INTRODUCTION

With increasing frequencies and reduced grid sizes numerical schemes such as FDTD take an excessive amount of time to complete their computations [1]. Numerous efforts that have been put forth to speed up the execution of this algorithm by utilizing parallel processors [2-6]. The conventional approach [7] has been to partition the FDTD domain into equal-sized sub-domains whose boundaries are Yee cells, as shown in Figure 1.

Due to the nature of FDTD, in order to calculate the E field values on the boundary, the outer H field values have to be transmitted by both sides. This results in idle time in case one processor is faster than its neighbor and has to wait for the message. The partitioning can be done in 1, 2, or 3 dimensions. Every side that is on a boundary will have to communicate with the surrounding processor. The smaller the size of the partition, the faster will be the increase in the surface to volume ratio (S/V). This is a caveat of the parallel processing approach because the communication to computation ratio is proportional to the surface to volume ratio. At some point, this ratio becomes so large that further partitioning does not result in any more gain.

Fixed speed-up measures how much faster the algorithm runs as it is divided among processors, while the problem size remains the same. It is written as [1]:

\[ S_f(P) = \frac{s+p}{s+w(P)+p/P} \]  

where P is the number of processors, s+p is the total time to run the entire algorithm on just one processor, split up into the respective serial and parallel portions, and w(P) is the additional work required to make the algorithm run on P processors. It is clear that the greater the amount of serialism, the worse the performance of the fixed speed-up. Another useful metric is the parallel efficiency. It measures the percentage utilization of the available parallelism and is simply the speed-up S(P) divided by the number of processors, P.

IMPLEMENTATION

A sequential version of FDTD has been implemented by the research group at the University of Minnesota [8]. This code simulates an electromagnetic wave propagating in a waveguide. This is accomplished by a mesh with dimensions \(nx \times ny \times nz = 133 \times 20 \times 9\) unit cells³ and the solution is performed for 1000 time steps. We converted this code to run in parallel and implemented it on a Beowulf parallel
computing cluster. This was done with the aid of MPI [9], a runtime and a set of message passing routines for parallel machines. Our implementation splits up the domain into equal portions along the $x$ direction and assigns each portion to a processor. Commands such as Mpi_send and Mpi_receive are used to transfer messages between processors. These commands constitute a blocking message passing mechanism. A processor will not continue execution until Mpi_recv has been executed on the processor where the message has been sent with Mpi_send. This is a very restricted form of synchronization, especially if more than 2 processors are operating. Two messages must be sent by a processor at each interface. These messages are an array of all the $hy$ values and an array of all the $hz$ values at the boundary. The dimensions of this array are $ny \times nz$. These messages introduce additional delay because the message has to be a one-dimensional array. Therefore a loop is required on the sending side to transfer the boundary values from a three-dimensional to a one-dimensional array. The initiation of a message involves a setup delay no matter how small the message is. Also, there is a delay for a confirmation of message delivery. Therefore, the transfer of a message involves a minimum of three constant delays: setup time, time for a message to get to the receiver across the communication channel, and time to receive a confirmation of delivery.

Aside from the S/V ratio, another factor that contributes to the decrease in parallel efficiency with the number of processors is the idle time experienced by each processor. This idle time is due to the small differences in time that each processor takes to compute its region. If the surrounding processors are faster, then they have to wait to receive the magnetic fields, resulting in idle time. The versatility of the FDTD algorithm allows the processors to step forward in time for a limited number of steps without receiving the data from the boundary processors. With this approach the algorithm later has to come back, receive the data, and recalculate the omitted values. This situation is illustrated in Figure 2.

With the approach shown in Figure 2 there is now a window of time created which gives the two processors some lag before they will encounter idle time. The size of this window depends on how far in time the process in Figure 2a is allowed to go. This modification depends on the assumption that each processor has random variations in the time it takes to compute a cycle. If the variation is not random then eventually one of the processors will reach the end of the window and will result in idle time. Also it is assumed that precise load balancing is implemented. The situation is illustrated in Figure 3.

MPI has several mechanisms for communicating between processors, such as blocking and non-blocking modes. In the blocking regime, the effect of the slow processor has immediate consequences on the neighboring processors, whereas in the non-blocking regime the consequences become evident gradually. We modified our code to utilize some of the non-blocking features of MPI such as MPI_Isend and MPI_Irecv. The TEST subroutine is used to test the channel for received messages.

RESULTS AND CONCLUSION

As can be seen from Figure 4, the proposed method outperforms the ordinary one by 10-15 percent. After some threshold speedup efficiency is reached, further
partitioning will not result in any more gain. The reason for the drop in the speedup efficiency is the rate of change of the surface to volume ratio. The speedup can be improved further by using alternative partitioning techniques.

As can be seen in Figure 1, there is a slight overlap such that the boundary values of the electric field are calculated by both partitions. If the common electric field is calculated by only one side then that side must communicate it to its neighbor. This approach will reduce the computational steps performed but might be more difficult to conceptualize and implement. Now one side of the boundary will communicate the electric field values and the other one will communicate the magnetic field ones. The computation time will be affected whereas the communication time will stay the same.

A way to reduce the communication time at the expense of added computation time is to allow overlap of the partitions [10]. With overlapping partitions the update of field values has to be done when each partition reaches the middle of the overlap through the process shown in Figure 2a. At that point the uncalculated values are exchanged by the partitions. The larger the size of the overlap, the fewer times will the system have to stop for communication breaks. However, each partition will do more redundant work, so a trade-off must be made. This has an advantage in the sense that the expensive communication setup time is not invoked during every iteration. Another advantage is that computation time can be sacrificed since it is usually cheaper than communication time which gets more expensive with smaller partitions.

**REFERENCES**


Figure 1. Conventional Partitioning Approach

Figure 2. How omitted values propagate in the FDTD domain.
Only two dimensions are shown. (a) Planar projection of one unit cell in time. Thick lines indicate points where values were not calculated. Arrows indicate the time dependence of uncalculated values. The leftmost uncalculated value is the result of missing data from adjacent processor. (b) The effect on the entire partition. Only values inside the triangle (cone) are calculated.

Figure 3. Comparison of ordinary and proposed approaches. The data was obtained with Jumpshot, a performance monitoring package. The oval indicates computation time. Part (a) shows the idle time experienced by the lower processor. Part (b) shows how the idle time is eliminated by using the window in the proposed method.

Figure 4. Speedup Comparison of the Approaches Discussed.