Conf.-750606--3

UCRL-76776
PREPRINT

LAWRENCE LIVERMORE LABORATORY
University of California/Livermore, California

THE MONTE CARLO PROBLEM AND PARALLEL COMPUTERS, AND HOW TO DO A FAST PARTICLE MOVER ON THE STAR 100

Kurt H. P. H. Sinz

June 4, 1975

This paper was prepared for presentation at the
7th Conference on Numerical Simulation of Plasma,
Courant Institute, New York University, New York, June 1975.

The Monte Carlo Problem and Parallel Computers

*and How To Do a Fast Particle Mover on The STAR 100**

Kurt H. P. H. Sinz

Lawrence Livermore Laboratory, University of California

Livermore, California 94550

ABSTRACT

Particle simulation problems of the Monte Carlo type are widely believed to be instrinsically highly scalar problems. In the absence of a definitive mathematical theorem to the contrary, this belief is based on the very apparent programming difficulties encountered on a vector machine. This class of problem is therefore thought to be ill-suited to highly parallel and vectorized computers. However, this paper demonstrates by several examples that a particle mover is fully vectorizable. In the case of the CDC STAR 100 it is found that the performance of such a particle mover is not hopeless but hopeful, and is in fact helpful.

One of the several possible vectorizations is estimated to yield a gain of a factor of 15 on the STAR over good serial coding on the same machine. This falls far short of the STAR's peak vector performance of 30 to 70 times scalar rates because certain fast vector instructions are not available and have to be simulated.

The current STAR algorithm outperforms the carefully handcoded 7600 by a factor of 3. This performance margin is achievable despite the 7600's fivefold superior scalar capability.

A more generally vectorized particle mover will always substantially outperform scalar coding on any machine equipped with a properly chosen set of fast vector instructions.

I. INTRODUCTION

In recent years much improvement in computer speed has come about by letting several processes go on in parallel. Examples are the CDC 6600 and the CDC 7600. An extreme example is the CDC STAR 100, which outperforms the 7600 by a factor of 7 when adding corresponding members of stringed arrays (vectors). This factor increases to 14 when using half words (32 bits). However, this type of performance margin is achievable only when the data is well ordered. The performance is worse than the 7600 by a factor of 5 or so when doing FORTRAN-like operations such as table look-ups, and so on. Within the machine the performance discrepancy between vector operations and scalar operations is a factor of 30 to 70.

Perhaps a useful working definition of what constitutes a modern parallel computer is a computer that is capable of obtaining several results in one machine cycle when operated in parallel mode, but generally only a partial result per machine cycle when operated in scalar mode. The STAR 100, the CRAY I, and the ILLIAC fall into this category as well as the TI-ASC (four pipe). On these machines there exists a discrepancy between scalar operations and the various brands of vector operations approaching one order of magnitude, or even two orders. It is apparent, therefore, particularly in the case of the STAR and in the case of the other vector machines as well, that a few percent of coding left unvectorized will seriously degrade the performance of the coding when compared to the machine's peak performance capabilities.

It is a widely held belief that a Monte Carlo particle simulation code, as used in controlled thermonuclear research, cannot be adapted to advantage to this type of computing environment. The main obstacles were table look-ups

and the apparently scalar, and possibly even recursive, accumulation process of sample points on a mesh to form, for example, charge distributions. This belief was the result of these latter, very obvious programming difficulties that were consistently interpreted as experimental evidence of nonvectorizability. However, a mathematical existence theorem providing a definitive answer to this question has been and still is unavailable. We therefore present a demonstration of at least one counter example to prove vectorizability.

Considering that increased machine speeds of the future seem to rely substantially on increased parallelism, particle simulation codes might not benefit fully from the resulting speed gains if such a vectorization were not possible. Since particle simulation codes are very important in controlled thermonuclear research, it is very desirable to accelerate their development by finding solutions to this problem within the framework of more readily achievable speed gains due to vector processors, as opposed to awaiting the advent of radically different machine organizations or super-fast memories.

In any event, major technological advances in fast memories, though helpful, can probably always be made to go much further by the superpositioning of efficient vector processors on such advanced memories to increase their computational yield achievable in scalar mode alone. In the case of STAR, this performance increase is a factor of 30 to 70 as mentioned above. A demonstration of vectorizability would, in turn, permit particle simulation codes to benefit fully from such performance gains.

Therefore, as an example of how one might deal with the particle pusher problem on a highly parallel vector computer, a fast vector solution in the framework of the STAR is presented.

Since other solutions are possible, two alternatives are discussed later. However, the solution that is suggested for the STAR is three times faster than the identical problem on the 7600 when handcoded for that machine.

The details of the algorithm are presented below. The choice of operations imply no start-ups whatever for vectors of lengths less than many thousands of elements. The algorithm also circumvents all table look-ups of the usual form, since their timings are poor.

Despite this desirable characteristic, it must be noted that only 10% of the time is spent doing the majority of the necessary arithmetic (less than 80 operations). This is less than 4% of the time required to execute the handcoded 7600 algorithm. The rest of the time is spent performing many slow instructions to avoid even slower table look-ups and recursive arithmetic. The performance of the present algorithm then falls far short of the STAR's peak performance capabilities. Unless this performance mismatch among the vector operations is seriously dealt with by the manufacturers (not at the cost of artificially holding down speed advances resulting from more parallel performance), controlled thermonuclear research will not be supported by the fastest possible particle simulation codes that proper machine development could facilitate.

We demonstrate a completely parallel way of solving the particle pusher problem that is hindered by the unnecessarily poor machine performance of certain vector instructions (other than table look-ups). Also, we make some hardware design suggestions that, hopefully, will also find consideration in the design of new parallel machines to speed theoretical controlled thermonuclear research in the future.

II. A MONTE CARLO PROCESS

As an example of Monte Carlo particle codes, a plasma particle mover has several characteristics that on the surface seem to be totally incompatible with the STAR architecture. The problem consists of many random particles moving through a mesh with electric and magnetic fields, represented at the mesh points, deflecting the particles' trajectories as they pass through the zones. At each time-step the cumulative effects of the particles in each zone must be found as they affect the updating of the fields in time.

The two main problems affecting the solution of this problem on parallel computers and the STAR in particular are, first, the table look-ups of the mesh values once a particle's zonal association has been established. The second problem is the accumulation of the particles' charges and currents on the mesh points. This involves another table look-up and additions that may be recursive if two successive particles are in the same zone and their mesh contributions cannot be treated in parallel. When handcoding the 7600, this problem can usually be circumvented by processing different scalar instructions in parallel - a possibility that arises from that machine's relatively limited parallel capabilities and the inherently lower losses when not using that facility. On more modern parallel computers, however, the economy of scale seems to demand that many operations of the same type be done at once. This characteristic gives rise to the isolation into single processes of the table look-up and the mesh accumulations, the recursion problem and, of course, memory conflicts. Machines currently under construction and future machines whose computational power resides mainly in some sort of vector processors are likely to encounter problems in this area as well, so that it is desirable to find solutions to this

problem in the framework of parallel computers. One such solution that relies strictly on the parallel facility of the STAR is given here, along with timing estimates that surpass the handcoded 7600 for the same problem by a factor of 3.

We restrict ourselves to the case in which the mean number of particles in a zone is about ten. The complication in this case is the large overhead of typically 400 add times associated with the initiation of each vector operation, so that vectors must be kept long. The algorithm would be somewhat different and much more efficient if a few start-ups could be amortized over, say, 50 to 100 particles per zone. Such an algorithm would be easy to construct and shall not be delved into here. Two alternate solutions are suggested for the present problem. Also, two hardware features are suggested for future parallel machines that would make the execution of this type of problem completely nonserial and nonrecursive, and therefore, parallel.

First the problem of a one-dimensional electrostatic particle mover will be dealt with because of its logical conceptual simplicity. Descriptions of certain programming algorithms are given in the appendixes.

III. A ONE-DIMENSIONAL PARTICLE MOVER

The motion of a given particle is determined by first finding what zone the particle is in. This in turn determines which zonal quantities (electric fields) besides initial conditions govern the particles' traversal of the zone. This piece of coding on the 7600 traditionally involves table look-ups, an operation that is inherently nonparallel. It is known to be very slow on the STAR since it exceeds 0.5 μs, which is enough time to perform about 50 half-word arithmetic operations in vector mode. For each

particle two fields have to be obtained so that this alone would cost minimally 1 μs per particle. This cost could easily go to 2 μs when memory conflicts are encountered.

The other problem mentioned above is the accumulation of the particles' charge contributions on the two neighboring meshpoints. This summing operation is somewhat troublesome even on the 7600 when coded in vector style, because two particles that are treated successively may want to contribute their charges to the same meshpoints, in which case the second addition has to be delayed and must be carried out serially instead of in parallel. Nevertheless, on the 7600 the necessary test in this innermost charge-summing loop is a viable proposition at a loop cost of 0.8 μs per particle when coded in FORTRAN. On STAR, this same DO loop could cost five or more times as much or 4 μs. This accumulation process is worse than even the table look-up, and makes the fact that the STAR can do the remaining equations of motion in the time that the 7600 does one single addition irrelevant. As a rule-of-thumb, the performance mismatch in the STAR between half-word vector operations and FORTRAN coding is 70 to 1. If this problem is to be solved on the STAR, or other similar computer, a way must be found to make a particle mover parallel even if many more operations are performed than seem to be required by the algebraic statement of the basic algorithm.

IV. RESTRUCTURING THE ALGORITHM

As a method of introducing some degree of parallelism, the ordering of particles has been proposed; the question of how to do the ordering having been left unanswered. If, however, some ordering scheme were successful, a small number of particles per zone such as an average of ten would make for very short vectors, and start-up costs alone could run three times certain

execution rates on the 7600. We therefore endeavor to propose a method that is free of start-ups for each zone and takes full advantage of the STAR's vector capability. The methods set forth for the one-dimensional example are trivially extendable to a two-dimensional problem. The timings for this latter problem are estimated to surpass the hundcoded 7600 by a factor of 3.

In the algorithm that follows, we indeed sort the particles every timestep, so that the particles that reside in a given zone are stored contiguously. A minor restriction that could easily be relaxed is the assumption that, in a given timestep, the particles move no further than one zone diameter. To determine which particles are in the i^{th} zone at time t, it is only necessary to look at the particles that were in the i-1, i, and i+1 st zone at time $t-\Delta t$.

To begin with, it is assumed that the particles are ordered and that their positions and velocities are given in two arrays: X and V. Electric fields, E, are given at the mesh points whose spacing is one. It is also assumed that an array of n_i's is given that specifies the number of particles in the i^{th} zone. In one-to-one correspondence with the X array, is a bit string, BN, whose runs of 1's with one terminating 0 state which contiguously stored particles are in the same zone. Thus, a run of n_i-1, 1's, and one 0, says that the i^{th} zone contains n_i particles. If only one particle resides in a zone, then there will only be one zero, and if there are no particles in the zone, there will be no bit. It is also desirable to have a bit string BZ stating which zones contain particles and which ones are empty; a condition that must be allowed for on a dynamic basis.

In the design of the overall algorithm, a very specialized random access capability of the STAR is exploited. This capability consists of skipping a random number of elements as flagged by a bit string. This can,

of course, function only as long as the basic ordering is not disturbed. The finer details of certain algorithms are relegated to the appendixes for whose understanding some familiarity with the STAR computer is assumed.

V. ALGORITHM DESCRIPTION

The main organizational difference between doing the problem on the 7600 as opposed to on the STAR is that on the 7600 the fields are brought to the particles by means of table look-ups, while on the STAR, in some sense, the pertinent particles are brought to each field value. There are two ways to do this: One is to store contiguously the particle attributes for each field value. The other method divides the particles into sets such that in each set at most one particle corresponds to a given zone.

Assuming the first ordering, the programming steps for an average particle density (n) of 10 particles per zone would proceed as follows. The timing estimates allow for up to 128 particles in any given zone and are given in machine cycles per particle with 25 cycles = 1 μs.

Step one. Form a vector of electric fields taken from the left edge of the zone, as discussed in Appendix A. Method 2 is preferable. Method 1 is preferable for an average particle density of 5 or less; likewise, form a vector of fields from the other side of the zone.

Time: 9 cycles

This time means that despite the STAR's slow memory, the table look-up problem has been effectively reduced to execute as well or better than the 7600 when the cost of ordering is ignored.

Step two. Find the new particle positions and velocities by completely trivial vector methods.

Time: 1-1/2 cycles

Boundary conditions typically are either periodic or reflecting. In either case, the ordering permits their handling by either phantom zones (a few additional mesh points at the edge of the mesh) or a physical copy from each end of the particle arrays to the other, or a combination of the two. When dealing with upward of 100,000 particles, this process is invisible in the timing.

Step three. Since the particles may now reside in different zones, proceed with the reordering of the particle attributes as discussed in Appendix B. This process absorbs the lion's share of the time. Included are the formation of the new n_i, BZ, and BN arrays, and finding the new n_{max}.

Time: 33 cycles

Step four. Now the charges need to be summed on the mesh. The process is described in Appendix C.

Time: 2-1/4 cycles

The approximate total time of 1.9 μs is competitive with the execution rate of the identical algorithm on the 7600 of a measured 2.6 μs obtained by using handcoded library loops. The STAR algorithm is foolproof with respect to the number of particles that may reside in a zone with only minor effects on the timing. It should also be noted that several instructions whose use has been suggested in the appendixes lag behind other vector operations by a factor of 16 in performance[1]. This is very harmful for the merge and sparse add operations and seems unnecessary since the data flow is completely monotonic, which is not the case with table look-ups. However, given the STAR as it stands, the time savings are modest compared to the 7600. The case lies a little differently with a two-dimensional particle mover.

VI. A TWO-DIMENSIONAL PARTICLE MOVER

The extension of the above algorithm to two dimensions is very straightforward. Now, however, there are three fields (two electric and one magnetic) with four values each that determine a particle's trajectory. Also, besides summing charges, two currents must be summed on four-mesh points for each particle. In the ordering, it is assumed that a given particle may travel no further than to any of eight zones adjoining the zone of origin. Again, this is a minor restriction that could be relaxed. To avoid looking at eight sets of particles for each zone, the following stratagem consisting of two one-dimensional ordering sweeps is suggested.

Assuming x-wise storage, first order all particles in each row according to their x-coordinates. This proceeds as above. With newly constructed intermediate arrays of n_i, BZ, and BN, now order the particles according to their y-coordinates using the identical algorithm with different offsets to look for particles that wish to travel "up" or "down". A particle that really wants to go to the zone in the southeast will then first be associated with the zone to its east on the first pass, and then with the zone in the south on the final pass. Again, the approximate timings are given in machine cycles per particle. The assumption is that the extreme number of particles per zone ranges from 0 to 128 with an average of 10.

Step 1. Establish 12 field values as described in Appendix A.

Time: 49 cycles

Step 2. Find the new particle positions. This step consists of less than 80 completely trivial vector operations.

Time: 20 cycles

Step 3. Reorder the particles. This step is equivalent to making two one-dimensional ordering passes as above and as described in Appendix B.

Now there are four particle attributes to sort. The new BN, BZ, and n_i arrays are formed.

Time: 102 cycles

Step 4. Sum the mesh contributions as in Appendix C. There are three quantities (charge and two current components) that must be summed on four-mesh points for each particle.

Time: 22 cycles

Again, the boundary conditions for the particles are very trivial to handle and shall not be treated here. The total time for this algorithm on STAR as it now stands is therefore estimated at 8 μs.

The handcoded version of the ZOHAR code that is currently being used at Livermore on the 7600 requires about 25 μs[2]. The STAR will then perform the problem with a speed advantage of 3 over the handcoded 7600.

VII. ALTERNATE METHODS AND HARDWARE SUGGESTIONS

Method 1

Another method of doing this problem is to effectively transpose the ordering. The data representation would be an array of dimension: The number of zones by the maximum number of particles in a zone for each particle attribute. A bit matrix indicates which entry in the decimal matrix is a particle and which is a "hole". This data representation has the advantage of the field look-up and the charge and current accumulation being completely trivial. These two processes would also be very fast, although effectively more particles are being processed. The speed gain in Steps 1, 2, and 4 would be a factor of about 4. The ordering, however, is complicated by the fact that the particles want to move and Boolean searches have to be made for the hole in the appropriate column. Thinking momentarily

in terms of a two-dimensional phase space, the physics of the problem might allow an ordering and distribution in the columns according to the particle velocity distribution, so that each row moves more or less in unison. In any case, the ordering is less cut-and-dried and the timing less predictable than the method discussed above.

Method 2

Another possible method that has implications for future machine design to solve this problem of mesh accumulation is as follows. Assume that only a very coarse ordering is performed every so-many timesteps, and that the accumulation loops are coded as one might do on the 7600. This can involve a test to determine whether the zone currently requested is still busy from the last request. Such a test on the STAR is exceedingly costly. The problem is aggravated by the fact that charge and current contributions to a given corner of the various zones form individual contiguous arrays. Efficiency in the respective accumulations is seriously diminished if one tries to alternate between the arrays or mesh corners to avoid the test for a busy from the last request. A solution that circumvents both of these difficulties is as follows:

Several scratch copies are made of the grid charge array, for example. This has the merit that successive particles could be summed into different mesh copies so no particle's contribution to the charge array has to be delayed to let the addition of the previous particle's charge complete. If the additions can be made to run in parallel, this constitutes a method whereby the charge-summing algorithm can be multiplexed. If there is some coarse sorting and a given array of particles can be guaranteed to reside in any of a few given zones, then the duplicate charge arrays can be held in the registers. This facilitates much more rapid access and totally avoids

the test of a busy, since enough copies can be kept to guarantee that the first is no longer busy when the last one finishes. Also, memory conflicts disappear. This concept is very applicable to the Cray I, with its many registers that can be used for this purpose. The problem of not being able to index registers can be easily handled by modifying a list of register instructions in vector mode and thus generating perfect instruction stacks. After a given group of particles has been processed, the multiple copies of the charge arrays are added in vector mode. This is another stratagem that permits the exploitation of parallel hardware for a process that, to all appearances, is serial.

An extension of this concept would be to multiplex the table look-ups of the field quantities. This would be accomplished by having multiple copies of certain fields in the registers. A table look-up instruction would now have to be constructed that, in one machine cycle, obtains several values from the tables, provided that no two look-up addresses are the same. This latter condition is guaranteed by pointing the different addresses in each one-cycle multiple look-up at the different copies of the tables. In the particle ordering, each particle is used only once, since particles are conserved so that the same look-up feature should be usable without multiple copies. The actual acceleration of the particles can be performed trivially in parallel by utilizing a vector capability. This then represents a method whereby the entire particle moving problem can, in principle, be done efficiently in parallel with a trivial number of extra operations and hardware modifications that seem within the grasp of current technology.

It appears that particle simulation codes could benefit fully from large advances in highly paralleled vector computing and are not relegated to awaiting the advent of radically new machine architectures.

VIII. VECTOR INSTRUCTION REQUIREMENTS

In the algorithms presented, several examples emerge that suggest simple new vector instructions. In particular, in Appendix A the broadcasting of the fields could be accomplished by a single instruction that broadcasts successive members of the field vectors in correspondence with runs of bits in the controlling bit string. This instruction should, in principle, execute at add rates and would reduce the run time of Step 1 in the algorithm above from 49 cycles to just a fraction.

Appendix D would be replaced by a single instruction that forms runs of bits of lengths specified by a sequence of integers. Again, this should be an extremely fast instruction. Overall, the relative slowness of certain vector instructions (a factor of 4 for compress and a factor of 16 for merge) is not tolerable and should be in better coincidence with the add times on a faster machine.

IX. CONCLUSIONS

We have given a fast, fully vectorized, two-dimensional plasma-particle moving algorithm that on the STAR-100 will outperform the same physics handcoded for the 7600 by a factor of three. This problem is of special interest as a worst test case for STAR because of its Monte Carlo nature. The algorithm presented, however, does not involve any "handcoded" register loops and relies entirely on vector operations. Moreover, the operations used are all characterized by completely monotonic data flow in address space and are fully parallel. Table look-ups as presently implemented do not fall into this category of vector instruction and have therefore been eliminated from the algorithm.

Some of the operations used are up to 16 times slower than vector odds, which is unnecessarily slow. This nonuniformity in timing together with the absence of a few other desirable vector instructions prevent the STAR from computing this particle-pushing problem at peak arithmetic rates. Hardware improvements in these two areas would facilitate a performance margin of ten over the 7600.

The discrepancy between vector performance and scalar performance could get substantially worse on future machines if their speed gains are obtained from increased parallelism as opposed to faster memories. However, even major advances in memory technology could serve to produce a much larger computational yield by the superpositioning of parallel hardware on such fast memories. A demonstration has therefore been given of the necessary logical reductions to obtain a fast algorithm for this problem on a parallel machine while pointing out the places of relatively poor performance. It is believed that large hardware speed advances, resulting from highly parallel vector processing, would thus fully benefit particle simulation codes. Also, alternate suggestions have been made of how to adapt future parallel hardware to the solution of this problem, or how to adapt the algorithm to other parallel hardware to demonstrate the viability of this Monte Carlo type of problem in a new, nonserial computing environment.

REFERENCES

* This work was performed under the auspices of the U.S. Energy Research & Development Administration.

1. J. Engle, private communication.

2. A. B. Langdon, private communication.

APPENDIX A: FORMING A VECTOR OF FIELDS IN CORRESPONDENCE WITH THE PARTICLES

Assuming that the particles are ordered zone-wise, two similar vector algorithms are presented that replicate the appropriate field values in correspondence with the particles. The object is to broadcast successive values of an array in correspondence with runs of bits in a controlling bit string. This seems to be a generalized broadcast instruction. A single fast instruction to this effect should outperform the two simulations given here by a factor of 16.

METHOD 1

Step 1. Compress the mesh array of fields under control of BZ to allow for empty zones.

Time: 1/n cycles

Step 2. Merge the result of Step I with 0's, using BN, yielding a vector of the form: $0., 0., 0., F_1, 0., 0., F_2 \ldots$

Time: 4 cycles

Step 3. Using the method of Appendix C, form running sums of these fields for each zone. This yields a data string of the form: $F_1, F_1, F_1, F_1, F_2, F_2, F_2, \ldots$

Time: 1-3/4 cycles for arithmetic

Total Time: 6 cycles per field per mesh point

1/2 cycle overhead

METHOD 2

Step 1. Form the running sum of the n_i's using logarithmic reduction and convert the resulting indices to full words to circumvent

a machine peculiarity. Assume that 4096 zones are processed in one vector.

Time: 3/n cycles

Step 2. Compress the mesh field array under control of BZ to allow for empty zones.

Time: 1/n cycles

Step 3. Clear a temporary array and use the indices obtained in Step 1 to transmit the fields of Step 2 into this array.

Time: 20/n + 1/4 cycles

Step 4. Use the method of Appendix C to form the field array as above.

Time: 1-3/4 cycles

7/16 cycles bit logic

Total Time: 4 cycles per field per mesh point

3/4 cycles overhead

APPENDIX B: HOW TO DO THE PARTICLE ORDERING

Two simple methods of doing the particle ordering are presented. Method 2 involves the use of an instruction that has a built-in start-up time for every zone and is therefore dealt with only briefly. The conceptual steps in Method 1 are very simple. First thinking in one dimension, bit strings are formed that tag whether particles are leaving their zones or not. Compresses perform a sorting into three groups according to whether the particles leave a zone by moving left or right or whether they are "stay homes". Performing the pertinent counts for each zone and building the appropriate bit strings allows the merging of the three sets of data strings

into the newly zone-wise-ordered arrays. The ordering scheme presented is foolproof as to the number of particles that may or may not exit from any combination of zones. Certain operations that are done only once for every ordering regardless of the number of particle attributes are labeled overhead in the timings. This item is unduly large because of the absence of the vector instruction simulated in Appendix D. Secondly a fast instruction is desirable that would form counts of bits in one bit string in correspondence with runs of bits in a controlling bit string thus performing the function of Step 4 below. Third the slowness of the compress instruction at 1/4 the half-word add rate makes Step 3 costly in the ordering of several particle attributes. The lack of executing speed of the merge and sparse add instructions at 1/16 half-word add rates are the most devastating obstacles to excellent performance.

The latter instruction alone accounts for 6 out of 9 cycles per particle attribute in the ordering. A tune up of these slow timings and the installation of various fast instructions would probably reduce the timings from 15 for the overhead and from 9 for each particle attribute to 1 or 2 cycles each. A factor of about 10 in the ordering algorithm presented below would therefore seem to be achievable.

METHOD 1

Step 1: Form the "floors" of the new X^1 coordinates in array IX^1.

Time: 1/4 cycles

Step 2: Compare IX^1 with the old IX to set bits for the particles that move into the zones to the left or right, or simply "stayed home". The resulting three-bit strings are called B_L, B_R and B_H.

Time: 9/16 cycles

Step 3: Compress each particle attribute under the control of B_L, B_R and B_H.

Time: 3 cycles per particle attribute

Step 4: Zero out an array and initialize different portions of words under the control of B_L and B_R and form the running sums under control of BN as discussed in Appendix C. Some economy can be realized by saving the intermediate bit strings from elsewhere in the algorithm. Compression under the control of the complement of BN yields an array that effectively contains the counts of particles that are zone emigrees in either direction.

Time: 3-1/2 cycles

Step 5: Expansion of the result of Step 4 under control of BZ brings the counts in one-to-one correspondence with the zones. Trivial vector operations yield the new zone population counts, regardless of whether certain zones are no longer empty or have just emptied out.

Time: 6/n cycles

Step 6: The population arrays of "left emigrees", "right emigrees", and "stay homes" permit the formation of bit strings as discussed in Appendix D. These three strings are generated in such a way that they have no intersection. Extra 0's between runs of 1's are innocuous in the intended use of these strings but distinguish the union of these strings from the new BN.

Time: 9 cycles

Step 7: Using the operation of "sparse add", the merging of the left emigrees and right emigrees into a single, ordered string is accomplished for each particle attribute. Using another sparse add of this result with the "stay homes" completes the ordering. In the timing, it is assumed that one quarter of the particles have moved in each direction which probably results in a high estimate.

Time: 6 cycles per particle attribute

Step 8: A comparison of the newly ordered particle coordinates with those of their neighbors determine the new BN. A conceptually preferable way of forming this string might be to do a few Boolean operations with the results of Step 6 and to do a compress of the appropriate bit strings.

Time: 1/4 cycles

Step 9: A comparison of the new zone populations with zero yields the new BZ.

Time: 1/4 n cycles

Total Time: 15 cycles overhead;

9 cycles per particle attribute

METHOD 2

Another method of doing the particle ordering involves the use of the transmit index record instruction. This instruction would make copies of the particles' attributes associated with each zone as well as those of the adjacent neighbors. Because of the fact that the number of particles in each zone is variable, irrelevant data from other zones is picked up that rounds out each record to a constant maximum length. A comparison with a trivially generated list of zonal values followed by a single compress completes the ordering.

The size of the intermediate scratch array will exceed substantially three times the number of particles.

The transmit index record instructions, however, implies an internal start-up and the timings of this method do not become favorable compared to Method 1, until the number of particles per zone reaches 50. The details of this method are therefore beyond the scope of this discussion.

APPENDIX C: FORMING RUNNING SUMS IN AN ARRAY CORRESPONDING TO RUNS OF BITS

Given a bit string such as BN, one wishes to obtain a running sum of items in a data string that corresponds to the runs of 1's and the respective terminating 0's. This summing requires somewhat recursive arithmetic and may always be a problem on vector machines unless it is permissible to permute the ordering of the summation process. A fast vector instruction to this effect seems desirable. In the present algorithm, the length n_{max} of the largest run must be known to determine the minimum number of passes through this logarithmic reduction scheme. If the sums are formed in place, the steps are:

Step 1. V = BN .CTRL. (V + V(1;))

where V(1;) means V is offset by one, and .CTRL. means that no result is stored when no enabling bit is set. This allows for the proper result in the last position (0's) of each run.

Step 2. B1 = B1. INT. BN(1;)

This operation replaces the trailing 1's in each run with 0's.

Step 3. V = B1 .CTRL. (V + V(2;))

Step 2 ensures that the sums (or zones) are kept separate.

Step 4. Continue $\log_2 n_{max}$ times to form the respective running sums and, of course, the various zonal subtotals.

This algorithm is used to broadcast fields, to sum charge contributions of particles, and to count the particles in each zone.

Total Time: 1-3/4 cylces for arithmetic;

7/16 cycles for bit logic

APPENDIX D: METHOD OF GENERATING A BIT STRING GIVEN A LIST OF COUNTS OF RUNS OF 1's

Given a list of indices, say, the n_i's, it is desired to generate a bit string with runs of n_i bits turned on. Extra 0's between these runs are irrelevant when used as control strings in the sparse operations referred to. In principle this entire appendix should be a single fast-vector instruction. However, we can simulate such an instruction by realizing that in binary format the number 2^m-1 constitutes a run of m 1's. The instruction simulation of this appendix assumes that the length of the maximum run is 128 so that it has to be distributed over several words. If the length of the maximum run were not to exceed 48 this instruction simulation would be very much faster. Numerous additional simulation algorithms are possible.

Step 1: Find n_{max} and the number of half-words, n_w, needed to contain n_{max} bits.

Time: 1/4 n cycles

Step 2: Construct a regular bit string and expand the n_i's into a full word array, allowing n_w full words for each n_i.

Time: 16/n cycles

Step 3: Using straightforward vector instructions, distribute over each group of n_w words the number 32 and a remainder to form the proper addends for the various n_i's.

Time: 6/n cycles

Step 4: Using the appropriate exponent and mantissa-manipulating instructions, form $2^m - 1$ in each full word. Step 3 guarantees that each $2^m \leq 32$.

Time: 2/n cycles

Step 5: A compress saving all lower half-words yields a bit string with runs of n_i bits turned on.

Time: 4/n cycles

Total Time: 3 cycles per particle

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, Virginia 22151
Price: Printed Copy $___*; Microfiche $2.25

*Pages	NTIS Selling Price
1-50	$4.00
51-150	$5.45
151-325	$7.60
326-500	$10.60
501-1000	$13.60

JH/rt