Accelerating a Particle-In-Cell Simulation using a Hybrid Counting Sort

K. J. Bowers[1]

Electrical Engineering and Computer Science Department
University of California at Berkeley

E-mail: kbowers@eecs.berkeley.edu

Received January 19, 2001; revised May 23, 2001

In this article, performance limitations of the particle advance in a particle-in-cell (PIC) simulation are discussed. It is shown that the memory subsystem and cache-thrashing severely limit the speed of such simulations. Methods to implement a PIC simulation under such conditions are explored. An algorithm based on a counting sort is developed which effectively eliminates PIC simulation cache thrashing. Sustained performance gains of 40 to 70 percent are measured on commodity workstations for a minimal 2d2v electrostatic PIC simulation. More complete simulations are expected to have even better results as larger simulations are usually even more memory subsystem limited.

Key Words: plasma simulation, particle in cell, cache thrashing, counting sort

1. INTRODUCTION

Particle-in-cell (PIC) simulation is a popular method for numerically modeling many body systems. The method's popularity derives from its conceptual simplicity, the relative ease at which simulations may be implemented and the fact that PIC simulations capture kinetic effects such as wave-particle interactions sometimes neglected by fluid models. Often, PIC simulations are implemented from first-principles (without the need for an approximate equation of state). However, these simulations often are computationally expensive with restrictive time step and mesh spacing limitations.

Detailed numerical analysis of PIC methods with an emphasis on plasma and beam simulation and a history of the technique may be found in Birdsall and Langdon's definitive book [1]. Hockney and Eastwood [2] also provide an excellent reference to the technique and demonstrate the application of PIC methods

[1]Presently at Agere Systems (formerly the Lucent Bell Labs Microelectronics Group), Room 1E332, 700 Mountain Avenue, Murray Hill, NJ 07974, kjbowers@agere.com

to a wide variety of problems including semiconductor physics and gravitational simulation.

In a PIC simulation, the motion of a macro-particle (representing many physical particles) is solved by numerical integration using fields interpolated from a mesh ("push"). The sources for the fields are obtained by interpolating the macro-particles' positions and velocities onto the mesh ("accumulate"). Typically, the push and accumulate are the dominant computational expense in both memory and time in such a simulation.

Modern workstations employ hierarchies to speed up memory accesses. At the bottom of the hierarchy is main memory and at the top is the processor. Between the processor and main memory are caches. The caches are generally labeled L1, L2, ... with lower numbered caches are higher in the hierarchy. Caches respond to memory transactions faster than main memory and store data recently used by the processor. If data requested by the processor is found in a cache, it is a "cache-hit" and the memory transaction completes quickly. If the data is not in a cache, it is a "cache-miss" and the memory transaction is much slower. Most processors transparently manage the caches—automatically deciding what data should be kept and anticipating future transactions ("prefetching"). In general, caches work poorly for sequences of memory transactions which randomly access large (compared to the cache size) amounts of memory. Such a sequence of transactions "thrashes" the cache.

In simulations which store the particles in a global array or similar data structure, adjacent particles in the array at any given timestep are usually located at random positions with respect to the mesh. Even if initially sorted by mesh location, the sorting eventually decays as simulation time advances due to particle drift, loss and creation. As a result, the memory reads and writes of field quantities in the particle push and accumulate thrash the memory caches heavily—hindering simulation performance.

In this article, a counting sort is hybridized with a particle push and accumulate to reduce this thrashing. A counting sort is an algorithm for sorting a length N list of values where only M different values are possible. The algorithm is order-N in time and order-M in auxillary storage. Counting sorts are discussed in many standard computer-science texts—see for example Cormen *et al* [3] (II.9.2). A counting sort may be implemented either in-place or out-of-place. The out-of-place sort creates a sorted copy of the input. The in-place sort works directly on the input (requiring less memory than an out-of-place routine).

In the hybrid algorithm developed here, the particles are sorted by mesh location simultaneously with the push and accumulate. The implementation requires a minimal number of extra computations as the push and accumulate already generate most of the information necessary to do a counting sort. Variants of the hybrid algorithm are explored and speed-memory tradeoff are discussed.

Counting sorts to accelerate a PIC simulation have been employed before. In Decyk *et al* [4], a out-of-place counting sort is done on one particle coordinate every couple of timesteps. Counting sorts are used more aggresively in this work by fully integrating the sort with particle operations, sorting on every particle position coordinate and by performing the sort every time step.

On commodity workstations in terms of the number of particles processed per second, typical performance increases for large simulations were in the range of 40 to 70 percent depending on problem size, processor and memory architecture. These performance increases are seen even though the benchmarks used are heavily biased against the hybrid routine. Emphasis is placed in this article on Pentium III-based workstations and clones as these are widely used low cost systems.

In addition to improved performance, having a sorted particle array makes implementing intra-cell collision algorithms much easier. An example of such an algorithm for Coloumb collisions in PIC simulations is given by Nanbu and Yonemura [5].

While this article focuses on efficient implementation of a PIC simulation, modeling techniques for reducing the computational expense of such simulations may be found in Kawamura *et al* [6].

A note on terminology: A particle advance refers to both pushing and accumulating the particle and a "MPAS" refers to millions of particle advances per second. A "flop" here is a single floating point operation. "MFLOPS" refers to millions of floating point operations per second. Similarly, a "mop" refers to a double precision memory operation (load or store) and "MMOPS" refers to millions of double precision memory operations per second. MMOPS is a measurement of "memory bandwidth"—the rate at which large blocks of data can be transferred. Memory subsystem performance is also characterized by "latency"—the time between a memory request and completion of that request. Modern processors generally try to hide latency by "pipelining" memory transactions. When pipelining, memory is requested well before it is needed so that the processor can do useful work while the memory transaction completes. For pipelining to work, the processor and the compiler must be able to efficiently "schedule" transactions so that processor is able to work while waiting for memory.

2. SIMULATION ALGORITHMS

The code used for the benchmarking in this article is a minimal 2d2v (two spatial dimensions, two velocity dimensions) electrostatic PIC simulation. A regular Cartesian mesh with periodic boundary conditions is employed. The diagnostic set is minimal; only the particle positions, velocities and mesh density are computed. The particle push is done with an explicit leap-frog method using fields bi-linearly interpolated from the mesh. The particle accumulate is done using bi-linear weighting.

The particle advance and accumulate are the same algorithms used by the code PDP2 (see Vahedi *et al* [7] and Vahedi and DiPeso [8] for a description of the code; see Birdsall and Langdon [1] for an analysis of this type of particle advance). Unlike PDP2, all operations are all done in double precision as is appropriate for larger simulations. In a large simulation at single precision, the field interpolation may be unacceptably coarse as many significant bits of the particle's position are needed to identify the mesh location and similar noticeable errors in the accumulation may occur due to this excessive truncation error.

Brief ANSI-C source code giving an optimized unsorted particle advance, an optimized sorted particle advance and an in-place counting sort may be found in the appendix. ANSI-C was chosen over FORTRAN because ANSI-C is more widely used and better supported on commodity workstations which are the emphasis of this article. In fact, many FORTRAN compilers on commodity workstations convert source code to C and compile the result with a C compiler. While FORTRAN is a staple in the supercomputing community and recent incarnations support object-oriented programming, the object-oriented language C++ (ANSI-C is a subset of C++) has been a particularly popular language for PIC simulation as object oriented coding techniques map well onto simulations which have to handle numerous types of boundary conditions; C and C++ based PIC codes include PDP1 (Verboncoeur *et al* [9]), PDP2 (Vahedi *et al* [7] and Vahedi and DiPeso [8]), OOPIC (Verboncoeur *et al* [10]), POOMA (Cummings and Humphrey [11]) and ICEPIC (Blahovec *et al* [12]).

3. SIMULATION IMPLEMENTATION

Before evaluating the hybrid counting sort algorithm, it is important to understand the limitations the processor and memory subsystem place on PIC simulation performance. The processor and memory capabilities also have a strong influence on how a PIC simulation should be implemented. As is shown below, the performance of the minimal PIC simulation described previously is almost entirely dictated by the memory subsystem on typical commodity hardware. More complete simulations (with additional diagnostics for example) are expected to be as reliant if not more so on the memory subsystem.

For the benchmark simulation, for every particle during a time step, the following memory operations are performed:

- Load the particle position and velocity (4 loads)
- Load the electric field for interpolation (8 loads)
- Store the updated particle position and velocity (4 stores)
- Load the particle density for weighting (4 loads)
- Store the updated density (4 stores)

The following floating pointing operations are performed per particle:

- Compute the particle mesh location and offset (6 flop)
- Compute the interpolated electric field (18 flop)
- Advance the position and velocity (6 flop)
- Handle periodic boundary conditions (4 flop)
- Compute the particle mesh location and offset (6 flop)
- Compute the density weighting (9 flop)

The flop counts are estimates that include operations such as converting a floating point number to an integer and back (used to find the mesh location and offset).[2]

The particle data is generally much larger than the size of the processor caches and usually particles are accessed sequentially only once per timestep during the

[2]Many compilers generate notoriously poor machine language for converting a floating point number to an integer. For the minimal PIC simulation here, additional performance gains (sometimes double) are obtained though the use of inline assembly for this operation alone. The benchmark results invoke such a compiler modification.

TABLE 1
Attainable Performance on a Typical Workstation: Larger numbers are better.

	Multiply (10^6 ops/s)	**Add** (10^6 ops/s)	**Total** (10^6 ops/s)
FPU performance			
3 cycle pipelined multiply add	396	396	798
'`dgemm`'	283	283	566

	Load (10^6 ops/s)	**Store** (10^6 ops/s)	**Total** (10^6 ops/s)
L1 cache performance			
Load	651		651
Store		664	664
Copy	258	258	515
Modify-in-place	482	482	963
L2 cache performance			
Load	427		427
Store	265		265
Copy	157	157	314
Modify-in-place	264	264	527
Main memory performance			
Load	97.3		97.3
Store		29.6	**29.6**
Copy	29.7	29.7	59.4
Modify-in-place	31.1	31.1	62.1

* The above data was measured on a dual-processor 800/133 MHz Pentium III with a dual channel 800 MHz ECC RDRAM memory subsystem. All operations are done with double precision numbers. The benchmark of simple unrolled loops was compiled using the GNU compiler '`gcc`' version 2.96 (compiler flags: '`-Wall -pedantic -ansi -O3 -fomit-frame-pointer -funroll-all-loops`') under the operating system Linux version 2.2.16. Only one processor was used for the benchmarks and the benchmarks make no use of non-portable platform specific features. The dominant limiting factor for a particle advance (stores to main memory) is highlighted.

particle advance (twice if accumulating in a separate loop). Thus, the particle advance is limited by the bandwidth of the main memory when accessing large amounts of sequential data. Through pipelining, the main memory latency is not a significant limiting factor.

Table 1 shows the attainable double precision floating point unit and memory subsystem performance on a Pentium III processor. Here 'attainable' indicates that such performance can be achieved in portable code without relying on special architectural features. Thus these measurements differ somewhat from the theoretical figures sometimes given in product specifications. Most algorithms can only reach a fraction of the attainable; for reference, the performance of the basic linear algebra subroutine (BLAS) '`dgemm`' is also given. '`dgemm`' (double precision real full matrix-matrix multiply) is an effective measure of the maximum sustained MFLOPS performance that a useful portable algorithm can achieve on a given platform. The '`dgemm`' measurement in the table is obtained from the platform self-tuning library ATLAS developed by Whaley and Dongarra [13].

For data which must be loaded from main memory, modified and stored (as is the case for the particle array), Table 1 indicates an implementation should perform roughly 9.1 floating operations per main memory operation ($\sim$ 566 MFLOPS sustained / 62.1 MMOPS sustained; these figures may be found in Table 1) to achieve similar performance to the reference '`dgemm`'. The PIC algorithm above performs between 2 and 6 floating point operations per main memory operation, depending on how the advance is implemented (2 corresponds to the worst case of a two pass advance with cache misses for the fields; 6 corresponds to the best case of a one pass advance with cache hits for the fields). Thus, the particle advance is heavily memory bandwidth limited.

Accordingly, the benchmark simulation is implemented to simultaneously keep the flop count and memory traffic minimal. Whenever tradeoffs between memory traffic and floating point operations occur, the benchmark simulation is implemented to reduce memory traffic. For example, the particle mesh location and offset calculation is done twice but could be done only once if the particle mesh location is saved between timesteps. However, this increases memory traffic by two integers loads and two integers stores per particle per timestep. Assuming 32-bit integers, this is equivalent to a savings of 6 flop at a cost of 2 mop. This results in a slower more memory subsystem dependent performance regardless whether or not the field accesses are cache hits or the particles are processed in one pass or two passes. A second example is the precomputation of interpolation coefficients for each cell to speed the push. Such an array saves many flops during the velocity update but doubles the total amount of field information that must be loaded from main memory during the push—eliminating performance gains.

The necessity of keeping memory traffic minimized indicates that instead of using a structure of arrays for the particle data appropriate for vectorized platforms (i.e. the particle coordinates and velocities are kept in separate arrays), an array of structures layout for the particle data results in higher performance on typical workstations (i.e. the particle data for coordinates and velocities are stored in one array in which the data for a specific particle is contiguous in memory). Also, the the array of structures should be accessed only once per timestep if feasible to reduce loads to main memory. Accordingly, the benchmark simulation packs the particles into a single array with all the data for a given particle in adjacent memory locations; the push and accumulate are done in a single pass also.

4. REASONABLE EXPECTATIONS

If the particles are located randomly with respect to the mesh, all the field accesses are likely to be cache misses in a large simulation. (A large simulation here is a simulation in which the electric field and density information are much larger than the processor caches.) The cache thrashing problem is excerbated as the mesh size increases. In a large simulation, the maximum performance on the benchmark platform described in Table 1 is expected to be approximately 2.0 MPAS. This figure was calculated assuming 16 loads (particle, electric fields and density) from main memory at 100 MMOPS, 8 stores (particle and density) to main memory at

30 MMOPS, 49 flops at 800 MFLOPS per particle. Explicitly:

$$2.0\,\text{MPAS} \approx \left(\frac{16\,\text{loads}}{100\,\text{MMOPS}} + \frac{8\,\text{stores}}{30\,\text{MMOPS}} + \frac{49\,\text{flop}}{800\,\text{MFLOPS}} \right)^{-1} \tag{1}$$

Actual performance in a large simulation is likely to be somewhat lower as the electric field and density accesses may also involve high memory latencies. However, this latency is somewhat mitigated if the compiler properly schedules instructions.

If the particles are sorted by mesh location, then the field and density accesses are likely to be cache hits as previously pushed particles are likely to have prefetched the needed field and density information. In such a simulation, the performance of the benchmark platform is expected to be 3.6 MPAS. This was found using the same method as the previous calculation under the assumption that the electric field and density memory accesses hit the L2 cache.

Thus, for the benchmark platform, sorting is expected to result in an approximate eighty percent performance increase if it is done at negligible cost for large simulations. However, common sorting techniques are very slow at sorting the particles. This is because the sorting routine passes through the particle array multiple times. For example, a ‘quicksort’ (an in-place sort) passes through the particles on average $\log_2 N$ times, where N is the number of particles. Thus for the 2d2v particle arrays here, quick sorting requires roughly $8N \log_2 N$ floating point memory loads and stores. Even for modest particle arrays, the memory bandwidth required for a quicksort is far greater than the bandwidth required to push the particles.

5. HYBRID COUNTING SORT

A counting sort is well matched to PIC simulation as it is an order-N operation which may be smoothly integrated into the particle push and accumulate. Pseudocode for an out-of-place counting sort follows:

Algorithm 1 (Particle Counting Sort).

```
Inputs
    $I$ is the particle array to sort containing $N$ particles
    $M$ is the number of mesh cells
Outputs
    $O$ is the sorted particle array
    $P$ is a particle allocation such that $O_{P_{i-1}+1}$ to $O_{P_i}$
            are all the particles in cell $i$
begin
    allocate $N$ particles for $O$
    allocate and set to zero $M$ integers for $P$

    for $n$ := 1 to $N$ do      Step 1: Count the number of particles in each cell
        $i :=$ compute the cell for particle $I_n$
        $P_i := P_i + 1$
    end for

    $k := 0$                            Step 2: Convert $P$ into an allocation
    for $i$ := 1 to $M$ do
```

```
            $j := P_i$
            $P_i := k$
            $k := k + j$
        **end for**

        **for** $n$ **:=** 1 **to** $N$ **do**                          *Step 3: Sort $I$ into $O$*
            $i :=$ compute the cell for particle $I_n$
            $j := P_i$
            $P_i := P_i + 1$
            $O_j := I_n$
        **end for**

        **return** $O$**,** $P$
    **end**
```

The above sorts the particles while only passing through the input particle array twice (for a 2d2v particle array, approximately $6 \sim 8N$ loads and $4N$ stores are performed). This algorithm takes advantage of the fact there are only a finite number of mesh locations. To explain the algorithm: The first loop counts the number of particles in each mesh cell. The second loop converts this count into an allocation which determines how the output particle array should be organized. After this loop is finished, P indicates where in the output array the next unsorted input particle should go. In the third loop, the input array is copied to the output array according to the allocation in P (and the allocation is accordingly updated). At the end of the routine, O contains a sorted particle array and P gives the indices where particles in a particular cell may be found in the output particle array.

This algorithm is well tailored to PIC simulation as the first loop is exactly analogous to the particle accumulate. In fact, in the unlikely case that a nearest grid point (NGP) weighting scheme is being used, the first loop of the counting sort is identical to the charge accumulation. Likewise, both the third loop and the particle push need to compute the input cell location. The second loop operates only on the P array and thus is a negligible expense.

A two-pass particle advance may be trivially modified to perform an out-of-place counting sort simultaneously by accumulating in the first loop of the counting sort and pushing in the third loop of the counting sort. Even though the counting sort does an extra integer load and store in the accumulate, this memory access is likely to be a cache hit as the particle array stays sorted while the simulation is running. Thus the benefits of a sorted particle array are obtained with negligible cost. This gives the simplest version of the hybrid counting sort.

The hybrid routine just described has four minor drawbacks. First, the sorting is not done in-place—increasing memory requirements for the simulation. For a N_s species simulation at least $N_s + 1$ particle arrays are necessary to allow the algorithm operate. Counting sorts may also be done in-place with a substantially more complicated algorithm (source code for an in-place sort is given in the appendix). While a particle push and accumulate may be hybridized with such an in-place algorithm, the resulting code is no faster than the unsorted code as the in-place algorithm effectively accesses the particle array randomly during the sort—

destroying the electric field and density memory access locality necessary for the sorting to give a performance boost.[3]

The second drawback is that the simple hybrid routine executes in two passes. As discussed previously, it is preferable to implement the particle advance in a single pass to reduce memory traffic. The hybrid counting sort can easily be generalized to a single pass routine which performs the count for the next time step while sorting the current time step. This modification is used in the code employed for the benchmarks of the following section. The source code for this variant is given in the appendix.

Third, the output particle array is sorted appropriately for the previous time step. If the sort is being done only for performance reasons, this is not an issue as most of the particles usually move much less than one cell every timestep (i.e. $v\Delta_t/\Delta_x \lesssim 1$). If the sort is being done in order to apply other algorithms needing an exactly sorted particle array, the hybrid counting sort above may be rearranged to count in the particle push and sort in the particle accumulate. This modification however is slower than the two pass hybrid sort and the one pass variant as the push and accumulate must execute separately (the field equations are usually solved after the accumulate and before the next push). Furthermore in this rearranged sort, both the push and accumulate read and write the particle arrays—increasing memory traffic. However, this variant can be faster than sorting the particles in an independent routine.

Lastly, the cell count must be kept accurate between the count and the sort. In particular in PIC-MCC simulations (Particle-In-Cell with Monte-Carlo Collisions) when a particle is created or destroyed through various reactions, the cell count needs to kept current in the MCC algorithm if it is performed after the count but before the sort. Similarly, handling particle boundary conditions where particles are created or absorbed is made slightly more complicated.

6. RESULTS

All the benchmarks use a simulation where the particles have a uniform density and a Maxwellian velocity distribution. The simulation parameters are consistent with the explicit leap-frog time step and finite-mesh instability mesh space conditions ($\omega_p\Delta_t \sim 0.2 < 2$ and $\Delta_x/\lambda_d \sim 1$ respectively) necessary for simulation stability and accuracy. (ω_p, λ_d are the plasma frequency and Debye length respectively; Δ_t and Δ_x are the simulation time step and mesh spacing.) The ratio of simulation particles to physical particles varies as the simulation size is changed. The mesh is doubly periodic and square. As the push and accumulate are the focus of the article, the solution of the field equations is omitted and the electric field is set to zero.

[3] An in-place counting sort algorithm is still useful if the extra memory requirements of the out-of-place algorithm are too high. The in-place algorithm is still an order N algorithm in time and thus the sorting may be done at a speed comparable to a particle advance. If the in-place algorithm is applied periodically, the performance gain of a sorted particle array persists for several timesteps as particles only move a little bit every time step. Periodic sorting to accelerate a PIC simulation was applied in Decyk *et al* [4].

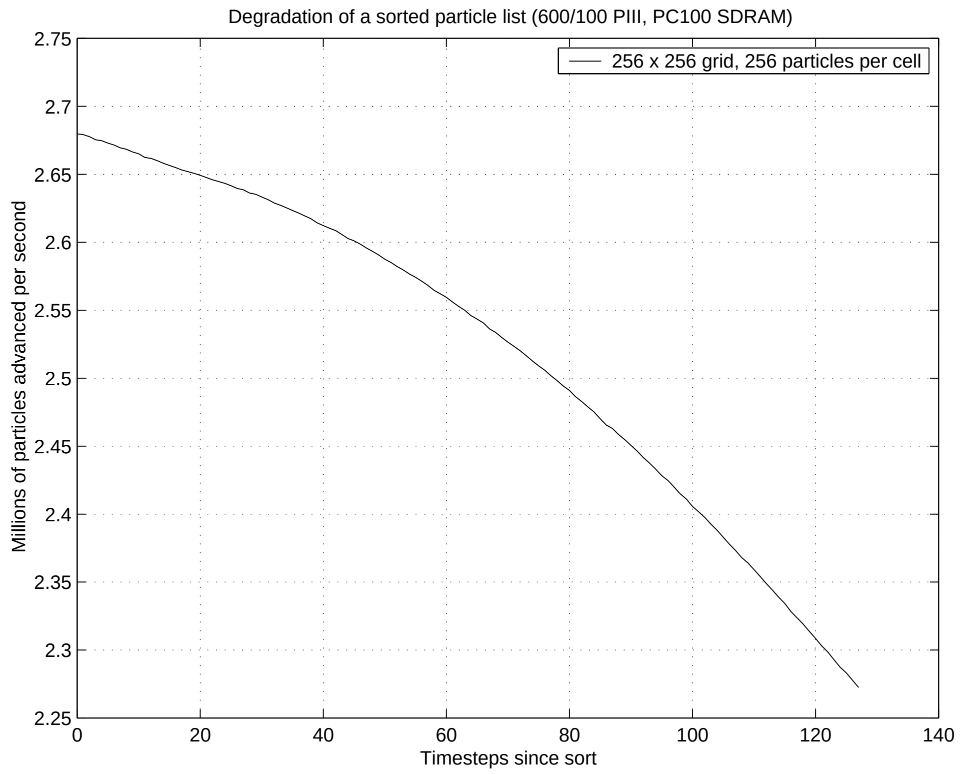

FIG. 1. Degradation of a Sorted Particle Array: As particles drift, a sorted particle array slowly becomes disorganized. Thus, successive timesteps take longer to compute because of increased cache thrashing. The temperature and density of particles used for this benchmark corresponds to a plasma with $\omega_p \Delta_t = 0.2$ and $\Delta_x/\lambda_d = 1$. The number of particles is 16,777,216. The mesh size is 256 by 256.

It should be noted that in simulations of beams or other applications, the memory access pattern is likely to be very different from a thermal plasma. The benchmarks here are most immediately applicable to PIC discharge modeling.

Figure 1 shows the loss of simulation performance as a sorted particle array becomes unsorted due to particle motion. Even over a short interval of 128 timsteps, a roughly 15 percent performance loss is seen. Over longer intervals, the performance loss approaches 50 percent. This gives strong motivation for pursuing efficient sorting techniques.

Figure 2 shows the performance of the unsorted advance routine on the test platform described in Table 1 (Pentium III 800/133 with a dual channel PC800 RDRAM memory subsystem). The measured performance shows distinct plateaus. The highest plateau corresponds to simulations that fit entirely inside the L1 cache. The middle plateau corresponds to simulations whose electric field and particle densities fit entirely inside the L2 cache. However, most simulations are far larger than either of these sizes. For the larger simulations, the performance is seen dropping off as the mesh size is increased. This drop off is relatively independent of number of particles as expected (typically particle arrays do not fit into any cache level even for modest simulation sizes).

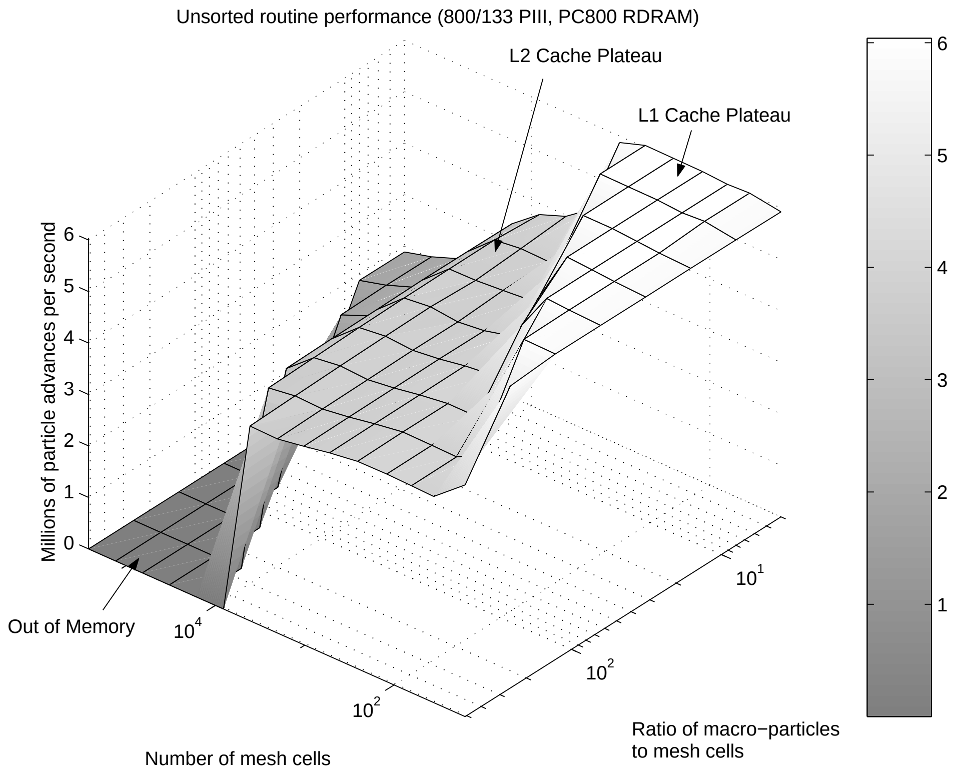

FIG. 2. Unsorted routine performance (RDRAM memory subsystem): This figure shows the performance of the unsorted advance routine on the test platform described in Table 1 (a Pentium III 800/133 with a dual channel PC800 RDRAM memory subsystem). The measured performance shows distinct plateaus corresponding to the performance of the cache supplying the field information for the particle push. For simulations which do not fit into cache, the performance is seen dropping off as the mesh size is increased.

Figure 3 shows the performance of the sorted advance routine on the test platform. The performance shows similar plateaus. However, the L2 cache plateau never rolls off. This indicates that the sorted algorithm effectively performs as though the fields fit within the L2 cache. The sorted algorithm gives consistent high performance regardless of total simulation size (even up to simulations which take all free memory).

Figure 4 shows the performance boost of the sorted routine. For simulations which fit entirely within cache, the sorted routine is marginally slower than the unsorted routine. This is unsurprising as the sorted routine has to do slightly more calculation than the unsorted routine. Similarly, the crossover for a simulation than fits within the L1 and an L2 cache simulation occurs much sooner in a sorted simulation due to the extra memory needed to store the particle array allocation. This accounts for the L1-L2 crossover performance dip seen. For large simulations, performance increases of approximately 70 percent are observed. This boost is consistent with the expected performance boost calculated previously. Likewise the benchmarks agrees nicely with the calculated expected performance of 3.6 MPAs.

Figures 5, 6 and 7 show a similar set of benchmarks on a platform utilizing a different memory subsystem. These results are included to show the sorting

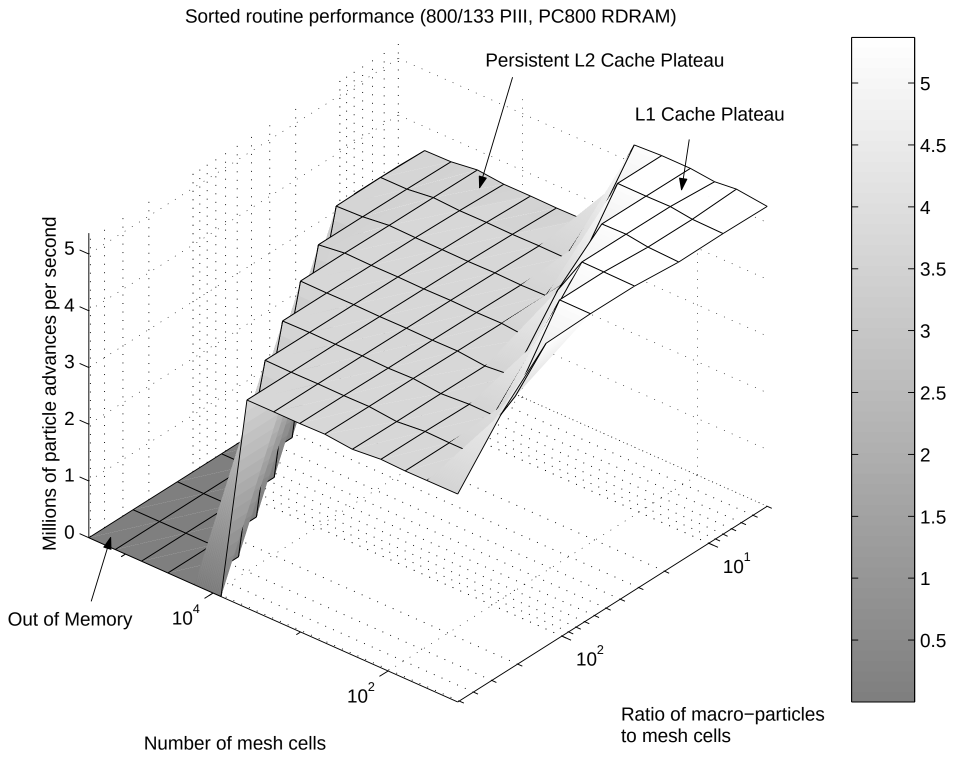

FIG. 3. Sorted routine performance (RDRAM memory subsystem): This figure shows the performance of the sorted advance routine on the test platform. The performance shows similar plateaus. However, the L2 cache plateau never rolls off—indicating the sorted algorithm effective performs as though the fields fit within the L2 cache.

performance when using an SDRAM memory workstation (SDRAM is the most common memory in commodity hardware). Also, these results cover even larger simulations as the SDRAM test platform had substantially more main memory than the RDRAM platform. The SDRAM platform is a Pentium III 600/100 with a PC100 SDRAM memory subsystem. The results show similar characteristics to the RDRAM system but the sorted algorithm receives heavier penalties in small simulations (owing to the slower processor) and the performance gains in large simulations are not as pronounced but still a respectable 40 percent.

7. SUMMARY

Analysis of system performance on commodity hardware indicates the memory subsystem of a workstation is the dominant factor limiting factor for speed in PIC simulations. Accordingly, in this article, an algorithm for accelerating the particle advance in PIC simulations was developed by optimizing memory accesses. The algorithm effectively the eliminates cache thrashing that occurs when interpolating from the mesh to the particles and when accumulating from the particles to the mesh. The algorithm combines a counting sort with a particle advance to sort the particles by mesh location simultaneously with the particle push and accumulate. A sorted particle array accesses the electric fields and densities in a nearly sequential manner—eliminating the cache thrashing.

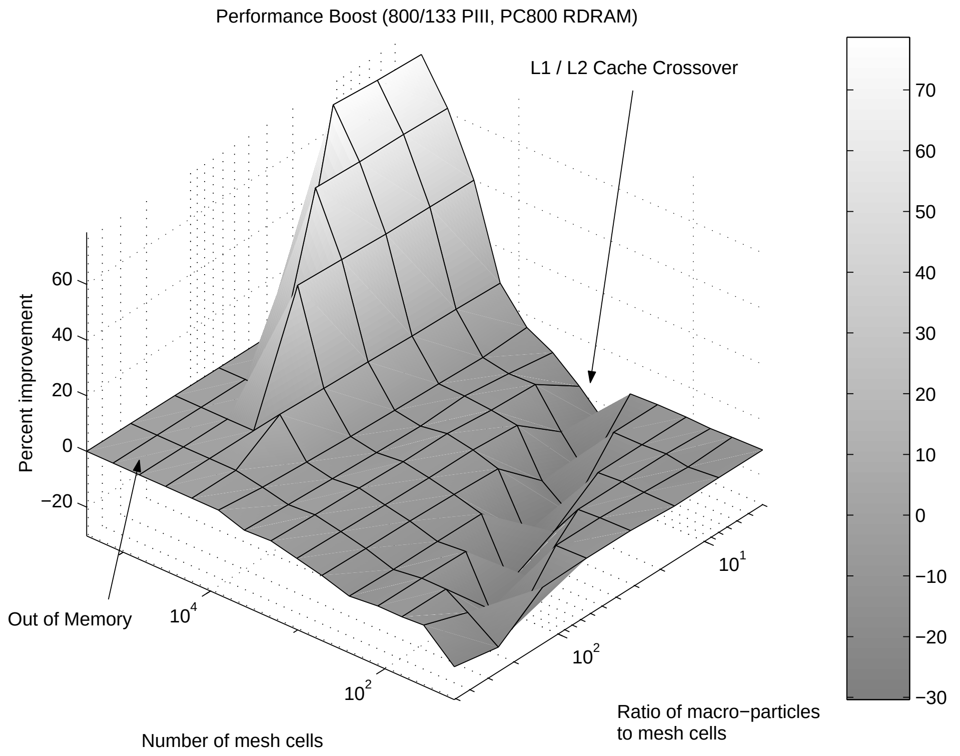

FIG. 4. Performance Boost (RDRAM memory subsystem): This figure shows the performance boost of the sorted routine. For simulations which fit entirely within either cache, the sorted routine is marginally slower than the unsorted routine. For large simulations, performance increases of approximately 70 percent are observed. The L1-L2 crossover is explained in the text.

Performance gains in large simulations of thermal plasmas on meshs containing over 10,000 cells (or roughly 100 cells along an edge) measured between 40 and 70 percent depending on processor and memory subsystem. The benchmarks compared two highly optimized particle advances for a minimal 2d2v PIC simulation (one unsorted and one sorted). Methods to implement the advance to achieve optimal performance on commodity workstations were discussed and realistic performance expectations were calculated from simple FPU measurements. Simulations of beams likely have very different memory access characteristics and may show different performance results.

It should be noted that the performance gains reported here are likely lower than the gains that might be obtained in a more complete simulation of a thermal plasma. This is because the benchmarks done here are heavily biased towards the unsorted case. For example, usually many diagnostics are taken during the accumulate beyond the density diagnostic computed in the benchmarks. This excerbates the cache thrashing of the unsorted advance over the benchmarks used here. A sorted routine however would not exhibit any extra cache thrashing. Likewise, the field solve routine (not included in the benchmarks here) is likely to flush the cache between calls to the particle advance. Thus, the unsorted small mesh benchmark cases overstate the actual performance that might be seen in a real simulation. Also, as the electric field and density mops occur almost entirely out of the L2

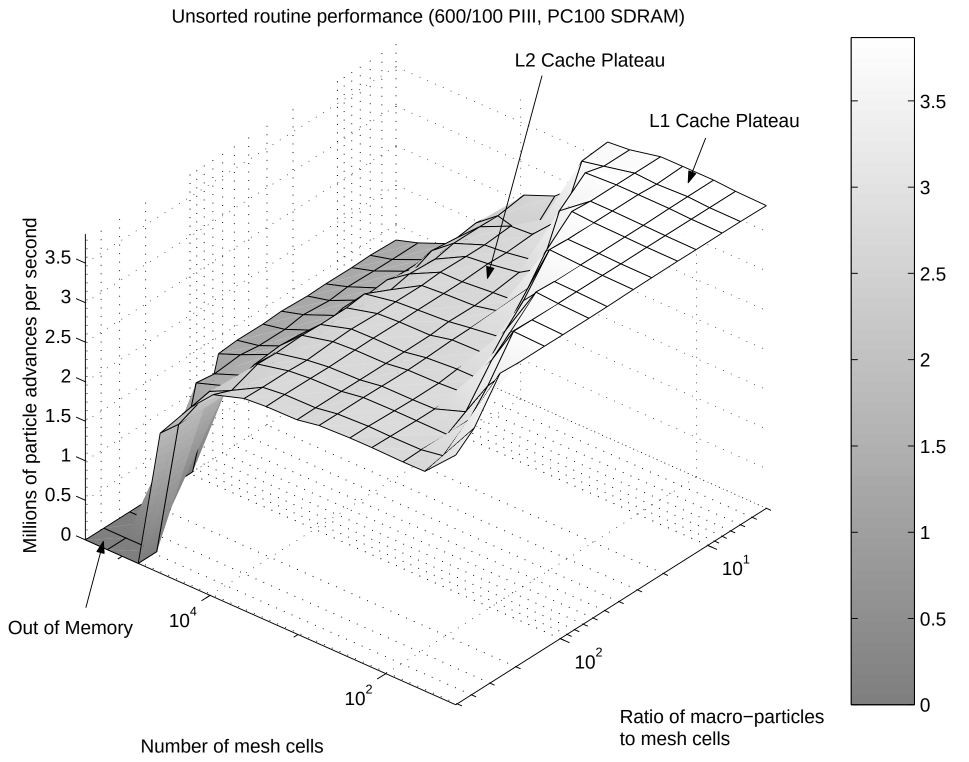

FIG. 5. Unsorted routine performance (SDRAM memory subsystem): This figure shows the unsorted benchmarks analogous to Figure 2 on the SDRAM test platform.

cache in the sorted algorithm, the sorted algorithm scales better to shared memory symmetric multi-processing systems as memory traffic to shared main memory is greatly reduced.

The principle drawback of the new algorithm is that the push and accumulate cannot be done in place. However, code for an in-place counting sort is presented in the appendix. By judiciously using the in-place sort periodically, the benefits of a reduction in cache thrashing can still be achieved without having to significantly modify already existing PIC simulations.

It should also be noted that the benchmarks done here were done in double precision. As the particle advance is limited by the memory subsystem, it is strongly suggested for performance reasons to use single precision whenever sufficiently accurate. However, as discussed previously, in large simulations single precision may not be numerically suitable.

APPENDIX: SELECT BENCHMARK SOURCE CODE

Here, the ANSI-C routines used for the benchmarks in this article are presented. The routine '`advance`' pushes and accumulates a particle array without sorting. The routine '`advance_sort`' performs the same mathematical operations as '`advance`' but also sorts the particles by mesh location. The routine '`sort_in_place`' performs an in-place counting sort by mesh location on a particle array.

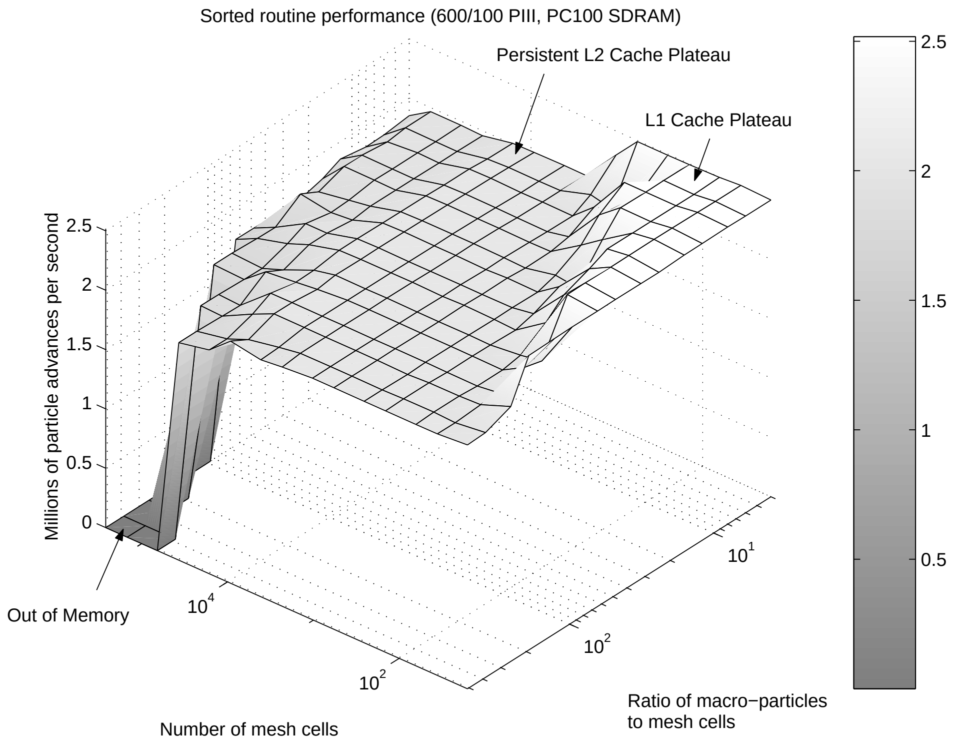

FIG. 6. Sorted routine performance (SDRAM memory subsystem): This figure shows the sorted benchmarks analogous to Figure 3 on the SDRAM test platform.

The push and accumulate routines are implementations of numerical algorithms similar to those used in the University of California - Berkeley Plasma Theory and Simulation Group code PDP2 described in Vahedi *et al* [7] and Vahedi and DiPeso [8]. However, both the unsorted and sorted routines have been heavily optimized for minimal flop count and memory traffic. As such, these routine use different data structures from their PDP2 counterparts and execute in one pass instead of two.

For reference, the particle advance routines apply to an unmagnetized 2d2v non-relativistic periodic electrostatic simulation. The particles are advanced using a second order accurate in time leap-frog algorithm. Thus the particle position and velocity are known at times separated by half a time-step. The particle positions and velocities are normalized to the mesh spacing and time step. Appropriate for an electrostatic simulation (which only needs the charge density to compute the fields), the accumulate routine only accumulates the normalized particle density.

The push and accumulate algorithms use bi-linear interpolation and weighting respectively. Bi-linear techniques may be implemented efficiently and produce acceptable simulation noise levels. Lower order methods (nearest grid point) are too noisy and high order methods are often difficult to implement, particularly near boundaries. However, the sorted routines accumulate the nearest grid point weighting as part of the hybridized sorting algorithm.

These routines were compiled using GNU '`gcc`' version 2.96 under the operating system Linux version 2.2.16 using aggressive optimizations (compiler flags: '`-Wall`

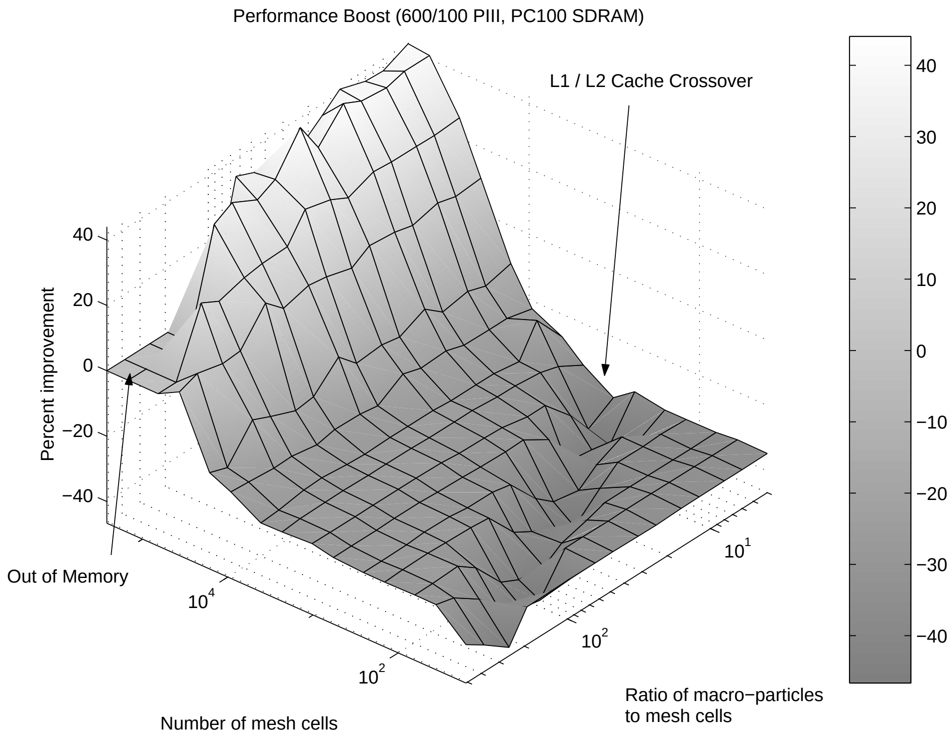

FIG. 7. Performance Boost (SDRAM memory subsystem): This figure shows the performance boost benchmarks analogous to Figure 4 on the SDRAM test platform. The large simulation gains are not as pronounced but still quite respectable. The smaller simulation cases show a significant performance loss but that is a consequence of the slower processor used on the SDRAM test platform.

`-pedantic -ansi -O6 -fomit-frame-pointer -funroll-loops -malign-double -march=pentiumpro`').

```
#include <stdlib.h>
#include <string.h>

/* Swap values (Note: Possible side effects) */
#define swap(a,b,t) do { t __tmp=(a); (a)=(b); (b)=__tmp; } while(0)

/* Allocate and clear an array with n members of type t */
#define alloc(n,t) (t *)calloc((n),sizeof(t))

/* Clear an array */
#define array_clear(ptr,n,t) memset((ptr),0,(n)*sizeof(t))

/* Convert a double to an int (Note: Compilers often generate
   unacceptable assembly for this operation. For high performance
   applications, this should be replaced with inline assembly) */
#define dtoi(d) ((int)(d))

/* p(n) points to the nth particle in the p array
```

```
   p(n)[0,1] is the nth particle x,y coordinate
   p(n)[2,3] is the nth particle x,y velocity
   All are normalized to the mesh spacing and time step */
#define p(n)     (p+(n)*span)
#define in_p(n)  (in_p+(n)*span)
#define out_p(n) (out_p+(n)*span)

/* copy_particle is hard-wired for the span = 4 (2d2v) case */
#define copy_particle( d, s, span ) do {                \
  d[0] = s[0]; d[1] = s[1]; d[2] = s[2]; d[3] = s[3];   \
} while(0)

/* Common code between the unsorted and sorted advance */

#define COMPUTE_CELL_AND_OFFSET(p)  \
    w1 = p[0];      w2 = p[1];      \
    i = dtoi( w1 ); j = dtoi( w2 ); \
    w1 -= i;        w2 -= j
#define PUSH_PARTICLE(out_p,in_p)                            \
    out_p[0] = in_p[0];   out_p[1] = in_p[1];                \
    w3 = ex[i] + w1*( ex[i+1] - ex[i] );                     \
    w4 = ex[j] + w1*( ex[j+1] - ex[j] );                     \
    out_p[2] = in_p[2] + e2ax * ( w3 + w2*( w4 - w3 ) ); \
    w3 = ey[i] + w1*( ey[i+1] - ey[i] );                     \
    w4 = ey[j] + w1*( ey[j+1] - ey[j] );                     \
    out_p[3] = in_p[3] + e2ay * ( w3 + w2*( w4 - w3 ) ); \
    out_p[0] += out_p[2]; out_p[1] += out_p[3]
#define ACCUMULATE_DENSITY()                                \
    w3 = w1*w2; w2 -= w3;                                   \
    density[i] += 1 - w1 - w2; density[i+1] += w1 - w3; \
    density[j] += w2;          density[j+1] += w3
#define APPLY_PERIODIC_BCS(p)                                       \
    while( p[0]<0 ) p[0]+=xmax; while( p[0]>=xmax ) p[0]-=xmax; \
    while( p[1]<0 ) p[1]+=ymax; while( p[1]>=ymax ) p[1]-=ymax

void advance(double *in_p, double *density,
             const double *ex, const double *ey,
             int N, int span, int Ncx, int Ncy,
             double e2ax, double e2ay) {
  int i, j, Ngx = Ncx+1;
  double *p, *stop, w1, w2, w3, w4, xmax = Ncx, ymax = Ncy;

  array_clear( density, (Ncx+1)*(Ncy+1), double );

  for( p = in_p(0), stop = in_p(N); p < stop; p += span ) {
    COMPUTE_CELL_AND_OFFSET(p);
    i += j*Ngx; j = i+Ngx;      /* i,j are offsets for ex and ey */
    PUSH_PARTICLE(p,p);         /* Push in-place */
    APPLY_PERIODIC_BCS(p);
    COMPUTE_CELL_AND_OFFSET(p);
    i += j*Ngx; j = i+Ngx;      /* i,j are offsets for density */
    ACCUMULATE_DENSITY();
  }
}

void advance_sort(double *out_p, const double *in_p,
                  double *density, int *ngp, int *pa,
                  const double *ex, const double *ey,
                  int N, int span, int Ncx, int Ncy,
                  double e2ax, double e2ay) {
  int i, j, k;
  const double *in, *stop;
  double *out, w1, w2, w3, w4, xmax = Ncx, ymax = Ncy;

  array_clear( density, (Ncx+1)*(Ncy+1), double );
  array_clear( ngp, Ncx*Ncy, int );

  /* Convert the input count contained in ‘pa’ to an allocation */
  for( i = k = 0; i < Ncx*Ncy; i++ ) { j=pa[i]; pa[i]=k; k+=j; }

  for( in = in_p(0), stop = in_p(N); in < stop; in += span ) {
    COMPUTE_CELL_AND_OFFSET(in);
    i += j*Ncx;                 /* Compute where to */
    out = out_p( pa[i]++ );     /* store the particle */
    i += j; j = i+Ncx+1;        /* i,j are offsets for ex and ey */
    PUSH_PARTICLE(out,in);      /* Push and sort to out */
    APPLY_PERIODIC_BCS(out);
    COMPUTE_CELL_AND_OFFSET(out);
    i += j*Ncx; ngp[i]++;       /* Count the output particle */
    i += j; j = i+Ncx+1;        /* i,j are offsets for density */
    ACCUMULATE_DENSITY();
  }
}

void sort_in_place(double *p, int N, int span, int Ncx, int Ncy) {
  int i, j, k, Nc = Ncx*Ncy, *pa, *pa_save;
  double *in, *out, *stop, *tmp, p_local[4];

  pa = alloc( Nc+1, int );      /* alloc clears   */
  pa_save = alloc( Nc+1, int ); /* the arrays too */

  /* Count the number of particles in each cell */
  for( in = p(0), stop = p(N); in < stop; in += span )
    pa[ dtoi(in[0]) + Ncx*dtoi(in[1]) ]++;
  /* Convert the cell count to an allocation in pa
     and save a copy of the allocation in pa_save */
  for( i=k=0; i<=Nc; i++ ) { j=pa[i]; pa_save[i]=pa[i]=k; k+=j; }

  i = 0;
  while( i < Nc ) {
    if( pa[i] >= pa_save[i+1] ) {
      i++; /* The current cell is done. Go to the next cell */
    } else {
      /* The current cell still contains unsorted particles. Get
         the next unsorted particle in the current cell for the
         next sorting cycle */
      in = stop = p( pa[i] );
      tmp = p_local;

      do {
        /* Figure out where to store the input particle.
           Update the current allocation accordingly */
        out = p( pa[ dtoi(in[0]) + Ncx*dtoi(in[1]) ]++ );

        /* The "ifs" are minor optimizations.
           They avoid needless memory loads and stores */
        if( out != stop ) {
          copy_particle( tmp, out, span );
          copy_particle( out, in, span );
          swap( tmp, in, double * );
        } else if( out != in ) {
          copy_particle( out, in, span );
        }
        /* Loop until we complete the cycle */
      } while( out != stop );
    }
  }

  free( pa_save ); free( pa );
}
```

ACKNOWLEDGMENT

Thanks to J. Brandon Laflen (Purdue University) for insights on sorting algorithms. Thanks to Dr. C. K. Birdsall and Dr. J. Verboncoeur for helpful discussions. This work was supported by DOE contract DE-FG03-97ER54446 and the Fannie and John Hertz Foundation Fellowship Program.

REFERENCES

1. C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation.* New York, NY: McGraw-Hill Inc., 1985.

2. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles.* Philadelphia, PA: Institute of Physics Publishing, 1988.

3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA: The MIT Press, 1990.

4. V. K. Decyk, S. R. Karmesin, A. deBoer, and P. C. Liewer, "Optimization of particle-in-cell codes on reduced instruction set computer processors," *Comp. in Phys.*, vol. 10, pp. 290–298, May-June 1996.

5. K. Nanbu and S. Yonemura, "Weighted particles in Coulomb collision simulations based on the theory of a cumulative scattering angle," *J. Comp. Phys.*, vol. 145, pp. 639–654, September 1998.

6. E. Kawamura, C. K. Birdsall, and V. Vahedi, "Physical and numerical methods of speeding up particle codes and paralleling as applied to RF discharges," *Plasma Sources Sci. Technol.*, vol. 9, pp. 413–428, May 2000.

7. V. Vahedi, C. K. Birdsall, M. A. Lieberman, G. DiPeso, and T. D. Rognlien, "Verification of frequency scaling laws for capacitive radio-frequency discharges using two-dimensional simulations," *Phys. Fluids B*, vol. 5, pp. 2719–2729, July 1993.

8. V. Vahedi and G. DiPeso, "Simultaneous potential and circuit solution for two-dimensional bounded plasma simulation codes," *J. Comp. Phys.*, vol. 131, pp. 149–163, 1997.

9. J. P. Verboncoeur, M. V. Alves, V. Vahedi, and C. K. Birdsall, "Simultaneous potential and circuit solution for 1d bounded plasma particle simulation codes," *J. Comp. Phys.*, vol. 104, pp. 321–328, February 1993.

10. J. P. Verboncoeur, A. B. Langdon, and N. T. Gladd, "An object-oriented electromagnetic PIC code," *Comp. Phys. Comm.*, vol. 87, pp. 199–211, May 1995.

11. J. C. Cummings and W. F. Humphrey, "Parallel particle simulations using the POOMA framework," in *8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

12. J. D. Blahovec, L. A. Bowers, J. W. Luginsland, G. E. Sasser, and J. J. Watrous, "3-D ICEPIC simulations of the relativistic klystron oscillator," *IEEE Trans. Plasma Sci.*, vol. 28, pp. 821–829, June 2000.

13. R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *ACM/IEEE SC98: 10th Anniversary. High Performance Networking and Computing Conference*, (Los Alamitos, CA), IEEE Computer Society, 1998.