Appendix C.

To be published in Proceedings of the 5th Distributed Memory Computing Conference, April 9-12, 1990

A 2D Electrostatic PIC Code for the Mark III Hypercube

R.D.Ferraro
P.C.Liewer
Jet Propulsion Laboratory / California Institute of Technology
4800 Oak Grove Dr.
Pasadena, CA 91109

V.K.Decyk
Department of Physics
University of California, Los Angeles
405 Hilgard Ave.
Los Angeles, CA 90024

FG03-84ER53173

Abstract

We have implemented a 2D electrostatic plasma particle in cell (PIC) simulation code on the Caltech/JPL Mark IIIfp Hypercube. The code simulates plasma effects by evolving in time the trajectories of thousands to millions of charged particles subject to their self-consistent fields. Each particle's position and velocity is advanced in time using a leap frog method for integrating Newton's equations of motion in electric and magnetic fields. The electric field due to these moving charged particles is calculated on a spatial grid at each time step by solving Poisson's equation in Fourier space. These two tasks represent the largest part of the computation. To obtain efficient operation on a distributed memory parallel computer, we are using the General Concurrent PIC (GCPIC) algorithm [1] previously developed for a 1D parallel PIC code.

Introduction

In previous work we have demonstrated the efficiency of a 1D PIC code on the JPL/Caltech Mark III Hypercube [1] We have now extended our work to a 2D implementation of an electrostatic PIC code for plasma simulations, using the General Concurrent PIC (GCPIC) algorithm [2]. The GCPIC algorithm is a generalization of the techniques employed in the 1D parallel PIC code which is applicable to many different parallel architectures. In this paper we describe its application to the implementation of the well benchmarked 2D electrostatic PIC code BEPSJ [3] on the Mark III Hypercube.

A plasma PIC code simulates the self consistent interactions of thousands to millions of electrons and ions in a computational box. There are two essential elements to an electrostatic PIC code. The first is the particle push, in which the positions and velocities of all of the particles are advanced in time subject to any external magnetic field and the self consistent electric, and their charges are interpolated onto the field grid. The second is the field solve, in which the electric field is updated based upon the new particle positions. These two code sections represent the vast majority of the computation. Additional computation is required for diagnostics which are done periodically throughout the simulation, but represent an ignorable fraction of the total computation time. Thus an efficient implementation of a PIC code requires an efficient implementation of the particle push and field solve. Since the particle push represents the major fraction of the computation time, it is essential on a distributed memory machine to have approximately equal numbers of particles in each processor. The field grid must be distributed as well for the purpose of solving for the new fields, and in a manner which is not necessarily the same as that needed to push the particles. We refer to these two decompositions as the primary (particle) and secondary (field) decompositions.

Our 2D PIC code is periodic in one dimension and may be periodic or bounded in the other dimension. As the particles are advanced in time, some may traverse the entire grid space during the course of the simulation. The simplest primary (particle) decomposition which handles this problem is the static decomposition, in which each processor keeps a copy of the entire field grid and the particles are partitioned at the beginning of the simulation among processors. This technique guarantees that load balance is maintained throughout the simulation, at the expense of redundant copies of the fields in every processor. Using the static decomposition, we have obtained efficiencies for the push in excess of 80%. The major inefficiency of this method results from the need to duplicate the charge array initialization in each processor and do a sum over processors when the charge array is updated.

The next level of sophistication is to partition the field grid as well as particles, so that each processor has a unique piece of the simulation space. Because of inhomogeneity in the particle density, this partition may in general be irregular in order to maintain load balance among the processors. However, there is a large class of

DISCLAIMER

2D problems which has the property of being relatively uniform along one coordinate direction, especially if the problem is periodic in that coordinate. In this case, a regular decomposition of the field grid among processors along the coordinate of uniformity (as shown in Fig. 1) will also result in a load balanced decomposition of particles. As the simulation progresses, some particles will traverse the entire simulation space. Since each processor now has only a part of the entire field grid, it is necessary to migrate particles from one processor to another as they evolve. This can result in particle load imbalance if the net flux of particles out of each processor is not zero. A particle load imbalance could develop during the course of the simulation, even though there is perfect load balance to begin with. Fortunately, for the class of problems for which this decomposition is appropriate, significant load imbalance *does not* develop due to the physics.

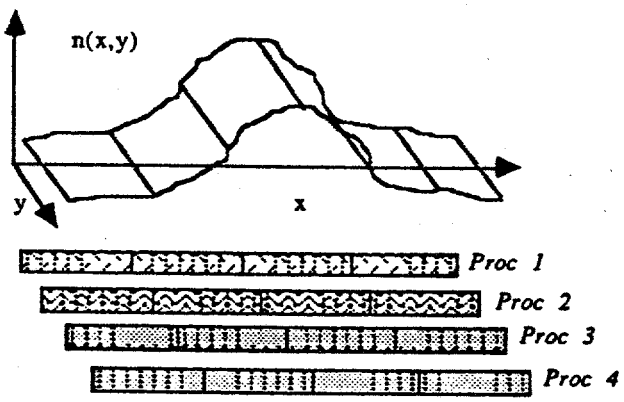

Figure 1. A regular particle partition for 4 processors. The y direction is the coordinate of relative uniformity in this case. Space is subdivided evenly among processors, leading to a load balanced partition of particles as well.

If the regular decomposition cannot be used effectively (some device physics problems can have large nonuniformities along all coordinate directions), a free form decomposition of the field grid may be necessary. These pieces can be of different size in general, since nonuniformities may develop during the course of a simulation. To maintain load balance, the distribution of particles and field grid must also evolve during the course of the simulation. We are in the process of implementing the same algorithm for dynamic load balancing as has been used for the 1D PIC code [4]. The grid space is partitioned as shown in Fig. 2 into slices so that each processor handles all of the x domain for a particular range of the y domain. Particles migrate between processors as they traverse the computational space. The grid space may be repartitioned as the density in the simulation evolves so that load balance is maintained.

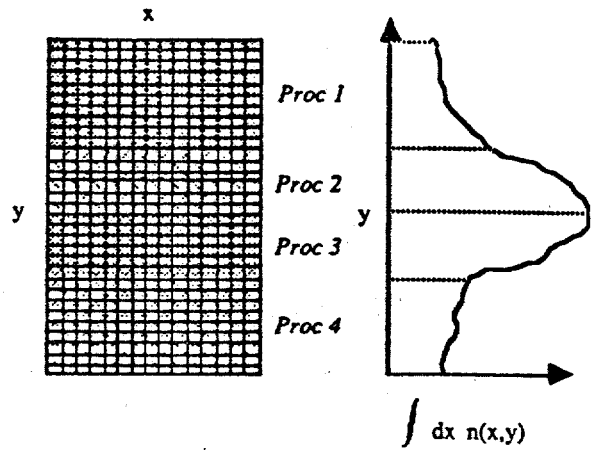

Figure 2. Field grid partition based on particle density distribution. Load balance requires that particles be distributed evenly among the processors. Thus each processor may have a different number of grid points.

The secondary (field) decomposition is made to update the field values at each time step. We calculate the new fields by solving Poisson's equation in Fourier space. For best performance in parallel, we compute the 2D FFT as two sets of 1D FFTs along each coordinate direction. For this solution method, the decomposition is a straightforward assignment of slices of the grid along one coordinate direction to each processor, as shown in Fig. 3. The

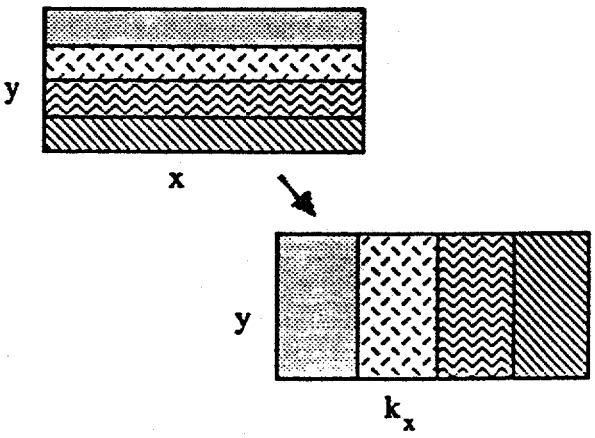

Figure 3. Field grid decomposition for the 2D FFT. Each processor has a strip of the field grid initially, such that it can do 1D FFTs in x for its subset of the y dimension. The results are then redistributed so that each processor now has a strip oriented along the y coordinate direction. 1D FFTs in y may now be performed for each processor's subset of k_x.

FFTs in the coordinate direction parallel to the long edge of the slice are performed. Then the grid is repartitioned into slices along the other coordinate direction, so that the second set of FFTs may be done.

Diagnostics are done in parallel, including graphics, by using one of the field decompositions described above. Phase space plots, for example, are parallelized using the primary decomposition, while contour plots of potential are done using the secondary field decomposition. The graphics software operates in parallel, with each processor drawing a separate portion of the graph corresponding to its part of the diagnostic.

Code Operation with the Regular Particle Decomposition

The main loop of the 2D code proceeds as follows. The field solver takes the real space charge distribution which has been interpolated onto the field grid and transforms it into k space using the 2D FFT algorithm mentioned above. Poisson's equation is solved in k space, and the x and y components of the electric field are computed from its solution. Then the two electric field components are transformed back to real space. Since the x space field grid decomposition is the same as the particle decomposition when using the regular grid primary decomposition, no addition grid rearrangement is required to begin the push. However, interpolating the field from the grid for all of the particles in the regular decomposition requires guard rows on both sides of the grid, since particles at a decomposition boundary require field information which is contained in a neighboring processor. This guard row information is exchanged between processor neighbors before the push phase begins. By mapping the processors into a logical ring, only nearest neighbor communication is required for the exchanges. The push phase of the simulation involves advancing the particles' positions and velocities one time step, then interpolating each particle's charge back onto the field grid using its updated position. Since some particle charge will be interpolated onto the guard rows, these rows must be combined with their counterparts in adjacent processors before the charge deposition is complete. Again, only nearest neighbor communication is required.

Results

In Table 1, we present timings for the two major code section which are executed at each time step of a simulation run. Two test problems of different size were timed. In each test case, the physics problem being modeled was the same (a lower hybrid plasma wave

Timings for Critical Code Sections
Mark IIIfp Hypercube

32 × 128 Field Grid
16,128 Particles

Number of Processors	1	2	4	8	16	32
Solver (sec)	.742	.427	.275	.205	.183	Note 1
Push (sec)	1.74	.861	.421	.207	.108	Note 1
per particle (msec)	107.9	53.6	26.1	12.8	6.7	

64 × 256 Field Grid
235,136 Particles

Number of Processors	1	2	4	8	16	32
Solver *(sec)*	Note 2	1.70	.996	.652	.498	.449
Push *(sec)*	Note 2	13.2	6.66	3.34	1.68	.849
per particle *(msec)*	Note 2	56.1	28.3	14.2	7.1	3.6

Note 1 - The FFT requires that n_x, the number of grid points in the x direction, be at least twice the number of processors.
Note 2 - The problem was too large to fit on one processor alone.

Table 1. Measured times for the two main code sections in BEPS. Solver and Push times are elapsed times, including communication. Per particle time is push time divided by the number of particles.

traveling along the periodic coordinate was excited by an antenna). The push phase in each case shows practically linear speedup as the number of processors are increased. The solver phase, which is dominated by three 2D FFTs, rapidly saturates in speedup. This is caused by an increase in the amount of communication required by the grid redistribution between 1D FFTs while the number of 1D FFTs done in each processor decreases. This is a problem with FFT based solvers in general, since information from each grid point must ultimately be combined with information from every other grid point in order to compute the transform.

In Fig. 4 we have plotted the efficiency of each code section for the two test cases as a function of the number of processors employed. We define efficiency E as

$$E = N\, T_N/T_1$$

where N is the number of processors, T_N is the execution time on N processors, and T_1 is the execution time on 1 processor. The solver efficiency drops dramatically as the number of processors is increased, due to the increasing communication to computation ratio mentioned above. The push efficiency remains very near

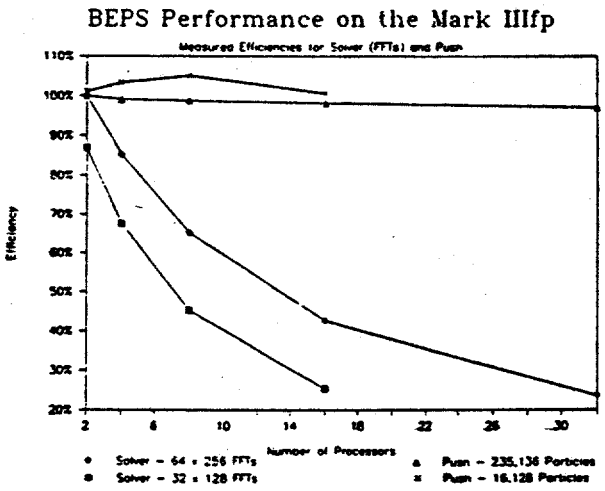

Figure 4. Measured code efficiencies. The push section of the code always runs close to 100%. The solver, which is dominated by 2D FFTs, suffers rapid efficiency degradation as the number of processors is increased.

100%, independent of the number of processors. This demonstrates that the communication time required for migrating particles between processors and exchanging guard row information is negligible compared to the computation involved in updating the particle positions and velocities. The efficiencies in excess of 100% achieved for the smaller test case by the push phase simply indicate that the algorithm being used in the push is not optimal for one processor. The Mark IIIfp has cash memory associated with the Weitek Floating Point Processors. As more processors are used, the number of particles and the size of the field grid each processor handles decreases, resulting in a lower probability of cash misses. The increase in performance of the hardware when using the cash memory more than makes up for the addition of communication overhead. The larger test case never gets subdivided sufficiently for this hardware effect to be noticed.

Since the primary (particle) decomposition remains fixed throughout the simulation, the possibility of particle load imbalance exists. In Fig. 5 we plot the percentage of load imbalance (%LI) observed in the smaller test case running on 16 processors. The physics of the problem was changed from a heating simulation to a current drive simulation, where particles are accelerated along the periodic coordinate. This number is defined as

$$\%LI = (n_{max} - n_{ave})/n_{ave}$$

where n_{max} is the maximum number of particles in any processor and n_{ave} is the average number of particles per processor. Even though particles are moving (rather

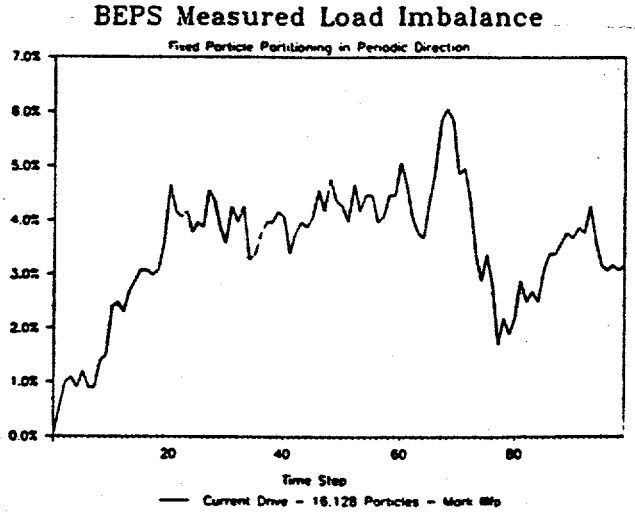

Figure 5. Measured particle load imbalance with the regular particle partition. The imbalance is defined as the largest percentage deviation of any processor's particle load from ideal at a given time step.

rapidly) between processors, the largest load imbalance observed during the first 100 time steps is about 6.2%. The load imbalance continues to oscillate around 3.5% for the rest of the simulation. This is clearly a simulation from the class where the fixed particle partition works very well. We believe, however, that the performance of the fixed particle partition on this problem is representative. Since, from a physics standpoint, it is quite difficult to develop and maintain large inhomogeneities in all coordinate directions in a plasma simulation, we also believe that the fixed particle partition is applicable to a wide variety of problems of interest to the fusion plasma and space plasma communities.

Dynamic Load Balancing

Of course not all simulation problems of interest are amenable to the fixed particle partitioning scheme. For these problems, some kind of irregular partition is necessary, and with it, the ability to dynamically balance the particle load among the processors. Fig. 6 illustrates

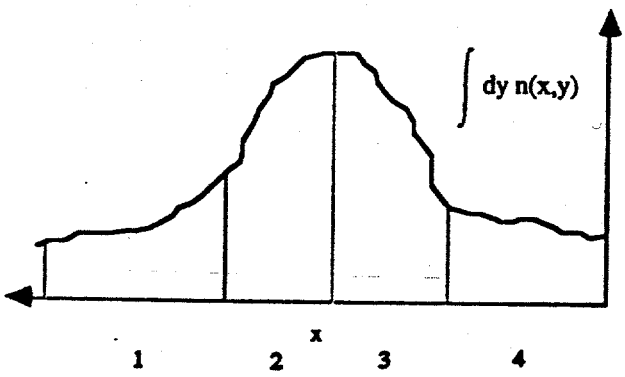

Figure 6. Dynamic load balancing without particle sorting. The charge density interpolated onto the grid is used to construct a density function. Partitioning is done based on this function.

a load balancing scheme which does not require particle sorting, *per se*. Assume that an irregular partition already exists which is load balanced. After the particles are advanced in time and passed among processors, some load imbalance may have developed. Rather than sorting the particles by coordinate to determine the new (load balanced) partition, the particles are interpolated onto the charge grid in the current partition. Before the field solve proceeds, the charge density is used to determine the new partition positions. The actual method of determining the new partition locations is not important, since it will scale with the grid size, rather than the number of particles. A parallel recursive bisection on the charge density appears to be an attractive choice. We are in the process of implementing a dynamic load balancing scheme for the 2D code.

Conclusions

We have implemented a 2D electrostatic PIC code for plasma simulation on the Mark IIIfp Hypercube Concurrent Computer. The code is completely parallelized, including diagnostics and graphics. We are currently using a regular primary (particle) partition, which is fixed throughout the entire simulation run. This decomposition exhibits very good particle load balance for a large class of plasma problems. Particle push efficiencies remain close to 100% with up to 32 processors. Solver performance, which is based upon FFT performance, degrades rapidly as the number of processors is increased.

Acknowledgements

This work is supported by DARPA, contract # NAS7-918

References

[1] V.K.Decyk, Supercomputer 27, 33 (1988).

[2] P.C.Liewer and V.K.Decyk, J. Comp. Phys., 85,302 (1989)

[3] V.K.Decyk and J.M.Dawson, J. Comp. Phys. 30, 407 (1979)

[4] P.C.Liewer, E.W.Leaver, V.K.Decyk, and J.M.Dawson, "Concurrent PIC Codes and Dynamic Load Balancing on the JPL/Caltech Mark III Hypercube", in *Proceedings of the 13th Conference on the Numerical Simulation of Plasmas*