JOURNAL OF COMPUTATIONAL PHYSICS 85, 302–322 (1989)

A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes

PAULETT C. LIEWER

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

AND

VIKTOR K. DECYK

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

Received June 23, 1988; revised December 14, 1988

We have developed a new algorithm for implementing plasma particle-in-cell (PIC) simulation codes on concurrent processors with distributed memory. This algorithm, named the general concurrent PIC algorithm (GCPIC), has been used to implement an electrostatic PIC code on the 32-node JPL Mark III Hypercube parallel computer. To decompose a PIC code using the GCPIC algorithm, the physical domain of the particle simulation is divided 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 updating the particles in its sub-domain. This algorithm has led to a very efficient parallel implementation of a well-benchmarked 1-dimensional PIC code. The dominant portion of the code, updating the particle positions and velocities, is nearly 100% efficient when the number of particles is increased linearly with the number of hypercube processors used so that the number of particles per processor is constant. For example, the increase in time spent updating particles in going from a problem with 11,264 particles run on 1 processor to 360,448 particles on 32 processors was only 3% (parallel efficiency of 97%). Although implemented on a hypercube concurrent computer, this algorithm should also be efficient for PIC codes on other parallel architectures and for large PIC codes on sequential computers where part of the data must reside on external disks. © 1989 Academic Press, Inc.

I. INTRODUCTION

VLSI technology has led to a dramatic decrease in the cost of computing power. By using this technology, concurrent or parallel processors with performance comparable to today's supercomputers can be built at a modest cost, thus potentially making supercomputers widely available. In the near future, concurrent processors with even greater performance will become available which, in principle, can be

0021-9991/89 $3.00
Copyright © 1989 by Academic Press, Inc.
All rights of reproduction in any form reserved.

used to address problems now beyond the limits of existing supercomputers. However, before the full potential of these machines can be realized, new algorithms must be developed for implementing codes efficiently on concurrent processors.

In this paper, we present a new algorithm for implementing one class of codes, plasma particle-in-cell (PIC) simulations [1, 2], on concurrent processors. In particle simulations of plasmas, the orbits of thousands to millions of individual plasma electrons and ions are followed as an initial value problem, where the particles move in the electric and magnetic fields calculated self-consistently from the charge and current densities created by these same particles. Particle simulations are used to study a wide variety of nonlinear problems in many areas of plasma physics research such as magnetic and inertial fusion, space and astrophysical plasmas, electron and ion beam propagation, free electron lasers and particle accelerators.

The general concurrent PIC (GCPIC) algorithm, introduced in this paper, is designed to make the most computationally intensive portion of a PIC code, the particle computations, run efficiently on a MIMD parallel processor with distributed memory. (In a multiple-instruction multiple-data (MIMD) parallel computer, each processor may execute a separate stream of instructions. This is in contrast to a single-instruction multiple-data (SIMD) parallel computer in which the same instruction is executed simultaneously in each processor.) To implement a code using the GCPIC algorithm, the physical domain of the simulation is divided 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 single sub-domain. The processor stores the particle and electromagnetic field information for this sub-domain and performs the computations for the particles in this sub-domain. Since each processor is responsible for approximately the same number of particles, the processors' loads are balanced for the dominant portion of the code, the particle computations, even when the particle spatial distribution is extremely non-uniform. Moreover, as the spatial distribution of the particles evolves in time, the sub-domains can be re-created to ensure that the processor loads remain balanced (*dynamic load balancing*). When a particle moves from one sub-domain to another, it must be passed to the appropriate processor.

The GCPIC algorithm for parallel decomposition has been used to implement a widely benchmarked UCLA 1-dimensional electrostatic PIC code [3, 4] on the 32-node JPL Mark III Hypercube, a concurrent processor with distributed memory. The resulting parallel Fortran code has proven to be quite efficient. For a problem with a fixed **total** number of particles, the increase in speed in going from 1 to 32 processors for the dominant portion of code, the particle computation (*push time*), was 29.5, or 92% efficiency for a *fixed problem size*. Here, the *push time* includes the time to update the particle positions and velocities (including the interpolation to find the forces at the particle positions) and the time to *deposit* (interpolate) the particles' contributions to the charge and/or current densities onto the

discrete grid. When the number of particles increased linearly with the number of processors used, so that the number of particles per processor was constant, the push algorithm was nearly 100% efficient. For example, the increase in time spent updating particles in going from a problem with 11,264 particles run on 1 processor to 720,896 particles on 64 processors was only 3%, or 97% efficiency for *fixed grain size.* Note that these efficiencies were found for a 1-dimensional PIC code. High push efficiencies should also be attainable in 2-and 3-dimensional codes using the GCPIC algorithm provided the fraction of particles moving between processors at each time step remains small. For these runs, the particle density remained relatively uniform in space and processor load imbalance did not become a problem. Although implemented on a hypercube concurrent computer, this algorithm should also be efficient for PIC codes on other parallel architectures with distributed memory and on sequential computers where part of the data resides in external memory.

The structure of this paper is as follows. In Section II, the GCPIC algorithm is described. In Section III, its implementation in a 1D electrostatic code running on the Mark III Hypercube is described and the portability of the code to other computers is discussed. In Section IV, the Mark III Hypercube hardware is described and the performance of the parallel code on the Mark III is analyzed. Timing comparisons with other supercomputers are also presented. Section V contains the conclusions.

II. GCPIC Algorithm

Plasma particle-in-cell (PIC) codes [1,2] calculate the orbits of $10^4 - 10^6$ plasma electrons and ions as an initial value problem, where the plasma particles are moving in the electromagnetic fields determined from Maxwell's equation (or a subset) with the plasma charge and/or current density as the sources. In PIC codes, the particles can be located anywhere in the spatial domain of the simulation, but the electromagnetic field equations are solved on a discrete grid.

Each iteration, or time step, in a PIC code consists of two stages. In the first stage, the *particle push,* all particle velocities and positions are advanced on time step using the present values of the electromagnetic fields using a discretized version of

$$\frac{d\mathbf{v}_i}{dt} = \frac{q_i}{m_i}\left(\mathbf{E} + \frac{\mathbf{v}_i \times \mathbf{B}}{c}\right) \tag{1}$$

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i. \tag{2}$$

Since the fields are defined only at grid points, an interpolation is used to find the fields and forces at the actual positions of the particles. The new plasma-generated

charge and current densities are calculated from the new velocities and positions of the particles using a discretized version of

$$\rho_q(\mathbf{x}, t) = \sum_{i=1, n_{\text{part}}} q_i \delta(\mathbf{x} - \mathbf{x}_i) \tag{3}$$

$$\mathbf{j}(\mathbf{x}, t) = \sum_{i=1, n_{\text{part}}} q_i \mathbf{v}_i \delta(\mathbf{x} - \mathbf{x}_i). \tag{4}$$

In the second stage of each time step, the electromagnetic fields at the new time are calculated by solving Maxwell's equations (or a subset) on the discrete grid with the new charge and/or charge densities as the sources. Many different varieties of PIC codes have been developed [1–3, 5–7]. The electromagnetic fields may be solved in Fourier space using fast Fourier transforms (FFT) or in configuration space using finite-difference techniques. A PIC code can be explicit or partially implicit [5] in time. The choice of methods is strongly problem dependent.

The general concurrent particle-in-cell (GCPIC) algorithm is a method for dividing such plasma particle-in-cell simulation problems among the N_p processors of a concurrent processor with distributed memory. In a distributed memory concurrent processor, each processor has its own local memory; there is no global or shared memory. The key feature of the GCPIC algorithm is that two distinct spatial decompositions of the physical domain are used to map the problem onto the parallel processors. The two decompositions mirror the two stages of a PIC code: a *primary decomposition* is used to divide the particles and particle computation efficiently among the processors and a *secondary decomposition* is used to divide the electromagnetic field computation among the processors. The primary decomposition is designed to make the dominant portion of a PIC code, the particle push, run efficiently in parallel. For the primary decomposition, the physical domain of the simulation is divided into N_p sub-domains such that these sub-domains have roughly equal numbers of particles. For problems with non-uniform particle densities, these sub-domains will be of unequal physical size, and, in general, will contain unequal numbers of grid points. Figure 1 shows such sub-domains for an eight node concurrent processor for 1- and 2-dimensional simulations for problems with uniformly spaced grid points. In both examples, the eight sub-domains have equal numbers of particles, but unequal numbers of grid points. Each processor is assigned a sub-domain and is responsible for storing and pushing the particles in this subdomain and, in addition, for storing the electromagnetic field arrays (charge densities, electric fields, etc.) for the grid points of its assigned sub-domain. As a particle moves to a new sub-domain, the particle is passed to the processor assigned that sub-domain in this primary decomposition. The primary decomposition is also used for most of the diagnostics.

The primary decomposition of the GCPIC algorithm, used to divide the particles among the processors, was motivated by two main considerations. To implement any code efficiently on a concurrent processor, one must (1) balance the computation (and memory) load uniformly among the processors and (2) minimize the

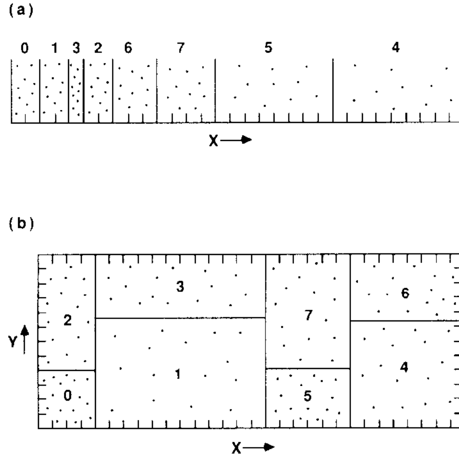


FIG. 1. Examples of primary sub-domains created for an 8-node parallel processor using the GCPIC algorithm for simulations with uniform grid spacing. Sub-domains have equal numbers of particles, but unequal physical size. (a) 1D simulation and (b) 2D simulation.

time spent in communication between the processors. The time and memory requirements of a PIC code are dominated by the particle computation. Therefore, the processor loads are most evenly balanced when they have nearly equal numbers of particles; the GCPIC algorithm accomplishes this by having the sub-domains in the primary decomposition created with nearly equal numbers of particles. To minimize inter-processor communication in a parallel PIC code, a processor should store the portions of the electromagnetic field grid arrays used in the interpolations in the particle computation for all of its assigned particles. Otherwise, the update of each particle could require one communication to a distant processor to obtain the forces and another to "deposit" the charge in the charge and current density array. Therefore, in the GCPIC algorithm, each processor stores the electromagnetic field arrays for the grid points in its physical sub-domain. Since these sub-domains may contain unequal numbers of grid points, the grid arrays for different processors may vary in size.

Dynamic load balancing can easily be incorporated into a code using the GCPIC algorithm: If processor load imbalance occurs during a run as the plasma density evolves, sub-domains of the primary decomposition can be re-created so that the processors particle loads are again balanced.

A hypercube computer of dimension d is an ensemble of 2^d independent processors, each with its own local memory. There is no shared or global memory. Communication is through channels which connect the processors in a hypercube topology. To assign the sub-domains of the primary decomposition to the processors in a hypercube, the hypercube is "configured" in the same spatial dimensionality as the simulation grid, i.e., the d-dimensional cube is symbolically reduced to a cube of the same dimension as the simulation by ignoring some of the hyper-

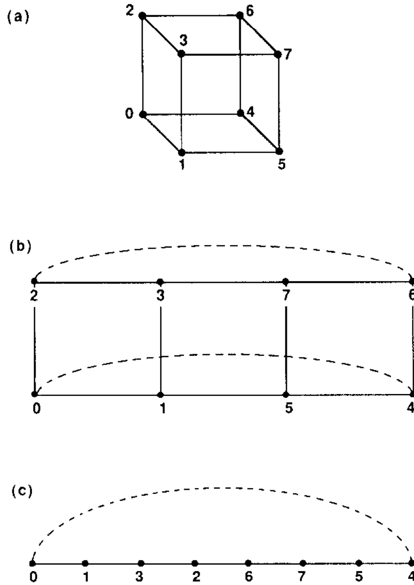


FIG. 2. Hypercube communications connections. Processors are located at large dots. (a) Communication connections for an 8-node (3-dimensional) hypercube. (b) An 8-node hypercube "configured" in two dimensions. (c) An 8-node hypercube "configured" in one dimension. In (b) and (c) the unused communication channels are not shown; dotted lines are communication channels used for periodic boundary conditions.

cube communication connections. This is illustrated in Fig. 2 where an 8-node ($d=3$) hypercube is shown configured in 3-, 2-, and 1-dimensions. It can be seen that lower dimensionality configurations are obtained by "cutting" some of the communication connections. In a hypercube, the processors are numbered so that directly connected processors differ by only one bit in the binary representation of the processor number, *e.g.*, processor 7 (*binary* 111) is directly connected to processors 3 (*binary* 011), 5 (*binary* 101), and 6 (*binary* 110). Nearest neighbor sub-domains are assigned to nearest neighbor processors as illustrated in Fig. 1 for both the 1- and 2-dimensional examples. Thus in one dimension, when a particle moves to a neighboring sub-domain, it is passed to a processor directly connected. In two dimensions, two passes will be required to pass particles which move to diagonal neighbors, e.g., in Fig. 1b, if a particle in sub-domain 7 moves to sub-domain 1, it would be passed first to processor 3 or 5 and then to processor 1.

The primary domain decomposition of the GCPIC algorithm, optimized for the concurrent particle computations, will not generally be the optimum decomposition for solving the electromagnetic field equations in parallel. Therefore, in the GCPIC algorithm, a secondary decomposition is used for the field solution. This secondary domain decomposition is chosen to make the concurrent field solution efficient and is used only for this and related diagnostics. The specific decomposition chosen will

be different depending on whether a fast Fourier transform (FFT) method, a finite difference method, or a multi-grid method is used. In general, we anticipate that the most efficient decomposition for the field solution will involve dividing the number of grid cells equally among the processors. Note that if this decomposition were also used for dividing the particles among processors, load imbalance could result whenever the number of particles per cell was non-uniform.

At each time step, prior to solving the field equations, the relevant grid arrays (charge, current densities) are passed among the processors and re-distributed as needed in the secondary decomposition. After the field equations are solved, the electromagnetic field arrays must be passed among the processors so that they are distributed as needed in the primary sub-domain decomposition for the parallel particle computation. *These two redistributions of grid arrays between the primary and secondary decomposition at each time step are inherent in the GCPIC algorithm and are necessary to make both the particle and field portions of a PIC code efficient on a parallel procesor.* Under certain circumstances, the primary and secondary decompositions will be the same and thus no redistributions are necessary. In this case, parallel code development would be simplified. The two decompositions will be the same, for example, if a finite difference method is used for the electromagnetic field solution and if an adaptive grid is used such that the number of particles per grid cell is uniform.

Although implemented on a hypercube parallel computer, the GCPIC algorithm should also be highly efficient on other parallel computers with distributed memory. The two main issues which motivated the GCPIC algorithm, processor load balance and minimum internode communication, remain the same for any parallel computer with distributed memory regardless of the topology of the internode connections.

The GCPIC algorithm should also be efficient for running large problems on sequential machines where portions of the data to be processed must be stored in external memory, e.g., on a solid state disk. In particular, the primary decomposition would be an efficient way of dividing up the particle and field data where now the sub-domain size would be determined by the size of the internal memory of the sequential computer. In this case, the individual sub-domains (including both particles and fields) would be brought off disk and into memory one at a time. Internode communication would be replaced by temporary storage used to pass information from one data block to another.

III. Implementation of GCPIC on the JPL Mark III Hypercube

The GCPIC algorithm for parallel decomposition has been used to implement a widely benchmarked UCLA 1-dimensional electrostatic PIC code [3, 4] on the JPL Mark III Hypercube in Fortran. In this code, named BEPS1, spatial variation is allowed only in the x direction and magnetic fields and forces have been neglected. Poisson's equation is used to determine the electric field from the

plasma charge density, Eq. (3). The velocity and position of each ion and electron ($i=1, n_{\text{part}}$) is advanced in time using the simple leap frog algorithm

$$v_i\left(t+\frac{1}{2}\Delta t\right)=v_i\left(t-\frac{1}{2}\Delta t\right)+\frac{F_i(t)}{m_i}\Delta t \tag{5}$$

$$x_i(t+\Delta t)=x_i(t)+v_i\left(t+\frac{1}{2}\Delta t\right)\Delta t. \tag{6}$$

Here, F_i is the force due to the x electric field, $F_i(t)=q_iE_i(t)$, at the position of the ith particle. The particle positions can be anywhere in the physical domain of the problem, but to calculate the electric field, the physical domain is represented as a discrete grid x_g with uniform spacing Δx. The particle positions are interpolated onto this grid to determine the charge density $\rho_q(x_g, t+\Delta t)$ at the grid points using a quadrupole interpolation. The resulting electric field $E(x_g, t+\Delta t)$ is determined by solving Poisson's equation

$$\frac{dE(x, t)}{dx}=4\pi\rho_q(x, t) \tag{7}$$

in Fourier space using a fast Fourier transform (FFT). In BEPS1, this equation is solved in Fourier space both for periodic and bounded systems; for bounded plasmas, the necessary homogenous solution is added to the FFT solution [8]. The force $F_i(t)$ on the ith particle is found by a quadrupole interpolation from the values of the electric field at the nearest grid points. In a sequential code, the time to push the particles scales linearly with the number of particles, n_{part}; the computation time to compute the field using the FFT scales as $n_g \ln n_g$, where n_g is the number of grid points. Typically $n_{\text{part}} \gg n_g$, and the time to push the particles dominates over the time to solve for the fields. In the benchmark case for this code, there are 11,264 particles and 128 grid points. For this case, about 90% of the cpu time was spent on the particle push when run sequentially. Of the remaining time, about 8% was spent on diagnostics and 2% on solving for the electric field.

A. *Primary Decomposition*

The GCPIC algorithm, without dynamic load balancing, has been used for the parallel implementation of the UCLA 1D electrostatic code for a problem with an initially *uniform* plasma density. A uniform grid spacing Δx is used in the code as required for the FFT solution of Poisson's equation. Because the particle density is uniform, the 1D grid was divided into N_p sub-domains of equal physical size for the primary GCPIC decomposition for this problem. Since the grid spacing Δx is a constant, these sub-domains also contain equal numbers of grid points. Each processor is assigned one sub-domain and is responsible for computing the updates for the particles in its sub-domain. If the total number of particles is n_{part} and N_p processors are used, each processor initially has $n_{pp}=n_{\text{part}}/N_p$ particles. However,

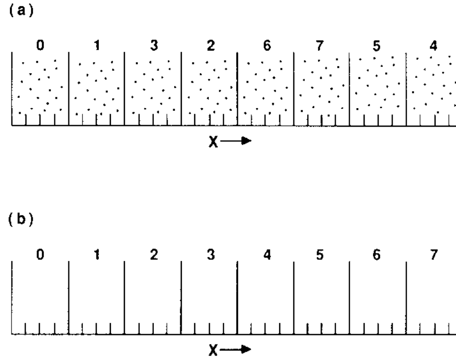


FIG. 3. (a) GCPIC primary decomposition for an 8-node machine showing nearest neighbor processor assignments for the 1D hypercube PIC code with uniform density and uniform grid spacing. (b) Secondary decomposition showing processor assignments for the Fourier transform portions of the code. Because of the two decompositions used, grid data must be redistributed among the processors at each iteration.

as the simulation evolves and particles move among processors, the number of particles per processor n_{pp} will vary.

For this 1D PIC code, the hypercube was configured as a 1D array of processors (cf. Fig. 2c), and the sub-domains were mapped directly onto the processor array, e.g., in nearest neighbor order. The sub-domains and processor assignments for this problem on an 8-node hypercube are shown in Fig. 3a. When a particle passes to a neighboring sub-domain, it is passed from its processor to a directly connected processor and thus all the communication needed for moving particles between processors is nearest neighbor. (Accuracy restrictions on the time step Δt ensure that the particle does not traverse an entire sub-domain in a time step.)

In the parallel code, two processor-dependent variables, x_{left} and x_{right}, define the range of possible coordinates for each processors' particles,

$$x_{\text{left}} \leqslant x_i(t) < x_{\text{right}}, \tag{8}$$

where $x_i(t)$ is the position of the ith particle at time t. The uniform distribution of particles in space at $t = 0$ is accomplished trivially in the parallel code by starting the distribution in each processor at a different spatial location (approximately at x_{left}). The velocities are assigned using a gaussian random number generator. It was decided not to do this in parallel in order to more easily generate the same velocity for each particle regardless of the number of processors actually used. (The user specifies whether to use 1, 2, 4, 8, 16, or all 32 nodes of the Mark III Hypercube.) Thus each processor generates the same sequence of random numbers, but only keeps those numbers needed to assign velocities for its particles.

Each processor also stores the charge, ρ_q and electric field E arrays for only the grid points in its primary decomposition sub-domain, $x_{\text{left}} \leqslant x_g < x_{\text{right}}$, and also for

guard cells at either end of the sub-domain which are needed for the quadrupole interpolations to and from the grid points. Excluding guard cells, in terms of a global *virtual* grid index j, each processor stores and has access to grid points defined by

$$j_{\text{left}} \leqslant j < j_{\text{right}}, \tag{9}$$

where $j_{\text{left}} = \text{INT}\ (x_{\text{left}} + 1.5)$ and $j_{\text{right}} = \text{INT}\ (x_{\text{right}} + 1.5)$. In addition, each processor stores one guard cell to the left of j_{left} and two guard cells on the right, j_{right} and $j_{\text{right}} + 1$ in order to do the interpolations and also parallel graphics. Within each processor, the value stored in the first location in a processors' array, e.g., $E(1)$, corresponds to the location $j = j_{\text{left}} - 1$ in a virtual global array; this is the processors' left-hand guard cell.

Each processor also computes the requested particle diagnostics (e.g., density and potential profiles, distribution function), for its assigned particles and grid points. In the Mark III hypercube, a controlling computer (control processor) combines and outputs the diagnostic information from the individual processors. For trajectory diagnostics and debugging, an extra word per particle can be allotted to store a particle's identity (i.e., a number in the range 1 to n_{part}). When used, this information is passed along with the particles' position and velocity, when it moves from one processor to another.

B. *Secondary Decomposition and Parallel FFT*

When using the GCPIC algorithm to decompose a PIC code on a parallel computer, generally a secondary decomposition is necessary for the field solution portion of the code. For this problem, a secondary decomposition is necessary because the processor assignments in the above primary domain decomposition are not appropriate for the hypercube FFT solution of Poisson's equation. To solve Poisson's equation, the hypercube PIC program uses the hypercube FFT code developed by Salmon and Williams, described in Ref. [9]. To use this code, the physical domain must be divided into N_p *equal* sub-domains and the sub-domains must be assigned to processors in order of the *processors number*, and not mapped directly onto the processor array as in the primary decomposition. This decomposition and processor assignment for the secondary decomposition are shown in Fig. 3b for $N_p = 8$. This secondary decomposition must also be used whenever the diagnostic routines utilize FFTs. For this particular problem with uniform density, both the primary and secondary sub-domains are of equal physical size; when a problem with non-uniform density is run, the primary sub-domains will not be of equal size (cf. Fig. 1), but the secondary sub-domains must still be of equal size because of the hypercube FFT code.

It can be seen that, in general, the decomposition needed for a concurrent FFT is *not* an efficient decomposition for the particles in PIC codes on parallel processors. Since the FFT requires uniform grid spacing and sub-domains of *equal physical size*, using this decomposition for dividing the particles among processors

would result in processor load imbalance whenever the plasma density was non-uniform.

To take the forward $(x \to k)$ Fourier transform of a spatial grid array, first the grid information is redistributed from the primary to the secondary decomposition for the hypercube FFT. The inverse $(k \to x)$ hypercube FFT of a k-space variable distributes the new grid information in the secondary decomposition. After the inverse $(k \to x)$ hypercube FFT is used, the new spatial grid information must also be redistributed among processors in the primary decomposition for storage and future use. In the present hypercube implementation, both types of global redistributions are accomplished by passing the processors information around the ring formed by the 1-dimensional configuration of processors (cf. Fig. 2c). Each processors' present spatial grid array is placed in a communication buffer. This is passed to the processor on the left and a new buffer is received from the neighbor on the right. $N_p - 1$ passes are made so all processors have access to all grid points. At each pass, each processor determines whether to store some or all of the information received in the incoming buffer before passing it on. The information needed to make these decisions (e.g., what grid points are arriving at each pass,) is calculated by each processor in a setup routine at the beginning of the code. Since these redistributions are always needed before an FFT and after an inverse FFT, they have been incorporated into a new Fourier transform subroutine for the parallel code. This routine performs the necessary redistributions and calls the hypercube FFT subroutines.

The inverse hypercube FFT in the $k \to x$ direction also utilizes and expects a particular initial distribution of the Fourier modes among processors (bit-reversed locations, cf. Ref. 9). The particle code uses various constant k-space arrays, e.g., the array of wave numbers $k(j)$, for Poisson's equation and for some of the diagnostics. These are calculated in a subroutine at the beginning of the sequential and parallel codes. In the parallel code, this subroutine now also distributes these k-space arrays among the processors as expected by the inverse hypercube FFT routine.

C. *Parallel Particle Update*

The sequential particle push subroutine has three major parts. Each part has a loop over particles, $i = 1, n_{\text{part}}$. First the electric field at the particle's position is found by interpolation, and the new particle position and velocity calculated via Eqs. (5)–(6). Second, the new coordinate is checked to see if it is out of the simulation bounds, and if so, corrected, e.g., for periodic boundary conditions, particles leaving one boundary are reintroduced at the opposite end. Third, the new charge density is computed by interpolation from the new particle positions.

In implementing the parallel particle push, two changes in the sequential push subroutine were made. The first was a trivial change in addressing a particle's nearest grid point, where a processor-dependent offset was now required. Each processor stores only the electric field E and charge array ρ_g for its primary subdomain (and guard cells), so an offset must be subtracted from the particle's nearest

grid point in the virtual global array in order to find the proper electric field values and to deposit the charge in the proper place in the charge density array. The nearest grid point now corresponds the position j in the local grid arrays where

$$j = \text{INT}\,(x_i(t) + 1.5) - j_{\text{off}}, \tag{10}$$

and where $j_{\text{off}} = j_{\text{left}} - 2$. [$j_{\text{left}}$ is defined following Eq. (9).]

The second, non-trivial, change was to replace the section of the push routine where the updated particle coordinates were checked with a much more elaborate scheme where particles which were now out of the processors' bounds are passed to other processors. This was implemented as follows:

If a processor's ith particle is out of bounds to the left, the particle's coordinates $x(i)$ and $v(i)$ are placed in a special buffer, *buffl*; the location (address) of the hole created in the particle position and velocity arrays by its departure is added to a list of holes, *ihole*. That is, if the ith particle is the first to leave the sub-domain, *ihole* $(1) = i$. Each processor has its own *ihole* array. Similarly, if a particle went out of bounds to the right, its co-ordinates are placed in *buffr* and its address/index placed in the processor's *ihole* array. When all particles have been checked, *buffl* is passed to the processor on the left and *buffr* to the processor on the right. At the same time, incoming particle buffers are received from the neighboring processors. The incoming particles are used to fill in the holes in the old particle arrays created by the departing particles. If there are more incoming particles than holes, and, thus this processor's number of particles n_{pp} is increasing, then the extra overflow particles are added to the bottom of the particle arrays. If there are fewer incoming particles than holes in the old particle arrays, and, thus, the number of particles in the processor n_{pp} is decreasing, then the remaining holes are filled, from the top down, by particles moved up from the bottom of the old particle arrays. As the filling of the extra holes progresses, any holes encountered at the current bottom of the particle arrays are skipped. When the location (address) of the hole being tested becomes greater than the current bottom particle address, the holes are all filled, and the filling procedure is stopped. In this fashion, the particle coordinates $x(i)$ and $v(i)$ are left in contiguous arrays with $i = 1, n_{pp}$. Alternatives would have required the use of indexing within the particle loops and additional particle storage space.

By accumulating the particle information for particles leaving a processor in the two buffer arrays *buffr* and *buffl* and making only two communication calls within the push subroutine (one to the neighbor on the right and one to the neighbor on the left), the communication time for exchanging particles is minimized because the overhead in starting up the internode communication is kept to a minimum.

One further change was necessary for the particle push portion of the code which is not part of the push subroutine itself. Because of the quadrupole interpolation of the charge, a portion of a particle's charge may be deposited in the spatial sub-domain of a neighboring processor, that is, it may be deposited in its guard cells (cf. Section IIIA). After the push, the charge accumulated in the guard cells is

passed to the appropriate neighbor and added to the proper elements in its ρ_q array to form the complete charge density for its sub-domain. The communication is done by two shifts of guard cell information, one to the neighbor on the left and one to the neighbor on the right.

D. *Parallel Poisson Solver*

For the case of periodic boundary conditions, the sequential Poisson solver (1) calculates $\rho_q(k)$ from $\rho_q(x_g)$ by calling an FFT; (2) obtains $E(k)$ by multiplying $\rho_q(k)$ by a factor $-iS(k)/k$, where $S(k)$ is the particle shape function [1, 3]; and (3) calls an FFT to transform $E(k)$ to $E(x_g)$.

In the parallel Poisson subroutine, the second portion was unchanged except for the trivial change in the loop index (each processor has $1/N_p$ of the Fourier modes). The first and third portions now call the new Fourier transform subroutine, described in Section B, which performs the necessary redistributions of data between the primary and secondary decompositions and calls the hypercube FFT routines. Note that times for the field solution portion of the code given in Section IV includes the time for the two redistributions of data between the primary and secondary decomposition.

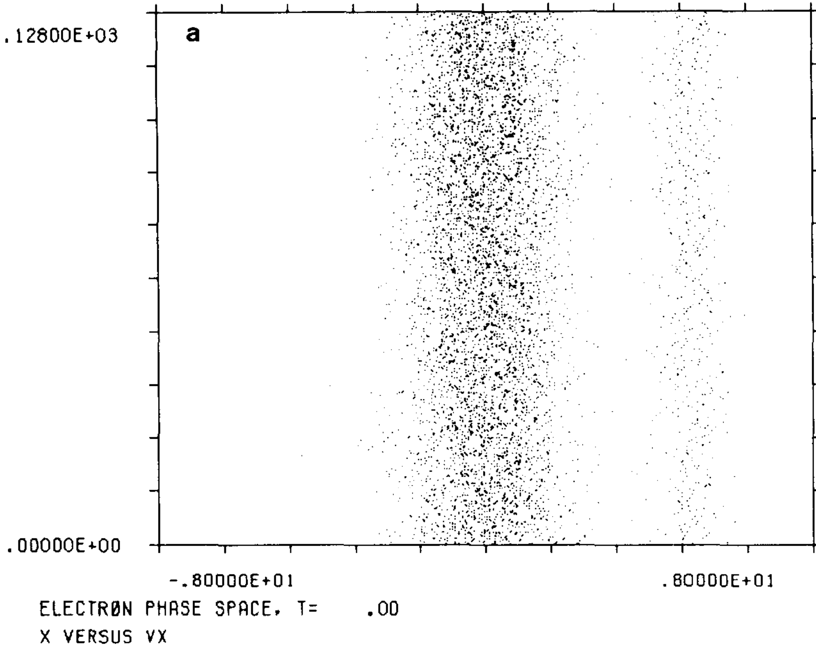
E. *Parallel Graphics*

Graphics in the parallel code were done using a Tektronix driver previously written in Fortran for sequential computers [10]. Graphs can be displayed in real time directly at a terminal (Tektronix emulator or SUN) or they can be saved in a graphics metafile for later viewing or printing on a variety of devices [10]. Figure 4 shows a sample phase space plot generated by the parallel code from the benchmark case discussed in Section IV.

It was necessary to make several small modifications to the sequential graphics routines for the parallel code because the information for creating the plots, such as phase space or electric field plots, is distributed among the processors.

In the parallel code, each processor creates the portion of the plot which corresponds to the data it stores, e.g., its particles for a phase space plot or its segment (grid points) of an electric field $E(x)$ vs x plot. One processor (processor 0) is also responsible for drawing the axes and labels. However, it is first necessary to determine the global maximum and minimum values of the function to be plotted and the result communicated to all processors so that all processors use the same scale when creating their portion of the plot. This is handled by a global communication routine. Another consideration was necessary for line plots such as $E(x)$ vs x. In order to make the curves continuous, each processor had to plot not only its grid points as defined by Eq. (9), but also its first guard cell on the right, j_{right}. However, each processor already has access to this information because the primary decomposition included this guard cell for use in the interpolations, so no further changes were needed.

With the current Mark III configuration, it is necessary for each processor to send its portion of the plot to the host or controlling computer which combines the



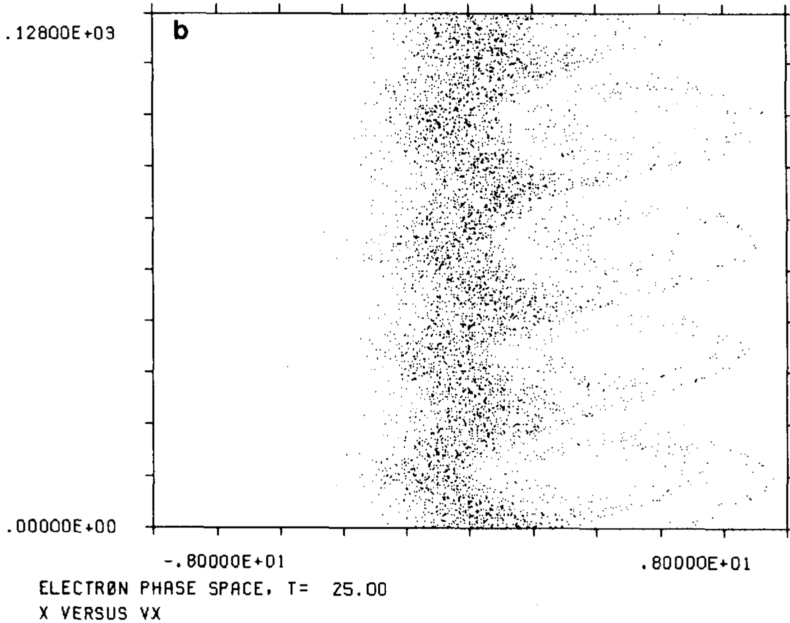


FIG. 4. Electron phase space from the Mark III code for the benchmark case of an electron beam-plasma instability: (a) initial phase space; (b) phase space near saturation of the instability.

diagnostic information and puts it in a file and/or displays it at a terminal. Because each portion of the plot is created independently in the nodes, the order it which the processors information is received is irrelevant.

F. *Portability of GCPIC Parallel Code and Algorithm*

The Mark III Hypercube parallel code, written in Fortran '77, runs under the CRoS III operating system [9]. (The code has also been implemented on an NCUBE hypercube parallel computer at Caltech which also runs under the CRoS III operating system.) Although communication among the processors is frequently used in the parallel code, in fact, only two system subroutines for internode communication were invoked. Thus, only replacements for these would be needed to port the code to another parallel computer. Additional systems subroutines were used for communication between the nodes and the controlling computer. These [illegible]

The two internode communication Fortran routines called are **kcombi** which combines data from all processors according to a user-supplied function, and **kcshft** (from *shift*) which passes data to, for example, the neighbor on the right while receiving data from the neighbor on the left. The **kcshft** routine is preferred since it is rapid and involves only nearest neighbor communication. The **kcombi** routine involves global communication among all processors. It is generally avoided except when calculating certain constants that all processors contribute to and need to know the results, e.g., finding the maximum and minimum of a variable for diagnostics or graphs.

Equivalent system calls on other parallel computers either exist or can be easily written using whatever parallel primitives are available. They can also be emulated for a large problem on a sequential computer e.g., a problem which needs an out-of-core solution, where a portion of the data to be processed must be stored in external memory, e.g., on a solid state disk. In this case, data associated with the separate sub-domains become separate blocks of externally stored data. Individual sub-domain data blocks (including both particles and fields) would be brought off disk and into memory one at a time for processing. The internode communication calls would be emulated by routines using temporary data storage to pass information from one data block to another.

IV. Performance Analysis of Mark III PIC Code

The performance of the parallel GCPIC Mark III hypercube code has been measured in three ways: (1) Comparison of the times to run the same problem on an increasing number of processors (Table I); (2) Comparison of the times to run problems with an increasing number of particles so that the average number of particles per processor is constant (Table II); and (3) Comparison of the time to run the benchmark problem with times on other computers (Table III). The first two

measure the performance of the parallel algorithm on one concurrent processor, the Mark III Hypercube, whereas the third measures the Mark III performance relative to other computers.

The Mark III Hypercube consists of 32 independent processors, each with its own local memory (4 Mbytes of dynamic random access memory and 128 Kbytes of static random access memory). There is no global or shared memory. Each processor consists of two Motorola MC68020 CPU's with a MC68882 coprocessor. The computation speed is approximately 200 Kflops per node. The rate for synchronous communication between nodes is 2 Mbytes/s per channel. For a brief time in the fall of 1988, a 64 node Mark III was available and used for the runs in Table II.

A. *Fixed Problem Size*

A hypercube can be run with different numbers of processors by varying the hypercube dimension, d, where $2^d = N^p$. Table I gives the time for the particle push portion of the code on the Mark III for 1, 2, 4, 8, 16, and 32 processors when the **total** number of particles was held fixed. The case run was the benchmark case [4], which is an electron beam–plasma instability, with 11,264 particles and 128 grid points. The initial density distribution was uniform, but becomes slightly nonuniform as the run progresses and the beam–plasma instability enters the nonlinear stage.

In the table, the time given is the particle *push time*, defined as the cpu time per time step per particle to advance the particles (including the force interpolation), to exchange particles among processors, and to deposit the charge on the grid. Since the number of particles per processor fluctuates as the run progresses, the push times given are the maximum over all processors averaged over 100 time steps of the benchmark case. In this run on 32 processors, 352 ± 20 particles per processor was observed. The speed up factor in Table I is defined to be

$$S(N_p) = \frac{\text{time on 1 processor}}{\text{time on } N_p \text{ processors}}.$$

TABLE I

Push Times on the Mark III for Fixed Total Number of Particles

Processors	Push time (μs)	Speed up	Efficiency (%)
1	210	—	—
2	106	1.98	99
4	52	4.0	99
8	27	7.9	98
16	14	15.3	96
32	7	29.5	92

It can be seen that the speed up in going from 1 to 32 processors was 29.5. This gives a parallel efficiency of $\varepsilon = S(N_p)/N_p = 90\%$ for a fixed total number of particles.

For the benchmark case with 128 grid points, the efficiencies for the field portion were very low: the speedup for the field portion in going from 1 to 32 processors was 5.3, giving a parallel efficiency of 16%. The low efficiency is due to both (1) the inherent inefficiency of the parallel fast Fourier transform for a 128 point transform run on 32 processors (only four grid points or modes per processor) and (2) the redistributions of the grid information between the primary and secondary decompositions (considered part of the field solution). The total efficiency of the benchmark problem in going from 1 to 32 nodes was 51%, reflecting the parallel inefficiency of both the field solution and the diagnostics. On 32 nodes, the parallel code spent 51% of the time in the push, 26% in the field solution, and 23% in the diagnostic routines.

The efficiencies of the push portion of the code presented here for the GCPIC decomposition are as high as the push efficiencies found in an earlier parallel decomposition of this code [11]. In the earlier decomposition, the particles were divided evenly among the processors and each processor stored its own copy of the entire grid arrays. No exchange of particles was necessary; instead, it was necessary to globally sum the charge density arrays over all the processors [11]. This decomposition, with each processors storing its own copy of the entire grid array, was also used by Lubeck and Faber [12] to implement a 2D PIC code on an IPSC hypercube. In this paper [12], a performance model for PIC codes on hypercubes was presented and the predictions compared with the results for the IPSC hypercube PIC code; agreement between the model and the observed scaling of the computation time with the number of processors used was excellent. A 2D PIC code implemented with the MYrias parallel computing system, which utilizes an extended parallel Fortran for simplifying parallel code development, has also essentially used the earlier decomposition [13].

B. *Increasing Problem Size/Fixed Grain Size*

In another set of runs, the number of particles and grid points was increased linearly with the number of hypercube processors used so that the number of grid points and the average number of particles per processor was held fixed. The benchmark case of 11,264 particles and 128 grid points was run on 1 processor, and a problem 64 times bigger (720,896 particles and 8128 grid points) was run on all 64 processors. For the runs, the initial density distribution was uniform. For a sequential code, the push time increases linearly with the number of particles and the field portion increases faster than linearly (roughly as $n_g \ln n_g$ due to the FFT scaling). Thus, for a perfectly efficient parallel code, as the number of particles increases linearly with the number of processors, the push time would remain constant; however, the total time would still increase due to the increase in the time for the FFT. For our parallel code on 64 processors, the efficiency of the particle push portion is nearly perfect: $\varepsilon = 97\%$. The observed 51% increase in the total times for

TABLE II

Mark III Timings for Increasing Problem Size

Processors	Particles	Grid points	Total time (100 steps) (s)	Total push time (s)	Push efficiency (%)	Push time per particle per timestep
1	11,264	128	329	272	—	241.5
2	22,528	256	344	273	100	121.2
4	45,056	512	359	273	100	60.6
8	90,112	1024	406	275	100	30.5
16	180,224	2048	406	281	97	15.6
32	360,448	4096	427	281	97	7.8
64	720,896	8128	497	281	97	3.9

the problem on 64 nodes compared to the problem on one node (Table II) is largely due to this increase in the time for the FFT.

Table II gives the problem size and the times for various numbers of processors used. Both the total run time and the total push time are given for 100 time steps. (The time to initialize the particles is not included.) Note that this is not the push time per particle per time step. Also given is the efficiency of the push, defined as

$$\varepsilon = \frac{\text{Time for } n_{\text{part}} \text{ particles on 1 processor}}{\text{Time for } N_p \times n_{\text{part}} \text{ particles on } N_p \text{ processors}}.$$

It can be seen that the total push time increased by only 3%, giving a parallel efficiency of 97% for fixed grain size. The total run time increased by 51%, most of which is due to the $n_g \ln n_g$ scaling of the FFT.

For these runs, the *push time*, the time per particle per time step, fell from 241 μs for 11,264 particles on 1 processor, to 4 μs for 720,896 ($=64 \times 11,264$) particles on 64 processors.

For the problem used to obtain the efficiencies for both fixed problem and fixed grain size case, the particle distribution remained fairly uniform in space during the run. The processors' loads did not become unbalanced and thus high efficiency was obtained for the particle push even though dynamic load balancing was not implemented. If the distribution had become less uniform, the efficiencies would have degraded in the absence of dynamic load balancing. In the future, we plan to investigate the efficiency of this algorithm when dynamic load balancing is utilized. In particular, it will be necessary to determine what overhead is associated with dynamic re-partitioning of the domain and at what level of processor load imbalance it becomes worthwhile.

In Tables I and II, only the efficiencies of the push portion of the code have been presented because this is the most important part of the code, taking roughly 90% of the cpu time in a sequential computer. In addition, the GCPIC parallel decomposition for the push portion (the primary decomposition) would be the same for

all types of PIC codes, whereas the secondary decomposition will depend on the particular method of solution used to solve for the electromagnetic fields.

C. Comparisons with Other Computers

Table III contains the timing information for the benchmark case ($n_{\text{part}} = 11{,}264$, $n_g = 128$) for several of the computers on which the UCLA benchmark code has been run. Times on additional computers can be found in Ref. [4]. Plots of electron [illegible] discussed in Section III.E). Two times are given in Table III: the total time and the push time per particle per time step, as defined above. It can be seen that running the benchmark on 32 nodes of the Mark III took 431 s, or 5.6 times longer than a Cray 2 using vectorization, and $\frac{1}{15}$ th as long as on the VAX. The push time on 32 nodes (7.2 μs per particle per time step) is 3.4 times slower than a Cray 2 using vectorization.

The Mark III used for the runs reported in this paper uses the Motorola 68882 chip as the floating point coprocessor. The new Mark III $+fp$ will also have a Weitek floating point processor to increase the speed. Preliminary runs on a

TABLE III

Benchmark Timings for 1D Electrostatic Code

Computer	Total time (2500 steps) (s)	Push time (per particle per timestep) (μs)	Push/total
CRAY XMP/48 (1 processor)			
Vector version	50	1.46	83.1%
Scalar version	127	4.07	90.3%
CRAY 1s, CFT 1.11			
Vector version	77	2.88	82.5%
Scalar version	174	5.42	87.5%
IBM 3090 VF			
Vector version	95	2.88	85.1%
Scalar version	185	6.0	92.1%
CRAY 2 (1 processor)			
Vector version	77	2.10	85.2%
Scalar version	302	10.10	94.3%
MARK III Hypercube			
32 nodes	431	7.2	50.7%
1 node	7071	210.4	82.2%
Alliant FX/8	564	12.56	62.7%
CDC 7600	671	21.26	89.2%
CONVEX C-1			
Scalar version	1731	56.08	91.2%
Vector version (comp. dir.)	704	19.491	78.4%
VAX 11/750,F.P.A.	6394	200.91	88.5%

4-node Mark III $+fp$ show that the speed up for the push time on one processor is roughly a factor of 8. Assuming a 10% loss in efficiency in going from 32 to 128 nodes, we can estimate that the push time will fall to 0.25 μs per particles per time step, making it about 8 times faster than a one processor Cray 2.

V. Conclusions

We have developed an efficient algorithm, termed the GCPIC algorithm, for dividing PIC simulations among the processors of a parallel computer. The efficiency of GCPIC has been demonstrated by implementing a 1D electrostatic code using this algorithm on the JPL Mark III Hypercube. Although the GCPIC algorithm was developed on a hypercube, it will be an efficient algorithm for load balancing particle codes on any parallel computer with distributed memory. In addition, it would be an efficient algorithm for large PIC codes on sequential computers where part of the data must reside in external memory, e.g., on solid state disks, where now the individual sub-domains would be brought off disk and into memory one at a time.

The central feature of the GCPIC algorithm is the use of two decompositions for the two portions of a PIC code: a primary decomposition used to make the particle push portion of the code efficient and a secondary decomposition used to make the electromagnetic field solution efficient. Inherent in the GCPIC algorithm is the need to redistribute the field arrays between the two decompositions at every iteration.

The code implemented on the Mark III uses a fast Fourier transform to solve for the field. If a finite difference field solution were used, a secondary decomposition with equal numbers of grid points per sub-domain would be most efficient, regardless of whether the grid spacing was uniform or non-uniform. For this secondary decomposition, nearest neighbor sub-domains would be assigned to nearest neighbor processors, e.g., the physical domain would be mapped directly onto the processor array. Only nearest neighbor communication would then be necessary within the field solution portion as well, as opposed to the global communication necessary for a Fourier transform solution. Note that if the field solution uses a finite difference algorithm and if a non-uniform grid is created such that each cell has an equal number of particles, then the primary (particle) and secondary (field) parallel decomposition become identical and no global redistributions of grid arrays are necessary.

Acknowledgments

The authors thank Professors Geofrey C. Fox (Caltech), John M. Dawson (UCLA), and R. W. Gould (Caltech) for their support. The help and support of the JPL hypercube group is also gratefully acknowledged, with special thanks to R. Calalo, J. Patterson, and B. Zimmerman. Part of the research described in this paper was performed by the Jet Propulsion Laboratory, California Institute of Technology, and

was sponsored by the U.S. Department of Energy under Contracts DE-FG03-85ER25009 and DE-FG03-85ER53173 and by Sandia National Laboratory under Contract 32-5531 through an agreement with the National Aeronautics and Space Administration. The research of one of the authors (VKD) was supported by Sandia National Laboratory under Contract 23-1540.

References

1. J. M. Dawson, *Rev. Mod. Phys.* **55**, 403 (1983).
2. C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation* (McGraw–Hill, New York, 1985).
3. V. K. Decyk, UCLA Center for Plasma Physics and Fusion Engineering Report PPG 708, 1983 (unpublished).
4. V. K. Decyk, *Supercomputer* **27**, 33 (1988).
5. A. B. Langdon, B. I. Cohen, and A. Friedman, *J. Comput. Phys.* **51**, 107 (1983); J. U. Brackbill and D. W. Forslund, *J. Comput. Phys.* **46**, 271 (1982).
6. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles* (McGraw–Hill, New York, 1981).
7. O. Buneman, C. W. Barnes, J. C. Green, and D. E. Nielson, *J. Comput. Phys.* **38**, 1 (1980).
8. V. K. Decyk and J. M. Dawson, *J. Comput. Phys.* **30**, 407 (1979).
9. G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, and J. Salmon, *Solving Problems on Concurrent Processors* (Prentice–Hall, Englewood Cliffs, NJ, 1988).
10. V. K. Decyk and L. Xu, in *Proceedings Twelfth Conf. on Numerical Simulation of Plasma, San Francisco, CA, 1987*, Paper PW 10.
11. P. C. Liewer, V. K. Decyk, J. M. Dawson, and G. C. Fox, *Math. Comput. Modelling* (*Proceedings Sixth International Conference on Mathematical Modelling*) **11**, 53 (1988).
12. O. M. Lubeck and V. Faber, *Parallel Computing* (to be published).
13. R. Foster, C. Thomson, and D. Wilson, "Parallel Programming without Tears," 3rd SIAM Conference on Parallel Processing for Scientific Computing, Los Angeles, CA, December 1987.