Conf-9010439--1

UCRL-JC--104063

DE92 003016

OBTAINING GIGAFLOP PERFORMANCE FROM PARTICLE SIMULATION OF PLASMAS

David V. Anderson, Bruce C. Curtis, Dana E. Shumaker
University of California
Lawrence Livermore National Laboratory
Livermore, CA

This paper was prepared for the
Proceedings of the Fifth International Symposium on
Science & Engineering on Cray Researrch Computers
October 22-24, 1990 London, England

June 1990

Lawrence Livermore National Laboratory

DISCLAIMER

OBTAINING GIGAFLOP PERFORMANCE FROM PARTICLE SIMULATION OF PLASMAS

David V. Anderson, Bruce C. Curtis, Dana E. Shumaker

National Energy Research Supercomputer Center
Lawrence Livermore National Laboratory
Livermore, California 94550 USA

Eric J. Horowitz

Computer Sciences Corporation, Baltimore Maryland USA

ABSTRACT

In the numerical simulation of plasma phenomena there are two fundamental approaches that are generally followed. In the continuum approach one models the evolution of the fluid moment equations derived from the appropriate Boltzmann equation of the plasma. Alternatively, in the particle approach a large group of simulated charged particles are moved according to the self-consistent electromagnetic fields which partly depend on the charge and current densities of these same particles. Although the particle simulation method has been traditionally the more expensive of the two, it is much more capable of giving adequate account of many important kinetic phenomena. With the advent of vector multiprocessor supercomputers, such as the Cray-2 or Cray Y-MP, we have learned to adapt particle simulation codes to exploit the parallel features of these machines. Yet, in spite of such developments, the particle simulation codes have remained much slower than the maximum machine speeds. We have investigated new techniques that further optimize these methods to bring the speeds of these particle simulations into the gigaflop range. Recent progress in this area suggests that the use of particle simulation methods will become competitive with the alternative fluid models especially when it is realized that gigaflop performance makes them much more affordable.

INTRODUCTION

Perhaps the simplest and most general methodology for simulating the behavior of plasmas is to follow the charged particles in their self-consistent electromagnetic fields.[1,2,3] To do this, one solves the coupled Newton-Maxwell equations for the modeled problem. Though straightforward, this approach is very intensive computationally both in regard to storage and time requirements. Thus, historically, most models of plasmas have employed a fluid representation inorder to make the calculations affordable. These particle simulation models are employed in the studies of astrophysics, particle accelerators, solid state physics, molecular dynamics, as well as plasma physics. In this paper we try to show how these particle codes can be made to run more efficiently on some contemporary shared memory supercomputers, to make them competitive with fluid codes.

A typical particle code is executed in four phases:

1. Interpolate the electromagnetic fields from the grid to the particle positions (*field interpolation*)
2. Using these fields, push particles by advancing positions and velocities according to Newton's law (*push*)
3. Interpolate from particles to deposit charge and current densities on the grid (*deposition interpolation*)
4. Solve the Maxwell field equations on the grid (*solve*)

In many particle codes the majority of the time is spent in the first three phases. Even when the solution of the field equations is most time consuming, we will assume that this phase can be straighforwardly optimized by standard techniques. In this paper we wish to emphasize the methods for the optimization of the first three phases, because we believe it holds the key to obtaining performance in the gigaflop range.

Vector Parallel Algorithms in QN3D

Prior to 1985 most particle codes were only partially vectorized, partly because the machines available had limited vector hardware and partly because the algorithms were not well developed. A notable exception to this was the work of Buneman et al[4] who wrote highly optimized particle codes in Cal assembly language for the Cray-1 computer. This situation began to change with the introduction of vector multiprocessor machines such as the Cray X-MP, Cray-2, Cray Y-MP, as well as some of Japanese manufacture. In the past few years, nearly all of the restrictions on vectorization and multitasking have been removed. The hybrid particle code QN3D, with fluid electrons and particle ions, pioneered several of these developments[5] and was fully parallelized. "Parallelized" denotes the simultaneous use of both vectorization and multitasking.

The first phase of the time step, as indicated in the list above, requires the interpolation of the fields, known on the mesh, to the particle positions. Since the particles, stored in so-called particle tables, tend to become randomly distributed in space, it follows that the fields used to move the particles must be accessed randomly from the computer memory. In terms of the written algorithm this is expressed in terms of an indirect index structure, such as *EX(I(N))*. This structure, which frustrated vectorization on the earlier vector computers, may now be indirectly vectorized on many of the newer vector computers by operations known as the "gather" or the "scatter." The gather mechanism is used to load memory items into the CPU while the related scatter operation is used to store into the memory from the CPU. QN3D employed the gather procedure to allow vectorization and used multitasking to carry out the *field interpolation* phase. It is crucial to note that this form of indirect vectorization is considerably slower, on the Cray-2, than normal direct vectorization of constant stride arrays; at best the gather operation can run at one fourth the normal vector speed. Its speed is usually worse than that because random access of memory tends to be slowed by memory bank and quadrant conflicts. But, in most cases, it is still faster than scalar coding.

During the next phase, the particle push, the orbits are all independent and thus trivially parallelized.

The third phase is a more complicated interpolation than used in the first phase and requires both a gather and a scatter operation to perform the deposition of charge and current density contributions from the various particles. There is a possible recursion in this step because different particles may try to increment the same mesh location simultaneously; consequently this aspect prevented vectorization and multitasking. One remedy to this problem, employed in QN3D[6], was to break the deposition phase into several sub-phases each of which accumulated the densities from a corresponding subgroup of particles. By choosing each subgroup such that no two particles occupied the same mesh cell it then became impossible for conflicts to occur. QN3D used both vectorization and multitasking within each sub-phase to parallelize the *deposition interpolation.*

In the last phase of QN3D we solve an approximate form of Maxwell's equations. Since we are trying to limit the scope of this paper to the issues of speeding up the other three phases of the calculation, we shall not discuss the field solver here.

Other workers have also been exploring similar ideas for optimizing the speeds of particle codes. For example, Heron and Adam[7] have developed a deposition algorithm which is organized such that most of the loops employ direct vectorization, thereby reducing some of the overhead of gather-scatter forms of vectorization.

Optimizing the Interpolation Phases in STORM

QN3D, though fully vectorized and multitasked was about 40 times slower than the peak speeds of a Cray-2. The *deposition interpolation* phase was the slowest of the four phases. Thus, in 1989, a new version of QN3D was developed that employed a different deposition technique in hopes of optimizing the *deposition interpolation* phase. It also had a new field solver and added a hybrid fluid component for one species of ions in the model. The code was renamed STORM, as it was quite different from its predecessor. We have built a special version of it which we have dubbed SQAL for the purpose of conducting studies into optimization in the Cray-2 environment.

Organization of this Paper

We begin by developing alternative techniques that allow a vector-parallel treatment and at the same time avoid the slower gather- scatter operations entirely. Some indirect indexing is still there, but it is in the outer loops.

Then we discuss a particle ordering scheme and show how it might allow us to get higher performance in our present computing environment and how it might extend into parallel computing environments.

The structure of this paper is as follows: We begin in Section 2 with a discussion of strategies for high performance and then in Section 3 we give concrete examples of implementing some of these ideas in the SQAL code. We propose other techniques to be tested in Section 4. In Section 5 we conclude with a discussion about how we expect to achieve gigaflop performance in our present computing environment.

STRATEGIES FOR HIGH PERFORMANCE

As stated above, the major obstacle to good performance in a particle code is the tendency for the particles to be randomly located in space which leads to the use of indirect indexing and which can lead to slowly executing code. That is, the CPU's idle most of the time while they wait for the memory accesses to complete. This is a critical issue for the Cray-2 and less important for the Cray Y-MP. And it may be important for the Cray-3. If the speed of accessing memory can be increased, there is the prospect of keeping the CPU's busy. Until we do that, it makes little sense to optimize the arithmetical aspects.

Our first objective in making these codes more efficient is to eliminate the use of gather-scatter vectorization by replacing it with direct vectorization (in which the stride is uniform.) We then develop efficient methods for accessing memory. Later, we shall consider a scheme that orders the particles by position.

SQAL EVOLUTION

A test code, dubbed SQAL, was built from the STORM code to allow us to test the various optimization ideas. The first version of SQAL stored the field arrays and deposition arrays such that each field component was stored in a separate 3D storage block. They were stored with what is known as natural ordering such that incrementing the index in the x direction corresponds to unit stride access of the memory.

Field Interpolations with Interleaving

For linear interpolations to a particle position, in the *field interpolation* phase, we must access the fields at the 8 surrounding grid vertices. These can be loaded from memory as 4 short vectors each of length 2. Very little improvement in speed can be obtained from the use of such short vectors.

We can improve the situation considerably by processing the particles 64 at a time and by using an interleaved storage scheme. What we do is store the fields such that their ordering in memory is

EX(N),EY(N),EZ(N),BX(N),BY(N),BZ(N),EX(N+1),..

such that twelve grid quantities are stored contiguously as we move from one grid vertex to the next one in the x coordinate. Here, N is taken to represent the grid offset index, which essentially labels the grid points using "normal" ordering (the x index turning fastest.) When we load this vector, of length 12, from memory we can potentially achieve good performance in direct vectorization mode which should outperform the gather-scatter modes used earlier. These field variables are then stored in a temporary field array FAVL(LCT,IVX,NL) where NL ranges over 64 particles per pass, IVX ranges over the 8 vertices of the enclosed grid cell, and LCT ranges over the 6 different field components.

Even with the various optimizations done in Fortran to get good performance, we found that the performance of the *field interpolation* was only about 23% faster than the original STORM This relates to the fact that our vectors are relatively short making the loop overhead rather expensive. Table 1. shows the various measured access rates compared to the theoretical maximum rates. The column labelled SQAL06 gives the speeds achieved using Fortran coding for the loading of the field quantities; later on we shall discuss the version SQAL08 that uses assembly coding and local memory to perform the loading. In Fortran our actual access rate of roughly 45 megawords per second is about 21 percent of the theoretical maximum rate of almost one access every clock period. The speeds of these versions of SQAL are shown in Table 2. If we assume that the access speed labelled IDEAL could be achieved, then the *Field Interpolation* speed of 59 MFLOPS would increase to about 106 MFLOPS. Even then, we do not have enough speed on a single processor to infer gigaflop speeds on an 8 processor multitasking machine. We must work harder to further optimize the code.

Memory Access Rates in Megawords/sec				
Code Segment	STORM	SQAL06	SQAL08	IDEAL
Field Interpolation Loop 12	≈ 33.2	44.6	75.8	217
Deposition Interpolation Loop 40	15.5	27.4	NA	217

Table 1: Measured access rates for STORM and two versions of SQAL are compared to the theoretical (ideal) rates for a Cray-2. In the *field interpolation* Loop 12 we read the fields on the grid into temporary arrays which are subsequently used in the interpolation of field quantities to the particle positions. SQAL06 uses four-way unrolled Fortran for accessing memory. SQAL08 improves on this by employing local memory for the grid indices and Cal-2 assembler to enhance the loading operations. For the *Deposition Interpolation*, Loop 40 does both loads and stores as it accumulates the charge and current densities.

Deposition Interpolations with Interleaving

The STORM code used a vectorized method for depositing charge and current densities to the grid from the particle quantities. Since each particle contributes to eight grid vertices and since there are four fields $(\rho, \mathbf{J_x}, \mathbf{J_y}, \mathbf{J_z})$, there are 32 quantities to be generated. This was accomplished by constructing 32 contiguous arrays, each spanning the entire grid. A gather-scatter loop was used to fill these arrays with the grid contributions. Since only one particle was processed at a time, there was no possibility for a conflict. But there was also no easy way to generalize this method for parallel computation. After all the 32 grid arrays were filled, the actual grid arrays of $(\rho, \mathbf{J_x}, \mathbf{J_y}, \mathbf{J_z})$ were constructed by summing these partial arrays.

Somewhat in analogy to the treament of the *field interpolation*, we modiifed the code to process the particles in groups of 64. That made it possible to generate all of the interpolation weights by direct vector mode. The 32 temporary arrays were interleaved into a larger array in which the the deposition could proceed into eight contiguous elements at a time. These 8 are the four quantities $(\rho, \mathbf{J_x}, \mathbf{J_y}, \mathbf{J_z})$ at some grid point followed by the same four quantities at the adjacent grid point in the x direction. These "interleaved" temporary arrays were then filled in direct vector mode. As before, the four primary arrays were constructed by summing over these temporary arrays.

Particle Code Performance In Unitasking Vector Mode						
Code Segment	STORM μsec	STORM MFLOPS	SQAL06 μsec	SQAL06 MFLOPS	SQAL08 μsec	SQAL08 MFLOPS
Field Interpolation and Push	4.8	48	3.9	59	3.0	76
Deposition Interpolation	6.9	14	3.8	26	3.8	26
Boundary Treatment	10.2	.5	3.4	1.5	3.4	1.5

Table 2: Speed measurements are given for the three essential regions of the particle codes STORM, SQAL06, and SQAL08. The fourth region, the *solve*, is not treated here. We give the performance in terms of the popular yardstick of micro-seconds per particle per timestep as well as in terms of the measured speed in mega-flops. The boundary treatment is mostly loads, stores and logic and with relatively little floating point arithmetic it is difficult to achieve a high megaflop rate.

In Table 2., in the row labelled *Deposition Interpolation*, we see that timings for this phase improved significantly, up 86% from the results of STORM. The improved speed of 26 MFLOPS is still far too slow to give the code performance anywhere near a gigaflop, even if we were able to multitask it at level 8. We are a factor of at least 4 too slow. Again, as in the revised *field interpolation* phase, we seem to be limited by the access rate to the common memory. Referring to Table 1. we see that much of the improved performance of the *Deposition Interpolation* may be attributed to the significantly improved memory access. Even so, it falls far short of the theoretical asymptotic rate, at about the 13 percent level. If we were to assume that the memory access rate was that given under "IDEAL" in Table 2, then the speed of this phase would increase to 107 MFLOPS.

It is clear that further modifications must be sought to increase these speeds. On the Cray-2 we can try to use local memory, we can optimize with assembly code, or we can seek longer vectors. One can also consider changing the entire code and data structure to minimize accesses to common memory. Lastly, there is the option to use a faster computer, such as a Cray Y-MP.

Using Assembler Code and Local Memory

In measuring the time spent in the different sections of the *field interpolation* routine (in SQAL05) we found that roughly 40% of the time is spent in the performance of the "arithmetical" loops and 60% of the time is used in reading the fields from common memory. In terms of microseconds per particle per time step we were using 2.26 to load the fields as compared to .98 for the interpolation and .66 for the push. It became apparent, that to make further progress we must optimize the loading of the fields.

Our first approach was to use Fortran optimization techniques. We unrolled the loop accessing the memory such that it processed four particles per pass. This version, SQAL06, gave a modest speed-up of this loop of approximately 9%. To better understand these slow speeds, we turned to the assembly code. We wanted to analyse the assembly code generated by the compiler to learn if it was optimal and if not recode the loop either with better Fortran or in Cal-2 assembly code.

The code generated by the compiler for the inner loop was quite good, but we discovered a memory bottleneck in the outer loop. A substantial amount of time was spent loading the locations of the grid cells in scalar mode. To remove the bottleneck we loaded the grid cell locations into local memory in vector mode in a separate loop. This netted an additional 41% speed-up. Although this was done in Fortran, it was based on our reading of the assembly code generated by the compiler.

In the inner loop, the limiting factor was the short vector length. Since the Cray-2 supports a memory bandwidth of [vector length] words in [vector length+8] clock periods, and the field interpolation loop used a vector length of 12, we were achieving only slightly better than one word every two clocks. We improved on that rate by loading four of the length 12 vectors into consecutive local memory locations, so the subsequent store into common memory could be done with a vector length four times greater. This modification, coded in assembly language, produced a maximum rate of nearly two words moved every three clocks and resulted in another 20% speed-up beyond our best Fortran version. Thus overall we have gained a speedup of 69% in the memory access rate. This can be seen in Table 1 by comparing the versions SQAL06 with SQAL08 in the row labeled *Field Interpolation Loop 12*. And compared to STORM, we have more than doubled the speed of this code segment.

We were working on a similar modification for the memory access of the *deposition interpolation*, as this paper was going to press. Thus the notation "NA" in Table 1.

THEORETICAL ESTIMATES

In the foregoing we presented results from our test code SQAL that showed that the use of interleaving, local memory, direct vectorization, and even assembler coded memory accesses substantially improved the speed of the interpolation phases. Yet this was insufficient to attain the speed on a single Cray-2 processor above 100 megaflops. In what follows we make estimates of speedups that might be obtained with further code modifications or by using a Cray Y-MP.

Critical Path Analysis for Cray-2 and Cray Y-MP

By knowing something about the performance of various instructions on the computer, we can make a best case estimate of how fast different code segments might run. We have made some simplifying assumptions such as:

1. Perfect overlap of adds and multiplies
2. One Direct Vector Memory Access per Clock Period
3. One Gather Access Every 4 Clock Periods (Cray-2)
4. One Path to Memory (Cray-2)
5. One Gather Access Every Clock Period (Cray Y-MP)
6. Three Paths to Memory (Cray Y-MP)
7. One Scatter Access per Clock Period
8. Ignore Loop Overhead
9. Ignore Startup and Memory Bank Conflicts

We have done such an analysis for the *field interpolation*, the *push*, and the *deposition interpolation*. Also, since the memory performance seems to be a critical issue for this particle code, we have also made estimates for the Cray Y-MP which has a slower clock cycle but a much faster memory in comparison to a Cray-2. Table 3. shows the limiting performance that could be obtained from STORM or SQAL under these assumptions.

Ordered Particle Scheme

It should be evident from the foregoing that the methods suggested for optimizing SQAL may not be adequate to bring it to a performance level near a gigaflop. Even if we assume that we could multitask the code at the highest level, neither the Cray-2 (with 4 CPU's) nor the Cray Y-MP (with 8 CPU's) will reach a gigaflop. However, the Y-MP will operate at a larger fraction of a gigaflop than the Cray-2. For this reason, we will now restrict our discussion to the Cray Y-MP.

Theoretical Asymptotic Best Case Speeds (MFLOPS)				
Code Segment	STORM (Cray -2)	SQAL (Cray-2)	STORM (Cray Y-MP)	SQAL (Cray Y-MP)
Field Interpolation and Push	180 (1.16)	215 (1.06)	286 (.74)	271 (.53)
Deposition Interpolation	133 (1.03)	191 (.83)	259 (.53)	207 (.76)
Combined Phases	157 (2.19)	204 (1.89)	277 (1.27)	241 (1.60)

Table 3: Speed estimates are given for the Cray-2 and for the Cray Y-MP under extremely optimistic assumptions. The numbers in parenthesis give the execution times in micro-seconds per particle. It is evident that the Cray Y-MP outperforms the Cray-2 inspite of its slower clock cycle. When combined with the multitasking potential of these machines, the Y-MP would appear to be the machine to use to obtain performance in the gigaflop range.

Also there is a serious problem when we consider how a code such as SQAL could be multitasked. The *field interpolation*, the *push*, and the *solve* are trivially multitasked because there is no data conflict. Unfortunately, there is a conflict in the *deposition interpolation* that must be resolved inorder to multitask that phase of the calculation. The problem is that two different processors may try to update the same grid points at the same time.

One approach to this problem recognizes that the errors generated by forcing the *deposition interpolation* to multitask are relatively small and therefore acceptable. Such an "asynchronous" version would generate irreproducible results which is less than satisfactory even if the errors are small.

The other approach uses a particle sorting scheme to eliminate any possible conflicts of multitasking. The idea here is to order the particles such they are grouped together according to the grid cell they occupy. When grouped this way, great economies result because the number of memory accesses required goes down dramatically. It also becomes advantageous to store portions of fields in local memory when they are reused many times. In this scheme they would be accessed as many times as there are particles in the cell being processed. We shall describe an algorithm that rebuilds the particle tables after each push in such a way that less than $4n$ operations are required to ensure that the n particles are properly sorted into their grid cells.

SQAL and STORM both carry the particle variables ip, jp, and kp in the particle tables; these indices specify the coordinates of the grid cell containing the particle. It is convenient to also carry the offset index

$$ic(n) = ip(n) + im * (jp(n) - 1) + im * jm * (kp(n) - 1)$$

where im and jm are the grid dimesnions in the x and y coordinates. The quantities *ip*, *jp*, *kp* and *ic* are computed as the last part of the *push* phase. By keeping the old value of *ic* as *icold* we can test (*ic*−*icold*) to immeadiately determine which particles have been pushed out of their former cells and which ones have been retained in them. Just after the push we reconstruct the particle tables to keep them ordered with respect to the grid cells. This can be done as follows:

1. Construct a transit table of particles departing cells
2. Sort the transit table ordering particles by cells
3. Construct transit pointers to old particle table
4. Construct retained pointers to old particle table
5. Allocate a new particle table
6. Process the grid cells in ascending order and for each cell:
   - Move the particle data of retained particles to new table
   - Move the particle data of departing particles to new table

As we envisage this method, we will not move the particle data until we have determined the mapping of the old "serial" numbers into the new ones. For *nm* particles, constructing the transit table will require *nm* integer adds as well as a few times *nm* logical operations. Of the *nm* particles only a fraction (perhaps 10%) will be departing particles. Say there are l of these. The sort will require on the order of $l \ln l$ operations of the integer arithmetic and logical varieties. There will be many fewer transit and retained pointers than particles, perhaps on the order of 20% of the number of particles. Once the pointers are set, moving the particle data requires $2 * nq * nm$ memory accesses. Here, *nq* is the number of attributes per particle- envisaged as 12 at least.

A great deal is gained by using these sorted particles. Instead of accessing each field grid quantity 8 times for every particle, this work can be reduced to 4 times for every cell. If there are 10 particles per cell, this implies a 20 fold reduction in memory accesses in the *field interpolation* phase. A similar reduction occurs in the *deposition interpolation* because the accumulating grid quantities are kept in vector registers until all particles in a cell have contributed. Not only does the time spent doing memory accesses decrease, but some of the arithmetic can be further optimized by keeping frequently reused field quantities in the local memory or registers.

It is the cost of the sorting and the mapping from the old particle table to the new one that partly offsets these gains. These costs are minimized by sorting only the departing particles and by the fact that the particle table data is not moved until pointers to the new particle table are determined.

Autotasking Implementation Issues

We mentioned above that the SQAL code could not be multitasked unless we were willing to use the so-called asynchronous mode. A code employing the just described sort procedure, however, can be easily multitasked by using autotasking. As in the earlier discussion, the *deposition interpolation* is the only phase that is not trivial to multitask. Here we partition the physical domain into as many subdomains as we have processors. Each sub-domain will have its own sub-table of particles. The only difficulty is that some particles cross the subdomain boundaries in the push. When this occurs, these particles and their attributes must be moved into other sub-tables in such a way that the ordering is preserved. This can be done by isolating from the table of departing particles those that have crossed into other subdomains. The pointers of these isolated particles are then passed to the tasks relevant to their new sub-domains. The number of such particles is quite small, down by a "surface to volume" ratio as compared to the already modest number of departing particles. Implicit in what we have said here is the fact that the sorting procedure itself is multitaskable since both the particle tables and the grid subdomains are partitioned among the several processors.

RESULTS AND CONCLUSIONS

We have obtained a signifcant improvement in the performance of the STORM code by making several modifications as evidenced in the sequence of SQAL codes. The code was restructured by interleaving both the field arrays on the grid and the particle tables; the interleaving allowed us to use direct vertorization in the place of gather-scatter constructs used in portions of STORM. Careful attention to the assembly code allowed us to improve the Fortran versions of SQAL and later to use some assembly code in the slowest code regions. Although we succeeded in doubling the speed of the three code regions we addressed, we are still short of demonstrating gigaflop speeds in the Cray-2 environment.

The picture is more optimistic if we consider the Cray Y-MP. From our critical path analysis a top speed of 241 Mflops per processor is the theoretical machine speed limit. If we could multitask this at level 8 we could have a top speed of 1.93 gigaflops. This is a big "if" because we have shown that the *deposition interpolation* cannot be multitasked unless one is willing to accept the slightly wrong answers that come from an asynchronous method.

We believe that a further restructuring of the code to use ordered particles will lead not only to better single processor performance, but to a fully multitaskable code as well. This should give us about one gigaflop performance on the Cray Y-MP. Even better news is that this version with ordered particles is naturally extendable to more massively parallel MIMD machines, whether built by Cray Research or others- particularly if the processors are vector processing units.

Acknowledgements
We thank Bruce Cohen, Alex Friedman, Dennis Hewett, and Alan Mankofsky for valuable advice on the many questions that arose in these studies. This work was supported by the U.S. Department of Energy for the Lawrence Livermore National Laboratory under contract W-7405-ENG-48. Cray-1 is a registered trademark and Cray-2, Cray X-MP, and Cray Y-MP are trademarks of Cray Research, Inc.

References

[1] A. Bruce Langdon and D. C. Barnes, "Direct Implicit Plasma Simulation," in "Multiple Time Scales," Eds. Brackbill and Cohen, Acadmeic Press (1985), 337

[2] R. J. Mason, J. Comput. Phys. **41** (1981), 233

[3] A. Friedman, A. B. Langdon and B. I. Cohen, Comments Plasma Phys. Controlled Fusion **6** (1981), 225

[4] O. Buneman, C.W. Barnes, J.C. Green, D.E. Nielsen, J. Comput. Phys. **38** (1980), 1-44

[5] Eric J. Horowitz, Dan E. Shumaker and David V. Anderson, J. Comput. Phys. **84** (1989), 279-310

[6] E. J. Horowitz, J. Comput. Phys. **68** (1987), 56

[7] A. Heron and J. C. Adam, J. Comput. Phys. **85** (1989), 284,301

END

DATE FILMED

12/27/91