Dynamic Alignment and Distribution of Irregularly Coupled Data Arrays for Scalable Parallelization of Particle-in-Cell Problems

Wei-keng Liao[†], Chao-wei Ou[†], and Sanjay Ranka[‡]

[†]2–120 Center for Science and Technology
School of Computer and Information Science
Syracuse University, Syracuse, NY 13244-4100

[‡]CSE Building, #301
Department of Computer Science
University of Florida, Gainesville, FL 32611

Abstract

Particle-in-cell (PIC) plasma simulation codes require two data arrays—particle array and field array—for storing the lists of particles and electromagnetic fields, respectively. In every iteration the two are updated based on the values of each other. The interaction between these two arrays is dynamic due to the movement of particles.

Efficient parallelization of PIC requires the two data arrays to be load balanced and the amount of communication generated due to the interaction between them to be minimized. This requires dynamic distribution and alignment of the two arrays. In this paper we present fast and efficient methods for achieving this task at runtime. The implementation and performance of a relativistic electromagnetic PIC plasma simulation code on the CM-5 are described.

1 Introduction

The particle-in-cell algorithm is a method widely used to simulate plasmas and hydrodynamics [4] [1]. The movement of particles is computed by the interaction between each pair of particles in a self-consistent system. Instead of directly calculating the interaction of particles, the PIC algorithm employs a regular computational mesh on the particle domain and assigns the particle attributes to nearby grid points of the mesh. The field equations are then solved on the mesh and the force which moves each particle to its new position is obtained by gathering the attributes of nearby grid points from the resultant fields on the mesh.

The PIC algorithm has two different data structures, particle array and mesh grid array. The mesh grid array is spatially homogeneous. The particle array can be nonhomogeneous. In each iteration the two are updated based on the values of the other. An efficient implementation on distributed memory MIMD computers requires distributing these two data structures over the processors such that off-processor accesses are minimized. A good load balance in parallelization should also ensure that each data structure is nearly equally distributed among the processors. Further, the data access patterns between two arrays may change dynamically (albeit incrementally) and therefore the particles may need to be redistributed frequently to reduce communication cost.

We assume the mesh grid is distributed along one or more dimensions using `BLOCK` distribution. This is necessary for effective parallelization of the field array. For partitioning the particle array, we present an efficient particle redistribution algorithm based on Hilbert indexing [10]. To improve the performance of redistribution, we used an incremental sorting algorithm that utilizes the previously sorted information to distribute the particle array efficiently. These algorithms maintain good spatial contiguity along multiple dimensions within decomposed particle sub-domains and thus reduce off-processor data accesses. We present experimental results which demonstrate that maintaining spatial contiguity in more than one dimension leads to reduced communication cost.

Although our repartitioning strategies are fast, it is not desirable to repartition the particles at every iteration. One strategy is to redistribute the particles periodically after every k iterations. Another strategy is to dynamically decide on when to perform this repartitioning based on monitoring the computation and communication costs of the previous iterations. We describe a simple redistribution strategy that performs comparably to or better than periodic distribution strategies. It has the added advantage that the user does not have to fine-tune program performance by choosing the optimal value of k.

The rest of the paper is organized as follows. Section 2 gives a brief description of the PIC algorithm. Section 3 discusses the particle and mesh domain partitioning strategies for parallel PIC algorithms. Section 4 describes a coarse-grained parallel machine model and also gives time complexity analysis for each phase of the PIC algorithm under the described machine model. Section 5 presents particle distribution and redistribution algorithms using index-based schemes and also describes the decision policy for dynamic particle redistribution. Experimental results

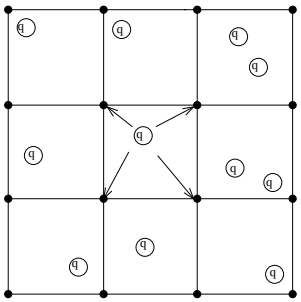


Figure 1: Scatter phase: particle contributes its charge to the grid points at the vertices of the cell where it lies.

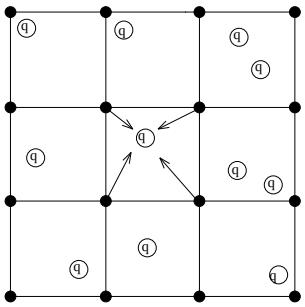


Figure 2: Gather phase: particle gathers contributions from the the grid points at the vertices of the cell where it lies.

and conclusions are given in Sections 6 and 7.

2 Particle-in-cell algorithm

In simulating the evolution of plasma we use a self-consistent model in which particles and mesh grid points mutually interact with each other without any external fields presented. That is, the charged particles making up a plasma move under the influence of the electro-magnetic fields that they generate. Instead of directly calculating the interaction of particles, the PIC algorithm employs a regular computational mesh on the particle domain and assigns the particle attributes to nearby grid points of the mesh. The appropriate field equations are then solved on the mesh, and the force on each particle due to the resultant fields is obtained by linear interpolation on the mesh. The particles are then moved under the influence of this force to a new position, ready to begin the next iteration.

The PIC algorithm follows the evolution of the particles in a series of time steps, each of which consists of four phases:

Scatter phase — Using a linear interpolation scheme each particle scatters its contributions to the current mesh grid points at the vertices of the cell in which it lies. A particle makes no contributions to other grid points. The contributions at each grid point are summed to form the current density there.

Field solve phase — Solve Maxwell's equations on the mesh to determine the electric and magnetic fields, $\mathbf{E}$ and $\mathbf{B}$. Each grid point on the mesh needs data from its neighboring grid points to calculate new values for $\mathbf{E}$ and $\mathbf{B}$.

Gather phase — The electric and magnetic fields at particles are obtained from their vertex grid points by a linear interpolation of particles' relative positions in the cell. Particles sum these contributions from the vertex grid points to generate the force. Figure 3 gives a simplistic description of the algorithms of scatter and gather phases.

n : number of particles
$vertex[4]$: stores 4 indices of particle's vertex grid points
$weight[4]$: stores weight of a particle to its 4 vertex grid points

Scatter( )
1 **for** $i \leftarrow 0$ **to** $n-1$
2 $vertex \leftarrow$ find indices of 4 vertex grid points of $particle[i]$
3 $weight \leftarrow$ calculate the weight of a particle to its 4 vertex grid points using linear interpolation
4 **for** $j \leftarrow 0$ **to** 3
5 $grid\_charge[\ vertex[j]\ ] \leftarrow grid\_charge[\ vertex[j]\ ] + weight[j] \cdot particle\_charge[i]$

Gather( )
1 **for** $i \leftarrow 0$ **to** $n-1$
2 **for** $j \leftarrow 0$ **to** 3
3 $particle\_fields[i] \leftarrow particle\_fields[i] + weight[j] \cdot grid\_fields[\ vertex[j]\ ]$

Figure 3: Scatter and gather phase algorithms in two-dimensional case.

Push phase — The force obtained from the gather phase moves particles to their new positions.

3 Parallelization

In a parallel PIC algorithm, the distribution of particles and mesh grid points over processors should be made such that the overhead generated from the parallelization is carefully controlled to obtain a reasonable speedup. In the scatter and gather phases, the off-processor data accesses produced from the interaction between particle and mesh grid arrays is the main portion of the interprocessor communication overhead.

Parallelization strategies for the PIC problem can be divided into two broad categories [13]:

Direct Eulerian method — Particle domain is divided spatially into several mutually exclusive regions (or sub-domains) and each processor is assigned one of these subdomains. Particles are then assigned to processors according to their positions and may migrate between processors' sub-domains as the system evolves.

Direct Lagrangian method — After particles are assigned to processors, this assignment remains fixed throughout the simulation. That is, particles will not move from one processor to another.

The above two approaches lead to two different implementation schemes for parallel PIC codes. Lubeck and Faber implemented a PIC algorithm for the two-dimensional electrostatic problem

using the direct Lagrangian method on a 128-node INTEL iPSC/1 hypercube [8]. To simplify the communication process Lubeck and Faber chose to replicate the mesh grid array so that each processor contains all the mesh grid data. This requires global communication operations to carry out the work of maintaining the same mesh grid array on every processor. In the scatter phase, the contributions of particles to the grid points are directly summed into the mesh grid array in each processor and then the mesh grid array is element-wise summed over all processors. This global sum operation maintains the same copy of mesh grid data in all processors. After the field solve phase, a global concatenation operation is necessary to broadcast the results of field values over all processors. The results presented by Lubeck and Faber show that the direct Lagrangian method is an efficient algorithm for small hypercubes. However, for large hypercubes the communication due to global operations on mesh grid array dominates the run time of the PIC simulation.

3.1 Domain partitioning strategies

Instead of replicating the mesh grids in each processor, the other option is to partition the mesh array into rectangular submeshes and assign a different submesh to each processor. In this case the contributions of particles to their grid points are sent directly to the owner processors. The main issue is to decompose two arrays to reduce the amount of off-processor data access. Efficient parallelization of each iteration during the system evolution requires that

1. the number of particles assigned to each processor be nearly equal;
2. the number of grid points assigned to each processor be nearly equal; and
3. the communication between the two data structures be minimized.

The first two conditions are required to maintain a good processor utilization for computations in the four phases, while the third condition is required to reduce communication. Achieving all three conflicting goals together is difficult. In the following we describe and analyze three potential ways to achieve the above goals:

Grid Partitioning — Partition the grid cells such that each processor gets an equal number of cells. The particles are then assigned based on the grid cells to which they belong.

Particle Partitioning — Partition the particle domain such that each sub-domain has an approximately equal number of particles. The grid cells are then assigned based on the corresponding sub-domain to which they belong.

Independent Partitioning — Partition the particle domain such that each sub-domain has an approximately equal number of particles. Also, partition the grid cells such that each processor gets an equal number of grid cells.

Initial Condition

	partitioning	Grid	Particle	Independent
computation load	field solve	balanced	unbalanced	balanced
	particle calculation	unbalanced	balanced	balanced
communication	type	local	local	non-local
	amount which is proportional to	size of internal boundaries	size of internal boundaries	sub-domain difference

After a Few Iterations

Direct Eulerian :

	partitioning	Grid	Particle	Independent
computation load	field solve	balanced	unbalanced	balanced
	particle calculation	unbalanced	unbalanced	unbalanced
communication	type	local	local	non-local
	amount which is proportional to	size of internal boundaries	size of internal boundaries	sub-domain difference

Direct Lagrangian :

	partitioning	Grid	Particle	Independent
computation load	field solve	balanced	unbalanced	balanced
	particle calculation	unbalanced	balanced	balanced
communication	type	non-local	non-local	non-local
	amount which is proportional to	sub-domain difference	sub-domain difference	sub-domain difference

Table 1: Comparison of computation load and communication patterns in each of three domain partitioning strategies. When both particles and grid arrays in a processor correspond to the same subdomain, the communication cost is proportional to the boundaries of the subdomain. Otherwise, the cost is proportional to the difference between the grid subdomain and particle subdomain assigned to the processor and the boundary of the grid domain.

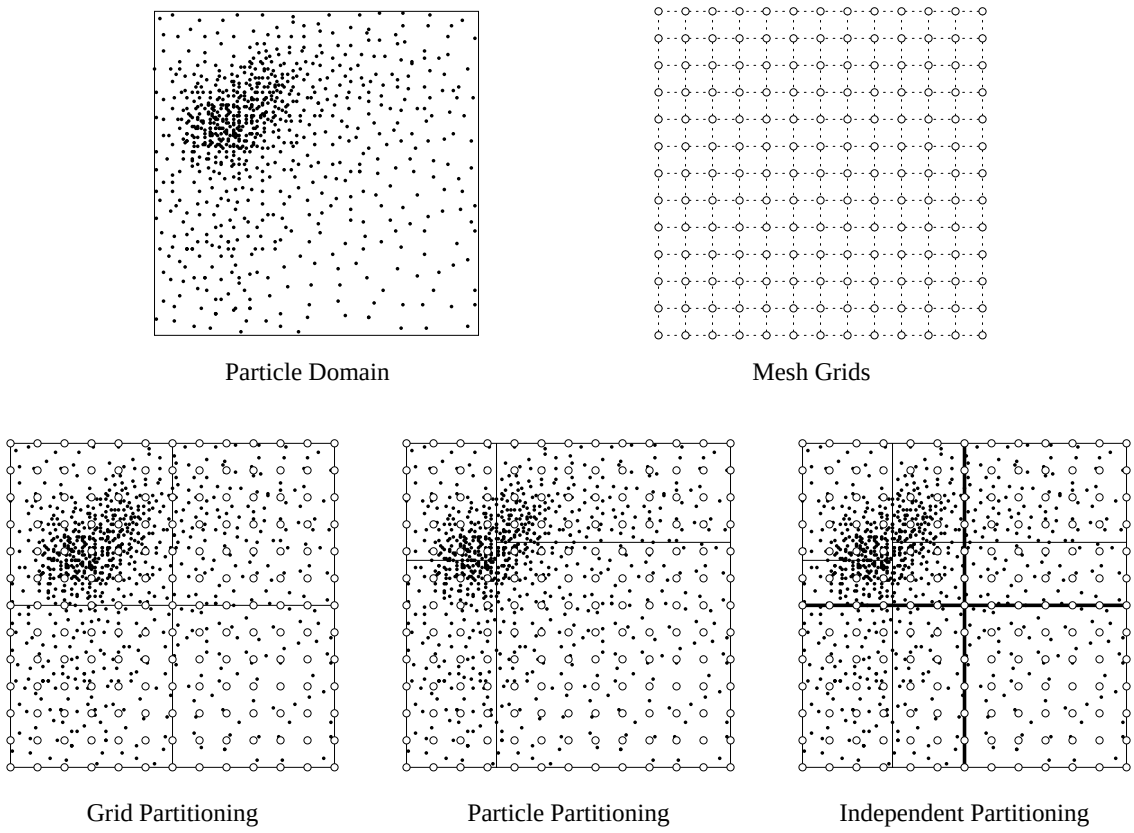


Figure 4: Three particle and mesh grid partitioning methods.

Table 1 analyzes interprocessor communication and computational load balance for the two particle movement methods for each of three domain partitioning strategies. Table 1 illustrates the principle difficulties in designing a scalable and efficient parallel PIC algorithm. The direct Eulerian method results in local interprocessor communication, but places no constraints on the computational load balance. The direct Lagrangian method imposes strict load balance, but makes no attempt to reduce communication. Because mesh grid data distribution is spatially homogeneous, while the particle data can be nonhomogeneous, the decision to choose a proper strategy for particle movement among processors represents a trade-off between communication cost and computational load balance.

Gledhill and Storey [3] adopted the direct Eulerian method on grid partitioning, which, as seen in Table 1, keeps the computation load of particle calculations unbalanced and communication cost low. When particle distribution is spatially highly nonhomogeneous, the problem of load unbalance will degrade the whole performance due to the low concurrent efficiency. Hoshino et al. used both direct Eulerian and Lagrangian methods on the grid partitioning strategy with scatter decomposition of grid points instead of the block partitioning used by Gledhill and Storey [5]. For a large number of processors, the problem of low efficiency still remains, because the computation load is unbalanced throughout the simulation.

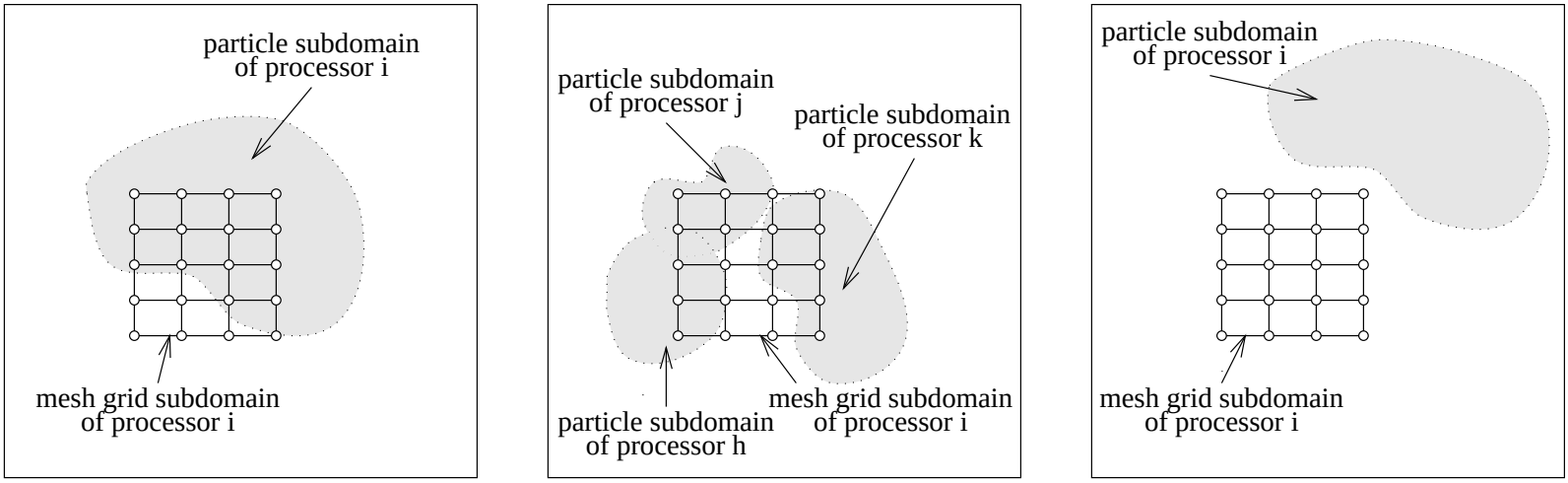


Figure 5: Illustration of particle and mesh grid subdomains in independent partition strategy.

Scalable implementation of the PIC problem for a large number of processors requires that load balance be achieved in all phases. The computational load in each phase is large enough that a load imbalance in any phase could become a bottleneck. For this reason we believe that scalable implementation of PIC codes would require using the direct Lagrangian method and an independent partitioning strategy that keeps the computation load, particles, and grid points strictly balanced. Of course, careful attention must be paid to the communication cost, else it could become a bottleneck.

We next describe a detailed model of the overall cost of the communication generated and strategies on how to minimize this cost while maintaining the load balance in the computation required for all four phases.

3.2 Independent partitioning

Figure 5 illustrates cases that may arise when the particle array and the grid array are mapped using independent partitioning strategy. Figure 5(a) shows the relative positions of two subdomains corresponding to processor i. The particle subdomain corresponds to a convex hull of all the particles assigned to a given processor (this convex hull typically expands as the particles move, assuming the assignment of the particles remains static). The area of particle subdomain outside the mesh grid subdomain represents data to be sent to other processors during the scatter phase. In the gather phase, a processor receives updates from those processors whose mesh grid subdomains are overlapped with the particle subdomain of the receiving processor. Figure 5(b) illustrates that processor i will receive messages from processors h, j, and k in the scatter phase. On the contrary, in the gather phase processors h, j, and k will receive messages from processor i. Further, one can assume that the amount of communication corresponds to the number of unique

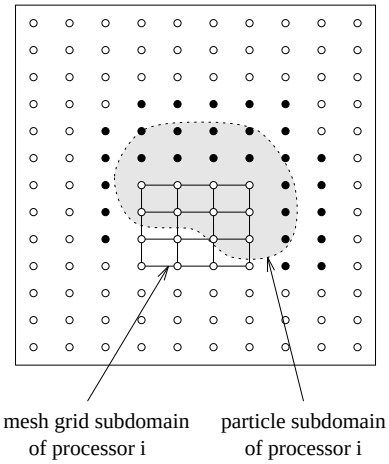


Figure 6: Ghost grid points for processor i.

points of the grid which overlap with the particle domain of another processor. This can be done by sending a sum of all the values and is discussed in detail in the next section.

Ideal partitioning of the two domains should be such that two subdomains for every processor overlap as much as possible. Clearly, in a highly nonhomogeneous distribution two subdomains for a processor may not overlap at all, as shown in Figure 5(c). Maximizing this overlap can be achieved in two ways:

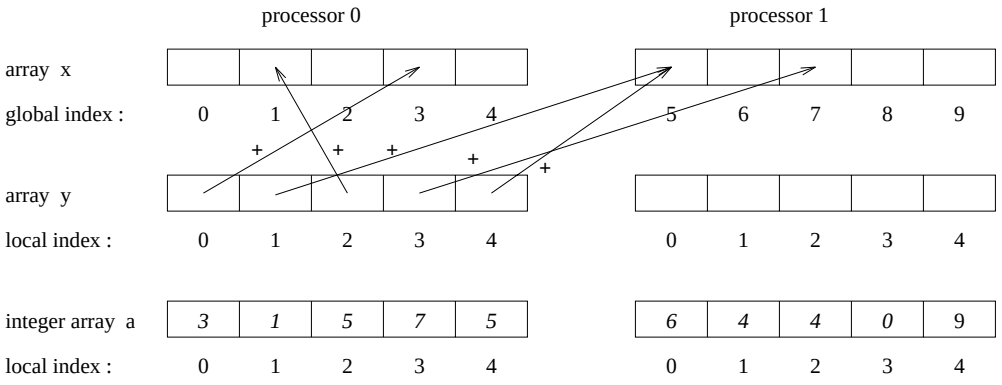
1. Distribute particles such that particles within a subdomain are as spatially close to each other as possible.

2. Align the particle and mesh subdomains with each other such that the overlaps between domains assigned to the same processors are close to each other.

In this paper we show that Hilbert index-based mappings of particles can be used to achieve these two properties simultaneously. Since grid and particle subdomains are partitioned independently and may not be fully overlapped (Figure 5(a)), **ghost grid points** are introduced here to store the particles' contributions to their grid points, which belong to other processors.

The PIC problem uses indirectly indexed arrays. Figure 7 gives an example of indirect indexing where the access pattern of array **x** is determined by the elements of array **a**. Several communication optimizations can be performed to minimize overhead [6]:

Removal of duplicated accesses — For each execution loop, the same off-processor data may be accessed multiple times, but only a single copy of that data can be fetched from an off-processor. The approach to minimize the communication cost by dividing the particle domain into compact subdomains generates duplicated data accesses. Removing duplicates can be done using a hash table [7] or a direct address table. Using a direct address table saves search time for checking duplicated data accesses, but takes memory space proportional to the number of mesh grid points. Figure 8 shows the use of a hash table and a direct address



[loops in each processor]

for $i \leftarrow 0$ **to** 5
$x[\ a[i]\ ] \leftarrow x[\ a[i]\ ] + y[i]$

Figure 7: An example of using an indirect indexed array.

table to remove duplicated off-processor accesses of the example from Figure 7. For this particular problem, the size of the mesh grids is small enough that a direct address table can be used to improve performance. It is worth noting that the PIC problem requires the use of this table for several hundreds of iterations, and hence the initialization cost of this table for a direct hash-based method can be ignored.

Communication coalescing — To reduce startup time, all data destined for the same processor should be collected into a single message.

4 Modeling the communication and computation time

We model a coarse-grained parallel machine as follows. A coarse-grained machine consists of several processors connected by an interconnection network. Rather than make specific assumptions about the underlying network, we assume a two-level model of computation. The two-level model assumes a fixed cost for an off-processor access, independent of the distance between the communicating processors. A unit computation local to a processor has a cost of δ. Communication between processors has a start-up overhead of τ, while the data transfer rate is $\frac{1}{\mu}$. For our complexity analysis we assume that τ and μ are constant and independent of the link congestion and distance between two processors. These assumptions closely model the behavior of the CM-5 on which our experimental results are presented. Although our algorithms are analyzed under these assumptions, most of them are architecture-independent and should be efficiently implementable on meshes and hypercubes.

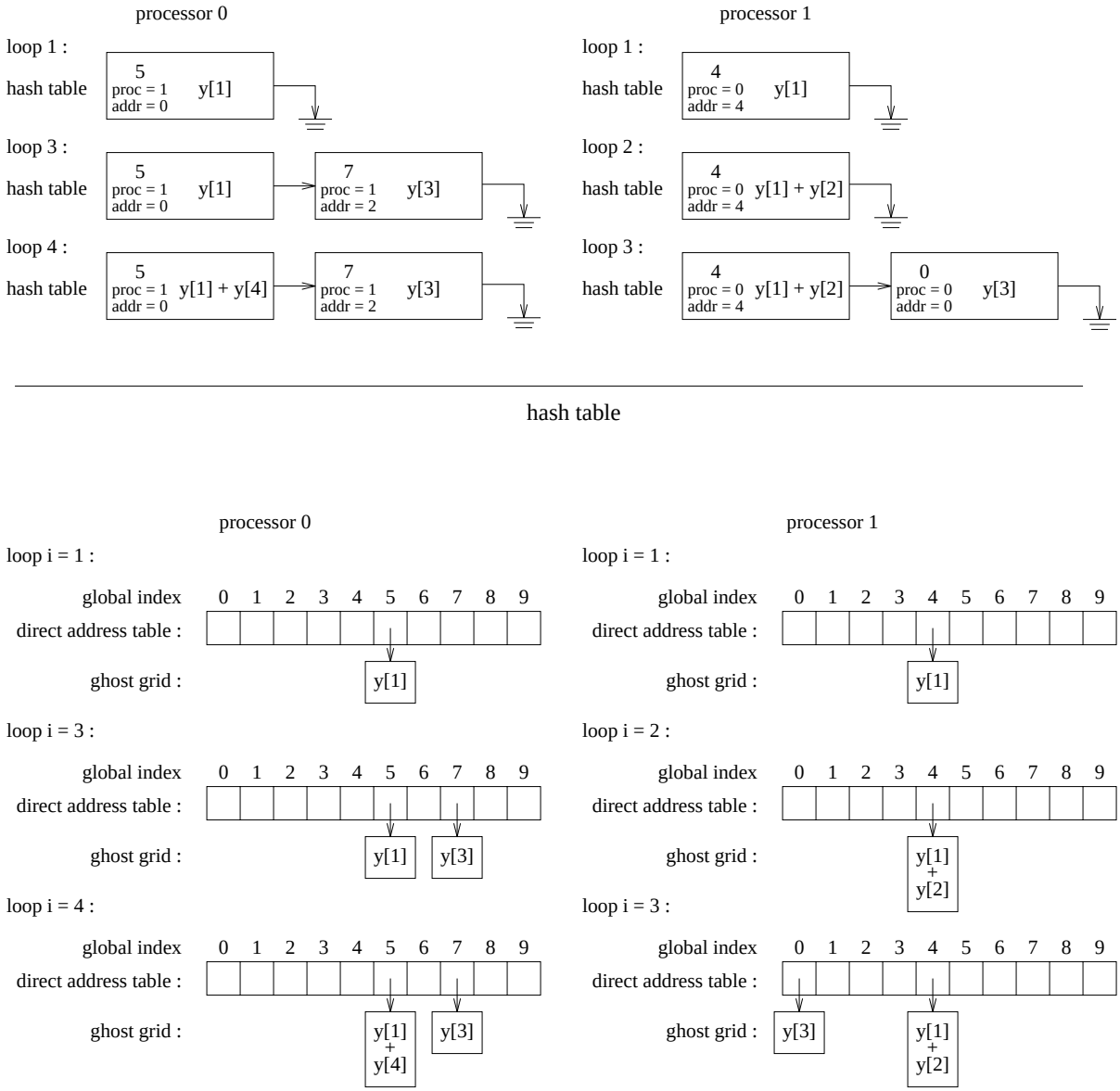


Figure 8: Using hash table and direct address table to remove duplicated off-processor data references in Figure 7.

Here we give a general time complexity analysis of the parallel PIC algorithm using the direct Lagrangian method, and the computational mesh is distributed over all processors. This analysis is developed for a two-dimensional case, and the three-dimensional case is similar. In this paper the particle's **cell** refers to the cell that encloses the particle. The following notation is used:

P: Total number of processors.

n: Total number of particles.

m: Total number of mesh grid points.

l_{grid}: Data size of one mesh grid point.

Scatter phase For each of the $\frac{n}{p}$ particles, the calculations on each of its four vertex grid points are the same, which include: (1) finding the global index of the vertex grid point to see if it is an off-processor access; (2) finding the particle weight at the vertex grid point by interpolating its relative spatial position; and (3) according to the weight, summing the particle's contribution to the current vertex grid point. Let T_{s_comp} be the computation time spent by a particle on one of its four vertex grid points; then the computation time for each processor in the scatter phase is $4 \cdot \frac{n}{p} \cdot T_{s\_comp}$. The communication cost is determined by the number of messages to be sent/received and their sizes. The number of messages to be sent is the number of processors whose mesh subdomains are covered by the ghost grid points, and the number of messages received is the number of other processors whose ghost grid points overlap with the mesh subdomain of the receiving processor. Both these numbers have an upper bound of $p-1$. The total amount of outgoing/incoming data depends on the number of ghost grid points whose upper bound is $u = \min(m - \frac{m}{p},\ 4 \cdot \frac{n}{p})$. Therefore, the upper bound of time spent in the scatter phase is

$$T_{scatter} \leq 4 \cdot \frac{n}{p} \cdot T_{s\_comp} + (p-1) \cdot \tau + u \cdot l_{grid} \cdot \mu.$$

Field solve phase Since a finite difference method is used to solve Maxwell's equations on the mesh grids, each grid point needs data from its four neighboring grid points. Only the grid points on the boundaries of the submesh in a processor will access data from the neighboring processors. Let T_{f_comp} be the computation time spent for updating grid point data and assume that the mesh is square and the number of processors, p, is also square. The message size along each of the four neighbors is $\frac{\sqrt{m}}{\sqrt{p}} \cdot l_{grid}$. The execution time of the field solve phase is

$$T_{fields} = \frac{m}{p} \cdot T_{f\_comp} + 4 \cdot \tau + 4 \cdot \sqrt{\frac{m}{p}} \cdot l_{grid} \cdot \mu.$$

Gather and push phase Every particle in the gather phase gathers the electric and magnetic field data from the four vertex grid points. The same ghost grid points generated in the scatter phase are used here to carry the necessary off-processor field data. The communication behavior is just the inverse of the scatter phase, except that two fields, $\mathbf{E}$ and $\mathbf{B}$, instead of one (charge

contributions) are the objects to be transfered among processors. Let T_{g_comp} be the computation time spent on a particle to add the fields' data from one of its four vertex grid points, and the execution time for each processor in the gather phase is $4 \cdot \frac{n}{p} \cdot T_{s\_comp}$. Therefore,

$$T_{gather} \leq 4 \cdot \frac{n}{p} \cdot T_{g\_comp} + (p-1) \cdot \tau + 2 \cdot u \cdot l_{grid} \cdot \mu.$$

Since the timing analysis is based on the direct Lagrangian method in which particles stay in the same processor during the evolution, the push phase has no interprocessor communication cost. Therefore, the time complexity of the parallel PIC algorithm is

$$\begin{aligned} T_{total} \quad \leq \quad & 4 \cdot \frac{n}{p} \cdot T_{s\_comp} + (p-1) \cdot \tau + \qquad u \cdot l_{grid} \cdot \mu \\ + \quad & \frac{m}{p} \cdot T_{f\_comp} + \qquad 4 \cdot \tau + 4 \cdot \sqrt{\frac{m}{p}} \cdot l_{grid} \cdot \mu \\ + \quad & 4 \cdot \frac{n}{p} \cdot T_{g\_comp} + (p-1) \cdot \tau + \qquad 2 \cdot u \cdot l_{grid} \cdot \mu \\ + \quad & \frac{n}{p} \cdot T_{push}. \end{aligned}$$

5 Particle distribution, redistribution and alignment with mesh

Since particles do not migrate between processors when using the direct Lagrangian method, as the system evolves the particle subdomain may extend and overlap with more mesh grid subdomains belonging to non-neighboring processors. When this occurs, the interprocessor communication cost will increase and degrade the whole simulation performance. Thus, after several time steps, it is necessary to redistribute the particles so that the particle subdomain of each processor is spatially coupled with its mesh subdomain and the interprocessor communication decreases.

In addition to maintaining an approximately equal number of particles in each processor for the next step of simulation, the results of the redistribution also have to align the mesh grid and overlapped particle subdomains as well as possible. The interprocessor communication cost will thus be reduced by placing particles in processors that are close to the processors holding the grids that make up the vertices of their cells.

5.1 Hilbert index-based particle distribution

The goal is to partition the particle array such that spatially contiguous particles are assigned to a given processor. Row-major indexing and Hilbert indexing are two of the several ways to index vertices in a two-dimensional grid. Three indexing schemes are shown in Figure 9 for a graph in which the set of vertices are arranged in a grid of size 8×8. Row-major indexing orders vertices such that if two points are along the same row, their indices are close to each other. However, the same property is not maintained along the other dimension. Hilbert indexing maintains the above property along both dimensions. This indexing scheme can be generalized to n-dimensions and used to convert an n-dimensional index into a one-dimensional index such that proximity in

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(a)

0	1	2	3	4	5	6	7
15	14	13	12	11	10	9	8
16	17	18	19	20	21	22	23
31	30	29	28	27	26	25	24
32	33	34	35	36	37	38	39
47	46	45	44	43	42	41	40
48	49	50	51	52	53	54	55
63	62	61	60	59	58	57	56

(b)

0	3	4	5	58	59	60	63
1	2	7	6	57	56	61	62
14	13	8	9	54	55	50	49
15	12	11	10	53	52	51	48
16	17	30	31	32	33	46	47
19	18	29	28	35	34	45	44
20	23	24	27	36	39	40	43
21	22	25	26	37	38	41	42

(c)

Figure 9: Three indexing schemes applied on a two-dimensional 8×8 mesh grid. (a) Row major array indexing, (b) row-major snakelike array indexing, and (c) Hilbert indexing.

the n-dimensions is generally maintained. Index-based algorithms for partitioning graphs have been described in [11].

For particle distribution in the PIC problem, we first apply the Hilbert index scheme to all cells in the mesh and to addresses of processors. An example of indexing 64 cells on a two-dimensional 8×8 mesh is given in Figure 10. When assigning each cell sub-block to the corresponding processor, cells i and $i+1$ are assigned to the same or adjacent processors, and each processor has a spatially contiguous cell subdomain assigned to it. In addition, cells in a processor are spatially as close to each other as their numerical indices. After indexing cells, we assign particles the same indices as the cells where they lie and sort particles in an increasing order based on the assigned indices. Since particles are arranged with their cells, the particle subdomain assigned in each processor also maintains the property of spatial contiguity. Therefore, the particle distribution not only becomes as simple as a linear alignment of the particle array, but also maintains a property of spatial overlapping between particle and mesh subdomain in each processor. The Hilbert index-based particle distribution algorithm has two major parts:

Particle indexing — Each particle is assigned an index of its global cell number, which is arranged using a Hilbert index-based order.

Sorting — The particles are sorted based on the above indices and the global particle array is distributed equally among all processors in an increasing indexing order. Several algorithms are available in the literature for parallel sorting [2] [12]. A sample-based sorting scheme can be used efficiently to perform the distribution by using an index-based method [11].

For close to uniform distribution, particle subdomains are generally aligned to their mesh subdomains because each particle and its cell have the same index. Assuming mesh applied on the problem space is regular, we divide the mesh into submeshes whose shapes are also regular. For

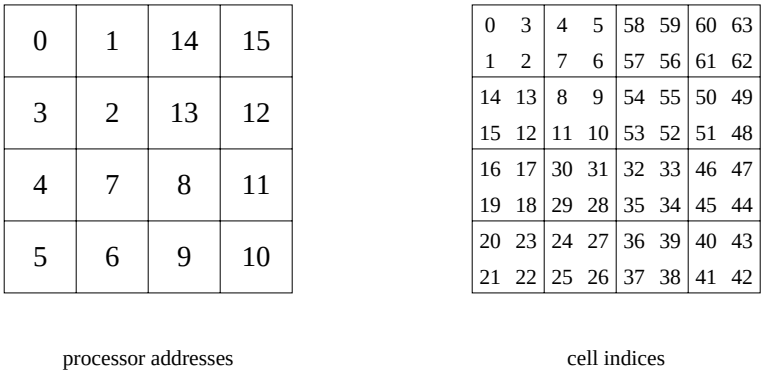


Figure 10: Hilbert indexing scheme is applied on 16 processor addresses and 64 cells in a mesh where each sub-block contains 4 cells and is corresponding to a processor.

example in a 2-dimensional case, a rectangular mesh is divided into smaller square or rectangular subdomains that are distributed among processors (see Figure 10). Since the Hilbert indexing scheme can keep indices spatially close to each other along multidimensions, we apply it on mesh and divide the mesh by sorting the indices among processors, which produces the mesh subdomains with shapes close to rectangular or square. The same results can be generated by applying the Hilbert indexing scheme on particles, obtaining particle subdomains with nearly rectangular or square shapes. For irregular distribution, maintaining an approximately equal number of particles among processors may cause particle subdomains to overlap with more non-neighboring mesh subdomains. In this case, the communication cost between non-neighboring processors dramatically increases, no matter what distribution strategy was chosen. However, applying the Hilbert indexing scheme can keep particles close to each other within a particle subdomain and thus the area occupied by the particle subdomain becomes smaller. The number of possible ghost grid points is reduced and so is the number of off-processor data accesses.

Another advantage of using Hilbert ordering is that the mapping of the next redistribution can be achieved very quickly. We adapted the bucket-based incremental sorting algorithm [10] to redistribute particles and to maintain its sorted increasing order (Figure 12). A sorted local particle array is divided equally into L buckets, and $L-1$ boundaries are obtained for the next stage. A global concatenation operation is performed in line 1 of **Bucket_incremental_sorting( )** to obtain the index boundaries of all processors at the previous sorting. Each element of the local particle array can be classified into three categories, depending on whether the index belongs to the same bucket as the previous, to a different bucket in the same processor, or to another processor. An all-to-many communication subroutine in line 20 is employed to move particles whose indices belong to other processors to their destinations. Lines 22–23 perform the sorting in each of the buckets and then merge with the received particles in line 24. An order-maintaining load balance operation moves extra particles to appropriate destinations such that the global order of concatenated particle array does not change [10]. Particle redistribution achieves better results by using the incremental sorting algorithm than by using the distribution algorithm at each step

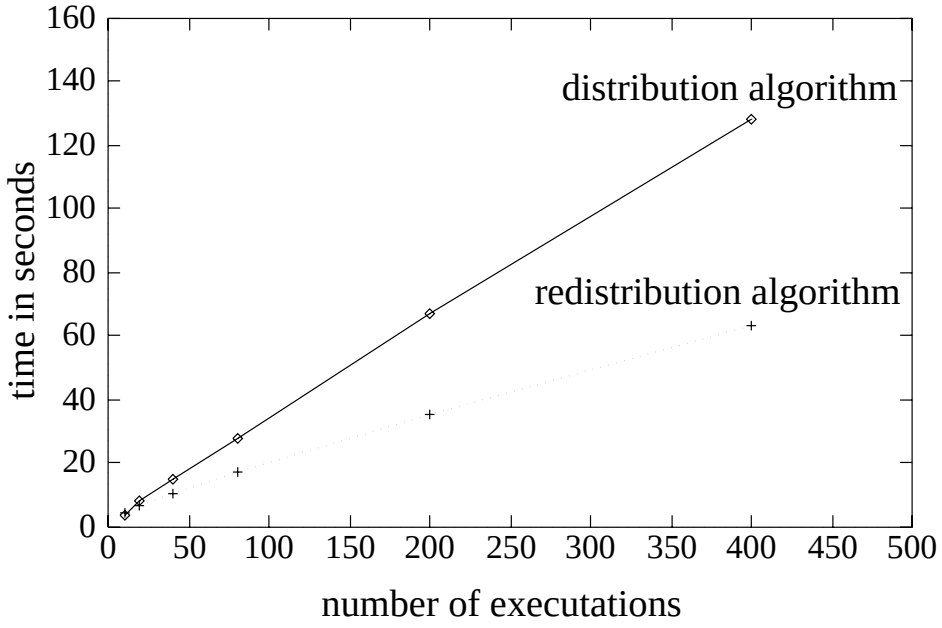


Figure 11: The timing results of distribution and redistribution algorithm.

of redistribution (Figure 11).

5.2 Decision policy for particle redistribution

This section outlines two redistribution decision policies that determine *when* particles need to be redistributed in order to decrease communication costs. The periodic redistribution policy redistributes particles in a fixed number of time steps, while the dynamic one redistributes particles when a certain criterion is satisfied.

Since the periodic redistribution method redistributes particles at intervals of a fixed number of steps, it is insensitive to variations in communication cost requirements. In addition, the periodic redistribution method requires a potentially impractical pre-runtime analysis to determine an optimal periodicity. The *Stop-At-Rise (SAR)* remapping decision policy was introduced in [9]. The SAR heuristic trades the cost of problem remapping against time wasted due to load imbalance. The SAR can also be applied on a PIC algorithm which uses the Lagrangian particle movement strategy to trade the cost of particle redistribution against the increased communication time without redistribution. Since computational loads across processors are strictly balanced, the increase in execution time for each iteration reflects the increase in communication time. We assume the communication time increases linearly as the number of iterations goes up since the last redistribution. Suppose the particles were last redistributed at iteration i_0 and the current iteration is i_1, let t_0 and t_1 represent the execution time at iterations i_0 and i_1, respectively. If the redistribution algorithm is triggered at i_1, then the execution time at $i_1 + 1$ is expected to be t_0 and the expected time saved according to this redistribution will be $(t_1 - t_0) \cdot (i_1 - i_0)$ (Figure 13).

```
Particle_redistribution( )
1   Hilbert_base_indexing(particle)
2   Bucket_incremental_sorting(particle)
3   Order_maintain_load_balance(particle)
4   span ← n/(L·p)
5   for  i ← 0 to L
6      local_bound[i] ← index of particle[i · span]

Bucket_incremental_sorting(particle)
1   global_bound ← global concatenate local_bound[L − 1]
2   span ← n/(L·p)
3   for  i ← 0 to n/p − 1
4      index ← index of particle[i]
5      case  index of
6         local_bound[i/span] ≤ index ≤ local_bound[i/span + 1] :
7               add particle[i] to self_list[i/span]
8         global_bound[my_id-1] ≤ index ≤ global_bound[my_id] :
9               bucket_num ← Binary_search (index, local_bound)
10              add particle[i] to self_list[bucket_num]
11        off-processor :
12              dest ← Binary_search (index, global_bound)
13              add particle[i] to send_list[dest]
14              table[my_id, dest] ← table[my_id, dest] +1
15  table ← global concatenate the my_id^th row of table
16  for  i ← 0 to p − 1
17     send_addr[i] ← send_list[i]
18  for  i ← 0 to p − 1
19     recv_addr[i] ← allocate memory of size table[i, my_id] to store the message from processor i
20  All_to_many_COMM (table, send_addr, recv_addr)
21  locally sorts recv_list based on its particle indices
22  for  i ← 0 to L − 1
23     locally sorts bucket self_list[i] based on its particle indices
24  particle ← merge self_list and recv_list
```

Figure 12: Particle redistribution algorithm.

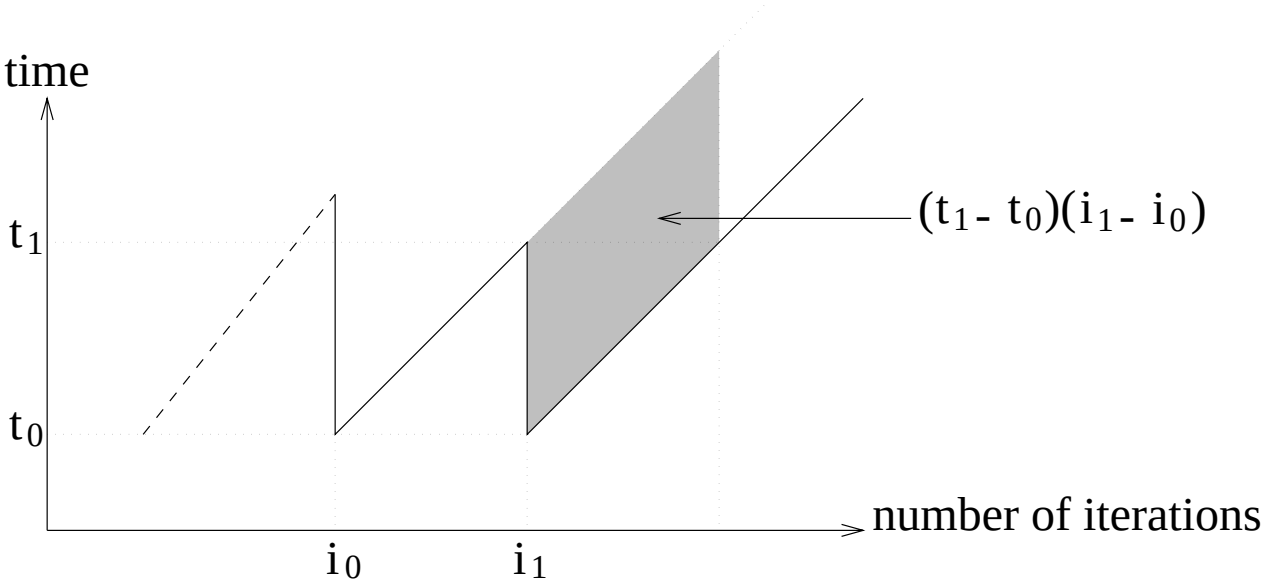


Figure 13: The condition to trigger the particle redistribution.

The shaded area is the time saved by executing particle redistribution. Although the execution time of particle redistribution is different each time, we adopt the previous redistribution time at i_0, $T_{redistribution}$ as the expected time for the current redistribution. It is worthwhile to trigger the particle redistribution subroutine when the expected saved time is larger than the time spent on the redistribution itself. Hence the condition that triggers a particle redistribution is:

$$(t_1 - t_0) \cdot (i_1 - i_0) \geq T_{redistribution}. \tag{1}$$

6 Experimental results

We present results of the simulation of a parallel PIC code for two different cases. The first one consists of simulation of uniformly distributed particles on a two-dimensional problem domain. The second case consists of irregularly distributed particles that are concentrated in the center of the domain (Figure 15). The distribution of particles chosen is highly irregular in order to study the effect of such distribution on the performance of our methods. We expect that most real applications will have intermediate distributions.

6.1 Static vs. periodic redistribution

We compared the performance of static and periodic particle redistribution strategies on two-dimensional PIC codes for a 32-node CM-5. Each simulation performed 2000 iterations. The periods of redistribution chosen were 200, 100, 50, 25, 10 and 5 loops. Figure 16 provides the execution time of three different pairs of numbers of grid points and particles. These results

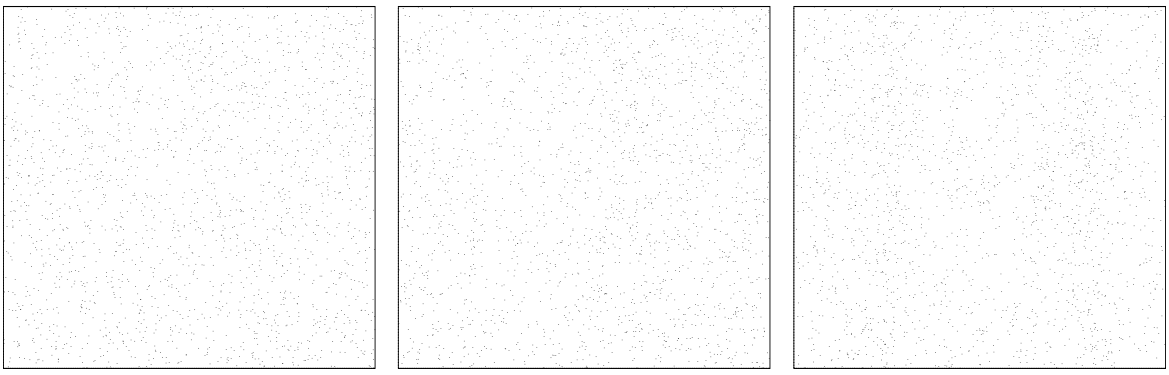


Figure 14: Uniform distribution.

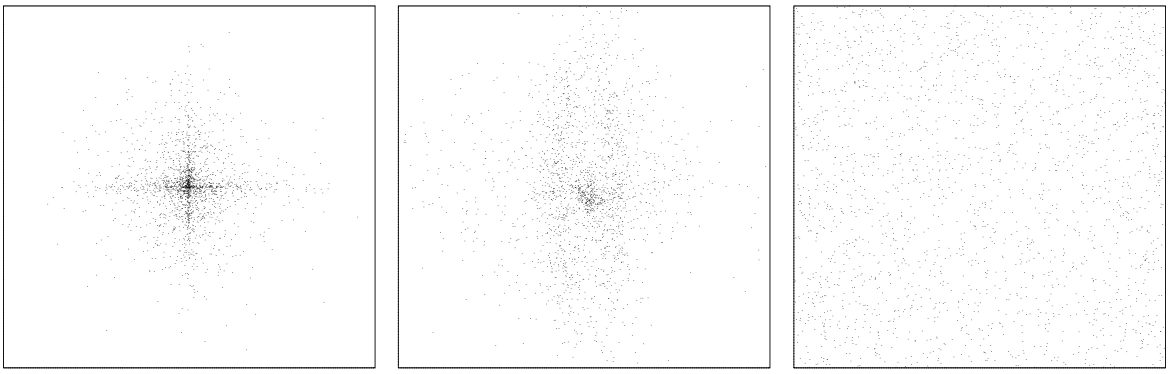


Figure 15: Irregular distribution.

indicate that all the periodic redistribution methods significantly outperform static ones. The period for which the best performance was achieved depended on the distribution of particles as well as other parameters.

The execution time, maximum amount of data sent or received by any processor, and number of messages sent or received by any processor are shown in Figures 17, 18, and 19 respectively. All of these correspond to simulation of 32768 particles with irregular distribution on 32 processors. The size of the mesh grid was fixed at 128×64 mesh. This corresponds to an average of 4 particles per cell. These graphs show that periodic redistribution leads to reduction in all the above parameters.

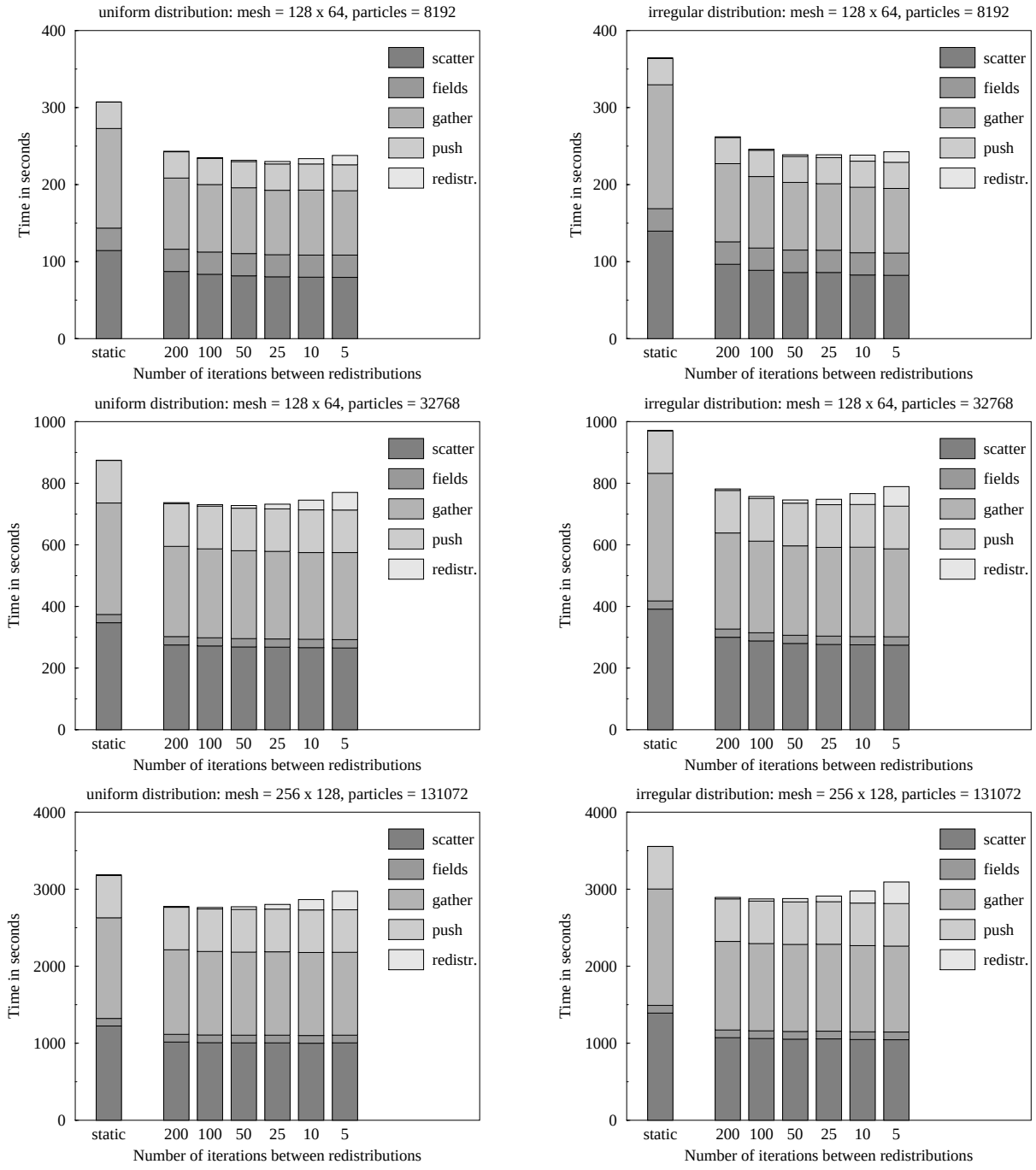


Figure 16: Total execution time for 2000 iterations on 32 nodes.

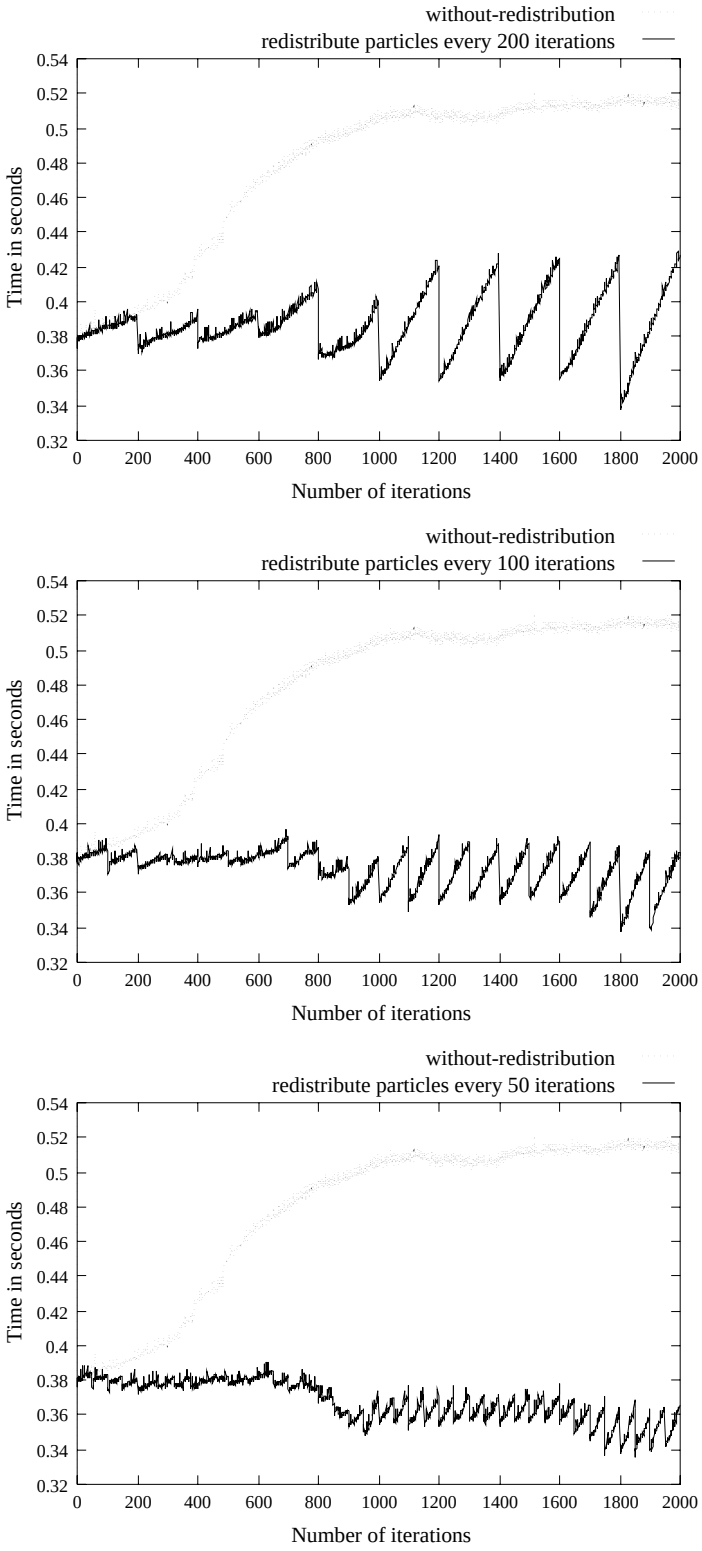


Figure 17: Execution time for each iteration. (Irregular distribution, mesh = 128×64, particles = 32768, processors = 32)

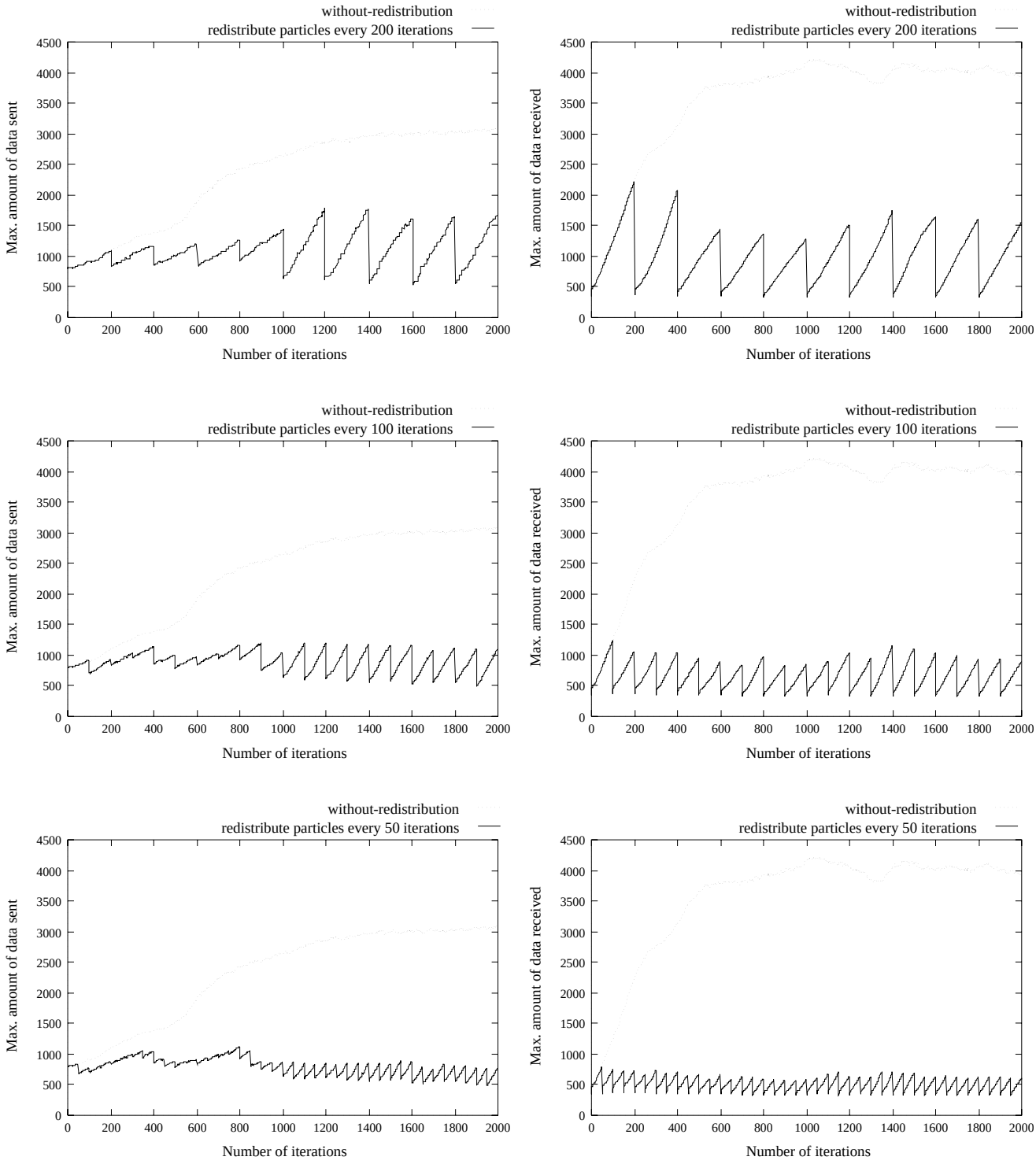


Figure 18: Maximum amount of data sent and received in the scatter phase. (Irregular distribution, mesh = 128×64, particles = 32768, processors = 32)

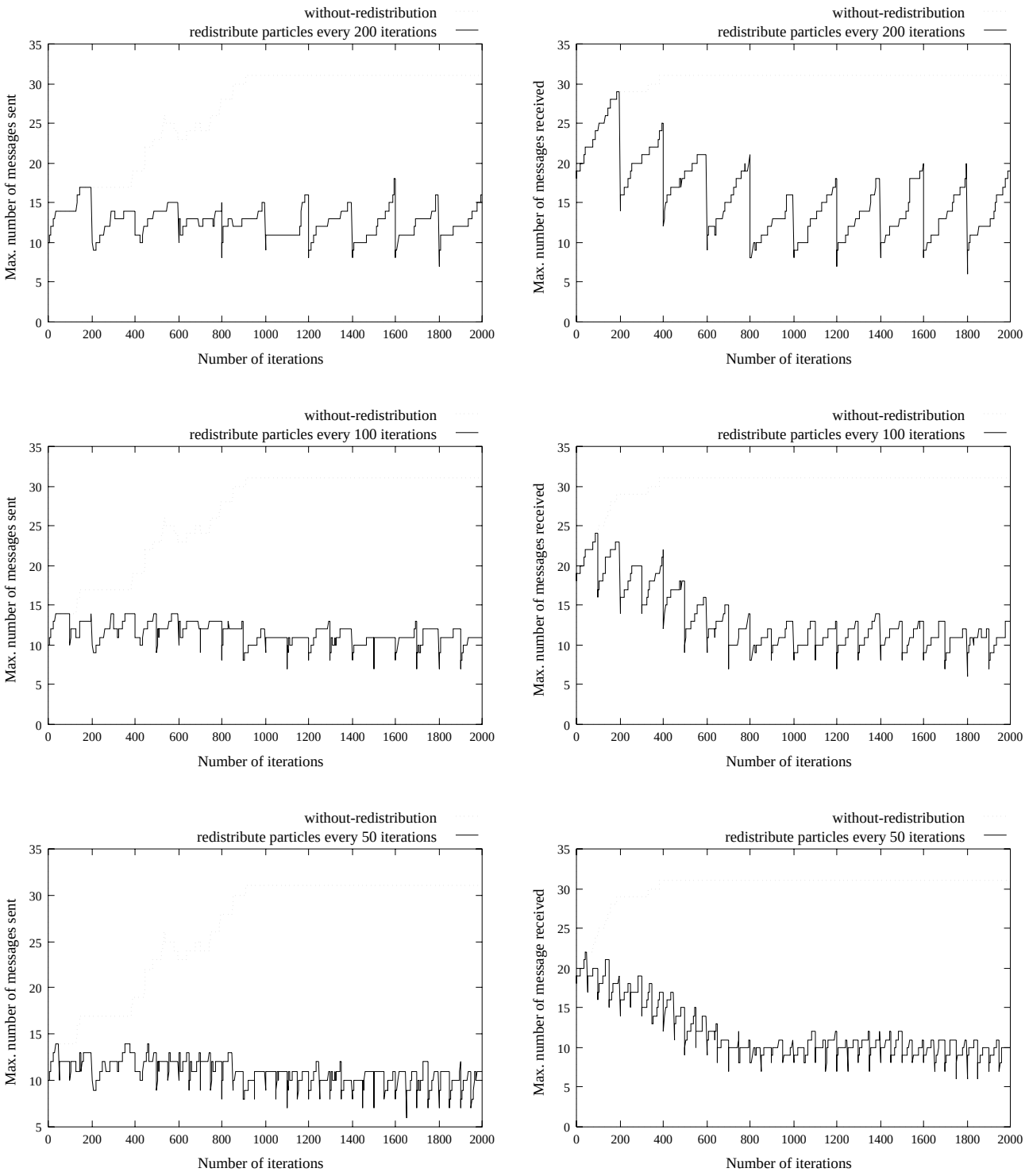


Figure 19: Maximum number of messages sent and received in the scatter phase. (Irregular distribution, mesh $= 128 \times 64$, particles $= 32768$, processors $= 32$)

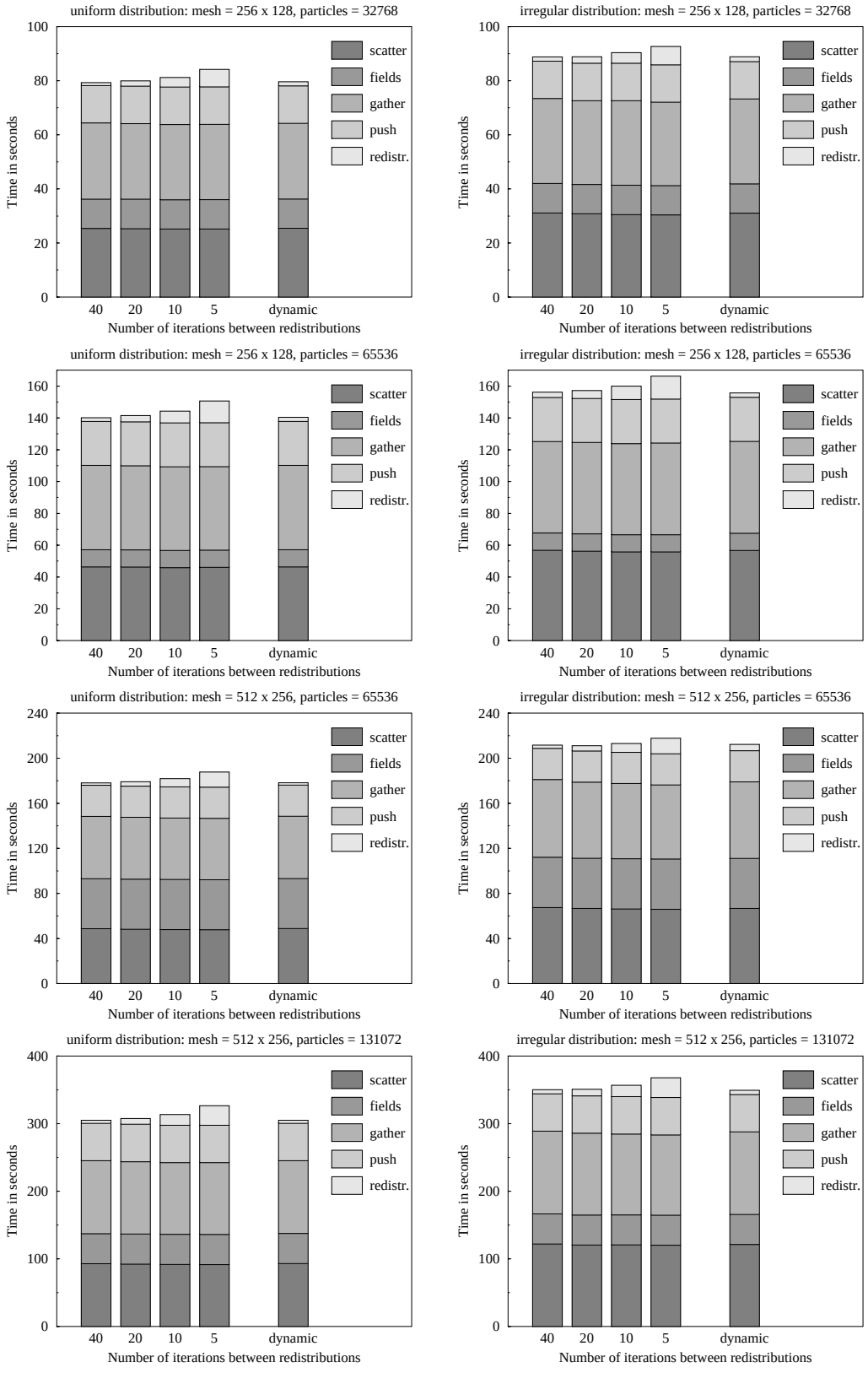


Figure 20: A comparison of periodic and dynamic redistribution for 200 iterations.

				number of processors		
distribution	mesh	particles	indexing	32	64	128
uniform	256×128	32768	Hilbert	72.47	38.77	19.83
			snake	72.74	39.66	20.50
		65536	Hilbert	132.65	71.67	36.44
			snake	133.76	72.74	37.64
	512×256	65536	Hilbert	169.85	91.51	45.58
			snake	170.41	91.03	46.57
		131072	Hilbert	292.55	157.15	79.26
			snake	294.87	159.09	80.82
irregular	256×128	32768	Hilbert	74.88	39.61	20.92
			snake	74.89	39.72	20.81
		65536	Hilbert	138.30	74.06	38.47
			snake	139.15	74.25	38.59
	512×256	65536	Hilbert	176.95	92.96	47.83
			snake	178.34	94.44	48.81
		131072	Hilbert	292.55	157.15	79.26
			snake	294.87	159.09	80.82

Table 2: Computational time (in seconds) of 200 iterations.

6.2 Periodic vs. dynamic redistribution

The dynamic redistribution method assumes linearly increasing execution time and attempts to reduce communication overhead by trading the redistribution cost and the increased execution cost since the last redistribution. Figure 20 compares this strategy for simulation of 200 iterations on the 32-node CM-5. Choosing an optimal period for the periodic redistribution is difficult. Increasing the frequency has an adverse effect on the total time (execution cost + redistribution cost). The performance of dynamic redistribution is close to the periodic redistribution with the best period. Since this strategy uses runtime information such as redistribution cost, execution cost right after the last distribution and execution time of the current iteration, it succeeds in adapting to the system behavior without any prior analysis.

6.3 Hilbert vs. snakelike indexing

We compared the performances of Hilbert and snakelike indexing schemes for partitioning particles. Dynamic redistribution strategy was used for both indexing schemes. Simulations for 200 iterations were performed on uniformly and irregularly distributed particles. Tables 2 shows the computational time of the 200 simulations. The corresponding overhead is given in Figures 21 and 22.

The overhead includes particle redistribution cost and communication cost in the phases of

scattering, field solving, and gathering. The amount of overhead becomes an indicator of redistribution quality for different indexing methods. The results presented here show that:

1. The Hilbert indexing scheme is better than snakelike indexing in all cases except for the case when the number of particles per processor are very small (32K particles on 128 processors with irregular distribution). It assigns particles spatially close to each other along multiple dimensions to reduce the off processor accesses. The subdomains created by a corresponding snakelike ordering are rectangular in nature with high aspect ratios. This results in boundaries with larger perimeters and greater communication cost.

2. Reasonable alignment between the mesh and particle arrays can be achieved by using the cell information of a particle in determining its spatial index. Although no explicit measurements were provided to demonstrate this, our experimental results show that the resulting communication remains the same or decreases with the increase in number of processors. The amount of decrease in overhead is larger for a larger number of particles. The above observation is true for uniform as well as irregular distributions.

3. When the number of particles per processor remains the same, similar efficiencies are maintained for a particular distribution (Table 3). For example, 32K particles on 32 processors and 64K particles on 64 processors (for a 256×128 mesh) have similar efficiencies. The same results are obtained in case of 64K particles on 64 processors and 128K particles on 128 processors (for a 512×256 mesh). Thus, assuming that the granularity of computation per processor remains the same, the indexing schemes scale extremely well for a large number of processors.

4. As discussed in Section 5, the Hilbert indexing scheme allows for using an incremental sorting algorithm to reduce the cost of redistribution. The total cost of redistribution using a Hilbert-based strategy was lower than snakelike-based ordering in all cases. This suggests the number of redistributions required when using a mapping which takes multiple dimensions into account is preferable. Further, the overhead of redistribution accounted for less than 20% of the total overhead for 128 processors.

The results in Table 3 demonstrate that good efficiencies were achieved for 128 processors, even for irregular distributions. Clearly, the CM-5 (without vector units) is not representative of a typical parallel machine, because the ratio of unit computation to unit communication is small. These efficiencies would be much smaller for a machine with more powerful nodes relative to the communication network. Maintaining similar efficiencies on such a machine would require a larger number of particles per processor.

			number of processors		
distribution	mesh	particles	32	64	128
uniform	256×128	32768	0.91	0.88	0.78
		65536	0.95	0.92	0.87
	512×256	65536	0.95	0.93	0.88
		131072	0.96	0.95	0.91
irregular	256×128	32768	0.84	0.79	0.72
		65536	0.89	0.84	0.79
	512×256	65536	0.83	0.78	0.72
		131072	0.88	0.82	0.77

Table 3: Efficiency of Hilbert indexing scheme.

7 Conclusions

The parallelization of PIC problems requires careful partitioning of particle and mesh domains to balance the computational load and minimize communication cost. We have presented a detailed analysis of the communication cost and computational load imbalance for several partitioning strategies. We have demonstrated that using an independent strategy can maintain load balance and keep the total communication cost at an acceptable level. We also demonstrated that using a Hilbert index-based scheme allows for fast partitioning and repartitioning of particles, while maintaining proximity of particles along more than one dimension.

8 Acknowledgments

We would like to thank David Walker for providing the sequential PIC codes. We would also like to thank the Army High Performance Computing Center at the University of Minnesota and the Northeast Parallel Architectures Center at Syracuse University for providing access to their CM-5. We would like to thank Elaine Weinman for proofreading this paper.

This work was supported in part by NSF under ASC-9213821, ARPA under contract #DABT63-91-C-0028, NAG-1485, and ONR under contract #N00014-93-1-0158. The content of the information does not necessarily reflect the position or the policy of the United States government, and no official endorsement should be inferred.

References

[1] C. Birdsall and A. Langdon, *Plasma Physics Via Computer Simulation*. McGraw-Hill, New York, 1985.

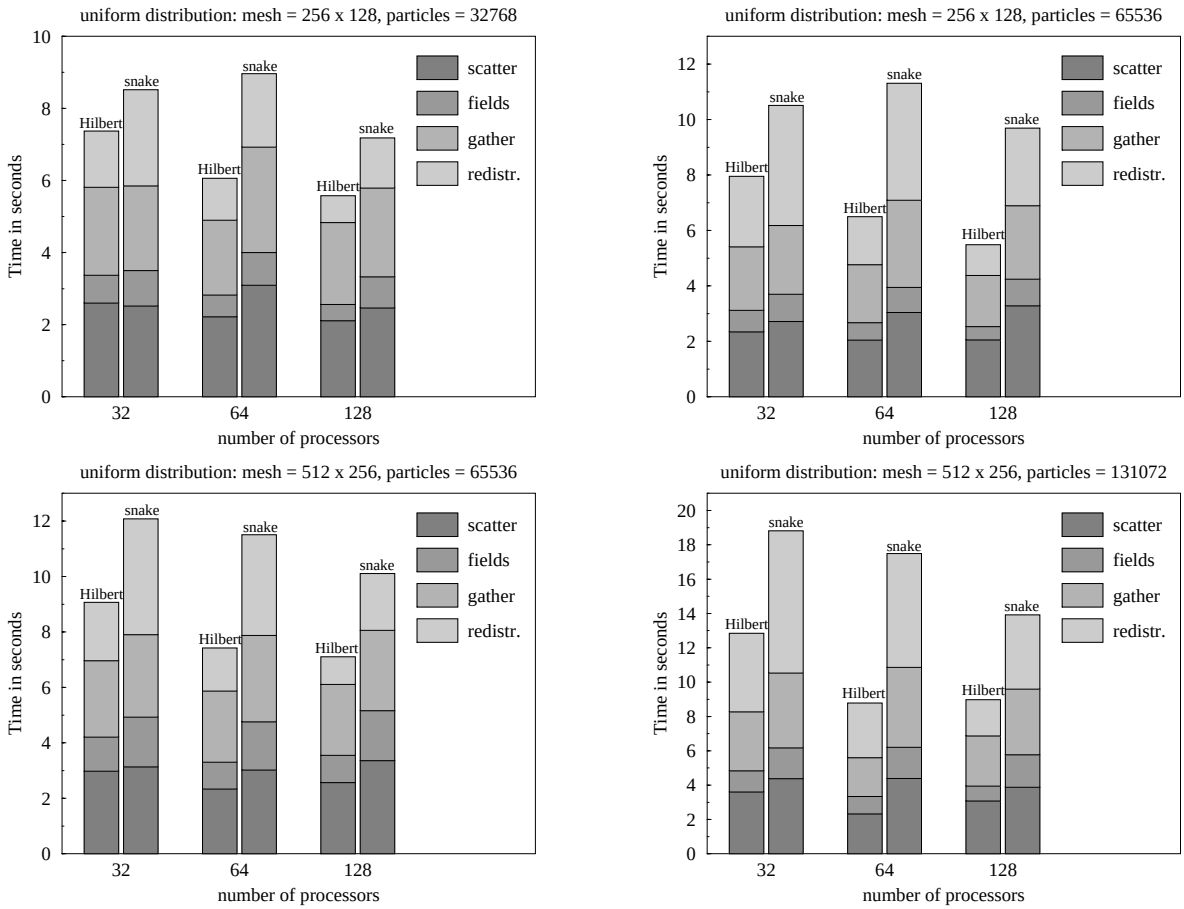


Figure 21: Overhead (execution time — computation time) of 200 iterations for uniform distribution.

[2] M. Bolorforoush, N. Coleman, D. Quammen, and P. Wang, "A Parallel Randomized Sorting Algorithm," *Proceedings of the 1993 International Conference on Parallel Processing*, August 1992.

[3] I. Gledhill and L. Storey, "Particle Simulation of Plasmas on the Massively Parallel Processor," *Proceedings of the First Symposium on Frontiers of Massively Parallel Computation*, pp. 37–46, 1987.

[4] R. Hockney and J. Eastwood, *Computer Simulation Using Particles*. Adam Hilger, Bristol, England, 1988.

[5] T. Hoshino, R. Hiromoto, S. Sekiguchi, and S. Majima, "Mapping Schemes of the Particle-in-Cell Method Implementation on the PAX Computer," *Parallel Computing*, vol. 9, pp. 53–75, 1988.

[6] Y. Hwang, R. Das, J. Saltz, B. Brooks, and M. Hodoscek, "Parallelizing Molecular Dynam-

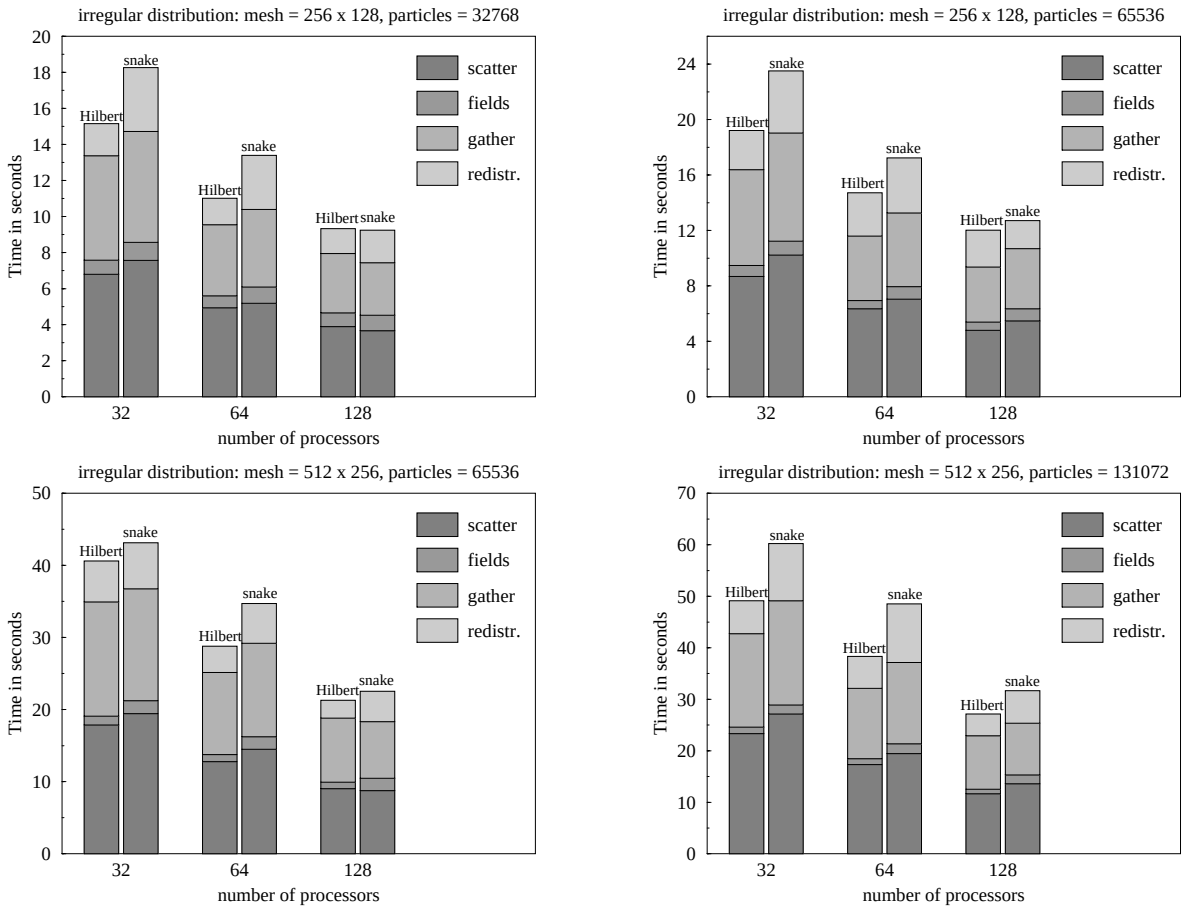


Figure 22: Overhead (execution time — computation time) of 200 iterations for irregular distribution.

ics Programs for Distributed Memory Machines: An Application of the CHAOS Runtime Support Library," Technical CS-TR-3374 and UMIACS-TR-94-125 Report CS-TR-3374 and UMIACS-TR-94-125, UMD, November 1994.

[7] Y. Hwang, B. Moon, S. Sharma, R. Das, and J. Saltz, "Runtime Support to Parallelize Adaptive Irregular Programs," *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing*. SIAM, May 1994.

[8] O. Lubeck and V. Faber, "Modeling the Performance of Hypercubes: A Case Study Using the Particle-in-Cell Application," *Parallel Computing*, vol. 9, pp. 37–52, 1988.

[9] D. Nicol and J. Saltz, "Dynamic Remapping of Parallel Computations with Varying Resource Demands," *IEEE Trans. on Computers*, vol. 37, no. 9, pp. 1073–1087, September 1988.

[10] C. Ou and S. Ranka, "Parallel Remapping Algorithms for Adaptive Problems," *Frontiers' 95*, pp. 367–374, February 1995.

[11] C. Ou, S. Ranka, and G. Fox, "Fast Mapping and Remapping Algorithm for Irregular and Adaptive Problems," *Proceedings of the 1993 International Conference on Parallel and Distributed Systems*, pp. 279–283, Taipei, Taiwan, December 1993.

[12] H. Shi and J. Schaeffer, "Parallel Sorting by Regular Sampling," *Journal of Parallel and Distributed Computing*, vol. 14, pp. 361–372, 1992.

[13] D. Walker, "Characterizing the Parallel Performance of A Large-Scale, Particle-in-Cell Plasma Simulation Code," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 257–288, December 1990.