CONF-9004159--9 **Appendix D.**

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

Dynamic Load Balancing in a Concurrent Plasma PIC Code on the JPL/Caltech Mark III Hypercube

FG03-84ER53173

P. C. Liewer and E. W. Leaver
Jet Propulsion Laboratory, California Institute of Technology
Pasadena, CA 91109

V. K. Decyk and J. M. Dawson
Physics Department, University of California
Los Angeles, CA 90024

Abstract

Dynamic load balancing has been implemented in a concurrent one-dimensional electromagnetic plasma particle-in-cell (PIC) simulation code using a method which adds very little overhead to the parallel code. In PIC codes, the orbits of many interacting plasma electrons and ions are followed as an initial value problem as the particles move in electromagnetic fields calculated self-consistently from the particle motions. The code was implemented using the GCPIC algorithm in which the particles are divided among processors by partitioning the spatial domain of the simulation. The problem is load-balanced by partitioning the spatial domain so that each partition has approximately the same number of particles. During the simulation, the partitions are dynamically recreated as the spatial distribution of the particles changes in order to maintain processor load balance.

Introduction

We have implemented dynamic load balancing in a one-dimensional electromagnetic plasma particle-in-cell (PIC) simulation code running on the Caltech/JPL Mark III Hypercube. In plasma particle simulations codes, the orbits of thousands to millions of interacting plasma electrons and ions are followed as an initial value problem as the particles move in electromagnetic fields calculated self-consistently from the particle motions. The electromagnetic fields are determined from Maxwell's equations, or a subset, with the plasma currents and charge density as source terms; these fields determine the forces on the particles. In a PIC code, particles can be located anywhere in the simulation domain, but the field equations are solved on a discrete grid. Each time step in a PIC code proceeds in two stages. In the first stage, the position and velocities of the particles are updated by calculating the forces on the particles from interpolation of the field values at the grid points; the new charge and current densities at the grid points are then calculated by interpolation from the new positions and velocities of the particles. In the second stage, the updated fields are found by solving the field equations on a discrete grid using the new charge and current densities. Generally the first stage accounts for most of the computation time because there are many more particles than grid points.

The one-dimensional electromagnetic code was implemented on the Mark III Hypercube using the General Concurrent Particle-in-Cell (GCPIC) algorithm[1]. The General Concurrent PIC algorithm is designed to make the most computationally intensive portion of a PIC code, updating the particles and the resulting charge and current densities, run efficiently on a parallel processor.

To implement a PIC code in parallel using the GCPIC algorithm[1,2], the physical domain of the particle simulation is partitioned into sub-domains, equal in number to the number of processors, such that all sub-domains have roughly equal numbers of particles. For problems with non-uniform particle densities, these sub-domains will be of unequal physical size. Each processor is assigned a sub-domain and is responsible for storing the particles and the electromagnetic field quantities for its sub-domain and for performing the particle computations for its parti-

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

cies. For a 1D code on a hypercube, nearest neighbor sub-domains are assigned to nearest neighbor processors. When particles move to new sub-domains, they are passed to the appropriate processor. As long as the number of particles per sub-domain is approximately equal, the processors' computational loads will be balanced. Dynamic load balancing is accomplished by repartitioning the simulations domain into sub-domains with roughly equal particle numbers when the processor loads become sufficiently unbalanced. The computation of the new partitions, done in a simple way using a crude approximation to the plasma density profile, adds very little overhead to the parallel code.

1D Electromagnetic PIC Code

The 1D electromagnetic PIC code was created from a parallel one-dimensional electrostatic code[1] by including the effects of magnetic fields on the particle orbits and including the calculation of the electromagnetic fields. This 1-d electromagnetic code, with kinetic electrons and ions, has been used to study electron dynamics in oblique collisionless shock waves such as in the earth's bow shock.

In this 1D code, variation is in the $x-$direction only. The orbit equations for the ith particle are

$$\frac{dx_i}{dt} = v_{x,i}$$

$$\frac{d\mathbf{v}_i}{dt} = \frac{q_i}{m_i}\left(\mathbf{E} + \frac{\mathbf{v}_i \times \mathbf{B}}{c}\right).$$

Motion is followed in the x direction only, but all three velocity components must be calculated in order to calculate the $\mathbf{v} \times \mathbf{B}$ force. Forces on the particles are found from the fields at the grid points by interpolation. The longitudinal (along x) electric field is found by solving Poisson's equation

$$\nabla \cdot \mathbf{E} = 4\pi\rho_q(x,t).$$

The transverse (to x) electromagnetic fields, E_y,E_z, B_y, and B_z, are found by solving

$$\frac{\partial \mathbf{B}}{\partial t} = -\frac{1}{c}\nabla \times \mathbf{E}$$

$$\frac{\partial \mathbf{E}}{\partial t} = c\nabla \times \mathbf{B} - 4\pi\mathbf{j}.$$

The plasma current density $\mathbf{j}(x,t)$ and charge density $\rho_q(x.t)$ are found at the grid points by interpolation from the particle positions. Only the transverse (y and z) components of the plasma current are needed. These coupled particle and field equations are solved in time as an initial value problem. As in the electrostatic code, the fields are solving by Fourier transforming the charge and current densities and solving the equation in k space, and advancing the Fourier components in time. External fields and currents can also be included. At each time step, the fields are transformed back to configuration space to calculate the forces needed to advance the particles to the next time step. Extending the existing parallel electrostatic code to include the electromagnetic effects required no change in the parallel decomposition of the code.

Dynamic Load Balancing

In the GCPIC electrostatic code[1], the partitioning of the grid was static. The grid was partitioned so that the computational load of the processors was initially balanced. As simulations progress and particles move among processors, the spatial distribution of the particles can change, leading to load imbalance. This can severly degrade the parallel efficiency of the push stage of the computation. To avoid this, dynamic load balancing has been implemented in the 1D electromagnetic code.

To implement dynamic load balancing, the grid is repartitioned into new sub-domains with roughly equal numbers of particles as the simulation progresses. The repartitioning is not done at every time step. The load imbalance is monitored at a user-specified interval. When the imbalance becomes sufficiently large, the grid is repartitioned and the particles moved to the appropriate processors as necessary. The load was judged sufficiently imbalanced to warrant load balancing when the number of particles per processor deviated from the ideal value n_{ideal} (= number of particles / number of processors) by $2\sqrt{n_{ideal}}$, *e.g.*, twice the statistical fluctuation level.

The dynamic load balancing is performed during the push stage of the computation. Specifically, the new grid partitions are computed after the particle positions have been updated, but before the particles are moved to new processors to avoid any unnecessary moving of particles. If the load are sufficiently balanced, the subroutine computing the new grid partitions is not called. The subroutine which moves the particles to appropriate processors is called in either case.

To accurately represent the physics. a particle can not move more that one grid cell per time step. As a result, in the static 1D code, the routine which moves particles to new processors only had to move parti-

cles to nearest neighbor processors[1]. To implement dynamic load balancing, this subroutine had to be modified to allow particles to be moved to processors any number of steps away. Moving the particles to new processors after grid repartitioning can add significant overhead; however, this is incurred only at time steps when load balancing occurs.

The new grid partitions are computed by a very simple method which adds very little overhead to the parallel code. Each processor constructs an approximation to the plasma density profile $\tilde{n}(x)$ and uses this to compute the grid partitioning to load balance. To construct the approximate density profile, each processor sends the locations of its current sub-domain boundaries and its current number of particles to all other processors. From this information, each processor can compute the average plasma density in each processor and from this can create the approximate to density profile (with as many points as processors). This approximate profile is used to compute the grid partitioning which approximately divides the particles equally among the processors. This is done by determining the set of sub-domain boundaries $xleft$ and $xright$ such that

$$n_{ideal} \approx \int_{xleft}^{xright} \tilde{n}(x)dx.$$

Linear interpolation of the approximate profile is used in the numerical integration. The actual plasma density profile could also be used in the integration to determine the partitions. No additional computation would be necessary to obtain the local (within a processor) $n(x_{grid})$ because it is already computed for the field solution stage, but it would require more communication to make the density profile global. Other method of calculating new sub-domain boundaries, such as sorting particles, require a much large amount of communication and computational overhead.

The effect of dynamic load balancing on the time for the push stage of the code (push time) is illustrated in the figures. Note that all of the overhead for dynamic load balancing is included in the time for the push time. Results are from a test problem in which the plasma was initialized with an extremely non-uniform "square wave" density profile: The plasma was uniform in the center quarter of the 1D simulation box, and zero elsewhere. The simulation box was bounded with solid walls and particles hitting the wall are reflected. As the run progresses, the plasma expands at roughly the sound speed and tends to distribute itself uniformly within the simulation box. Results for the time for the push stage are shown from test cases with 5120 particles run on 8 processors with

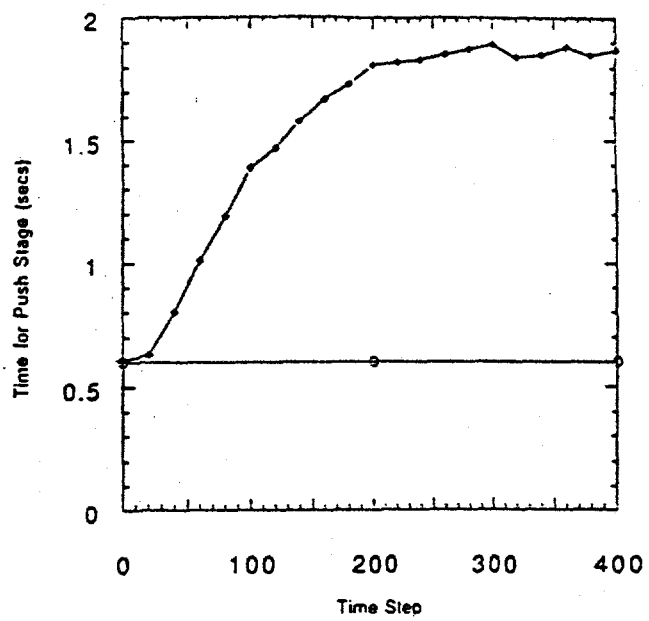

Figure 1: Time for push stage of code at a function of time step in the run. Upper curve is time for push stage for the test case with fixed grid partitioning (no dynamic load balancing). Lower curve is the ideal push time *e.g.*, the time if the load is perfectly balanced.

and without dynamic load balancing. In both runs, the grid was initially partitioned with equal numbers of particles in each of the 8 sub-domains. Since the density was uniform in the central portion of the box, the six "interior" processors each had sub-domains of equal size; the two end processors, however, had much large sub-domains because they were also responsible for the outer portions of the simulation domain with initially no particles.

Results for the time in seconds for the push stage as a function of time step for the case without dynamic load balancing are shown in the upper curve of Fig. 1. The lower curve illustrates the push time if the computational load is perfectly balanced (approximately 0.6 secs for this test case). Early in the run, as the plasma expands outward toward the walls, particles flow into the empty outer portions of the simulation domain and the end processors accumulate an increasingly large number of particles. The push time (without dynamic load balancing) is linearly proportional to the maximum number of particles per processor. As the number of particles in the end processors increases, the time for the push increases proportionately. By the end of the run, when the plasma is

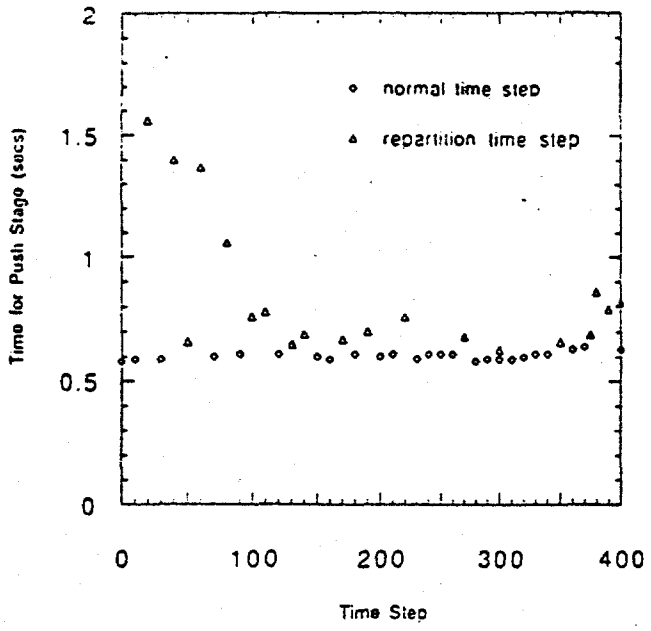

Figure 2: Time for push stage of code at a function of time step in the run for the test case with dynamic load balancing. Normal time steps (no load balancing) are plotted with a diamond (◇) and time step in which load balancing occurred are plotted with a triangle (△).

roughly uniformly distributed in the simulation box, the push time has saturated at a level slightly over three times the ideal push time.

Results for the push stage of the code for the test case with dynamic load balancing are shown in Fig. 2. Not every time step is shown in the figure. For this run, the load balance was tested every five steps to see if the load imbalance was sufficient to warrant load balancing. The steps on which load balancing (repartitioning and moving particles appropriately) occurred are shown with a triangle (△); normal time steps are shown with a diamond(◇). Note that for most of the time steps, the push time is close to the ideal push time of 0.6 secs. The overhead for dynamic load balancing can be seen on the time steps with load balancing, especially at early time. This overhead is due primarily to moving the particles to the new processors after the new sub-domains are computed. The very low overhead on some of the load balancing time steps shows how little overhead is incurred by the computation of the new sub-domains themselves. For this test case, the total time for the run without dynamic load balancing was 1.8 times as long as the run with dynamic load balancing.

Conclusions

Dynamic load balancing has been implemented in a concurrent 1D electromagnetic PIC code using a method which adds very little overhead to the parallel code. The code was implemented in parallel using the GCPIC algorithm in which particles are divided among processors by partitioning the spatial domain. Partitions are created dynamically during a run so that the sub-domains have approximately equal numbers of particles. The grid partitions are computed from an approximate plasma density profile, rather than from the particle data, making the computation of the new sub-domain boundaries fast and simple. This method should extend to two- and three- dimensional PIC codes, where the new sub-domains could be calculated by recursive bi-section using the density profile. We plan to implement this method of dynamic load balancing in a two-dimensional electrostatic GCPIC code, described by Ferraro *et al.* [2]. Most of the overhead for the dynamic load balancing results from passing the particles to the appropriate processor. Thus the amount of overhead for the load balancing will depend in part on how rapidly the spatial distribution of particles is changing. For all cases run to date, the parallel efficiency of the code always improves when dynamic load balancing is used. For the test case presented, the run time was 1.8 times longer when dynamic load balancing was not used.

Acknowledgements

Part of the research described in this paper was performed by the Jet Propulsion Laboratory, California Institute of Technology and was sponsored by Sandia National Laboratories, Albuquerque and by Caltech President's Fund Grant No. PF-317. The UCLA research was supported by Sandia National Laboratory under Contract 23-1540. and by Caltech President's Fund Grant No. PF-317.

References

[1] P. C. Liewer and V. K. Decyk, J. Comp. Phys. 30,407(1989).

[2] R. D. Ferraro, P. C. Liewer and V. K. Decyk. "A 2D Electrostatic PIC Code for the Mark III Hypercube," (*this proceedings*).