Hybrid Partitioning for Particle-in-Cell Simulation on Shared Memory Systems

Nasser G. Azari and Soo-Young Lee
School of Electrical Engineering
Cornell University, Ithaca, New York 14853

Abstract

Particle-in-cell (PIC) is a simulation method widely used in many important scientific areas; such as plasma physics, semiconductor device physics, global climate modeling, and galaxy dynamics. In general, these simulations are extremely computationally intensive and, therefore, very time consuming even on supercomputers. We consider a PIC algorithm which simulates the behavior of charged particles in an electromagnetic field. This study is performed in order to explore parallel processing issues, such as relationships between speedup and problem partitioning schemes, problem size and time duration of each iteration for PIC method on different multiprocessors. A new partitioning scheme, hybrid partitioning, is introduced. Hybrid partitioning has evolved out of two general approaches to PIC problem decomposition on multiprocessors, partitioning particles and partitioning the space. We chose the shared memory multiprocessor environment for analyzing our parallel (distributed computing) algorithms. Two different BBN Butterfly machines (GP1000 and TC2000) were employed as testbeds.

1 Introduction

Computer simulation is a reasonable approach to determining the complex behavior of systems which are difficult to test directly or analyze by standard analytic methods. Particle simulations are frequently used in such areas as kinetic plasma physics, semiconductor device physics, global climate modeling, and galaxy dynamics [3,8].

PIC simulation, one of the principal types of particle simulation, is considered as one of the most time consuming problems and typically run on supercomputers. For example, a complete world climate simulation can take up to 200 hours of CPU time on a CRAY-1 supercomputer. PIC simulation on vector machines is relatively less efficient since the dominant portion of a PIC code is nonvectorizable. There is also a large amount of data involved with most of the PIC simulations. Even the largest existing supercomputers can model only relatively few million particles in high dimensions, while even a very small region of most plasmas of interest can contain $10^{12}$ particles. Therefore, PIC models are not generally used for a very large-scale phenomena. Some of the techniques usually employed to model the systems are (a) reduction of the dimension of the problem, (b) reduction of the problem size by studying a limited number of the particles (a typical plasma simulation problem contains only $10^4$–$10^5$ particles).

The simulation space (2D or 3D) is represented by a grid. Particle-in-cell simulations follow two types of quantities: grid quantities and particle quantities. These two types of quantities allow us to divide the operations in implementing a PIC simulation into four groups. These operations are the following:

1. updating of grid quantities using other grid quantities,
2. updating of particle quantities using other particle quantities,
3. obtaining particle quantities from grid quantities (gather),
4. obtaining grid quantities from particle quantities (scatter).

The first of these operations is often vectorizable and thus very rapidly performed on a vector machine such as a CRAY supercomputer. The second one is always vectorizable, and is thus also fast. The last two operations are termed gather-scatter operations. In these operations, grid quantities from grid points surrounding the locations of the particles are interpolated into
the particles to serve as particle-advancing quantities (gather), or particle quantities are interpolated and accumulated on the grid to serve as grid source quantities (scatter) (see Fig. 1). Since it is not known a priori which particles will be in the vicinity of which grid points, the gather-scatter operations are in general not vectorizable. As a result, these operations are relatively slow on a vector machine. It is not unusual for these operations to take 80% to 90% of the total CPU time used by the algorithm. While there have been some studies on vectorizing some parts of the gather-scatter problem on different machines, the methods are not universally applicable to all supercomputers [9,13]. A recent effort is focused on parallelizing this class of problems in the most efficient manner using the available multiprocessors.

Figure 1: Directions of gather (g) and scatter (s) are shown for one particle on the grid. w1, w2, w3 and w4 are weight factors.

In our earlier work [1,2], we have parallelized the PIC method on a distributed memory system (iPSC/2) using two major approaches, partitioning particles and partitioning the space. We also suggested a partitioning method which is considered to be a compromise between these two general approaches. This partitioning method, which we call hybrid partitioning, achieves a relatively larger speedup while not requiring a large memory space.

In the hybrid partitioning, for a given number of PEs employed in the system \( N \), one optimal partitioning could be distinguished which achieves the highest speedup. Our performance analysis shows a significant performance improvement (40%-50%) for this scheme over other data allocation schemes.

In Section 2, a brief introduction to the plasma simulation algorithm is provided. Section 3 is devoted to the two general approaches to parallelizing of the PIC. Parallelizing the problem using the hybrid partitioning on shared memory systems and the results obtained on BBN Butterfly (GP1000 and TC2000) are contained in Section 4.

2 Review of the Particle-in-Cell Algorithm

The original code follows two spatial dimensions and all three velocity components. It solves as grid equations, Faraday's Law,

\[
\frac{\partial B}{\partial t} = - c \nabla \times E
\]

Ampere's Law,

\[
u_e = \frac{1}{\epsilon n_i} \left( \frac{\nabla \times B}{4\pi} - \frac{J_i}{c} \right)
\]

and the inertialess electron momentum equation,

\[-m_e \frac{du_e}{dt} = 0 = E + \frac{u_e \times B}{c} - \frac{T_e \nabla \rho}{\epsilon n_i}
\]

assuming quasineutrality \( n_e \approx n_i \). The ion momentum equation,

\[
\frac{dv_i}{dt} = \frac{e}{m_i} \left( E(x_i) + \frac{u_i \times B(x_i)}{c} \right)
\]

and particle advance,

\[
\frac{dx_i}{dt} = v_i
\]

are solved as particle equations. In these equations, \( m_e, T_e, n_e, \) and \( u_e \) are electron mass, temperature, number density, and fluid velocity, respectively, while \( n_i, J_i, v_i, \) and \( x_i \) are the ion number density, current density, particle velocities, and particle positions, respectively. This type of code may be characterized as a 2-1/2 dimensional quasineutral Darwin hybrid plasma code. The equations are solved using a predictor-corrector method. The variables \( B, E, u_e, n_i, \) and \( J_i \) are defined on a two-dimensional grid, while \( x_i, v_i, E(x_i), \) and \( B(x_i) \) are ion particle quantities. The most time-consuming portions of this algorithm are the following:

1. accumulating ion number and current densities, \( n_i \) and \( J_i \), of the ion particles on the grid
2. interpolating electric and magnetic fields, \( E(x_i) \) and \( B(x_i) \), to the particle positions \( x_i \) in order to push the particles

Additional details of this algorithm are given in References [4,6,14].
3 Two Basic Parallelization Methods

Particle-in-cell simulations are designed to follow the behavior (particle motion, fields change, etc.) of the system in each time step (iteration). A new iteration can be initiated only after the current iteration has been completed; that is, any two iterations cannot be executed at the same time. Therefore, the only way to speed up this simulation is to partition the computational task in each iteration among processing elements (PES).

In the following, we review the advantages and the disadvantages associated with two major task partitioning approaches. In our earlier work [1,2], we called these two partitioning schemes, partitioning grid space, and partitioning particles (see Fig. 2). Hybrid partitioning effectively utilizes the tradeoff between the memory space and the performance of these two schemes.

3.1 Partitioning Grid Space

In this partitioning scheme, we parallelized the simulation by partitioning the grid into subgrids with the number of subgrids being equal to \( N \). To have a balanced load distribution on PES, subgrids should contain an equal number of particles. One PE is assigned to each subgrid, that is, each PE operates on one subgrid and the particles in it. Since particles move freely in the grid space and migrate between subgrids, partitioning grid space can give rise to unequal work load on PES as the PIC simulation progresses. Consequently, one immediate concern with this scheme is to balance the load when the number of particles in a subgrid varies considerably with subgrid. To keep the load balanced on PES, we need to check the load distribution periodically and adjust the partitions if necessary (dynamic load balancing). The overhead of dynamic load balancing strategy relates to several factors such as the problem size, the technique used, etc. [5,12,15]. Also, in this approach (partitioning grid space), there is communication overhead for exchanging the particles which cross the boundaries, and the fields on the grid points located on the boundaries.

3.2 Partitioning Particles

One major drawback in partitioning the spatial domain (grid space) is a potential poor load balance. In order to avoid this, we may consider partitioning particles. That is, all particles in the system are evenly distributed among PEs (no matter where they are located on the grid). Each PE keeps track of the same set of particles throughout the entire simulation. In this type of partitioning, to keep the load balanced and the communication overhead low, one has to store the entire grid on each PE. Therefore, the partitioning particle scheme will require a larger memory space than the former scheme.

After each iteration, each PE has the complete results (next positions, etc.) for the particles assigned to it; however, the grid on each PE contains only partial results of fields (contribution by its particles to the grid points). Then, to compute the contribution of all the particles in the system to the grid, the grid values have to be updated in all PEs; that is, the grids in all PEs are superimposed (summed) in each iteration to obtain the complete field results.

4 Hybrid Partitioning

The objective of the hybrid partitioning is to provide a (efficient) task partitioning scheme by which one can reduce the size of memory required on each PE while not sacrificing the system performance (speedup or efficiency). This may be achieved by combining the two
general schemes above. That is, by partitioning the space one can save the memory space on each PE, and by partitioning the particles one may attempt to obtain a well-balanced load distribution which would lead to a high efficiency.

\[
\begin{align*}
M_t &= \text{Total memory access time} \\
M_l &= \text{Local memory access time} \\
M_r &= \text{Remote memory access time} \\
N_m &= \text{Total number of memory accesses} \\
M_a &= \text{Average memory access time}
\end{align*}
\]

One effective way to reduce \( M_a \) is to cut down the number of remote memory accesses, since \( M_r \) is always larger than \( M_l \) (5 times more on GP1000). In our earlier work [2], we have showed that a proper control of data allocation can reduce the remote memory accesses and produce a higher efficiency. This approach is taken also in the hybrid partitioning.

In the hybrid partitioning scheme, the grid is partitioned into contiguous blocks (subgrids), where each block contains the same number of rows of grid points (aligned with the horizontal axis). A group of PEs is assigned to each block. The number of PEs in a group, referred to as group size (GS) is equal to \( N \) divided by the number of blocks (see Fig. 3). Each block (the fields on the grid points in the block) is stored in the local shared memory of each PE which is assigned to that block (see Fig. 3). That is, each PE has its own copy of the block. The particles in each block are initially partitioned evenly among PEs in that block. Each PE keeps track of the initially assigned particles wherever they migrate as the simulation progresses. To illustrate the block allocation and the PES grouping, an example for 16 PEs and a GS of 4 is shown in Fig. 4.

We define the following terms. A partitioning instance is a case of task partitioning for a given \( N \) and a specific GS (these partitioning instances are shown in Fig. 3 for 8 PES). An optimal partitioning instance is a partitioning instance which provides the highest speedup.

4.3 Distinct Partitioning Instances
Three major distinct categories of partitioning instances are listed in the following:

a) \( GS = 1 \) (the left most column in Fig. 3).

b) \( GS = N \) (the main diagonal in Fig. 3, partitioning particles scheme).

c) \( 1 < GS < N \) (all the partitioning instances between the main diagonal and the left most column in Fig. 3).
Category a) is a simple data allocation technique which allows control in allocating the data on the system. In this category no duplication of data is attempted, called no-duplication (ND) technique for convenience. A contiguous block of the grid rows allocated to each PE and only one PE is assigned to each block. That is, the number of blocks is equal to $N$.

In category b), the grid is not partitioned (each PE has a copy of the entire grid). Note that this case corresponds to the partitioning scheme mentioned in Section 3.

In category c), each PE within a group of PES stores the same block which is shared by that group.

In categories a) and c), since each block is shared by two or more PEs, partial field results on grid points of the copies of that block need to be accumulated. The current density and the ion number density, $J$ and $n_i$, on the grid points in each block have to be updated by superposition (E and B fields are then computed using $J$ and $n_i$). The superposition in this case could degrade the performance if $N$ is large. The overhead in the updating stage is very much machine dependent.

For the categories a) and c), data coherency is to be ensured since particles assigned to different PEs could be positioned in the same grid cell and consequently two or more PEs may attempt to update a grid point at the same time which may result in a wrong value. In the contrast, one would not be concerned about data coherency in category b) since each PE only deals with its own copy of grid.

Data Coherency

One way to assure the correctness of the computation result is to place locks around the critical section of the algorithm for the scatter operation. Depending on how the locking is implemented, it could degrade the performance considerably. Locking the instructions one-by-one (using atomic operation) in the critical section of the algorithm would generally be considerably more efficient than locking the entire critical section since the average time for which each PE has to wait between attempts for setting the lock is longer for the latter case. There is still a large performance degradation (10%-30%) involved with atomic operations. One alternative solution to atomic operations could possibly be to prevent the situation in which two or more PEs attempt to update the grid points. By sorting the particles based on the blocks in which they are located, one can perform the updating step in an order in which no grid points in a block would be updated by more than one PE at the same time.

4.4 Implementation Results and Performance Analysis

As a comparison reference we consider an allocation routine (UsAllocScatterMatrix) provided by the Uniform System (library of subroutines for BBN Butterfly), called built-in (BI) allocation technique for convenience. In this technique, the system scatters the grid by row (not in contiguous blocks) among PEs (more precisely, the shared memory modules of PEs). Users have no control on data allocation. Since the grid points are scattered over the entire system, to update the particle values (gather) and the grid point values (scatter) we need to access different shared memory modules. The number of remote memory accesses for each PE is directly related to the number of particles which do not reside in the grid cells along the rows that are stored in the same PE. $M_{i}$ could vary in each iteration for a PE and also among PEs, therefore, creating unbalanced load distribution on PEs.

Two different BBN Butterfly systems (GP1000 at Cornell University and TC2000 at Argonne National Laboratory) are employed. To represent the simulation space, a 32-by-32 uniform (square) grid is adopted. The grid is assumed to be wrapped around (periodic) in both directions. Different numbers of particles (8192, 16384 and 32768) denoted by $N_p$ are considered. Five different time step durations ($10^{-9}$, $5 \times 10^{-10}$, $10^{-10}$, $5 \times 10^{-11}$ and $10^{-11}$) denoted by $DT$ are tested. For a larger $DT$, particles travel a longer distance over the simulation area. The number of time steps (iterations)
is 100. The measured performances are provided in Tables (1-3) and Figures (5-11). The speedup is plotted as a function of N where the speedup is the sequential execution time divided by the parallel execution time. From the results the following may be observed.

i) A significant speedup improvement has been achieved by the hybrid partitioning over built-in data allocation (39%-49%) while the improvement for the no-duplication technique over the built-in technique is marginal (3%-7%). This is because \( M_a \) is decreased much more in hybrid partitioning than in no-duplication scheme (see Fig. 5, Table 1).

ii) As the GS increases (for a given N) the superposition time increases while the gather-scatter time decreases (see Figures 6 & 7, Tables 2 & 3). This result is observed since the larger GS is the larger the block size becomes which in turn reduces the number of remote memory accesses and increases the number of grid points to be superimposed. The highest speedup is obtained for the partitioning instance (optimal) for which the superposition overhead increase starts to offset the gather-scatter time decrease (the peak point of each curve in Figures 6 & 7, the highlighted values in Tables 2 & 3).

iii) A larger \( N_p \) results in a larger speedup (see Fig. 8). This is because the larger the ratio of computation time over communication time (less overhead) becomes, the higher the efficiency becomes for a given N.

iv) The smaller \( DT \), the higher the speedup becomes for a given number of time steps. \( M_a \) is smaller for a smaller \( DT \) since the average number of particles which leave the blocks is generally smaller (i.e., the distance travelled by each particle is smaller). This can be observed in Fig. 9 where speedup for a given N saturates when \( DT \) increases.

v) The optimal GS (for given N) tends to decrease when \( DT \) decreases (compare Figures 6 and 7). By decreasing GS the superposition time decreases equally for any given \( DT \) while we observe a smaller increase in gather-scatter time for a smaller \( DT \). Therefore, for a smaller \( DT \), the superposition overhead increase may get offset by the gather-scatter time decrease in an instance with a smaller GS (shift to the left in Fig. 3 is expected).

vi) Although the total simulation time for TC2000 is much smaller than for GP1000, smaller speedups are observed (see Fig. 10). This is because the communication (switch) in TC2000 is about 3/2 times faster than that in GP1000 (38 vs 24 MBytes/Sec per path) while the computation time is about 7 times faster. This difference makes the ratio of computation time over the communication time smaller for TC2000 and consequently reduces the efficiency for this machine (TC2000).

vii) It is confirmed that though the execution time per iteration (time step) increases as the simulation progresses (more particles cross the block boundaries), it tends to become steady eventually (see Fig. 11). This observation shows that the particles assigned to each PE would be almost uniformly spread over all the blocks, if the simulation time last long enough (the curves in Fig. 11 saturate).

<table>
<thead>
<tr>
<th>PE</th>
<th>Built-In (B)</th>
<th>No-Duplicate (ND)</th>
<th>Hybrid/Optimal (H)</th>
<th>Speedup Improvement (ND over B)</th>
<th>Speedup Improvement (H over B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1.44</td>
<td>1.69</td>
<td>2.00</td>
<td>3%</td>
<td>39%</td>
</tr>
<tr>
<td>4</td>
<td>2.76</td>
<td>3.94</td>
<td>5.98</td>
<td>7%</td>
<td>43%</td>
</tr>
<tr>
<td>8</td>
<td>5.97</td>
<td>7.75</td>
<td>7.85</td>
<td>7%</td>
<td>43%</td>
</tr>
<tr>
<td>16</td>
<td>10.18</td>
<td>10.96</td>
<td>15.53</td>
<td>7%</td>
<td>43%</td>
</tr>
<tr>
<td>32</td>
<td>16.72</td>
<td>18.20</td>
<td>28.23</td>
<td>9%</td>
<td>43%</td>
</tr>
</tbody>
</table>

Table 1: Speedup comparison of different data allocations

<table>
<thead>
<tr>
<th>PE</th>
<th>Built-In (B)</th>
<th>No-Duplicate (ND)</th>
<th>Hybrid/Optimal (H)</th>
<th>Speedup Improvement (ND over B)</th>
<th>Speedup Improvement (H over B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1.45</td>
<td>1.94</td>
<td>2.67</td>
<td>3%</td>
<td>39%</td>
</tr>
<tr>
<td>4</td>
<td>2.84</td>
<td>2.91</td>
<td>3.77</td>
<td>7%</td>
<td>43%</td>
</tr>
<tr>
<td>8</td>
<td>5.69</td>
<td>5.75</td>
<td>5.67</td>
<td>7%</td>
<td>43%</td>
</tr>
<tr>
<td>16</td>
<td>10.15</td>
<td>11.10</td>
<td>10.68</td>
<td>7%</td>
<td>43%</td>
</tr>
<tr>
<td>32</td>
<td>14.96</td>
<td>15.62</td>
<td>20.07</td>
<td>7%</td>
<td>43%</td>
</tr>
</tbody>
</table>

Table 3: Speedup in hybrid partitioning, highlighted values are the optimal partitioning instances (GP1000, \( N_p = 8192, DT = 10e-10 \))
Figure 5: Speedup comparison for different data allocations (GP1000, $N_p = 16384$, $DT = 10^{-11}$)

Figure 6: Hybrid partitioning (GP1000, $N_p = 8192$, $DT = 10^{-10}$)

Figure 7: Hybrid partitioning (GP1000, $N_p = 8192$, $DT = 10^{-11}$)

Figure 8: Hybrid partitioning for different number of particles (GP1000, optimal, $N_p = 16384$)

Figure 9: Speedup for 5 different time steps (GP1000, optimal, $N_p = 16384$)

Figure 10: Speedup comparison for GP1000 and TC2000 (optimal, $DT = 10^{-11}$)
5 Conclusion

The goal in decomposing PIC methods for parallelization has been to achieve a large speedup with a reasonable memory space. The partitioning schemes previously proposed mostly emphasize only one of two important issues, speedup or memory space.

The hybrid partitioning scheme presents different partitioning instances, which introduces a flexibility in choosing the speedup and memory space for a given number of PEs. That is, the speedup and the required memory space are functions of the group size. In this paper, we have shown the effectiveness of the hybrid partitioning in achieving a relatively high speedup while the memory space required is not detrimental for a large problem (e.g. a 3D problem). We have also analyzed the effects of the parameters on the speedup, such as the number of PEs, the relative speeds of processors & memories & interconnection network, time step duration, the number of particles. Our future work will include implementing the hybrid partitioning on a distributed memory multiprocessor system.

References


