor, the addition of extra memory to the system is generally not an available option for a user. Distributed computing has the distinct advantage that to obtain more memory one need only find a lightly loaded, networked workstation, or a networked PC, within one's organization. Typically, there are many such machines available, even if only during the night and at weekends.

Improving the Effectiveness of the Computation

In order to improve the predicability of execution times of the distributed algorithm, we decided to assess the number of executing processes on each host using the well known UNIX command *uptime*[16], which provides a one line summary of the number of users and the average load on the processor for the previous five, ten and fifteen minutes; unfortunately, this approach proved to be ineffective in our circumstances. Some workstations in our laboratory act as file servers holding various commercial EM simulation packages; even when a simulation is being performed on another machine, the file server consumes CPU cycles and appears loaded. A more useful UNIX command is *top* which tabulates running processes and gives the percentage CPU consumption associated with each. This, of course, produces several lines of output which must be analysed to find the appropriate information, a task that was accomplished by using a UNIX C

shell script.

This script was successful in selecting the lightly loaded hosts prior to FDTD execution. In addition to configuring the optimum virtual machine, the script compiled and ran the program on the available hosts. The code was modified so that each task was spawned onto the next host in the configuration so that all available hosts were used even though we use dynamic groups. This ensured that one, and only one, task was given to each workstation and thus helped to ensure that the impairment in effectiveness, described in the previous section, was eliminated. This improvement is of limited use, however, because in the duration of execution, the load on each of the networked workstations is likely to change due to the presence of other users. It was imperative to allow for the transient nature of the load, so methods of combating the negative effects of a multi-user system were considered.

The power of a portable parallel environment such as PVM lies in the fact that computational resources can be utilised which would be otherwise idle. However, to convince the owner of a private workstation that it should become part of a larger, virtual machine it is often necessary to ensure dedicated use of that host when the user requires it. It was with this scenario in mind that the following task migration scheme was developed. The migration of a complete task from a heavily loaded host onto a lightly loaded one is a more severe option than obtaining load balancing through load redistribution but it is easier to achieve.

In the virtual machine one has, in general, one incidence of the 'first' and one incidence of the 'last' program executing but several incidences of the 'middle' program executing. In order to simplify matters, only the incidences of the 'middle' program were condidered as candidates for migration. To enable the scheme, a fourth program source, 'middle-migrated' was created; this was designed a little differently from the 'middle' program so that it could receive data defining the calculation in the midst of a loop over timesteps and then continue executing from some arbitrary timestep.

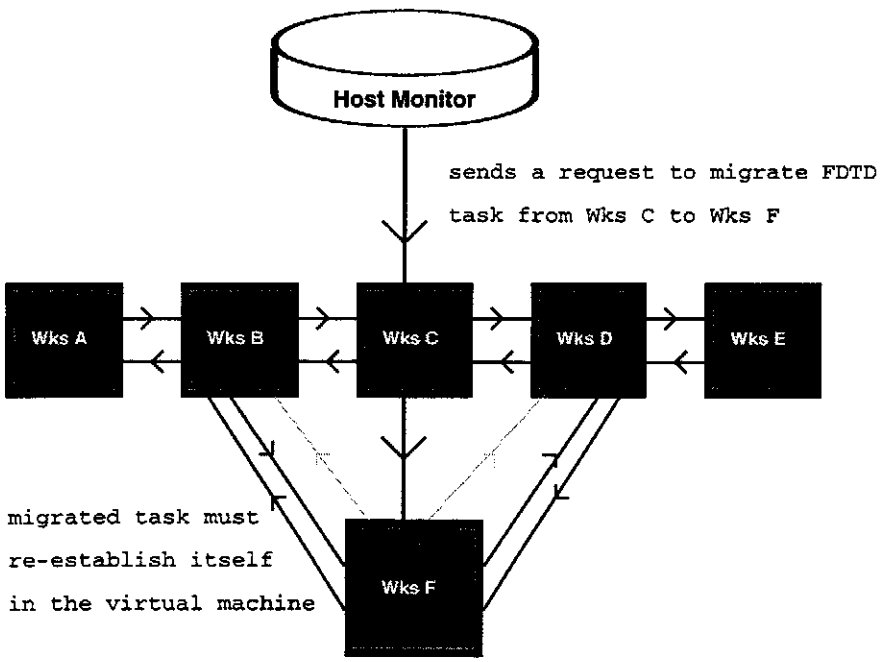

Figure 5: Schematic illustration of the communications involved in a task migration from workstation C to F. F must establish itself with workstations B and D as a nearest neighbour node.

At the start of an FDTD calculation, 'first' FDTD program spawns a *monitor* program after the FDTD calculation has been started, that is after some appropriate number of timesteps (defined by the user). The monitor then runs independently of the FDTD codes and continually assesses the status of each configured host. The monitor program is itself a master process controlling several slaves. In this case the slaves are

UNIX C-shell scripts which are spawned on each workstation in the virtual machine in turn. The scripts execute the *top* command with the output being directed to a file; this file is edited, using *sed* amd *awk* within the script finally producing a disk file which contains some integer data related to the load on the processor. This data is subsequently read from the file by the *monitor* program, a feature that depends on the network file system (NFS) that is in use in our laboratory (using NFS means essentially that all workstations share a common disk area). The efficiency of using NFS as a data sharing mechanism has been illustrated for image processing calculations using a network of SUN SPARC10 workstations[17].

A task migration is effected when the monitor program detects an overloaded machine and prompts the offending task to relocate on a host deemed to be free. The decision criteria programmed was that

> when two jobs are running on a host and one is an FDTD executable, then the FDTD process should be migrated off that host.

This was moderated by the condition that migration was not carried out until an FDTD executable was found to be in need of migration on three sucessive occasions. This condition was needed in order to prevent spurious migrations due to very short lived processes that might be present in the virtual machine. In figure 5 we show, the sequence of events that takes place when a task migration is effected.

In a non-migrating FDTD calculation, the communications among the various tasks are synchronous. In the load balancing situation several communications become asynchronous in nature, for example,

- between the monitor and the 'middle' programs.
- between a terminating task and the *migrated* one that replaces it.
- at the termination of the monitor program, something which takes place when the 'first' program reaches the end of the loop over timesteps, a message is passed asynchronously from the 'first'program to the 'monitor'program.

We have enabled these communications with the *pvmfprobe()* routine. A key issue was whether, or not, the extra probes and barriers used in the task migration scheme actually affected execution times. At first a probe and a barrier were used on each time step as it was felt that for computationally intensive FDTD applications these would have negligible effect, in other words the time spent updating electromagnetic fields was far in excess of that used for task synchronization. For coarse grids, or substantially faster CPUs, this would not be the case but then the distributed algorithm is ineffective for those anyway. The migrating and non-migrating FDTD codes were run for fifty timesteps with a selection of grid sizes; the execution times are compared in table 1. As expected, the more the computation involved in each time step, the smaller

Grid spacing (δ) in metres	Migrating Code	Non-migrating Code	Increase in the execution time
0.00616	5.82	2.12	2.74
0.00198	41.47	21.29	1.94
0.00148	88.66	54.18	1.64

Table 1: Comparison of execution times for the migrating and non-migrating FDTD codes, discussed in the text, for various sizes of mesh. All times are given in seconds.

the effect of the probes. The effect is still excessive for the finest mesh and this would put into doubt the advantage of parallelizing the FDTD algorithm at all. It is clear, upon reflection, that probing on each timestep is overkill and every ten steps would be a much more reasonable approach. Initial tests show that this produces an efficient task migration procedure.

Conclusions

The performance of our distributed implementation of the FDTD algorithm has been found to be sensitive to the dynamic load on the *virtual machine* used. The presence of other active tasks on any one, or more, of the nodes used causes severe degradation in performance. To overcome this limitation, a dynamic loadbalancing strategy is used in which tasks migrate onto lightly loaded nodes. Notwithstanding any performance degradation, distributed computing provides an easily scalable memory environment, a feature which is generally not economic on massively parallel systems.

The full task migration process described in this paper is effective when enough workstations are available so that a free machine can be found. It is not so useful for a small number of host machines when it is likely that all available hosts will be employed at the start of an FDTD execution. A more useful approach in these conditions is to use data redistribution. It is envisaged that the above programs could be modified so that instead of ruling out a host with one non-FDTD process running, we could steal half of its available CPU. Thus a free machine would calculate x nodes while a semi-free machine would compute $x/2$ to achieve a load balance on the homogeneous network. The dynamic load-balancing would thus involve a reshuffling of the amount of work on all machines when one has to be altered. At first sight, this seems so much more involved than full task migration to be unfeasible. However, the large increase in work needed by the monitor will not be a problem as it can run virtually independent from the FDTD codes. The number of communications during a load redistribution is undoubtedly larger than in a full task migration. These data transfers will occur concurrently and thus be quicker. It should therefore be a relatively simple matter to extend the task migration to produce a very efficient load balancing tool. This would not be limited to a homogeneous network because it is based on standard UNIX and PVM commands.

Acknowledgments

This work was supported, in part, by the UK Engineering and Physical Sciences Research Council under contract GR/L23215.

References

[1] Yee K S 1966, "Numerical Solution of Initial Boundary-value Problems Involving Maxwell's equations in isotropic Media ", IEE Trans Ant. Prog. Vol: AP-14, May 1966 pp.302-307.

[2] Huang T W, Houshmand B and Itoh T 1993, "Microwave structure characterization by a combination of FDTD and system identification methods", *IEEE MTT-S Int. Microwave Symposium Digest*, vol. 2, pp. 793-796.

[3] Kunz K S and Lee K M 1978, "A three-dimensional finite-difference solution of the response of an aircraft to a complex transient EM environment I: The method and its implementation", *IEEE Trans. on Electromagn. Compat.* vol. 20, pp. 328-333,

[4] Chebolu S, Mittra R and Becker W D 1996, "The analysis of microstrip antennas using the FDTD method", *Microwave*, vol. 39, pp. 134-150.

[5] Ghandi O P, Sullivan D M and Taflove A 1988, "Use of the finite-difference time-domain method in calculating EM absorption in man models", *IEEE Trans. Biomedical Engineering* vol. 35, pp. 179-186

[6] Buchanan W J 1993, "Simulation of radiatin from a microstrip antenna using three dimensional finite-difference time-domain (FDTD) method." *Eighth International Conference on Antennas and Propagation*, IEE Conf. Pub. No. 370, pp639-642

[7] Chew K C and Fusco V F 1995 "A Parallel Implementation of the Finite Difference Time Domain Algorithm" *Intl. J. of Numerical Modelling:El. Networks, Devices and Fields* 8 293

[8] Taflove A 1995 *Computational Electrodynamics: The Finite Difference Time Domain Method*, (Boston: Artech House) ISBN 0-89006-792-9, pp545-84

[9] Davidson D`B, Ziolkowski R W and Judkins J B 1994, "FDTD modelling of 3D optical pulse propagation in linear and non-linear materials", *Second International Conference on Computation in Electromagnetics*, IEE Conf. Pub. No. 384, pp166-169

[10] Varadarajan V and Mittra R 1995 "Finite Difference Time Domain (FDTD) Analysis using Distributed Computing" *IEEE Microwave and Guided Wave Letters* 4 144-145

[11] Rodohan D P, Saunders S R and Glover R J 1995 "A Distributed Implementation of the Finite Difference Time-Domain (FDTD) Method" *Intl. J. of Numerical Modelling:El. Networks, Devices and Fields* 8 283

[12] Reale F, Bocchino F and Sciortino S, "Parallel Computing on UNIX Workstation Arrays" *Comp. Phys. Comm.* **83** pp. 130-140, 1994

[13] Geist G A, Beguelin A, Dongarra J, Jiang W, Manchek R and Sunderam V 1994, *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory Technical Report ORNL/TM-11616.

[14] Malard J 1995, *MPI: A Message Passing Interface standard*, Technical Report, Edinburgh Parallel Computer Centre.

[15] Robbins K A and Robbins S 1996, *Practical Unix Programming: A Guide to Concurency, Communication and Multithreading*, (Prentice Hall: NJ) ISBN 0-13-443706-3 p401-18.

[16] "Using HP-UX", HP 9000 Workstations, Hewlett Packard, Part No. B2910-9001, Edition 1, 1994.

[17] Ward G, "Parallel Rendering on the ICSD SPARC-10's"
The Radiance Synthetic Imaging System, http://radsite.lbl.gov/radiance/refer/Notes/parallel.html.