JOURNAL OF COMPUTATIONAL PHYSICS 109, 329–341 (1993)

Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code

ROBERT D. FERRARO AND PAULETT C. LIEWER

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California 91109

AND

VIKTOR K. DECYK

Physics Department, University of California Los Angeles, Los Angeles, California 90024

Received July 22, 1991; revised April 21, 1993

We have developed and implemented a load balancing scheme for particle-in-cell (PIC) codes which use the GCPIC algorithm on MIMD computers. The algorithm has been applied to a two-dimensional electrostatic plasma PIC simulation code using 1D partitions for particles and fields. The code is a parallelized version of a well-benchmarked sequential plasma PIC code which uses quadratic field and charge interpolation and an FFT-based Poisson solver. This code has been run on the Caltech/JPL Mark IIIfp, NCUBE2, and Intel i860 Gamma hypercube computers. Particle load balance is maintained by readjusting the GCPIC primary partitions whenever any processor's particle count exceeds a certain imbalance threshold. The new partition boundaries are calculated based upon a 1D number density function computed from the grid deposited charge distribution. Dynamic load balancing is found to be effective when the execution time of the particle push dominates over execution time for the field solve. For a large class of plasma simulations, we find that dynamic load balancing is not required; static initial partitions intelligently chosen can work just as well. Parallel efficiency for the particle push is very high (90–100%) and scales with number of processors. Parallel efficiency for the FFT-based field solver is poor in this implementation and directly impacts the effectiveness of the load balancing scheme.

I. INTRODUCTION

The most powerful general purpose supercomputers today are of the massively parallel, distributed memory type. These machines consists of many processors, each with their own local memory, working either synchronously (SIMD—single instruction multiple data architectures) or asynchronously (MIMD—multiple instruction multiple data architectures) on a single problem. Programming these machines [1] efficiently generally requires special techniques or algorithms which differ from their corresponding sequential implementations, due to the existence of distributed memory and the latency of accessing memory which is not local to a given processor. These techniques may be nothing more complicated than properly mapping the major data arrays onto the collection of processors and their memories, as is the case for finite difference and finite element algorithms [2]. In other cases, an entirely new approach to parallelism, such as functional decomposition [3], might be required.

For plasma PIC simulations, some special considerations enter into the strategy for implementation on parallel computers [4]. A plasma PIC simulation [5] follows the evolution in time of the trajectories of thousands to millions of charged particles in their self-consistent electromagnetic fields, as well as any externally applied forces which might be present. The particles may have positions anywhere within the simulation space, while the fields and forces are defined at discrete points on a grid within the space. The code consists of two major sections, which we refer to as the *push* and the *field solve*. In the push, the positions and velocities of all of the particles are advanced some small amount in time under the influence of forces which are interpolated to the particle positions from the grid. The charge (and current, for electromagnetic simulations) at the new particle positions is interpolated back onto the grid for the field solve. Then the field solve, using the new gridded charge distribution, advances the field quantities in time. To implement a plasma PIC code on these massively parallel machines, it is clear that somehow both the particles and the fields must be appropriately distributed among the processors and their memory.

We have previously developed an algorithm for implementing PIC simulation codes on MIMD computers which we have called the GCPIC [6] algorithm. In GCPIC, particles are divided among processors by partitioning the simulation space among processors so that each partition contains approximately the same number of particles. This algorithm has been shown to be an effective means for parallelizing PIC simulations in both 1D and 2D [7, 8]. It

0021-9991/93 $5.00

was recognized that during execution of a simulation, particles could potentially migrate among processors creating situations where the number of particles assigned to each processor deviated substantially from the average. This condition of particle load imbalance would lead to a slowing down of the code execution, since processors must synchronize their activities at several times during the main loop. The execution time of the loop in GCPIC is determined by the sum of the execution times between synchronization points of the slowest (most heavily loaded) processors. We have now developed a method for dynamically balancing the computational load among processors as the simulation proceeds. An important consideration in any algorithm for load balancing particles is that it scale at most linearly with the number of particles, since the rest of the PIC algorithm already does so. Therefore, any method which requires particle sorting (an $n \log n$ process) is undesirable. Our method is based upon a one-dimensional grid based number density function computed from the interpolated charge distributions. Its direct cost, therefore, scales with the size of the grid and is independent of the number of particles.

In many PIC simulations, a plasma's tendency toward uniform density will make load balancing unnecessary. It is also unnecessary in other cases, such as tokamak modeling, where the expected density distribution is known and the domain can be decomposed at the start to give load balancing. However, in many other cases, particularly device computations, the density distribution changes dramatically and dynamic load balancing will be essential to obtain efficient code.

In the sections that follow, we will briefly review the GCPIC algorithm as it has been applied to our 2D electrostatic PIC code. We will look at the code efficiency on both particle load balanced and unbalanced problems. Then the load balancing algorithm will be presented and its performance discussed. Finally, we will make some remarks concerning the general load balancing problem for 3D PIC simulations in the conclusions.

II. THE 2D PARALLEL ELECTROSTATIC PIC CODE

The parallel code [7] is based on a well-benchmarked 2D sequential electrostatic PIC code from UCLA called BEPS [9]. This code may be bounded or periodic in the x direction and periodic in the y direction. A static externally imposed magnetic field is allowed, and three velocity space components for each particle are tracked. The field solver is FFT-based. The code includes integrated graphics for potential, density, and velocity space diagnostics. It has also recently been extended to electromagnetic simulations [10].

In the GCPIC algorithm, there are two major data decompositions which map particles and fields to processors. In the *primary decomposition*, particles are assigned to processors based upon physical space *partition boundaries* so that all processors have approximately equal numbers of particles. This decomposition is designed to make the particle *push* section load balanced. So that each particle is uniquely assigned to a single processor, the partitions are constructed so as to be non-overlapping. In order to do the grid interpolation required for force evaluations and charge deposition, the field grid must also be distributed among processors such that all grid points within the partition boundaries of a given processor, plus additional points outside and adjacent to the partition boundaries, are present in the processor's memory. This results in the need to duplicate some grid points in multiple processors. We refer to these duplicate grid points as *guard cells*.

The second data decomposition is designed to make the field solve load balanced, and is referred to as the *secondary decomposition*. Since the field solve computation is proportional to the number of grid points, the secondary decomposition assigns approximately equal numbers of grid points to processors in a manner which is most efficient for the field solver method. For our FFT-based solver, each grid point is uniquely assigned to a single processor. In this case it is clear that at least guard cell information must be communicated among processors when switching between the push and field solve phases of GCPIC, and, in the general case, some major redistribution of the field data must take place.

The particle decomposition among processors also presents a situation not encountered on sequential or shared memory computers. After a particle's position and velocity have been advanced in time, it may find itself outside of its parent processor's partition boundaries. Depending upon the number of guard cells available and the distance outside the partition, the deposition of the particle's charge may require grid locations not available in its parent processor. Therefore, after the position and velocity update, but before the charge deposition, particles which have crossed partition boundaries must be migrated to the processors appropriate to the partitions which they have entered.

Applying GCPIC to the 2D code results in an iteration scheme as detailed in Fig. 1. There are three main sections which are carried over directly from the sequential version. The sequential push is now split into two parts: a push, in which all particles' positions and velocities are time advanced; and a deposit, in which all particles' charges are interpolated onto the grid. The third section is the Poisson solver, but with the 2D FFTs replaced by 2D parallel distributed FFTs. GCPIC introduces two field-related communications, or *redistributions*, in converting field information between the primary, or particle, decomposition, and the secondary, or field, decomposition. There is also the particle mover which communicates particles

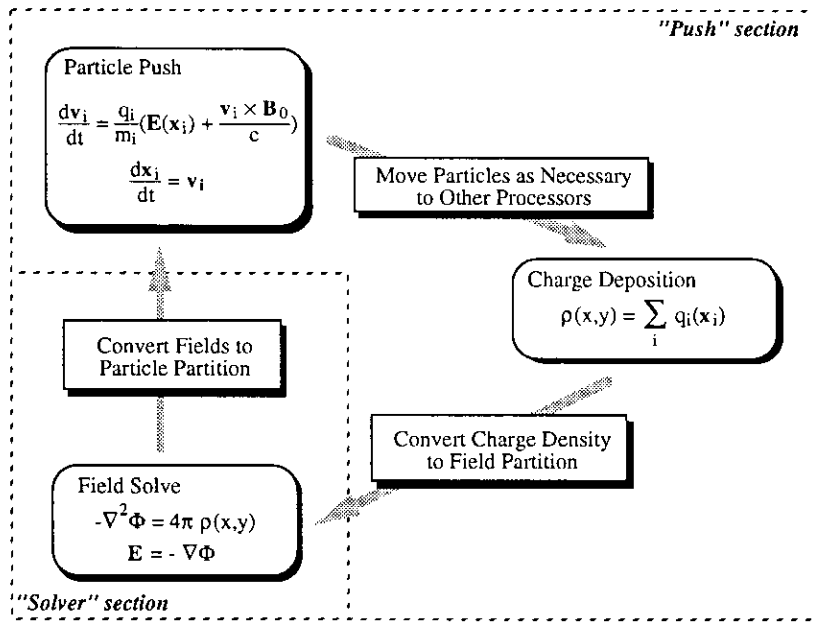

FIG. 1. The major tasks in the main loop of the General Concurrent PIC Algorithm. The items in rounded boxes are found in a sequential PIC code, while the items in square boxes are additions for operation on concurrent computers.

which have exited their processor's partition boundaries to their new processor assignments.

In the present implementation, the 2D parallel FFT used in the Poisson solver consists of two sets of 1D sequential FFTs done in parallel on each dimension, in the same way that 2D FFTs may be vectorized. The field decomposition for the solver is therefore by rows, as in Fig. 2a. Each processors has all of the x coordinate for a range of y; we refer to this decomposition as the x-space field decomposition. Groups of rows are assigned to individual processors, which can perform the $x \rightarrow k_x$ FFTs in parallel without inter-processor communication. Then a redistribution of data takes place (which amounts to a global array transpose) so that each processor now has complete columns. The $y \rightarrow k_y$ FFTs can now be done in parallel without inter-processor communication. The k-space field decomposition is shown in Fig. 2b. It is in this decomposition that Poisson's equation is solved and the electric field components calculated. The inverse 2D FFT proceeds in exactly the reverse of the forward transform. A total of three FFTs are necessary: one

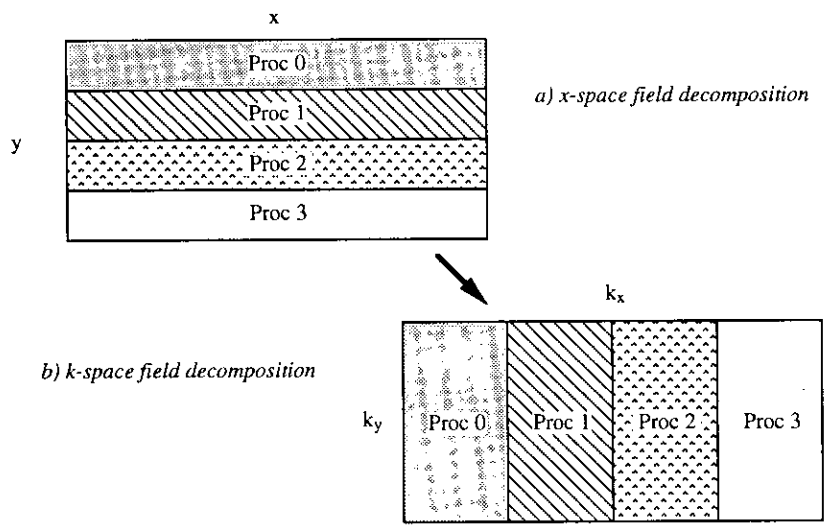

FIG. 2. The two field decompositions illustrated for four processors. In real space, the grid is distributed among processors in groups of contiguous rows. In transform space, the grid is distributed in groups of contiguous columns. The 2D parallel FFT transforms the grid from one partition to the other.

forward FFT for the charge density and two inverse FFTs for the two electric field components.

Implicit in this field solver implementation is the assumption that the number of grid points in each dimension is some integer multiple of the number of processors, since we do not currently use distributed 1D FFTs. Our 1D FFTs are restricted to power-of-two lengths; therefore the number of processors which we use is also restricted to be a power of two.

The particle decomposition used in our initial implementation was a 1D decomposition matched to the x-space field decomposition. We refer to it as the *static regular* partition, since the partitions were of equal width and remained fixed throughout the course of a simulation. A four-processor example is illustrated in Fig. 3. The partition boundaries are defined by

$$y_l = p\frac{L_y}{N_{\text{proc}}}, \tag{1a}$$

$$y_r = (p+1)\frac{L_y}{N_{\text{proc}}}, \tag{1b}$$

where y_l is the left boundary, y_r is the right boundary, L_y is the length of the system (an integer in normalized coordinates) in the y direction, N_{proc} is the number of processors, and p is the logical processor number, beginning with zero. Note that these boundaries are integers. (The logical processor number is a means of identifying a processor's position in a communication topology; in this instance, it corresponds to a processor's position in a ring of processors where one processor has arbitrarily been designated as Processor 0). The partition itself is defined by

$$y_l \leqslant y < y_r. \tag{2}$$

The quadratic interpolation scheme uses a nine-point stencil centered on the grid point nearest the particle. For particles in a partition defined by Eq. (1), the field grid rows defined by

$$n_{y\,\text{left}} = y_l - 1, \tag{3a}$$

$$n_{y\,\text{right}} = y_r + 1, \tag{3b}$$

are required to perform the interpolation. (Values of $n_{y\,\text{left}}$ below zero and $n_{y\,\text{right}}$ above $L_y - 1$ are handled according to the boundary condition for the y dimension, which for our code is periodic.) This amounts to requiring one guard cell row to the left and two guard cell rows to the right; the asymmetry here is due to the fact that the left bounday is in the domain, whereas the right boundary is in the next processor's domain (Eq. (2)).

The static regular 1D partition has several advantages. The transformation between particle partition and field partition involves only the guard cell rows, since the grid points which lie within a processor's particle partition boundaries are the same points as those of the processor's field partition. By reserving extra rows at the beginning and end of the field arrays, only the guard cell data need actually be moved in the transformation. The logical processor numbers can be selected so as to map processors into a physical ring on the communications network; only nearest neighbor communication will then be required for the field transformations, as well as for the particle mover. Finally, by partitioning the periodic direction of the simulation, the likelihood of significant load imbalance is reduced. While this paper treats only 1D parallel decomposition of the simulation domain, the methods presented here can be extended to higher dimensional simulations and decompositions. This will be discussed in the conclusions.

We have measured the parallel efficiency of our implementation on the Caltech/JPL Mark IIIfp Hypercube and on the Intel iPSC/860 Gamma hypercube. On each machine, we have run a lower hybrid heating simulation [11] consisting of 235,136 electrons and ions on a 64×256

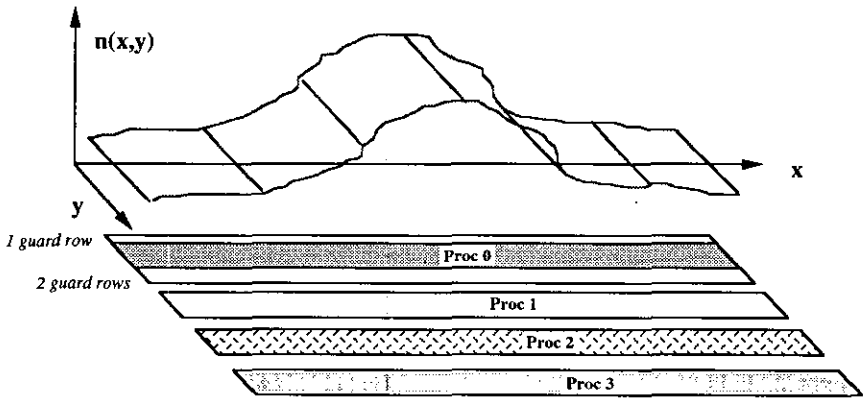

FIG. 3. The field grid assignment according to the static regular partition. Each processor receives an equal number of contiguous rows of the field grid. In addition, each processor requires additional rows of guard cells for interpolation of charge and field quantities. The guard rows must be exchanged with neighboring processors. A four-processor example is illustrated. Guard rows are depicted only for processor 0.

TABLE I

Time per Iteration for the Push and Solver Sections of the Parallel Code for the Lower Hybrid Heating Simulation Run on the Mark IIIfp and Intel i860 Gamma Machines for Various Numbers of Processors

Processors	Mark IIIfp		Intel i860 Gamma	
	Push (s)	Solver (s)	Push (s)	Solver (s)
1	26.4	3.39	5.27	.62
2	13.2	1.70	2.65	.40
4	6.66	1.00	1.34	.31
8	3.34	.65	.68	.29
16	1.68	.50	.35	.32
32	.85	.45	.19	.35

field grid for 500 time steps. The simulation was repeated while varying the number of processors employed. The execution time of the push section and the field solve section of the main loop (defined for GCPIC in Fig. 1) were timed and averaged for the first 10 timesteps, and are presented in Table I. A parallel efficiency number E can be defined for the code sections as

$$E = T_1/nT_n, \tag{4}$$

where n is the number of processors, T_n is the execution time on n processors, and T_1 is the execution time on one processor. These numbers, calculated from Table I, are plotted in Fig. 4. The push efficiency is close to 100% through 32 processors for both machines. The Mark IIIfp curve is quite flat, indicating that overhead due to interprocessor communication is unimportant in this case. The Intel machine exhibits a declining efficiency curve, due to communication overhead. The two machines behave differently because their computation speed/communication speed ratios are different (the Intel processors are perhaps a factor of 4 faster than the Mark IIIfp processors, while the communication speeds are about the same). The field solver is dominated by three 2D parallel FFTs, which rapidly become dominated by inter-processor communication, hence the precipitous drop in solver efficiency. Again, the Intel machine is lower in efficiency due to the difference in speed ratio.

The fact that the FFTs are communications-dominated can be seen in Table I; note that for the Intel, the solver time begins to *increase* with the number of processors above eight processors. This increase results from the increase in the time to perform the data transpose illustrated in Fig. 2. The 0.62 s solver time on one Intel node results from an approximate FFT cpu time (for the three FFTs) of 0.5 s and a guard cell filling time of about 0.1 s. The guard cell time is constant as the number of processors increases and the actual FFT-cpu time decrease inversely with the number of processors. On eight processors, the three costs are about comparable; beyond this, the time for the data transpose dominates the solver time. (Our one processor's FFT cpu time is considerably larger than the Intel-supplied one pro-

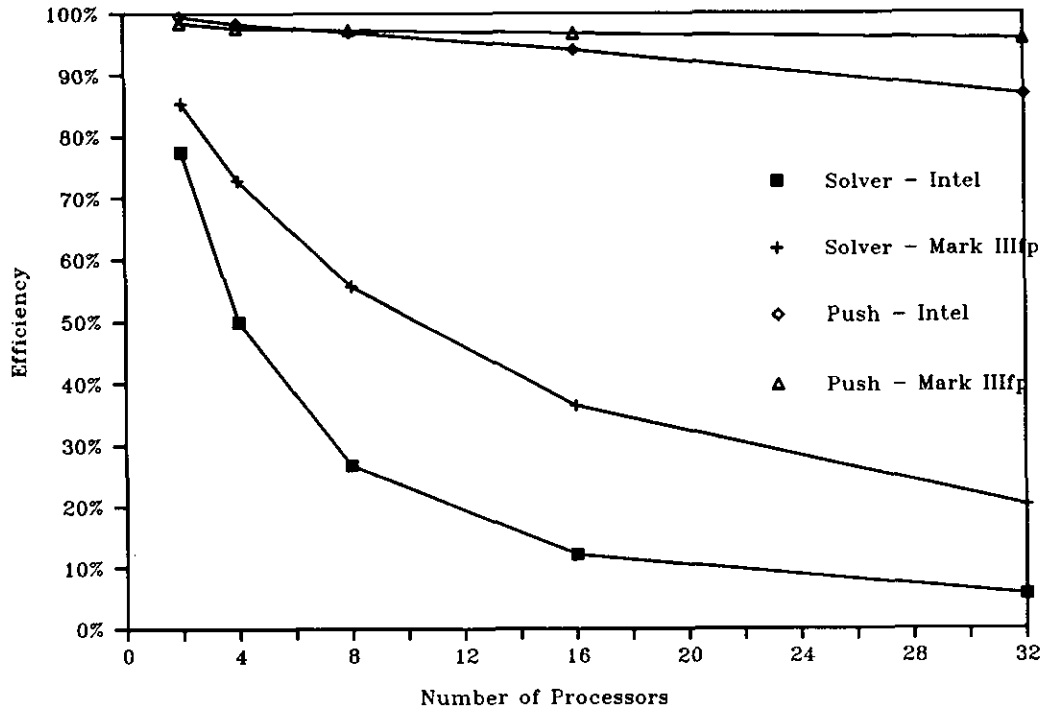

FIG. 4. Parallel efficiency versus the number of processors for the push and solver code sections as measured on the Mark IIIfp Hypercube and the Intel i860 Gamma. The Intel machine has lower efficiency in both cases due to its higher computation/communication speed ratio.

TABLE II

Push Time per Particle and Total Execution Time for a Lower Hybrid Heating Simulation Using 235,136 Particles on a 64×256 grid

	Push time (ms/particle)	Total execution time (s)
Mark IIIfp—32 processors	3.6	845
Intel i860 Gamma—32 processors	.8	481
Cray 2–4 processors (multitasked push)	1.5	268

Note. The simulation was run for 500 time steps.

cessor FFT, which was not available at the time this work was done; however, since the actual FFT cpu time is a small fraction of the solver time, this is unimportant for the work here.)

For comparison, we have also run the original sequential code on a four-processor Cray 2 with a multitasked push. The total execution time was measured along with the per-particle push time. These numbers, along with the corresponding hypercube machine numbers are listed in Table II. Although the Intel machine achieves the best per-particle push time, the Cray 2 has the lowest total execution time for the simulation. This is entirely attributable to the poor performance of the 2D FFTs on the hypercube machines.

III. THE DYNAMIC LOAD BALANCING ALGORITHM

In the lower hybrid heating simulation, a standing lower hybrid wave is excited by an antenna along the y direction. When the wave is driven at resonance, a large amplitude standing wave can be excited. If the simulation is continued long enough, the ponderomotive force due to the large amplitude standing wave will begin to dig out a density cavity in the plasma. This cavity will grow until the antenna detunes from the allowed lower hybrid modes (due to the non-linear density modification). The lower hybrid wave, along with the cavity will then collapse on an acoustic time scale. Eventually, the antenna will begin to resonantly drive a new lower hybrid wave, repeating the cycle.

The density cavities can be seen in the ion density contours of Fig. 5 which were plotted at $t = 228\omega_{pe}^{-1}$ (1140 time steps) into the simulation. For this case, a total of 524,288 particles were used on a 128×128 field grid. These density cavities are aligned parallel to the 1D regular static partitions used in the hypercube code, resulting in significant particle load imbalance during part of the simulation. We have plotted in Fig. 6 the maximum and minimum number of particles (electrons and ions) in any processor at each time step during the first 1500 time steps in a run done on a 64-processor hypercube. At the worst point, there is

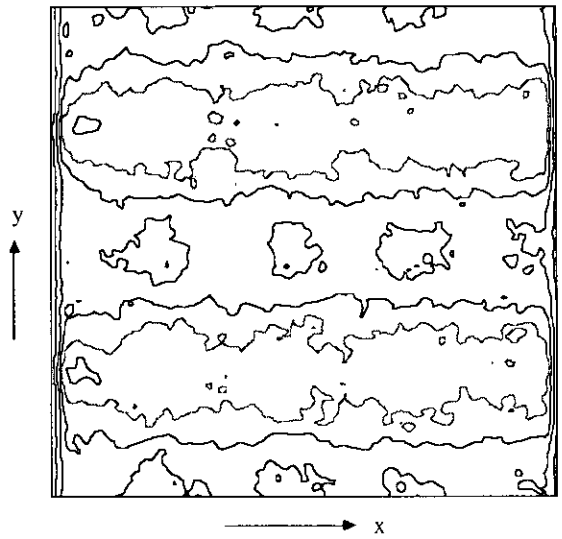

FIG. 5. Ion density contours plotted at $t = 228\omega_{pe}^{-1}$ in a lower hybrid heating simulation. A standing lower hybrid wave is being driven along the y axis. The ponderomotive effect digs troughs in density which are parallel to the particle partitions. The non-uniform distribution of particles among partitions results in load imbalance.

approximately a 7000 particle difference between the processor with the largest particle count ($\sim 12{,}000$) and the processor with the smallest particle count ($\sim 5{,}000$), which is a factor of 2.4 difference in loads. If this were a static situation throughout the course of the simulation, we could initially set our partitions for particle load balance which could remain fixed throughout the run. But since the load imbalance evolves, some kind of dynamic repartitioning is necessary in this case to maintain optimal push efficiency.

Sorting the particles by coordinate would allow an algorithm to determine the partition boundaries which give perfect load balance. However, the best sorting algorithms are order $N_p \log(N_p)$ processes, where N_p is the number of particles. Even though repartitioning is not done at every iteration step, the cost of an order $N_p \log(N_p)$ repartition would quickly overtake the order N_p push as the number of particles N_p becomes large. Perfect load balance is not even necessary, since at the next time step the load balance will begin to degrade. What is desired is a low cost method for determining new partition boundaries which gives *good* load balance.

Our method for defining new partition boundaries uses the existing charge deposition to the grid which takes place at every time step to construct a grid-based 1D number density function along the partition direction. The charge density arrays for the electrons and ions are summed over their x coordinates and, after being weighted by the inverse of their species charge to produce a number density, the sums are accumulated in a number density array. In Fig. 7

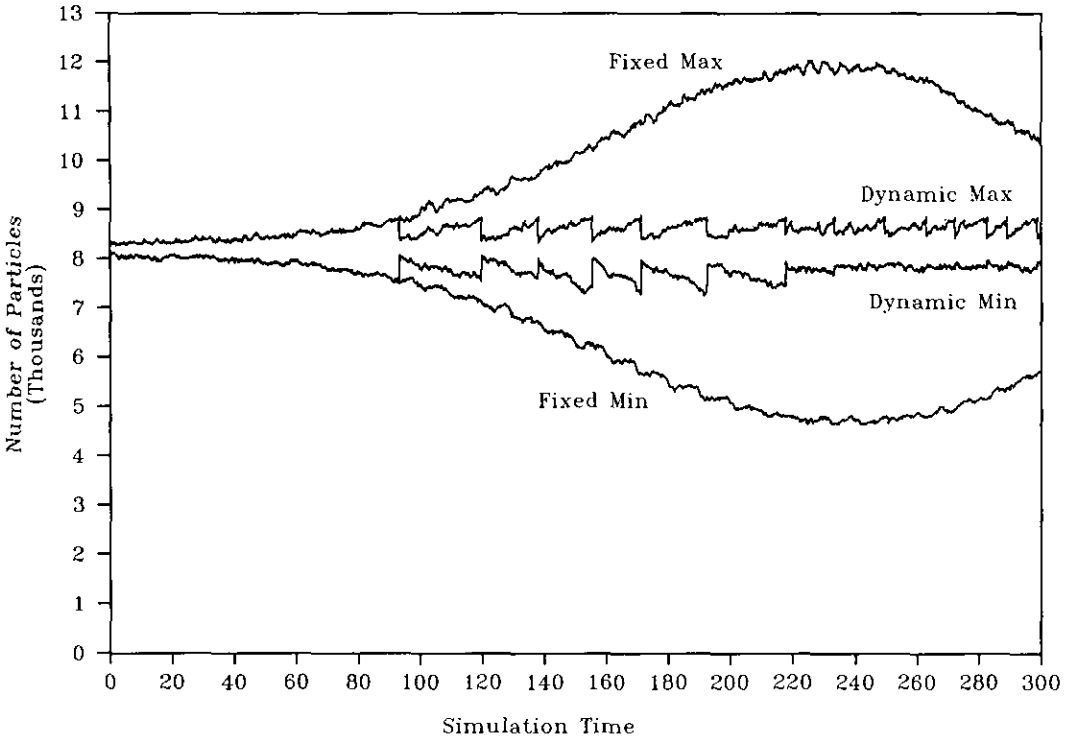

FIG. 6. The maximum and minimum particle counts in any processor at each timestep during the lower hybrid heating simulation for the cases of static fixed partitions and dynamic load balanced partitions run on 64 processors. There are five timesteps per unit simulation time. With the fixed partitions, particles concentrate in processors whose partitions happen to coincide with the density peaks depicted in Fig. 5, resulting in load imbalance. Dynamic repartitioning limits the maximum excursion from ideal load balance.

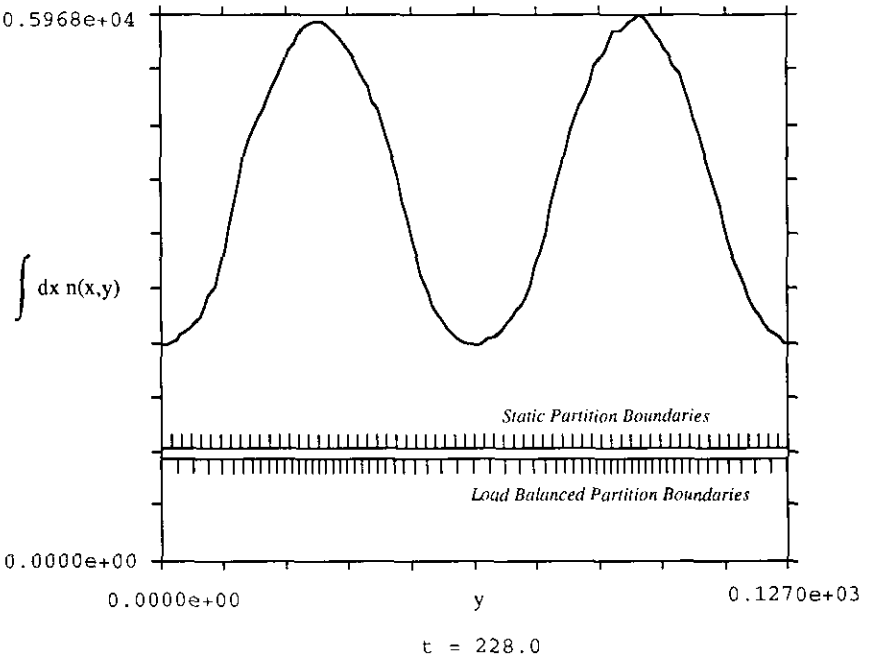

FIG. 7. The 1D number density function used in calculating partition boundaries for dynamic load balancing. The function was computed at the same timestep as the density contours of Fig. 5. The location of the particle partition boundaries for a static regular decomposition and a dynamic load balanced decomposition computed from the function have been added for comparison.

we plot the number density function calculated at the same timestep as the contour plot of Fig. 5. The new partition boundaries are calculated by solving the following equation for y_l in each processor:

$$p\frac{N_p}{N_{\text{proc}}} = \int_0^{y_l} dy\, n(y). \tag{5}$$

Here, again, N_p is the total number of particles in the simulation, N_{proc} is the total number of processors, p is the logical processor number, y_l has been previously defined as the left partition boundary, and $n(y)$ is the grid based density function described above. The right partition boundary for each processor is simply the left partition boundary of the next higher number processor, i.e.,

$$y_r(p) = y_l(p+1) \tag{6}$$

and we have defined

$$y_l(N_{\text{proc}}) \equiv L_y. \tag{7}$$

Again, L_y is the system length in the y direction. The number density function is computed in parallel piecewise by processor in the old partition, then globally combined so that each processor has its own copy of the complete function. Each processor computes its own left boundary in parallel from Eq. (5) and then receives from its right neighbor its right boundary. The cost of this operation is independent of the number of particles. Rather, it scales with the number of grid points. Particle partition boundaries for 64 processors which resulted from the density function of Fig. 7 are noted in the figure, along with the boundaries for the static regular partition.

Determining when repartitioning is necessary is a trivial task. Each processor knows what its ideal particle count should be and can compare that number to its current particle count. An imbalance threshold is set at the beginning of the simulation. At every time step, before the charge deposition, each processor determines whether its current particle count exceeds the ideal by more than the imbalance threshold. If any processor signals a threshold exceeded condition, a repartition is undertaken.

For dynamically changing particle partitions, the transformation between the particle and field partitions is no longer a simple nearest neighbor exchange of guard cell rows. The field grid rows required for interpolation in each processor are now given by the expressions:

$$n_{y\,\text{left}} = \text{INT}(y_l + 0.5) - 1, \tag{8a}$$

$$n_{y\,\text{right}} = \text{INT}(y_r + 0.5) + 1, \tag{8b}$$

where INT is the truncate to integer function. A general means of exchanging grid information among processors must now be provided. In some cases, the section of the grid assigned to a processor in the field partition will be completely different from the section assigned to it in the particle partition. To accomplish the transformation from one partition to the other, we maintain tables of row destinations for both transform directions which are computed at repartition time. Processor-addressed generalized communication calls available on the hypercubes are used in conjunction

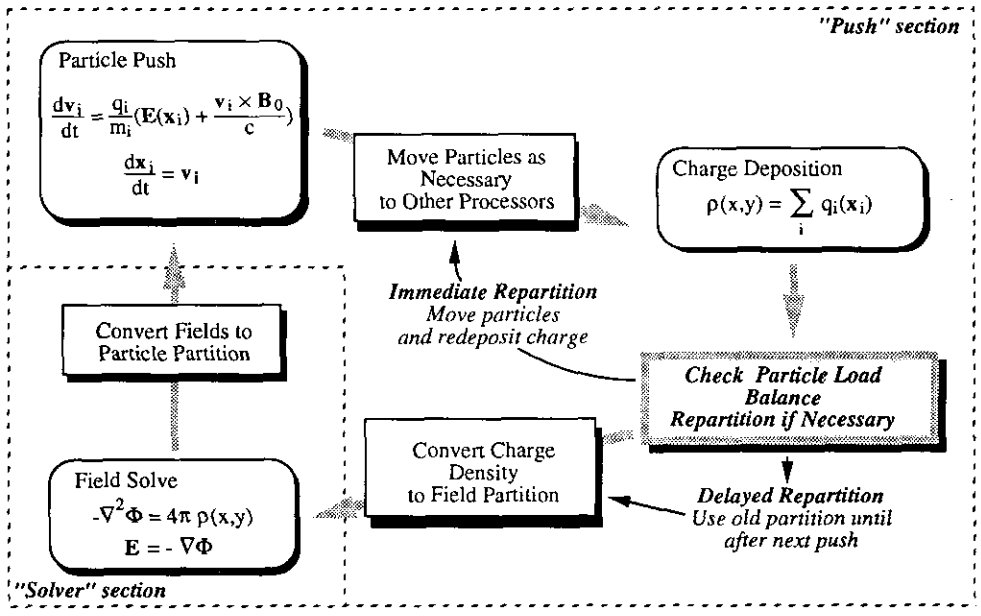

FIG. 8. GCPIC with dynamic load balancing. An additional step is performed in the timestepping loop to check for particle load imbalance. If load balance is within acceptable limits, the loop continues normally. If the load imbalance threshold is exceeded, repartitioning takes place. Immediate repartitioning requires backing up to the particle mover in the loop. Delayed repartitioning only notes the number density function for use at the next timestep.

with these tables to send each row of field grid or charge grid to its appropriate destination. This method of communication is more costly than the nearest neighbor exchange which can be used in the regular partition case.

To implement repartitioning for dynamic load balancing, the iteration loop illustrated in Fig. 1 is modified to that of Fig. 8. After the charge deposition, particle load balance is checked. If a repartition is required, the 1D number density function is computed from the current 2D charge density grids. Two possible ways of proceeding now are possible:

An immediate repartition could be undertaken. In this case, the new boundaries and field transformation tables are computed. Then control is transferred back to the particle mover to recheck current particle locations and move particles to new processors as necessary. Next a second charge deposition is done in the new partition. The load balance check is disabled for this second pass to prevent any possibility of an infinite loop condition, so control passes to the field solver.

The other possibility is to delay the repartitioning until the next particle move which follows every push. In this case, the 1D number density function is saved until after the next particle advance, but before particles are moved among processors due to excursion across particle boundaries. At this point the new partitions and field transformation tables are computed. Then the particle move proceeds normally with the new partition boundaries.

The delayed repartition requires no extra charge deposition as does the immediate repartition, but it will never achieve as good a rebalance result since the partition information used is always one time step old. This will turn out to be unimportant, as we will demonstrate in the next section. Dynamic repartitioning when subcycling is used for one of the species is also easily accomplished using this scheme. One can simply force an extra push of the subcycle species coincident with the load balancing or delay the load balancing until the next subcycle push. We would favor the delayed approach for reasons of simplicity.

The question of when to rebalance remains an issue. It is counterproductive to attempt load balancing at every time step, since there is some overhead associated with the construction of the 1D number density function and the computation of new partitions and field transformation tables. The optimum threshold level is a complicated function of the size of the grid, number of particles, and the cost of communications. Therefore, we have adopted an experimental approach to determining a threshold for load balancing. We report results for several threshold levels in the next section.

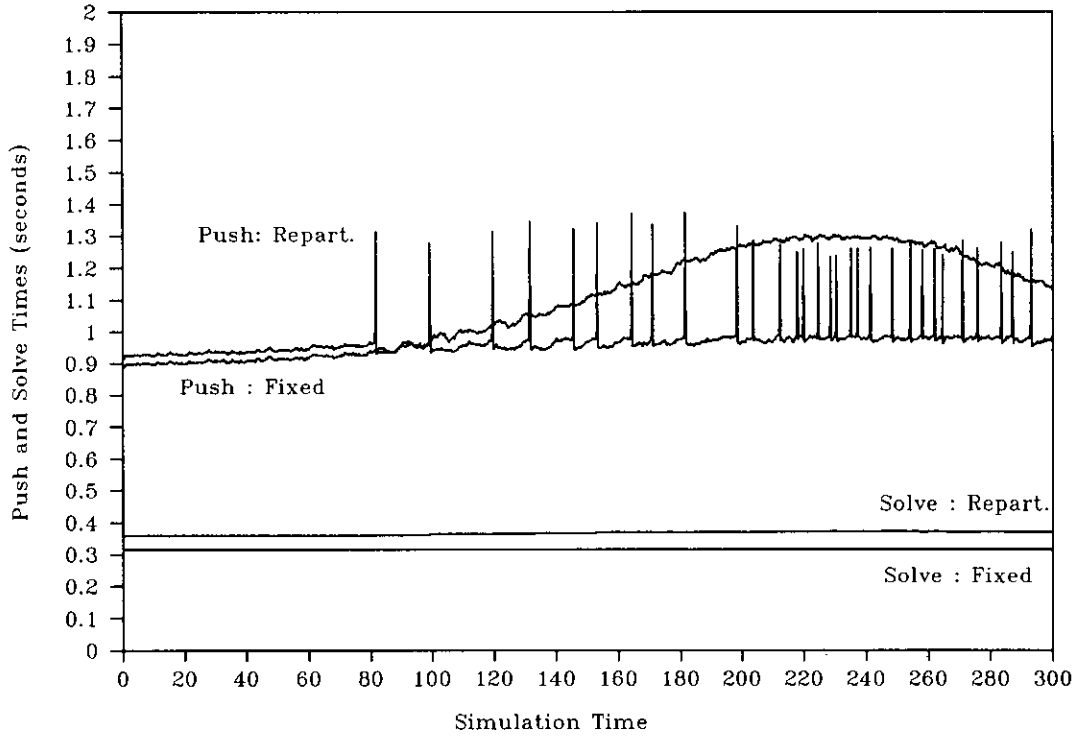

FIG. 9. Push and solver execution times at each timestep using static and dynamic particle partitions, run on a 64 processor Mark IIIfp Hypercube. The simulation parameters were the same as those of Fig. 5. For dynamic partitioning, both immediate and delayed repartitioning methods were run. Solver execution times were virtually identical for both dynamic partitioning methods; only one curve is distinguishable. The push time is shown for delayed repartitioning. The spikes in the repartition push execution times correspond to timesteps at which new partitions were calculated. The immediate repartition push times for normal timesteps are essentially the same as those of the delayed repartition push. For repartition time steps, the immediate repartition increases the cost by about 30%.

IV. RESULTS

We have implemented the load balancing scheme described above in our 2D parallel code and compared code performance with and without load balancing for several situations. In Fig. 6 we have also plotted the maximum and minimum processor particle loads at each time step for the lower hybrid heating simulation run with dynamic load balancing. A load imbalance threshold of 0.08 was used for this run (i.e., when the number of particles in any one processor exceeded the ideal particle load by 8%, load rebalancing was undertaken). Load rebalancing does not take place for some time, and then only sporadically at first. The points at which a rebalance takes place are clear from the vertical jumps in particle number. Obviously, the load imbalance never significantly exceeded 8% in this run, while the fixed partition run suffered from imbalances which were 50% at times.

The overhead of using dynamic load balancing may be observed by looking at the execution times for the push and field solve code sections in comparisons to those from the fixed (static regular) partition code. In Fig. 9 we have plotted these times at every timestep for the lower hybrid heating simulation of Fig. 5 described previously. Runs were done with fixed partitioning, dynamic load balancing with immediate repartitioning, and dynamic load balancing with delayed repartitioning. An imbalance threshold of 6% was used in the dynamic load balancing cases.

Figure 9 shows the times for the field solve and the push times for the cases static case and the case with delayed repartitioning. The overall improvement in the push time with load balancing is apparent. The push time for the static fixed partition case shows a gradual increase in the push time as the particle distribution becomes nonuniform. The push time for the delayed repartition case shows small increases preceding the spikes which occur at time steps with repartitioning; following repartitioning, the push time drops back down to approximately the minimum "load balanced" level. The overhead for a delayed repartitioning step is actually spread over two consecutive time steps, since the calculation of the 1D number density function takes place at the time step before the actual repartitioning. The time to do this calculation is too small to be observed here.

The times for the field solver were virtually identical for the immediate and delayed repartitioning runs. The difference in times for the field solver between the fixed regular partition and dynamic partition cases (Fig. 9) is due entirely to the difference in communication schemes used in transforming from the field decomposition to the particle decomposition. (Recall that we have assigned the field to particle partition transformation to the field solver and the particle to field partition transformation to the push.) The fixed partition run uses nearest neighbor exchanges which involve only the guard cell information, while the dynamic partition run requires the general communication scheme involving all field quantities. This cost is also apparent in the time difference between the fixed and dynamic partition push seen in Fig. 9 at the beginning of the run, where the particle load is still uniform across processors. The overhead of the generalized communications scheme makes the dynamic load balancing code slower than the fixed partition code for problems which do not exhibit particle load imbalance. Thus dynamic load balancing is not always a desirable feature to be included. It should be emphasized that this overhead is present in every timestep and is the result of having to allow for the transformation between a regular partitioning of the grid (required for load balancing the field solver) and an irregular partitioning of the grid (required for load balancing the push when large density inhomogeneity is present). Even if the partitions never change, simply allowing for the possibility adds a fixed additional cost to the solver and the push. This cost scales with the ratio of grid size to the number of processors and is independent of the number of particles.

For the load-balanced cases, the cost of an immediate repartition push time step is about 30% higher than the delayed repartition push time step, due to the extra particle reassignment and charge deposition. For normal push time steps, the push times for the two schemes are essentially the same. For time steps when no repartitioning is required, the push time differs from ideal by the small residual load imbalance and the partition transformation overhead. There are a similar number of repartition steps in each scheme during the course of the runs. The total execution times for the two cases, however, are almost identical (within a few tenths of a percent). Apparently, the immediate repartition scheme makes up the cost of the extra charge deposition and particle reassignment by achieving a better average load balance (and push execution time). This characteristic appears to be a general one; in every comparison we have done between the immediate and delayed repartitioning, we have failed to observe a significant difference in total execution time between the two.

To judge the effectiveness of dynamic load balancing, we ran the same physics simulation (lower hybrid heating at resonance) with dynamic load balancing and with static regular partitioning, for several different grid sizes, total particle counts, and number of processors. In all cases, the maximum particle load imbalance observed during the simulation using the static regular partitioning exceeded 50%. For a given grid size, total number of particles, and total number of processors, we ran the dynamic load balanced code with several imbalance thresholds and compared the total execution time to that for the static regular partition code. The run length was chosen in each case so that the simulation ran through one complete cycle of cavity formation and collapse. The total execution time for each run was measured and the ratio of load-balanced run time

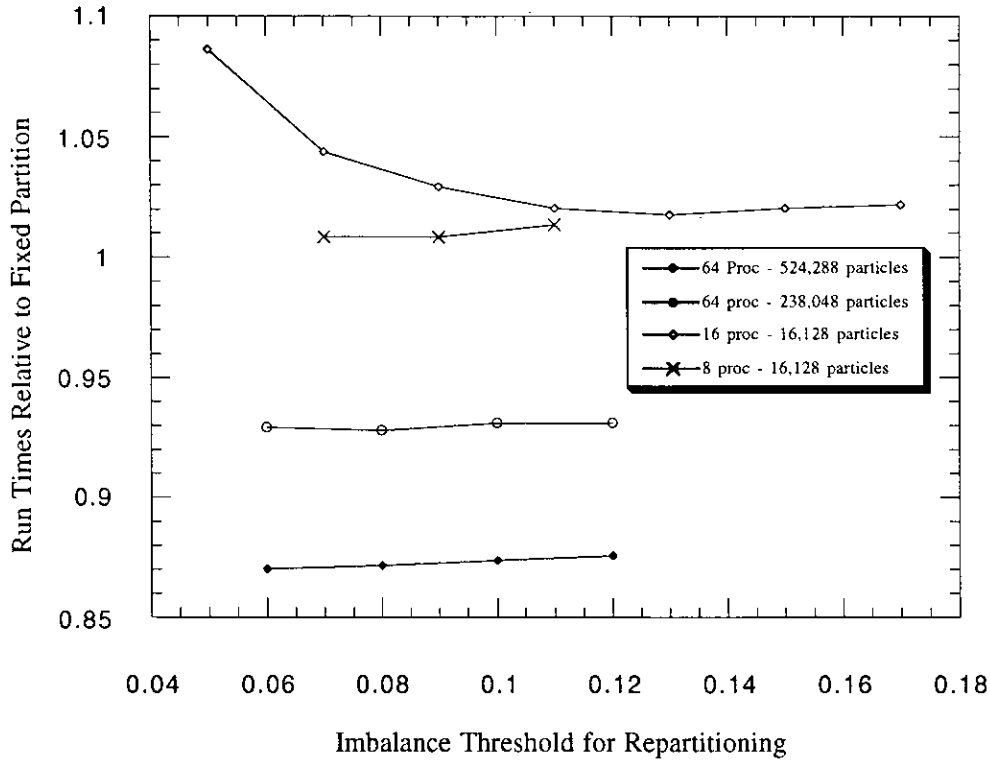

FIG. 10. Relative total run times for several dynamic load balancing cases as a function of imbalance threshold. The points plotted are the ratio of total execution times on a Mark IIIfp Hypercube for dynamic load balanced runs compared to the static partition run using the same simulation parameters. The runs with 16,128 particles were done using a 32×128 grid. The runs of 64 processors were done using a 128×128 grid. Dynamic load balancing can be counterproductive if the particle push does not dominate the main loop, as is the case for the runs using 16,128 particles.

to regular partition run time was computed. In Fig. 10 we have plotted these ratios for the various test cases as a function of the imbalance threshold. The most striking result shown here is that load balancing is not always better than the static regular partitioning we have used previously. For the smaller simulations, the extra overhead from the use of generalized communications for the partition transformations nullified any gain due to better particle load balance. It was necessary to go to large numbers of particles and higher numbers of processors before dynamic load balancing became effective. The second observation to be made is that the variation of run time with imbalance threshold is fairly flat over some range, indicating that the particular value of the threshold is not critical for best performance.

Dynamic load balancing appears to be useful only when the particle push time dominates the main loop execution time *and* significant load imbalance develops during the simulation. These conditions depend upon machine characteristics. Refer again to Table I. On the Mark IIIfp for the benchmark problem, the particle push time always dominates the iteration loop. Dynamic load balancing is *potentially* useful here. On the Intel i860 Gamma machine, the field solver dominates on 32 processors. Dynamic load balancing would be counterproductive for this case. The push dominance condition places a lower limit on particle number density in the simulation. Assuming that the FFT-based solver is executing at its most inefficient limit of two 1D FFTs per processor, this means particle number densities on the order of eight per cell are the approximate break-even point for the Mark IIIfp Hypercube. On other MIMD computers, this number will be different due the different computation/communication speed ratio. For higher ratio machines, like the Intel Gamma machine, the number will be higher. The cost for dynamic load balancing is dominated by the cost of the generalized communications required for the field redistributions. Any improvement must come from a more efficient field redistribution.

V. CONCLUSIONS

We have implemented a method for dynamic load balancing of particles in a 2D plasma PIC simulation code which runs on MIMD computers. The code has previously used a static regular 1D partition of particles among processors. The 2D gridded charge density functions which arise from the interpolation of particle charge to the field grid in preparation for the field solver are used to construct,

in parallel, a 1D number density function. New partition boundaries which result in particle load balance among the processors are computed from this function. Load balancing is undertaken whenever any single processor's particle count exceeds the ideal particle count by some threshold fraction, typically around 6%. The variation of total execution time as a function of threshold fraction is fairly constant throughout some range, provided the fraction is large enough.

Load balancing is found to be counterproductive, compared to the static regular partitions used previously, unless the particle push dominates over the field solver, and significant particle load imbalance develops. Since the particle push parallelizes with high efficiency, while our FFT-based field solver does not, the condition of push dominance is not as easily achieved as it is on sequential machines. The particle push must dominate execution because the extra overhead associated with generalized communication of field quantities between the particle decomposition and the fields decomposition is substantial. Gains in push efficiency must exceed this overhead cost before dynamic load balancing becomes useful. For fusion or space plasma simulations with this field solver, a judicious choice of static regular partitions will result in acceptable particle load balance in all but the most non-uniform of density variations. It is only for large dynamic density variations that load balancing will ever be required when using this field solver. A means of redistributing field quantities between the particle decomposition required by the push and the field decomposition required by the field solver which entails substantially less overhead could make dynamic load balancing useful in a larger number of cases.

A more efficient parallel field solver would increase substantially the number of cases in which the particle push still dominates the iteration loop, and hence it would make dynamic load balancing of use. Recall that the field solver time was dominated by the communication time involved in the field data transpose required by the FFT, leading to a poor overall code efficiency. If one uses an explicit finite-difference method to advance the field equation, which does not require a transpose, the efficiency of the field solver will improve dramatically. Explicit finite-difference solutions of Maxwell's equation for electromagnetic scattering problems have shown very high ($>90\%$) parallel efficiency [2]. The only communication required in this case is the communication of the guard cells and the communication to move data from the particle to the field decomposition, both of which are much smaller than the transpose time. Thus the method presented here should scale well to finite-difference electromagnetic PIC codes.

Although this work has only treated a 1D parallel decomposition, the basic scheme can be extended to higher dimensional decompositions. The generalization of this load balancing scheme to 2D partitions could be done as a two-stage decomposition. The 1D partitions we have discussed here could be further subdivided along the direction orthogonal to these divisions. Alternatively, 2D load balanced partitions could be created from the density distribution using the recursive bisection algorithm [1]. In a 3D PIC simulation, one-, two-, or three-dimensional partitions could be created by either of these schemes. Thus we conclude that the method presented here should lead to an efficient parallel implementation of three-dimensional electromagnetic PIC codes with dynamically changing particle distributions.

ACKNOWLEDGMENTS

The authors thank Professor John Dawson for his continued interest and encouragement in this project. Part of the research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored in part by Sandia National Laboratory, DARPA under Contract NAS7-918, and the National Science Foundation under cooperative agreement CCR-8809615, through an agreement with the National Aeronautics and Space Administration. One of the authors (V.K.D.) was supported in part by the National Science Foundation under Contract PHY89-22487 and in part by the Department of Energy under Contract DE-FG03-86ER53223.

REFERENCES

1. G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors*, Vol. 1 (Prentice–Hall, Englewood Cliffs, NJ, 1988).
2. R. D. Ferraro, "Solving PDEs for Electromagnetic Scattering Problems on Coarse Grained Concurrent Computers," in the PIER volume *Computational Electromagnetics and Supercomputer Architecture*, Elsevier Science, New York, in press.
3. J. E. Patterson, R. D. Ferraro, and L. Sparks, "High Performance Remote Sensing Data Analysis Using Parallel Computation," Proceedings, AIAA/NASA Second International Symposium on Space Information Systems, Sept. 17–19, 1990, Pasadena, CA.
4. D. W. Walker, *Concurrency Pract. Exper.* **2**, 257 (1990).
5. C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation* (McGraw–Hill, New York, 1981).
6. P. C. Liewer and V. K. Decyk, *J. Comput. Phys.* **85**, 302 (1989).
7. R. D. Ferraro, P. C. Liewer, and V. K. Decyk, in *Proceedings, 5th Distributed Memory Computing Conference* (IEEE Comput. Soc., Press, 1990).
8. P. C. Liewer, E. W. Leaver, V. K. Decyk, and J. M. Dawson, in *Proceedings, 5th Distributed Memory Computing Conference* (IEEE Comput. Soc., Washington, DC, 1990).
9. V. K. Decyk, *Supercomputer* **27**, 33 (1988).
10. T. Krücken, P. C. Liewer, R. D. Ferraro, and V. K. Decyk, in *Proceedings, 6th Distributed Memory Computing Conference* (IEEE Comput. Soc., Los Alamitos, CA, 1991), p. 452.
11. V. K. Decyk, J. M. Dawson, and G. Morales, *Phys. Fluids* **22**, 507 (1979).