JOURNAL OF COMPUTATIONAL PHYSICS 61, 138–153 (1985)

Molecular Dynamics on Vector Computers*

FRANCIS SULLIVAN

Center for Applied Mathematics,
National Bureau of Standards, Washington, D.C.

RAYMOND D. MOUNTAIN

Center for Chemical Physics,
National Bureau of Standards, Washington, D.C.

AND

JULIE O'CONNELL†

TRW, *Inc., Cleveland, Ohio*

Received September 12, 1984; revised January 8, 1985

An algorithm has been developed for computer simulation of molecular dynamics. The algorithm, called the "method of lights," is based on sorting and on reformulating the way in which neighbor lists are constructed. It uses data structures compatible with either traditional scalar computer architecture or specialized vector statements which perform computations in parallel. The algorithm has been implemented on the CYBER 205[1] computer. Tests indicate that the method reduces running time over standard methods in scalar form, and that "vectorization" produces an order-of-magnitude decrease in execution time. © 1985 Academic Press, Inc

1. INTRODUCTION

Computer simulation of molecular dynamics is by now a well-established and important technique in condensed matter physics and chemistry. Simply stated, it consists in solving Newton's equations for a collection of atoms or molecules which are assumed to interact according to some postulated (usually pair-wise) force law. The solution of these equations is then used to determine thermodynamic and transport properties of a dense system, such as a liquid. Greater speed and wider

* Certain commercial equipment, instruments, or materials are identified in this paper in order to adequately specify the computational procedure. Such identification does not imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the materials or equipment are necessarily the best available for the purpose.

† Present address: The Catholic University of America, Washington, D.C.

0021-9991/85 $3.00

availability of computers has led to increased use of simulation techniques, making possible recent rapid advances in the understanding of the liquid state [10].

For very large simulations, vector-processing "supercomputers" offer great potential for reducing run time. In order to make effective use of these machines, algorithms must be designed to be "vectorizable," that is, compatible with language-based and machine-based structures for parallel processing. A new algorithm based on sorting has been developed. The algorithm, called the "method of lights," can be implemented with either scalar or vector arithmetic; it is more efficient than existing techniques even on scalar machines. Tests indicate that on vector machines the method greatly reduces running time and thus helps to make very large scale simulations a practical possibility.

Determination of the forces acting between the particles as functions of the particle positions is a time-consuming calculation which must be performed at each time step in a simulation. Every particle interacts with every other so that, in principle, for n particles there are, $O(n^2)$ forces which need to be computed at each time step. In most cases, however, the force falls off rapidly as a function of distance, and a particle can be assumed to interact only with its nearby neighbors. Typically, a neighbor list is constructed and periodically updated as the simulation progress [11]. While impressive reductions in execution time have resulted using the neighbor list method, the algorithms have not been designed to take advantage of the architecture of vector machines. We have concentrated on reformulating the structure of the neighbor list and the procedures for updating it.

In the following section we give a general description of the algorithm and some details concerning implementation. This information is general and applicable to any computer. In Section 3 we take up the "vector" version of the algorithm. The principal task is to vectorize the method for generating the neighbor list described in Section 2; however, we also modify the sorting method and the computation of particle accelerations used in the integration routine. Timing data are given for the vector version. In Section 4 we comment briefly on some other possibilities for neighbor list algorithms. The remainder of the present section presents a general description of the model and the methodology for calculating interactions between particles.

The model consists of an ensemble of interacting molecules in a 2- or 3-dimensional "box." Periodic boundary conditions are imposed so that in effect all of space is filled with identical boxes. This periodicity must be taken into account in determining neighbors, as is illustrated for two dimensions in Fig. 1. Particles near, for example, the upper right corner of the box are neighbors of those in the lower left corner.

Interactions between particles are determined by a potential law. We use a soft sphere potential, so that the (purely repelling) potential averages for the ith molecule are given by

$$V(\mathbf{r}_i) = \sum_j \left(\frac{\sigma}{R_{ij}} \right)^{12} .$$

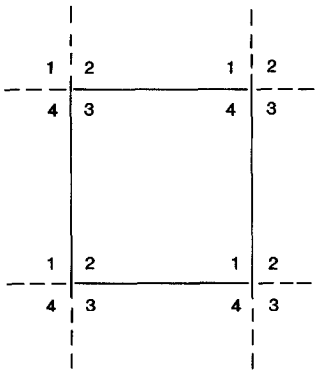

FIG. 1. Periodic boundary conditions.

Here j ranges over all particles different from i, R_{ij} is the distance from particle i to particle j and σ is the unit of length. The acceleration of molecule i at position $\mathbf{r}_i$ is, thus,

$$\mathbf{a}_i(\mathbf{r}_i) = -\nabla V(\mathbf{r}_i)/m.$$

Here m, the molecular mass, can be taken as equal to 1. The method of integration we use is that recommended by Beeman [2].

In the simulations since length is in units of σ and energy is in units of ε, the unit of time is

$$\tau = (m\sigma^2/\varepsilon)^{1/2}.$$

In the sum for the potential of the ith molecule, terms for which $\mathbf{R}_{ij}/\sigma > R = 1.5$ are set to zero and the neighbors of a given particle are just those particles with distance less than or equal to R. In computing the force on particle i, we consider those particles, j with $R_{ij} < R + \delta$. Because of the extra thickness δ and the fact that the step size is sufficiently small so that particles do not move beyond the δ between updates, the neighbor information needs to be updated only once every few time steps and saved in the neighbor list.

In software written for conventional scalar machines, it is common to save storage by structuring this list as a 1-dimensional array in which sets of neighbors for each particle are separated by zeroes. This provides efficient utilization of storage, since pairs of neighbors need to be stored only once. However, this device is not efficient for parallel processing. Hence for vectorization we make the list a 2-dimensional array. More detail is given in Section 3.

2. THE METHOD OF LIGHTS

The procedure for determining neighbors is based on sorting the particles independently according to values of each coordinate. Information from the

separate sorted orders can be used to obtain X, Y, and Z ranges which are then combined to generate a cubical box neighborhood for each particle which includes the sphere of radius of $R+\delta$.

For simplicity, we limit discussion to the 2-dimensional case and all lengths are scaled by the length of the side of the box. Implementation in the model for three dimensions is based on the obvious extensions. (See Fig. 2.)

Because all of the forces were set to zero for which $R_{ij}/\sigma > R$, the neighbors included in the box but in the sphere have no effect on the acceleration. The first step is to sort the particles according to X- coordinate values, and independently by Y values. While the sorting can be done by a conventional $O(n \log n)$ method, improvements are achievable for both scalar and vector algorithms. The neighbor list is retained for several time steps before updating it so that on all steps after the first the lists are already approximately ordered before the sort is performed. To take advantage of this approximate ordering we use the "Smoothsort" algorithm due to Dijkstra [5]. This algorithm is $O(n \log n)$ for a completely disordered list but $O(n)$ for an ordered list, with a smooth transition from one to the other. Our program for scalar machines incorporates Dijkstra's method. In the vector case, however, we use a method similar to the non-contingent "Diamondsort" [4]. Although Diamondsort does not take advantage of the existing ordering, it is extremely efficient on vector machines [8].

A square neighborhood for each particle is described by pairs of pointers which indicate ranges in the lists of sorted particles. We are given lists of coordinates X_i, Y_i, $1 \leqslant i \leqslant N$. In sorting the X and Y coordinates we need to retain the original indices as well as generate the sorted ones; hence, we define auxiliary arrays LOCX and LOCY and use indirect addressing. These arrays contain the original indices in sorted order so that

$$[X_{\mathrm{LOCX}_1}, < X_{\mathrm{LOCX}_2}, \ldots, < X_{\mathrm{LOCX}_N}]$$

and

$$[Y_{\mathrm{LOCY}_1}, < Y_{\mathrm{LOCY}_2}, \ldots, < Y_{\mathrm{LOCY}_N}].$$

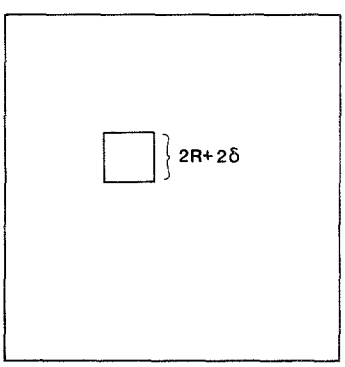

FIG. 2. Square neighborhood.

The pointers which describe the square neighborhood cell are generated as follows: given a pointer into the array of increasing X values only two more pointers, JB and JE, are required to indicate which particles have X coordinates within $R+\delta$ of particle LOCX_I. These pointers are saved in $\mathrm{JBS}_{\mathrm{LOCX}}$ and $\mathrm{JES}_{\mathrm{LOCX}}$, respectively. Then i is incremented and JB and JE are incremented until ranges for LOCX_{I+1} are found. (See Fig. 3.)

The periodic boundary conditions are treated by adding or subtracting 1.0 in the calculation of distance. For example, when the upper index JE reaches N, it is reset to 1 and 1.0 is added to $X_{\mathrm{LOCX_{JE}}}$, as shown in Fig. 4.

The algorithm for updating and saving of pointers is essentially a "DO WHILE" loop. Advancing the JE pointer, for example, is done as follows:

```
      ADDX = 0.0
                              {for 1 ⩽ K ⩽ N}
      SX(K) = X(LOCX(K))
            ⋮          {Initialize JE and JB for particle LOCX(1)}
      DO 50 I = 2, N
   30 IF((SX(JE) + ADDX) − SX(I).GT.R + δ) GO TO 40
        JE = JE + 1
        IF (JE.GT.N) THEN
          JE = 1
          ADDX = 1.0
      END IF
      GO TO 30
            ⋮              {Similar code for JB}
   40 JBS (LOCX(I)) = JB
      JES(LOCX(I)) = JE
   50 CONTINUE
```

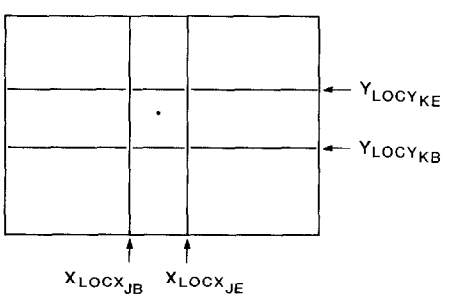

FIG. 3. X and Y ranges.

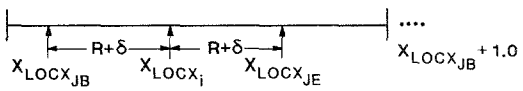

FIG. 4. Periodic boundary in X.

Pointers KB and KE for the Y-ordered list are generated similarly. Thus, the cell associated with particle i consists of the list of all particles j such that

$$X_{\mathrm{LOCX}_{\mathrm{JBS}_i}} \leqslant X_j \leqslant X_{\mathrm{LOCX}_{\mathrm{JES}_i}}$$

and simultaneously,

$$Y_{\mathrm{LOCY}_{\mathrm{KBS}_i}} \leqslant Y_j \leqslant Y_{\mathrm{LOCY}_{\mathrm{KES}_i}}$$

with appropriate modifications for the periodic boundary conditions. This list, in fact, is the intersection of the sets of indices associated with the X and Y neighbors and its computation can be facilitated by using an additional array RGY of indirect addresses. For each j, RGY_j is the rank position in the Y-sorted order at which j appears, so that

$$\mathrm{RGY}_{\mathrm{LOCY}_j} = \mathrm{LOCY}_{\mathrm{RGY}_j} = j.$$

This formulation provides a simple method for generating RGY. It is helpful to think of RGY as pointing from the array Y_j into the sorted order and of LOCY as pointing from the sorted order back to Y_j.

Using array RGY, the neighbors of particle i are just the LOCX of those js satisfying

$$\mathrm{JBS}_i \leqslant j \leqslant \mathrm{JES}_i$$

and simultaneously

$$\mathrm{KBS}_i \leqslant \mathrm{RGY}_{\mathrm{LOCX}_j} \leqslant \mathrm{KES}_i.$$

(Again, the obvious modifications must be made in order to handle the periodic boundary conditions.)

The theoretical complexity of this method depends mostly on how R, the radius of interaction, varies as a function of n. For the scalar case the sorting time is between $O(n)$ and $O(n \log n)$. Determining the arrays JBS, JES, KBS, and KES is $O(n)$. As we have seen, filling the neighbor list requires computing, for each i, the intersection of the sets of indices of X and Y neighbors. This time is proportional to the size of these sets, which in turn is proportional to nR. Thus, the asymptotic complexity should look like $O(n^2R)$. In general, since the number of particles per unit area or volume is approximately constant, the average number of neighbors

per particle is constant, i.e., nR^2 is constant. Hence, for a 2-dimensional problem, $O(n^2R)$ is between $O(n)$ and $O(n^{3/2})$. In three dimensions nR^3 is the constant and n^2R grows like $n^{5/3}$.

3. Vectorization of the Method of Lights

The FORTRAN available on the CYBER 205 has been augmented with a number of functions to permit explicit manipulation of vectors. These allow the programmer to specify vector computations which would not ordinarily be recognized as such, even by an optimizing compiler. However, vectorization and, in particular, use of the augmented FORTRAN requires that programs be designed to take advantage of these capabilities.

We briefly summarize the vector notation. Vectors are similar to, but not exactly the same as, 1-dimensional arrays. A vector is designated by specifying its starting point and its length. Thus, if $I(1; 10)$ is the vector which corresponds to the entire 10-element array I; $I(3; 5)$ is a vector consisting of elements three to seven of I_j and $M(2, 3; 3)$ is the vector of elements $(M(2, 3), M(3, 3), M(4, 3))$. Vectors can be referred to either explicitly, as in these examples, or by assigning a "designator" to the vector. Designators are vector variables which are assigned to specific vectors using an ASSIGN command. The statement

$$\text{ASSIGN DES, } I(4; 2)$$

assigns the designator name DES to the vector $[I(4), I(5)]$. Both notations are used in the examples in this section.

Procedures for vectorization were applied in three segments of the code: (1) identifying particle neighbors to create the list; (2) sorting by X, Y, and Z coordinates; (3) calculating the accelerations. In some cases, the original algorithm required significant restructuring in order to apply vector functions. Timings for each segment before and after vectorization are shown in the table below. Times are for the 3-dimensional case with $n = 1000$ and $R = 1.5$: These timings indicate that an order of magnitude improvement is achievable with explicit vectorization. (See Table I.)

A. *Neighbor Lists*

Vectorization of the algorithm for finding neighbors concentrated on the procedures for determining the intersection of the X, Y, and Z ranges of each particle. It is based on two vector functions: ("gather") Q8VGATHR and ("compress") Q8VCMPRS. The "gather" function specifies that a vector is to be filled with values from a second vector, and the values in the new vector are to be arranged according to an index contained in a third vector. Thus, the statement

$$\text{U} = \text{Q8VGATHR(V, I; U)}$$

TABLE I

CYBER 205 Timings

Program segment	Execution time/Time steps (s)	
	Scalar version	Vector version
Neighbor table update	0.98	0.07
Sorting	0.06	0.01
Acceleration calculation	0.22	0.04
Total time/time step		
With neighbor table update	1.29	0.14
Without neighbor table update	0.31	0.07

assigns values to sequential locations in vector U from locations in vector V. Index I indicates the locations in V from which values are taken so that the effect is the assignement $U(J) = V(I(J))$. For example, for

$$V = [1.0, 3.4, 7.1, 9.5]$$
$$I = [3, 1, 1, 2]$$

U will be assigned the values

$$U = [7.1, 1.0, 1.0, 3.4].$$

The "compress" function is similar to "gather" except that the index vector I is replaced by a bit array B. In this case, U is assigned from the values of V for which the corresponding entry in B is "1." (See Fig. 5.)

Finding neighbors of a particle i consists of a series of "compress" and "gather" functions using arrays LOCX, LOCY, and LOCZ, and ranks RGY and RGZ. The

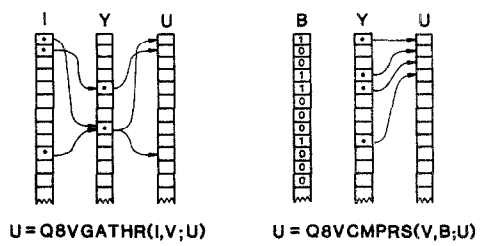

FIG. 5. "Gather" and "compress."

process begins by setting a bit vector BT from the X-indices JBS_i and JES_i. The statements

```
N = ⟨number of particles⟩
BT(1:N) = B'O'
JBU = JBS(I)
JEU = JES(I)
LN = JEU − JBU + 1
BT(1:LN) = LOCX(JBU; LN).NE.I
```

set BT_i to 1 for $JBS_i \leqslant j \leqslant JES_i$ and $LOCX_j \neq i$, and "0" otherwise. If the interval [JBU, JEU] crosses a periodic boundary, we have JEU < JBU, so that the test becomes

```
BT(1; JEU) = LOCX(1; JEU).NE.I
LN = N − JBU + 1
BT(JBU; LN) = LOCX(JBU; LN).NE.I.
```

X-neighbors of particle i are placed in a list designated DNBRX using a "compress" from LOCX according to BT:

```
DNBRX = Q8VCMPRS (LOCX(1; N) JBT; DNBRX).
```

Y-locations of the X-neighbors are obtained using a "gather" from RGY according to DNBRX:

```
DLNBR = Q8VGATHR (RGY(1; N), DNBRX; DLNBR).
```

Now DLNBR contains a list of the Y-rankings of the X-neighbors. These rankings are compared with the Y-indices for particle i, KBS_i, and KES_i, and the bit vector BT is set to reflect the results of these tests:

```
NNX = ⟨number of X-neighbors⟩
KBU = KBS(I)
KEU = KES(I)
BT(1; NNX) = (KBU.LE.DLNBR).AND.(KEU.GE.DLNBR).
```

For the periodic boundary condition with KBU > KEU, the test becomes

```
BT(1; NNX) = (KBU.LE.DLNBR).OR.(KEU.GE.DLNBR).
```

X–Y neighbors of particle i are obtained with a "compress" from DNBRX, the X-neighbors, into DNBRXY according to BT. Proceeding in a similar manner, a second "gather" from RGZ to DLNBR according to DNBRXY gives the Z-

rankings of the X–Y neighbors. These are compared with Z-indices for particle i, MBS_i, and MES_i, and BT is set to reflect the results. The final "compress" from DNBRXY according to BT yields the list of particles in the set

$$[JBS_i, JES_i] \cap [KBS_i, KES_i] \cap [MBS_i, MES_i].$$

The timings labeled "neighbor table update" include the time to find the intersection and to enter the neighbors in the output table, but do not include sorting time or time to determine the arrays of indices. Thus, these timings reflect precisely the effect of the modification described.

B. *Sorting*

The "smoothsort" algorithm is based on sequential procedures and cannot be restructed to take advantage of parallelism. Consequently, a different sorting algorithm, the Batcher sort, was selected for the vectorized implementation. The Batcher sort is similar to "Diamondsort" [4]. It uses operations which are highly compatible with parallel computation techniques and are relatively straightforward to implement with vector FORTRAN.

C. *Acceleration Calculation*

The last segment to which vectorization was applied is the acceleration computation. The acceleration algorithm uses a table look-up and an interpolation procedure to determine acceleration as a function of the distance between particles.

Implementation of the algorithm for acceleration with vector arithmetic requires modification of two data structures. First, rather than string them in a 1-dimensional array, the lists of neighbors are described by a 2-dimensional table in which the neighbors of particle i are listed in row i. Since every particle has at least some neighbors, the initial columns of the neighbor list will contain entries for every particle. Thus, the vector computation can be implemented via column-wise operations. Incremental acceleration is calculated in parallel for each particle with respect to the neighbors in a single column of the neighbor list. Corrections for the final "incomplete" columns are done in scalar mode.

In the scalar code, the periodic boundary conditions must be queried each time a distance between two particles is calculated. The resulting conditionals make implementation of the distance calculations with vector arithmetic awkward, if not impossible. In our alternative method, six additional arrays store the status of each particle with respect to the boundary conditions. The contents of these arrays are updated in the neighbor algorithm as it determines the indices (JBS, JES, etc.) particle. In fact, these arrays are obtained merely by saving values of the terms ADDX, etc., which are mentioned in Section 2. As a result, the first part of the neighbor algorithm (sorting and updating boundary conditions) is executed at each time step, even though it may not be necessary to construct a new neighbor list. Values

in the six arrays are 0 or 1, depending on whether the corresponding particle has neighbors which result from the periodic boundaries. Thus

$XA_i = 1$ if some of the neighbors of particle i in the positive direction lie across the periodic boundary,

$= 0$ otherwise

and similarly,

$XS_i = 1$ if some of the neighbors of particle i in the negative direction lie across the periodic boundary,

$= 0$ otherwise

(see Fig. 6).

In computing distances between particles, $XA_i = 1$ and $XS_j = 1$ for particles i and j implies that 1 must be added to the difference $X_j - X_i$ to account for the boundary condition. Similarly, $XS_i = 1$ and $XA_j = 1$ implies that 1 must be subtracted from the difference. It is assumed that R is small enough that XA_i and XS_i cannot simultaneously be 1 so that the X-, Y-, and Z-distances between particles i and j can be computed using the formulas

$$(X_i - X_j) + (XS_i \cdot XA_j - XA_i \cdot XS_j)$$
$$(Y_i - Y_j) + (YS_i \cdot YA_j - YA_i \cdot YS_j)$$
$$(Z_i - Z_j) + (ZS_i \cdot ZA_j - ZA_i \cdot ZS_j).$$

This eliminates the need for conditional statements in the distance calculation. As we mentioned, incremental acceleration is calculated for each particle with respect to the neighbors in a single column of the neighbor table. X, Y, Z distance components are computed as above, using a series of "gather" statements for indirect addressing from a single column of the neighbor table. This is done by using the

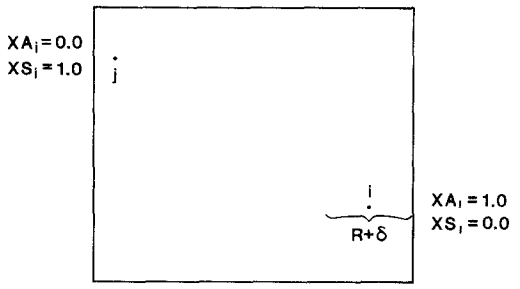

FIG. 6. Recording boundary conditions.

current column of the neighbor list as the index, and then storing the result in a temporary vector XT (see Fig. 7).

Similarly, parameter values for the neighbors are gathered from XA and from XS and stored in temporary vectors. Element-by-element multiplications, additions, and subtractions can then be performed using vector arithmetic statements to compute the X, Y, and Z distance components from each particle to the neighbor in the kth column of the neighbor list.

The table look-up procedure used for the force is somewhat awkward to implement with vector statements. It can be accomplished by means of "compress," "gather," and "expand" functions. The "expand" function is the inverse of "compress," depositing data from one array into a larger array according to a bit vector index. The table look-up is done by compressing the list of indices which point into the table (to eliminate 0 indices), gathering values from the table according to the compressed index, and then expanding into the appropriate locations of a working array. Since vectorization calculations are invoked only where all entries in a column of the neighbor table are non-zero, the effectiveness of vectorization is dependent on the current composition of the neighbor list and varies somewhat as the neighbor list is rebuilt.

4. Other Methods

In this section we discuss briefly some alternate methods for building the table of neighbors. We have tested several of these and the others have been treated in the literature. None of the methods are obviously vectorizable, but all of them have interesting aspects.

A. *Radix Sorting*

For each particle we first compute integer coordinates,

$$P(I)=\left\lfloor \frac{X(I)}{R+\delta} \right\rfloor, \qquad Q(I)=\left\lfloor \frac{Y(I)}{R+\delta} \right\rfloor, \qquad S(I)=\left\lfloor \frac{Z(I)}{R+\delta} \right\rfloor;$$

and then do a radix sort on these coordinates. That is, we put elements in "buckets"

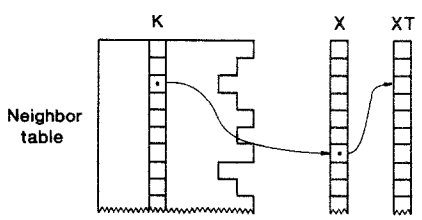

FIG. 7. Column-wise calculation of forces.

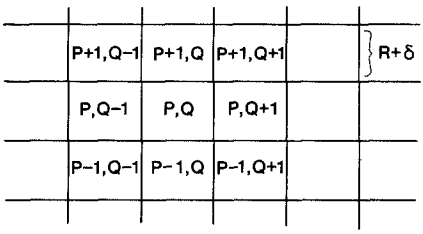

FIG. 8. P, Q stencil.

according to the S value; gather up in the S-bucket order; put in buckets according to the Q value; gather up in the Q order, etc. The object is to associate each particle with its rank in the (P, Q, S) ordering of cells.

In two dimensions we can think of the P, Q indices as associated with a "stencil" of lines (see Fig. 8). A particle in cell P, Q has possible neighbors in that cell and in the eight cells in the stencil touching it.

From the stencil we get a table of neighbors by working with the cell indices to search contiguous cells. Pointers keep track of the cell boundaries. One way to do to this is to use a pointer to the blocks of indices for cells contiguous to the P, Q cell containing particle i so that for particles with cell coordinates P, Q, one needs to search the 9-cell stencil, while keeping track of list boundaries using the set of nine moving pointers. As usual, some bookkeeping is necessary for cells on the boundary.

In the 3-dimensional case one must search a 27-cell stencil for each particle. For "medium sized" problems this a major drawback because, although the radix sort has time complexity $O(n)$, we must do a "local" $O(n^2)$ search of the stencil. The cost of this search again depends on how R varies as a function of n. Also, the bookkeeping for the periodic boundary conditions increases in the 3-dimensional case. It is likely, nevertheless, that radix sorting can be quite efficient for very large problems, especially if it is combined with some method for "automatically" determining when a cell boundary has been crossed. If position coordinates are computed in fixed point, for example, boundary crossings can be indicated by the changing of a digit in the coordinate value. It is also possible to use much smaller

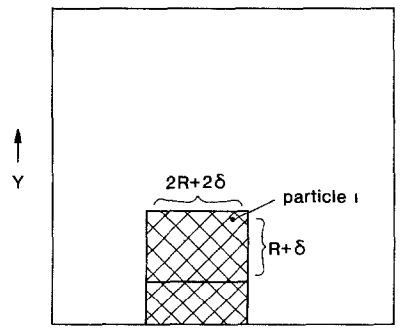

FIG. 9. The algorithm for the method of shadow.

cells to refine the $(R+\delta)\times(R+\delta)$ cell, and then keep track of the stencil by a method similar to that used to find the boundaries JBS_t, JES_t, etc., in the method of lights. This avoids searching neighboring cells because we can merely store an entire $(R+\delta)\times(R+\delta)$ cell as in the method of lights.

Hockney and Eastwood [6] describe a related method which uses linked lists to chain together particles in each of the radix cells. In this way sorting is avoided and pointers to call boundaries are not needed. However, there is some overhead associated with the linked lists, and the use of indirect addressing probably rules out vectorization of this algorithm. Rabin [9] uses a mesh of cells of side $2\times(R+\delta)$ to replace the search of neighboring cells with merges.

B. *The Method of "Shadows"*

This method was inspired by work of Hopcroft, Schwartz and Sharir [7]. In the 2-dimensional case we first sort the particles in Y order. If we are considering, say, particle i, the shadow of i consists of all those j with $Y_j \leqslant Y_i$ and $|X_j - X_i| \leqslant R+\delta$. Among these js, those with $Y_i - Y_j \leqslant R+\delta$ are neighbors of i and *all others* can be discarded. That is, any j such that $R+\delta+Yj<Y_i$ cannot possibly be a neighbor of a particle with a larger Y coordinate than Y_i. On the other hand, particle i itself might be a neighbor of particles with larger Y coordinates. Hence, the index i must be saved.

A data structure for dynamic storage allocation efficiently implements the saving and discarding of particles during the Y order search. Since it will have to be repeatedly searched for shadows of succeeding particles, the data structure should be amenable to this range searching. Several such data structures are discussed by Bentley and Friedman [3].

We have used 2:3 trees in tests. These are trees such that every internal node has either two or three descendants and all leaves have the same depth. We assume that the data are stored at the leaves sorted in X-coordinate order from left to right. Internal nodes contain information on the range of their descendants. For instance, two integers (L, M) can be used to indicate the largest element to be found on the left and middle subtrees, respectively. This is enough information to guide the search down the tree. Nodes are added and discarded using an "adopting and splitting" algorithm which is discussed in [1] (see Fig. 9).

```
st: collect from T all X neighbors of particle i
    if in Y range of i
    then
      add to i-neighbor queue
    else
      delete from the tree T
    endif
      add particle i to the tree T
    i ← next particle in Y order
    go to st
```

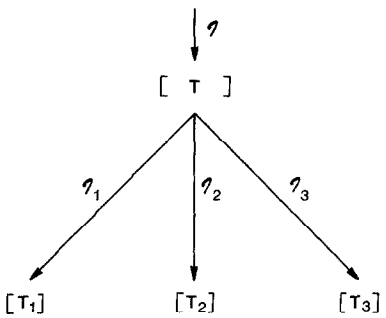

FIG. 10. Range search on the tree T.

In the range search the 2:3 tree is used as follows: The X range of particle i can be thought of as an interval of X values $\mathscr{I}$, positioned at the top of the tree T, which also represents an interval. The interval $\mathscr{I}$ must pass down the tree splitting into disjoint subintervals which in turn generate two or three more intervals, eventually stopping at nodes whose descendants are the X-neighbors of i. Given $\mathscr{I}$ and T, the general step is:

If $\mathscr{I} \subset T$, pass down one level by generating new intervals from the sons of T;
If $\mathscr{I} = T$, collect all descendants of T;
If $\mathscr{I} \cap T = \phi$, discard this branch in the search.

(see Fig. 10.)

For simulations using a few hundred particles, this method is somewhat slower than the method of lights. The difference in time is probably due to the extra overhead involved in handling the tree. On the other hand, for larger simulations on scalar machines the method of shadows may be attractive because the ongoing deletion of nodes prevents the search tree T from growing too large.

REFERENCES

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, Mass., 1974.
2. D. BEEMAN, *J. Comput. Phys.* **20** (1976), 130.
3. J. L. BENTLEY AND J. H. FRIEDMAN, *Comput. Surv. II* **4** (1979).
4. H. BROCK, B. BROOKS, AND F. SULLIVAN, *BIT* **21** (1981), 2.
5. E. DIJKSTRA, *Sci. Comput. Programming* **1** (1982).
6. R. HOCKNEY AND J. W. EASTWOOD, "Computer Simulation Using Particles," McGraw–Hill, New York, 1981.

7. J. E. HOPCROFT, J. T. SCHWARTZ, AND M. SHARIR, "Efficient Detection of Intersections Among Spheres," in press.
8. B. MOSSBERG, *in* "Symposium on Cyber 205 Applications," Colorado State Univ. Fort Collins, Colo., 1982.
9. M. RABIN, "Algorithms and Complexity," Academic Press, New York, 1976.
10. P. SCHOFIELD, *Comput. Phys. Commun.* **5** (1973), 17.
11. L. VERLET, *Phys. Rev.* **159** (1967), 98.