JOURNAL OF COMPUTATIONAL PHYSICS 68, 56–65 (1987)

Vectorizing the Interpolation Routines of Particle-in-Cell Codes*

ERIC J. HOROWITZ

National Magnetic Fusion Energy Computer Center, Lawrence Livermore National Laboratory, Livermore, California 94550

Received December 17, 1985; revised March 22, 1986

A discussion of the interpolation routines in particle-in-cell codes is presented indicating the problems in vectorizing them. Solutions to these problems are then discussed with the timing results indicating the effectiveness of the solutions. A comparison of our methods to those of Nishiguchi *et al.* (*J. Comput. Phys.* **61**, 519 (1985)) is presented. Finally, multitasking is briefly addressed. © 1987 Academic Press, Inc.

I. INTRODUCTION

Partice-in-cell (PIC) codes [1] have always been attractive computational tools due to their inherent simplicity and wide applicability. However, the use of PIC codes for large 3-dimensional problems has been inhibited for a number of reasons. The most obvious is the restriction on the number of particles due to the memory size of the computer. Though we could use buffered memory to store the information of numerous particles, such a process would be prohibitively slow. But there is a more subtle problem related to the structure of PIC codes. In general, a PIC code is a cycle consisting of four steps:

1. Solving fields on a grid.
2. Interpolating fields to particle positions.
3. Advancing particle positions and velocities consistent with the fields.
4. Interpolating particle charge and current densities to the grid.

It would be desirable to generate efficient vectorized code for all of these steps, but the two interpolation steps (2 and 4) contain constructs which were not vectorizable previously. We present a new algorithm that exploits new vector hardware features of the Cray-2 and the Cray-XMP 48 so that these steps are now vectorizable with a sizable reduction in CPU time required. For simplicity, we describe a 2-dimensional algorithm, but this discussion applies to three dimensions as well. In fact, the timing analysis was done using a 3-dimensional version of the following

* Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

0021-9991/87 $3.00

algorithm. It should be noted that timing results were obtained using the Cray-2 "BMA" at the National Magnetic Fusion Energy Computer Center (NMFECC) using the CIVIC compiler.

II. Field Interpolation

To find the fields at the position of a particle we must interpolate the fields from the grid points surrounding the particle. Thus, for linear weighting, the x component of the magnetic field at the particle in Fig. 1 would be

$$\begin{aligned}\mathrm{BPX(N)} = {} & \mathrm{A1} * \mathrm{BGX(I, J)} + \mathrm{A2} * \mathrm{BGX(I+1, J)} \\ & + \mathrm{A3} * \mathrm{BGX(I, J+1)} + \mathrm{A4} * \mathrm{BGX(I+1, J+1)},\end{aligned}$$

where N is the particle index, BPX is the field at the particle and BGX is the field on the grid. To code this algorithm, we would try something like:

```
      DO 10 N = 1, NMAX
            A1 = (XG(I(N) + 1) - XP(N)) * (YG(J(N) + 1) - YP(N))/A
            A2 = ···
            A3 = ···
            A4 = ···
            BPX(N) = A1 * BGX(I(N), J(N)) + A2 * BGX(I(N) + 1, J(N))
                   + A3 * BGX(I(N), J(N) + 1) + A4 * BGX(I(N) + 1, J(N) + 1)
            BPY(N) = A1 * BGY(I(N), J(N)) + ···
            BPZ(N) = ···
            EPX(N) = ···
            EPY(N) = ···
            EPZ(N) = ···
   10 CONTINUE
```

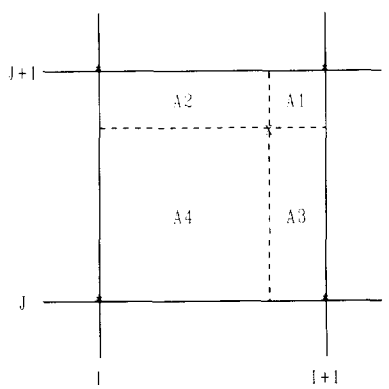

FIG. 1. Linear weighting in two dimensions for a uniform grid.

where XP(N) is the x coordinate position of particle N, XG(I) is the x coordinate position of grid point I and A is the area of the grid cell. The important point to notice about such a loop is that the indexes of many of the arrays are arrays themselves, e.g., BGX(I(N), J(N)). A construct where such arrays appear on the right side of an assignment statement is called *gathering*; on the left side of an assignment statement, it is called *scattering*. Fortunately, the Cray-2 multiprocessor has hardware instructions to vectorize these constructs. This particular loop will run a factor of 6 faster when it is vectorized than when it is not [2]. The actual CPU time required to do this loop per particle per time step is reduced from 16 to 3 μs on the Cray-2.

III. Charge and Current Deposition

To find the charge and current densities on the grid, we reverse the above procedure and obtain similar coding, except for some surprises. A first attempt at the deposition might look like:

```
      DO 20 N = 1, NMAX
         A1 = (XG(I(N) + 1) - XP(N)) * (YG(J(N) + 1) - YP(N))/A
         A2 = ···
         A3 = ···
         A4 = ···
         D(I(N), J(N)) = D(I(N), J(N)) + A1 * Q/A
         D(I(N) + 1, J(N)) = D(I(N) + 1, J(N)) + A2 * Q/A
                 ⋮
   20    CONTINUE
```

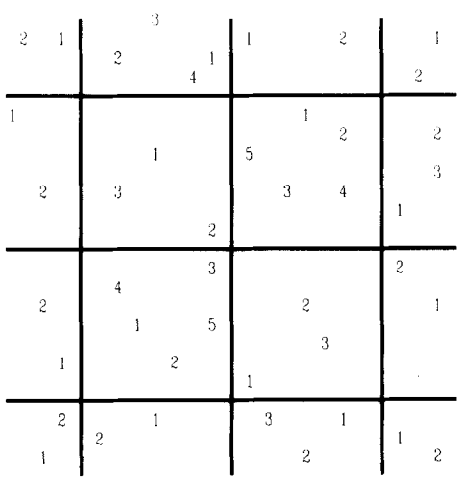

FIG. 2. The numbers represent the particle positions and the groups to which the particles belong.

where D(I, J) is the charge density at grid point I, J. This loop represents the contribution of a particle to the charge density of the four grid points surrounding it. Note that D appears on both sides of the equation since we are computing the running sum of contributions over all particles that are in any given cell.

The problem with vectorizing such a loop is that several particles can contribute to the density of a particular grid point, and thus an element of the array D(I, J) will be assigned several values as we sweep through the particles. The compiler will note this point and will not vectorize the above loop.

To circumvent this problem, we can sort the particles into groups such that within a group, no two particles occupy the same cell (see Fig. 2). The sorting algorithm presented here requires a one-pass sweep through the particles in a non-vectorized loop. The algorithm is

```
      DO 30 N = 1, NMAX
            NPIC(IC(N)) = NPIC(IC(N)) + 1
            IGN = NPIC(IC(N))
            NIG(IGN) = NIG(IGN) + 1
            IP(NIG(IGN), IGN) = N
30    CONTINUE
```

In words, the steps are;

(1) We are given IC(N), which is the cell number occupied by particle N. A cell is numbered by its lower left grid point. The grid point (I(N), J(N)) can be related to the 1-dimensional quantity IC(N) by IC(N) = I(N) + IMAX * (J(N) − 1). With IC(N) we can find the number of particles in this cell, NPIC(IC(N)), by keeping a running sum.

(2) Once we know how many particles are in cell IC(N), we also know the group number of particle N. We label this quantity IGN. This step is not necessary since IGN = NPIC(IC(N)) but it is notationally convenient.

(3) We can now determine how many particles are in group IGN, NIG(IGN), by keeping another running sum.

(4) Finally, the index of the particle that occupies the NIG(IGN)th position in the IGNth group, IP(NIG(IGN), IGN), is simply *N*. Notice that we do not shift particle indexes in the sorting loop; we simply generate an ordered list in the array IP. To use this information, we compute the charge deposition by groups:

```
      DO 50 IGN = 1, IGNMAX
CDIR$ IVDEP
            DO 40 M = 1, NIG(IGN)
                  N = IP(M, IGN)
                  A1 = ···
                  A2 = ···
                  ⋮
```

```
                D1(I(N), J(N)) = D1(I(N), J(N)) + Q * A1
                D2(I(N) + 1, J(N)) = D2(I(N) + 1, J(N)) + Q * A2
                :
40          CONTINUE
50      CONTINUE
```

The inner loop above is very similar to the original loop. CDIR$ IVDEP is a compiler directive that tells the compiler to ignore the vector dependence and to vectorize the inner loop. Note that we have introduced the quantities D1 D2... in place of D. These are necessary to take care of a vector dependence not removed by the sorting algorithm, namely, the dependence caused by two adjacent cells which have common grid points. The contribution of particles in each of these cells to the common grid points will cause a vector hazard. By separating the contribution of a particle to its lower left grid from its lower right grid point, etc., we eliminate this conflict. To get the total charge density on the grid, we must sum all of these contributions, i.e.,

```
        DO 60 IG = 1, IGRIDMAX
            D(IG) = D1(IG) + D2(IG) + D3(IG) + D4(IG)
60      CONTINUE
```

This entire procedure seems like a lot of trouble but it is well worth the effort. Timing runs on a 3-dimensional version of this routine using 3000 to 30,000 particles indicate that there is a gain of more than a factor of 3 in speed-up, including the sorting of the particles. The sorting requires about 20% of the time while the deposition loop requires about 65% with an additional 15% for initializing arrays [2]. The total CPU time required for the deposition routine to process one particle one time step is reduced from 13.5 to 4.5 μs.

IV. Modifications to the Sorting Loop

The four-line sorting loop described in Section III (loop 30), requires the array IP. This array has dimensions IGRIDMAX * IGNMAX, where IGRIDMAX is the total number of grid cells and also represents the maximum number of particles in any group, and IGNMAX is the maximum number of groups. One could expect that some cells might have many more particles than other cells. It would not be wise to make IP large enough to hold as many groups as the maximum number of particles in any cell, as this would lead to a large amount of wasted storage. A better idea would be to select some number, larger than the average number of particles per cell, as the maximum number of groups allowed. The index of any particle that would have a higher group number than the maximum allowed would be placed into another array. All such particles would be processed in a scalar loop. To do this, the sorting loop would be modified as follows:

```
      DO 70 N = 1, NMAX
         IF (NPIC(IC(N)).LT.IGNMAX) THEN
            NPIC(IC(N)) = NPIC(IC(N)) + 1
            IGN = NPIC(IC(N))
            NIG(IGN) = NIG(IGN) + 1
            IP(NIG(IGN), IGN) = N
         ELSE
            NSP = NSP + 1
            ISP(NSP) = N
         ENDIF
70    CONTINUE
```

where NSP keeps track of the number of scalar particles and ISP is the index of the NSPth particle.

The storage required for this sorting method is $(2^D * F + IGNMAX + 1) *$ IGRIDMAX. D is the dimension of the system and F is the number of field quantities you want to collect (1 for the density and 3 for the current densities, for example). Thus $2^D * F$ represents the extra arrays required to eliminate the vector conflicts caused by particles in adjacent cells contributing to the same grid point. The 1 is for the NPIC array. The other arrays are of insignificant size. By using scalar particles, we can reduce IGNMAX and save storage.

We could further modify the sorting algorithm to completely eliminate excess storage and allow for an arbitrary number of groups. The basic idea is to stack the groups contiguously into the IP array and to keep a pointer array to tell us where the groups begin in the IP array. The cost will be more computations, but this scheme will work better in situations of highly non-uniform densities. The scheme is:

```
      DO 80 N = 1, NMAX
         NPIC(IC(N)) = NPIC(IC(N)) + 1
         IGN = NPIC(IC(N))
         IGRP(N) = IGN
         NIG(IGN) = NIG(IGN) + 1
80    CONTINUE
      IGNMAX = 0
      DO 90 IG = 1, IGRIDMAX
         IGNMAX = MAX(NPIC(IG), IGNMAX)
90    CONTINUE
      IPOINT(1) = 1
      IPINDEX(1) = 1
      DO 100 IGN = 2, IGNMAX
         IPOINT(IGN) = IPOINT(IGN - 1) + NIG(IGN - 1)
         IPINDEX(IGN) = IPOINT(IGN)
```

```
100     CONTINUE
        DO 110 N = 1, NMAX
              IP(IPINDEX(IGRP(N))) = N
              IPINDEX(IGRP(N)) = IPINDEX(IGRP(N)) + 1
110     CONTINUE
```

In words, the steps are:

LOOP 80:

(1) Find the number of particles in cell IC(N) as before.

(2) For convenience, define the temporary IGN as before.

(3) Find the number of particles in group IGN as before.

(4) Save the group number of particle N in IGRP(N).

LOOP 90:

Calculate the size of the largest group.

LOOP 100:

(1) Set the pointer, IPOINT(IGN) to the beginning of each group stacked in IP.

(2) Initialize the index of the positions of the particles in IP. See loop 110 below.

LOOP 110:

(1) Place the index of particle N into IP.

(2) Increment the index of IP for the group corresponding to particle N.

Now the deposition loop is:

```
          DO 130 IGN = 1, IGNMAX
CDIR$ IVDEP
                DO 120 M = IPOINT(IGN), IPOINT(IGN) + NIG(IGN) - 1
                      N = IP(M)
                      A1 = ···
                      A2 = ···
                      :
                      D1(I(N), J(N)) = D1(I(N), J(N)) + Q * A1/A
                      D2(I(N) + 1, J(N)) = D2(I(N) + 1, J(N)) + Q * A2/A
                      :
120             CONTINUE
130     CONTINUE
```

The extra storage required for this scheme is $2 * NMAX + (2^D * F + 1) *$ IGRIDMAX. Thus the difference between this scheme and the previous one is $2 * NMAX - IGNMAX * IGRIDMAX$. If we let IGNMAX be some factor mul-

tiplied by the average number of particles per cell, i.e., IGNMAX = IFC * NMAX/IGRIDMAX, then the difference simplifies to (2 − IFC) * NMAX. Thus if we can set IFC to something less than 2, then the first scheme will be more space efficient. If, however, we had a situation in which several cells had a very high density while other cells were depleted, then this last method would be more appropriate. The CPU time required to sort one particle one time step is 0.9 μs for the first scheme and 1.4 μs for the second.

V. Comparison to Another Scheme

Nishiguchi *et al.* present a scheme which also vectorizes the particle deposition loop yet requires no particle sorting [3]. Direct comparison of their method to the ones presented here in regards to CPU time required is not possible since they modelled a 1-dimensional system and used a different machine. However, we can compare the storage requirements and the number of calculations required.

As stated in their article, their method requires LVEC * IGRIDMAX * F additional storage, where LVEC is the vector length of the inner deposition loop. If we consider a sizable test case with 10^6 particles, 10^5 grid cells, 3 dimensions and 4 particle quantities to interpolate to the grid, then we will have, on the average, 10 particles per cell and thus we will let IGNMAX = 15. The storage requirement for our two modifications of the sorting algorithm would be 4.8 and 5.3 million words, respectively. If we use LVEC somewhere between 32 and 64, as indicated by Nishiguchi for a case with about 10 particles per cell [3], we find that the storage required for this algorithm would be between 12.8 and 25.6 million words. In either case, one might be able to use some of this memory in another part of the code, but our experience to date with the development of a particle code [2] reveals that only about 2.5 million words could be easily shared with another part of the code.

Besides comparing memory requirements, we could compare the number of executed statements in each algorithm. Our deposition loops are almost identical, and would be done the same number of times. In place of a predeposition sort, Nishiguchi has a postdeposition "recollection" of the particle quantities. This recollection is vectorized but requires F * LVEC * IGRIDMAX statement executions. For the test case, this would be anywhere from 12.8 to 25.6 million operations. Our first algorithm has only three essential steps and thus requires 3 million operations in scalar mode. If we say that vector statements will be executed 10 times faster than scalar statements of comparable complexity (as a rough estimate), then we find that Nishiguchi's scheme is faster, but only by a factor of 1.17 to 2.34. These factors are doubled if their scheme is compared to our second scheme.

Thus one must consider the resources available as well as the problem at hand to determine which scheme is appropriate. For very large systems, our schemes are a bit slower but require much less storage.

VI. Multitasking

Our particle code [2] is being developed to run on the Cray-2 at the NMFECC. As such, it is appropriate to mention how the interpolation routines of this code can take advantage of the multitasking capabilities of this machine in addition to its ability to vectorize gather-scatter constructs.

To multitask the interpolation of the fields from the grid to the particles, we simply break up the interpolation loop (loop 10) into several subloops and run the subloops simultaneously. We can do this because each iteration of loop 10 assigns field quantities to the position of a unique particle, and this assignment does not depend on the other particles.

To multitask the deposition loops (either loops 40 and 50, or 120 and 130), we must be careful not to break up the outer loop. Each iteration of the outer loop depends on the values that were assigned to the arrays from the previous iteration. However, we are free to break up the inner loop since each iteration assigns values to arrays at unique grid cells, as guaranteed by the sorting algorithm.

Multitasking does not reduce the CPU time charge, but it can reduce the wall clock time and the memory charge, both of which could be substantial for a large code.

VII. Summary

We now have machines that can vectorize gather-scatter constructs with about a factor of 6 reduction in CPU time over scalar coding. This feature is very important in particle-in-cell codes which contain gather-scatter constructs in the interpolation routines.

To take advantages of this vectorization in the charge and current deposition routine, we must sort the particles to eliminate vector conflicts. Although the sorting is a scalar operation, the computer time it requires is small enough to make the sorting worthwhile.

Others have suggested a deposition algorithm that is vectorizable and does not require a particle sort. Even though it is completely vectorized, the total CPU time required for it is not much less than for our scheme, and the memory requirements are much greater for a sizable case.

As well as being vectorizable, the interpolation routines of particle-in-cell codes lend themselves readily to multitasking.

Acknowledgments

I wish to acknowledge the continuing support of my supervisors, Dan Shumaker and David Anderson, as well as that of my advisor, John Killeen. Special thanks are due Larry Berdahl, whose knowledge of the compiler features and the hardware configuration of the Cray-2, as well as his interest and enthusiasm, was indispensible to the success of this work.

REFERENCES

1. C. K. BIRDSALL AND A. B. LANGDON, "Plasma Physics via Computer Simulation" (McGraw–Hill, New York, 1985).
2. E. J. HOROWITZ, D. V. ANDERSON, AND D. E. SHUMAKER, *Bull. Am. Phys. Soc.* **30** (1985).
3. A. NISHIGUCHI, S. ORII, AND T. YABE, *J. Comput. Phys.* **61**, 519 (1985).