

ORNL/TM-12890

Engineering Physics and Mathematics Division

Mathematical Sciences Section

**A NEW SHARED-MEMORY PROGRAMMING PARADIGM FOR  
MOLECULAR DYNAMICS SIMULATIONS ON THE INTEL  
PARAGON**

E.F. D'Azevedo  
C.H. Romine

Mathematical Sciences Section  
Oak Ridge National Laboratory  
P.O. Box 2008, Bldg. 6012  
Oak Ridge, TN 37831-6367

Date Published: March 1995

Research supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy

Prepared by the  
Oak Ridge National Laboratory  
Oak Ridge, Tennessee 37831  
managed by  
Martin Marietta Energy Systems, Inc.  
for the  
U.S. DEPARTMENT OF ENERGY  
under Contract No. DE-AC05-84OR21400

## Contents

1	Introduction . . . . .	1
1.1	DOLIB (Distributed Object Library) . . . . .	1
1.2	SOTON_PAR . . . . .	2
2	Overview of Parallelization of SOTON_PAR with DOLIB . . . . .	3
2.1	Geometric Hashing . . . . .	3
2.2	DOLIB implementation of <code>movout</code> . . . . .	4
2.3	Force evaluation . . . . .	6
2.4	Dynamic Load Balancing . . . . .	7
2.5	Shifted Force Potential . . . . .	8
3	Parallel Performance . . . . .	8
4	Appendix . . . . .	12
4.1	subroutine <code>movout</code> . . . . .	12
4.2	subroutine <code>adjustxyz</code> . . . . .	13
4.3	subroutine <code>hashxyz</code> . . . . .	15
4.4	subroutine <code>setxlist</code> . . . . .	19
4.5	subroutine <code>setuplist</code> . . . . .	22
4.6	subroutine <code>set4list</code> . . . . .	23
4.7	subroutine <code>copyxyz</code> . . . . .	25
4.8	subroutine <code>force</code> . . . . .	29
4.9	subroutine <code>forceall</code> . . . . .	31
4.10	subroutine <code>forcecol</code> . . . . .	33
4.11	subroutine <code>forcecal</code> . . . . .	45
5	References . . . . .	51

## List of Figures

- 2.1 Top view of columns used in processing column  $(i, j)$ . Only two new columns are needed in processing column  $(i + 1, j)$ . . . . . 7

# A NEW SHARED-MEMORY PROGRAMMING PARADIGM FOR MOLECULAR DYNAMICS SIMULATIONS ON THE INTEL PARAGON

E.F. D'Azevedo

C.H. Romine

## Abstract

This report describes the use of shared memory emulation with DOLIB (Distributed Object Library) to simplify parallel programming on the Intel Paragon. A molecular dynamics application is used as an example to illustrate the use of the DOLIB shared memory library. SOTON\_PAR, a parallel molecular dynamics code with explicit message-passing using a Lennard-Jones 6-12 potential, is rewritten using DOLIB primitives. The resulting code has no explicit message primitives and resembles a serial code. The new code can perform dynamic load balancing and achieves *better* performance than the original parallel code with explicit message-passing.

## 1. Introduction

This report describes the use of shared memory emulation provided by DOLIB (Distributed Object Library) [3] for parallel programming of large-scale Molecular Dynamics (MD) codes on the Intel Paragon distributed memory supercomputer. SOTON\_PAR [7], a parallel MD code that uses explicit message passing and Lennard-Jones atoms (6-12 potential), is rewritten using DOLIB shared memory primitives. The new MD prototype code can perform dynamic load balancing and achieves *better* performance than the original code that uses spatial decomposition. This report may also serve as a guide for scientists or programmers considering the use of DOLIB shared memory emulation in more sophisticated molecular dynamics simulations or particle-in-cell methods.

While this prototype code uses a simple Lennard-Jones 6-12 potential, more complicated force evaluation (such as including angular (three-body) forces for silicon and torsional (four-body) forces for organic polymers or proteins) can easily be added using the DOLIB shared memory framework. DOLIB's dynamic load balancing capability is especially attractive for more demanding MD simulations.

### 1.1. DOLIB (Distributed Object Library)

DOLIB (Distributed Object Library) is a set of FORTRAN and C callable routines to emulate shared memory on distributed-memory environments such as Intel multiprocessors and PVM clusters of workstations.

DOLIB supports runtime dynamic creation and destruction of one-dimensional arrays. Explicit `gather` and `scatter` operations provide access to array elements. DOLIB is portable since no language extension is introduced and no preprocessor, compiler or operating system support is required. DOLIB provides an atomic accumulate operation `axpby` ( $y(index(:)) \leftarrow \alpha * x(:) + \beta * y(index(:))$ ), where  $x$  is a local array,  $y$  is a globally shared array, and  $index$  is an array of indices for  $y$ . `axpby` is intended for use in finite element matrix assembly, with no need for explicit `lock/unlock`. Note that  $\alpha = 1$ ,  $\beta = 0$  corresponds to a `scatter` operation. DOLIB also provides automatic caching of *read-only* data to reduce

message traffic. However, the programmer has the responsibility of flushing the cache to maintain coherency during concurrent updates.

A global array in DOLIB is stored as fixed-size pages in a block wrapped mapped fashion across all processors. DOLIB translates requests for remote data (`gather`) or update (`scatter`) into the appropriate message sent to the “owner” processor of that data page. These DOLIB requests are then serviced by the IPX [6] message system.

DOLIB supports a *generalized* atomic update operation (`axpbyz`),

$$\begin{aligned} z(:) &\leftarrow y(\text{index}(:)), \\ y(\text{index}(:)) &\leftarrow \alpha * x(:) + \beta * y(\text{index}(:)) \\ x(:) &\leftarrow z(:) \end{aligned}$$

Operation `axpbyz` can also be used to implement a vector of counting semaphores.

## 1.2. SOTON\_PAR

The original version of SOTON\_PAR<sup>1</sup> MD code ran on the Intel iPsc/2. The version we obtained was substantially modified by David Walker to run efficiently on Intel Paragon multiprocessors. SOTON\_PAR models short range inter-atomic interactions by using a link-cell (geometric hashing) algorithm where all particles are hashed into three-dimensional  $N_b \times N_b \times N_b$  bins. The bin size is at least the cut-off distance ( $r_c$ ) used in the short-range force evaluation. SOTON\_PAR exploits the symmetry of Newton’s Third Law by examining atoms in only 13 (instead of 26) neighboring cells. It uses a spatial decomposition [8] and requires message exchanges with neighbor processors at each time step as atoms migrate across processor boundaries. Many of the MD algorithms used in SOTON\_PAR are described in the book *Computer Simulation of Liquids* [1]. The code also uses a “shifted-force” [9] Lennard-Jones 6-12 potential to avoid a discontinuity at the

---

<sup>1</sup>The source code for SOTON\_PAR is available from the CCP5 archive at [ftp://ftp.dl.ac.uk/ccp5/SOTON\\_PAR/mdp11i3cu.master](ftp://ftp.dl.ac.uk/ccp5/SOTON_PAR/mdp11i3cu.master)

cut-off distance.

The code consists of two parts: the “host” code runs on the service partition and controls the I/O, allocation and loading of nodes etc.; the “worker” code runs on the compute nodes and performs most of the work in geometric hashing and force evaluations.

The code assumes an  $N_c \times N_c \times N_c$  FCC (Face Centered Cubic) lattice scaled to the canonical unit cube  $[-0.5, 0.5] \times [-0.5, 0.5] \times [-0.5, 0.5]$ , with a total of  $N = 4N_c^3$  atoms. A periodic boundary condition is imposed on all sides. A simple Verlet leap-frog time stepping scheme is used to update the particle positions.

## 2. Overview of Parallelization of SOTON\_PAR with DOLIB

We have written a new parallel MD prototype code based on SOTON\_PAR using DOLIB shared memory primitives. The new code shares much of the overall structure of SOTON\_PAR with most of the host code for performing I/O executed by node 0. We believe the new code is much easier to write and understand without the complexities of explicit message passing code. We have also added several enhancements: (1) initialize with Boltzmann distribution for faster convergence to equilibrium, (2) a more robust parallel cold-start mechanism even for very large configurations, (3) an option for dynamic load balancing, (4) an option for using double precision for force computation.

We shall discuss two of the most time consuming computational kernels in SOTON\_PAR: `movout`, which performs geometric hashing, and `force`, which performs the force evaluations.

### 2.1. Geometric Hashing

The link-cell method [4] is commonly used to speed up MD force calculations by minimizing the number of neighboring atoms that must be checked for possible interaction (based on a cut-off distance of  $r_c$ , beyond which particle interaction is assumed to be negligible). At each time step, all the atoms are *hashed* into

three-dimensional bins or cells of side length at least  $r_c$ . Each particle can thus interact only with atoms within the same bin or the 26 surrounding bins. Hashing the atoms can be easily parallelized and requires only  $O(N)$  work. Only 13 of the neighboring bins need to be examined if Newton's Third Law ( $f_{ij} = -f_{ji}$ ) is applied. The link-cell method is memory efficient, requiring only  $O(N_b^3)$  storage for the 3-D bins, 9 real vectors of length  $N$  ( $x, y, z, v_x, v_y, v_z, f_x, f_y, f_z$  for the positions, velocities and forces, respectively) and an integer vector of length  $N$  for storing the linked list.

Another common technique is to construct and maintain for *each* atom, a list of neighboring atoms [10]. This list is updated every few time steps. The advantage provided by the neighbor list is that once the list is built, examining it for possible interaction is much quicker than checking all other atoms in the neighboring bins. However, the neighbor list for each atom commonly grows quite long (70 or more atoms per list), and the resulting high cost in memory is prohibitive for very large scale MD calculations.

SOTON\_PAR is based on a spatial decomposition and uses the link-cell method. Each processor is responsible for a spatial region and the corresponding subset of 3-D bins. SOTON\_PAR uses message passing code to exchange atoms with neighbor processors as atoms migrate across processor boundaries. A highly non-uniform distribution of particles may result in a serious load-imbalance.

## 2.2. DOLIB implementation of movout

The link-cell algorithm in SOTON\_PAR constructs a linked list of atoms for each bin. We implement the link-cell method by performing a reordering or renumbering so that all atoms in a bin are contiguously numbered, *e.g.*, if each bin has 10 atoms, then after the reordering, atoms 1 to 10 are contained in the first bin, atoms 11 to 20 are contained in the second bin, *etc.* The reordering permits the use of the most efficient DOLIB contiguous block `gather/scatter` operations on long vectors.

Two passes are required for the geometric hashing and reordering. The first



pass performs the geometric hashing and stores the result of the mapping. For each bin, we compute the number of atoms to be assigned. We then set up and allocate storage for each bin using a pointer array `xlist`. The second pass then performs the actual reordering and data movement. Note that vectors  $(f_x, f_y, f_z)$  are used as temporary storage and will be cleared again for force computation.

The geometric hashing and reordering is performed in several routines:

**adjustxyz:** After a time step, particles close to the boundary might exit the canonical box  $[-0.5, 0.5] \times [-0.5, 0.5] \times [-0.5, 0.5]$ . We implement a periodic boundary condition by adjusting the  $(x, y, z)$  coordinate to reintroduce the particle at the opposite face. Hence,

$$\begin{aligned} \text{if } (x > 0.5) \quad x &= x - 1.0 \\ \text{if } (x < -0.5) \quad x &= x + 1.0 \end{aligned}$$

**hashxyz:** Geometric hashing is performed as a simple divide and integer truncation operation. An atom with coordinates  $(x, y, z)$  will be hashed to 3-D bin  $(i, j, k)$  by

$$\begin{aligned} i &= 1 + \text{int}((x - (-0.5))/r_c) \\ j &= 1 + \text{int}((y - (-0.5))/r_c) \\ k &= 1 + \text{int}((z - (-0.5))/r_c) \end{aligned}$$

The result of the hashing is stored in array `imap` for use in the second pass. The array `imap` requires  $N_b^3$  words of memory.

**setxlist:** We use a pointer array `xlist` to mark for each bin, the beginning position (or label) of each contiguous block of atoms assigned to the bin.

Let  $m_i$  be the number of particles assigned to the  $i^{\text{th}}$  bin, then  $xlist(1) = 1$ ,  $xlist(2) = 1 + m_1$ ,  $xlist(3) = xlist(2) + m_2$ ,  $\dots$ ,  $xlist(k+1) = xlist(k) + m_k$ , *etc.* The array `xlist` requires  $N_b^3$  words of memory.

**setuplist:** We shall use `xlist` as pointer to the next available storage position for each bin. Thus as we assign particles into bins, `xlist` has to be incremented in parallel with an *atomic* operation. We avoid the use of inefficient `lock/unlock` by using the DOLIB `axpbyz` operation.

**copyxyz:** Once the assignment mapping is determined, vectors  $(x, y, z)$  and  $(v_x, v_y, v_z)$  are permuted using  $(f_x, f_y, f_z)$  as temporary storage. Vectors  $(f_x, f_y, f_z)$  are then cleared again before the force computation. The arrays  $(x, y, z)$ ,  $(v_x, v_y, v_z)$  and  $(f_x, f_y, f_z)$  require  $9N$  words of memory for  $N$  atoms. If the forces  $(f_x, f_y, f_z)$  are to be computed in double precision,  $12N$  words are required.

**adjustxlist:** Finally we reset `xlist` to point to the beginning position of each contiguous block.

Note that as the MD system converges to an equilibrium state, most of the particles will be hashed into the same bins to which they belonged in the previous time step, thus requiring only a small amount of data movement.

In Section 3 we list the proportion of time spent in each routine.

### 2.3. Force evaluation

We take advantage of Newton's 3rd Law ( $f_{ij} = -f_{ji}$ ) to compute a force interaction only once for each atom pair. This means only 13 (instead of 26) neighboring bins must be examined. Lomdahl [5] describes the selection of 13 neighbors as a particular "3D interactions path", though in fact *any* stencil of 13 neighboring bins can be chosen, if consistently used.

For simplicity, we consider a two-dimensional partition of the computation. Thus, we group all force interactions for each column of  $N_b$  cells of the 3D mesh of bins as one unit. The  $N_b \times N_b$  columns are then *block* partitioned and assigned to individual processors. Because all atoms in each bin are contiguously ordered (see Section 2.2), DOLIB's efficient contiguous block `gather/scatters` can be used. Moreover, we exploit data reuse by selecting the 13 neighboring cells to be

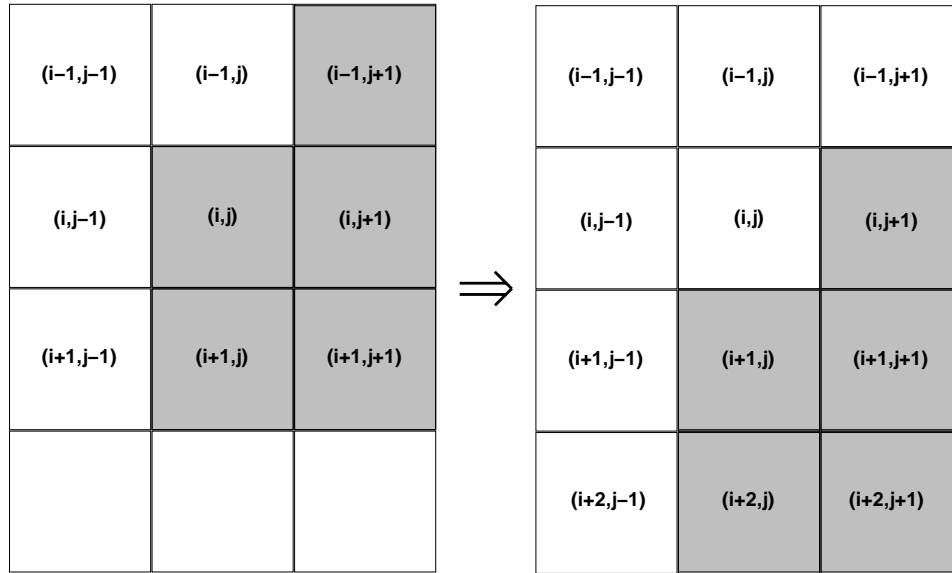


Figure 2.1: Top view of columns used in processing column  $(i, j)$ . Only two new columns are needed in processing column  $(i+1, j)$ .

in 5 neighboring columns (see Figure 2.1). To process column  $(i, j)$  we require data from neighboring columns  $(i-1, j+1)$ ,  $(i, j+1)$ ,  $(i+1, j)$ ,  $(i+1, j+1)$ ; to process the next column  $(i+1, j)$ , data in columns  $(i+1, j)$ ,  $(i, j+1)$ ,  $(i+1, j+1)$  can be reused. Only data for two new columns  $(i+2, j)$  and  $(i+2, j+1)$  must be brought in. This organization reduces the amount of communication required for `gather/scatter` operations.

Another advantage of this reorganization is that force evaluation in each of the bins requires no further communication. Automatic thread parallelization by the Intel MP node Paragon compiler is thus simplified.

## 2.4. Dynamic Load Balancing

We include an option to estimate a work measure for each column, using this to distribute columns to processors to avoid load imbalance. One simple measure of work load is the total number of atoms in the column. For a uniform distribution of atoms in the domain, this is generally sufficient to attain good load-balance. However, for nonuniform distributions, a more reasonable work measure for each

bin is to count the total number of possible atom-atom interactions among other particles in the 13 neighboring bins. Both of these work estimates are provided as options to the load-balancing routine.

The load balancing strategy then computes the overall total and average amount of work for each processor, assigning columns to processors to satisfy this average using a greedy algorithm. Future versions will provide a more sophisticated bin-packing algorithm. However, for the simple Lennard-Jones fluid simulations, each bin has approximately the same number of atoms and induces almost no load imbalance.

### **2.5. Shifted Force Potential**

SOTON\_PAR implements a shifted force Lennard-Jones 6-12 potential [1, page 145] where a small linear term is added to the potential so that it is continuous and its derivative is zero at the cutoff distance. The code also computes the virial and potential energy along with force calculations. Each atom-pair takes 8 flops and 1 compare to determine whether they are within the cut-off distance. If they are, then another 29 flops, 2 divides and 1 square root operation are further required. In our experience, the accumulation into total potential energy may encounter catastrophic cancellation that can yield as few as 4 significant digits of precision. We have implemented an option using separate double precision variables  $v_{pos}$  and  $v_{neg}$  for accumulating positive and negative potential values, which reduces the amount of numerical cancellation and produces more consistent results as the number of processors is varied.

## **3. Parallel Performance**

Our parallel codes were tested on an Intel MP node Paragon system. Each MP node contains 3 CPUs and at least 64MBytes of memory in a local shared memory configuration. By default, one CPU is a dedicated message co-processor. The second CPU runs the main computational thread. The third CPU can be utilized by

automatic thread parallelization by the Paragon MP Fortran compiler. The MP node can also be configured in “turbo” mode where all three CPUs are dedicated for computation but with some degradation in communication performance. All our MD computations were done in single precision in non-turbo mode, although we have also implemented a double precision option for force computations.

We tested our code with a benchmark problem described in Plimpton [8, page 23] for a Lennard-Jones 6-12 potential with reduced density  $\rho = 0.8442$ , and reduced temperature  $T = 0.72$ . The system is initialized with a *fcc* lattice and randomized velocities chosen from a Boltzmann distribution. The integration time step is 0.00462 in reduced units, and cut-off distance is  $r_c = 2.5\sigma$ . Plimpton estimates there are about 55 neighbors interacting with each atom at every time step (out of about 177 atom-pairs examined). If we count a divide as 5 flops and a square root also as 5 flops, then (from Section 2.5) each atom requires approximately  $9 * 177 + 44 * 55 = 4013$  flops for force evaluation. Updating the velocities, positions and accumulation of total kinetic energy in `movea` and `moveb` requires another 30 flops per atom.

Table 3.1 shows averaged run times per time step for a 500,000 atom simulation ( $50 \times 50 \times 50$  lattice). Initialization and setup time are excluded. We achieved faster run times without the overhead of computing work measure and dynamic load balancing, since the problem is already well-balanced. The serial run was performed using the same “parallel” code on a single processor. Note that no messages were generated in the DOLIB `gather/scatter` since on a single processor these are translated into memory copies.

For the 500,000 atom problem, our code achieved a speed of 0.175ms/atom per time step on a single processor (with parallel threads) with no messages generated. About 75% of the overall time (per time step) was spent in `force` computation and about 20% in performing reordering in `movout`. Within `movout`, about 42% of time was spent in performing 3-D bin hashing (first pass) in `hashxyz`, about 17% in allocating storage with DOLIB `axpbyz` operation in routine `setuplist` and about 35% in performing (second pass) actual reordering of vectors ( $x, y, z$ ,

$v_x, v_y, v_z$ ) in routine `copyxyz`. On 64 processors with parallel threads, the speed decreased to about 0.31ms/atom per processor per time step. About 60% was in `force` and 30% in `movout`. Within `movout`, about 27% was in `hashxyz`, 21% in `setuplist` and about 38% was in `copyxyz`. Note however, there is message passing and servicing of remote memory requests even within the `force` computations.

By comparison, the original SOTON\_PAR host/node codes required about 17.1 sec on 8 processors for a 256,000 atom ( $40 \times 40 \times 40$  lattice) with no parallel threads. This is about 0.534ms/atom/processor. Our code (with no parallel threads) takes about 10.4 sec per time step. This is about 0.333ms/atom. With parallel threads enabled, the average time reduced to 7.7sec per step, or 0.25ms/atom/processor.

On a much larger problem with 32 million atoms ( $200 \times 200 \times 200$  lattice) our code takes about 205.3sec on 32 processors and 103.7sec on 64 processors, which is about 0.207ms/atom per processor. This yields an approximate overall rate (including message passing overhead) of 19.5MFlop/sec for each node (with parallel threading enabled). Although we used single precision in our codes, our run times compare quite favorably with the times of 0.24ms/atom/processor on the Intel Delta [8] and 0.26ms/atom/processor [2] (performed in double precision) on a 1024 processor CM-5<sup>2</sup> both using a spatial decomposition and linked-cell method.

## Acknowledgments

The authors take this opportunity to express appreciation to Bob Marr, Ron Peierls and Joe Pasciak for the IPX package, which simplified the development of DOLIB. We also thank Bill Shelton and David Walker for their insight and advice on Molecular Dynamics Simulations.

---

<sup>2</sup>The code used on the CM-5 computes about 7,000 flops per atom.

Table 3.1: Averaged runtimes (in sec) per timestep for 500,000 nodes problem.

Processor	1 thread	2 threads
1	137.5	95.4
2	73.1	51.8
4	37.6	27.2
8	19.7	13.2
16	10.3	7.1
32	5.6	4.3
64	3.3	2.4

## 4. Appendix

In this appendix, we list the RATFOR source code for some of the key routines used in movout permutation and force computations.

### 4.1. subroutine movout

```
#include "stdinc.h"

        /* routine to setup hash table and other data structures for
           force calculations */

subroutine movout()
#include "globals.h"
#include "parallel.h"
{
    logical useminspace;
#ifdef USE_MINSPACE
    parameter(useminspace=TRUE);
#else
    parameter(useminspace=FALSE);
#endif

    STRING name; STRING ctype;
    integer psize, gsize, blocksize;

        /* ===== start execution ===== */

#ifdef USE_PROFILE
    call profstart('movout');
#endif

    if (useminspace) {
        /* disable everything */

        call dodisable(0);

        call dogsync();

        call dodestroy( IGfxyz );

        call dogsync();

        name = 'imap(natoms)' // char(0);
        ctype = 'integer' // char(0);
        gsize = natoms; psize = pagesize; blocksize = 1;

        call dodeclare( IGimap, name, gsize, ctype, psize, blocksize);

        call dogsync();

    };
};
```



```
/* make sure all xyz coordinates are within canonical box */
call adjustxyz();

/* first pass in hashing to count number of atoms in each bin cell */
call hashxyz();

/* setup up pointer list */
call setxlist();

/* setup up numlist */
call setuplist();

/* second pass in hashing to copy xyz coordinates
   into the 'permuted' array */
call copyxyz();

/* reset xlist */
call adjustxlist();

#ifdef USE_PROFILE
    call profend('movout');
#endif

    return;
}

end
```

## 4.2. subroutine adjustxyz

```
#include "stdinc.h"

/* perform adjustment of xyz to be within canonical box of
   [xmin,xmax] x [ymin,ymax] x [zmin,zmax] */

#define logdev 16

subroutine adjustxyz()
#include "globals.h"
#include "parallel.h"
{
    intrinsic int,max,min,abs,sqrt;

    integer ipage,npages,ipagestr;
    integer ni,i,istrt,iend,ysize;

/* simulation in box [-0.5,0.5] */

    real xmin,xmax,xadjust;
    real ymin,ymax,yadjust;
    real zmin,zmax,zadjust;
```

```
parameter(xmin=-0.5,xmax=0.5,xadjust=(xmax-xmin));
parameter(ymin=-0.5,ymax=0.5,yadjust=(ymax-ymin));
parameter(zmin=-0.5,zmax=0.5,zadjust=(zmax-zmin));

real one,zero;
parameter(one=1.0,zero=0.0);

real xyz(3,pagesize);

#define rx(i) xyz(1,i)
#define ry(i) xyz(2,i)
#define rz(i) xyz(3,i)

logical isxok,isyok,iszok, isallok, isformatted;

/* ===== start execution ===== */

#ifdef USE_PROFILE
call profstart('adjustxyz');
#endif

call dogsync();

npages = int( natoms/ pagesize ) + 1;
ipagestrt=myid;

call dodisable( IGxyz );

/* scatter operation */
call openrbuf( IGxyz, one, zero );

doloop4(ipage,ipagestrt,npages,nproc) {
  istr = 1 + ipage*pagesize;
  iend = min(natoms, istr+pagesize-1);
  isize = iend - istr + 1;
  if (isize <= 0) { break; };

  call dowait(dobrgather( IGxyz, 3*isize,
                        idxyz(1,istr), xyz ));

  if (IDEBUG >= 3) {
    write(logdev,*) 'adjustxyz: xyz before update ';
    write(logdev,9001) (istr+(i-1),rx(i),ry(i),rz(i),i=1,isize);
    format((1x,i6,',',3(1x,1pe14.5)));
  };

  9001
  doloop(i,1,isize) {

    isallok = TRUE;

    if (rx(i) < xmin) { rx(i) = rx(i) + xadjust; isallok = FALSE;};
    if (rx(i) > xmax) { rx(i) = rx(i) - xadjust; isallok = FALSE;};

    if (ry(i) < ymin) { ry(i) = ry(i) + yadjust; isallok = FALSE;};
    if (ry(i) > ymax) { ry(i) = ry(i) - yadjust; isallok = FALSE;};

    if (rz(i) < zmin) { rz(i) = rz(i) + zadjust; isallok = FALSE;};
    if (rz(i) > zmax) { rz(i) = rz(i) - zadjust; isallok = FALSE;};
  };
};
```

```
if (isallok) { next; };

/* ----- */

ni = istr + (i-1);
if (IDEBUG >= 2) {

    isxok = (xmin <= rx(i)) & (rx(i) <= xmax);
    isyok = (ymin <= ry(i)) & (ry(i) <= ymax);
    iszok = (zmin <= rz(i)) & (rz(i) <= zmax);
    isallok = isxok & isyok & iszok;

    if (!isallok) {
        write(*,*) ' ** adjustxyz: invalid xyz after adjustment';
        write(*,*) 'ni,rx,ry,rz',ni,rx(i),ry(i),rz(i);

        isformatted = TRUE;
        call outconf( isformatted ); /* formatted dump */
    };

    ASSERT(isallok,
        ' ** adjustxyz: invalid xyz after adjustment',ni);
}; /* if (IDEBUG >= 2) */

/* buffer the update */
call storebrbuf( IGxyz,3, idxyz(1,ni), xyz(1,i) );

}; /* end do i */

}; /* end do ipage */

call closerbuf(IGxyz);

call dogsync();

#ifdef USE_PROFILE
    call profend('adjustxyz');
#endif

return;
}

end
```

### 4.3. subroutine hashxyz

```
#include "stdinc.h"

/* perform hashing of xyz to count number of atoms in each bin */

#define USE_WCACHE 1
```

```
subroutine hashxyz()
#include "globals.h"
#include "parallel.h"
{
    intrinsic int,max,min,abs,sqrt;

    integer ipage,npages,ipagestrt;
    integer i,istrt,iend,ysize;
    integer ix,iy,iz;

    integer ival,gsize,psize;

    integer ilist(pagesize), mcount;

    logical usedivide;
#ifdef USE_DIVIDE
    parameter(usedivide=TRUE);
#else
    parameter(usedivide=FALSE);
#endif

    logical usewcache;
#ifdef USE_WCACHE
    parameter(usewcache=TRUE);
#else
    parameter(usewcache=FALSE);
#endif

    logical usesort;
#ifdef USE_SORT
    parameter(usesort=TRUE);
#else
    parameter(usesort=FALSE);
#endif

    integer ibin,icount;
    logical issame;

    /* simulation in box [-0.5,0.5] */

    real xmin,xmax,xadjust;
    real ymin,ymax,yadjust;
    real zmin,zmax,zadjust;

    parameter(xmin=-0.5,xmax=0.5,xadjust=(xmax-xmin));
    parameter(ymin=-0.5,ymax=0.5,yadjust=(ymax-ymin));
    parameter(zmin=-0.5,zmax=0.5,zadjust=(zmax-zmin));

    real rxi,ryi,rzi, rcutinv;
    real xyz(3,pagesize);

#define rx(i) xyz(1,i)
#define ry(i) xyz(2,i)
#define rz(i) xyz(3,i)

    integer list(pagesize), ivalue(pagesize);

    /* ===== start execution ===== */
}
```

```
#ifdef USE_PROFILE
    call profstart('hashxyz');
#endif

call doenable( IGxyz );
call dodisable( IGncount );

/* zero out global array ncount(:) */
ival = 0;
gsize = nbinx*nbiny*nbinz; psize = pagesize;
call givecfill( IGncount, ival, gsize, psize );

if (!(usewcache)) {
    call ifill( pagesize, 1, ival, 1 );
};

if (!(usedivide)) {
    rcutinv = 1.0/rcut;
};

npages = int( natoms/ pagesize ) + 1;
ipagestrt=myid;

if (usewcache) { call openibuf( IGncount, 1,1); };

doloop4(ipage,ipagestrt,npages,nproc) {
    istr = 1 + ipage*pagesize;
    iend = min(natoms, istr+pagesize-1);
    isize = iend - istr + 1;
    if (isize <= 0) { break; };

    call dowait(dobrgather( IGxyz, 3*isize,
        idxyz(1,istr), xyz ));

    doloop(i,1,isize) {

        if (usedivide) {
            ix = 1+int((rx(i) - xmin)/rcut );
            iy = 1+int((ry(i) - ymin)/rcut );
            iz = 1+int((rz(i) - zmin)/rcut );
        }
        else {
            ix = 1+int((rx(i) - xmin)*rcutinv );
            iy = 1+int((ry(i) - ymin)*rcutinv );
            iz = 1+int((rz(i) - zmin)*rcutinv );
        };

        if (IDEBUG >= 2) {
            if (ix < 1) { ix = ix + nbinx; };
            if (ix > nbinx) { ix = ix - nbinx; };

            if (iy < 1) { iy = iy + nbiny; };
            if (iy > nbiny) { iy = iy - nbiny; };
        }
    }
}
}
```

```
    if (iz < 1) { iz = iz + nbinz; };
    if (iz > nbinz) { iz = iz - nbinz; };
};

list(i) = idxncount(ix,iy,iz);

}; /* end do i */

/* save hash results into imap */
call dobiscatter( IGimap, isize, idximap(istrt), list );

/* --- accumulated into hash table */

if (usewcache) {

/*
  attempt compression, instead of adding
  ncount(:) by 1 icount times,
  simply add by icount only once
*/
  if (usesort) {
    call shell( list, isize );
  };

  mcount = 0;

  ibin = -1; icount = 0;
  doloop(i,1,isize) {
    issame = (ibin == list(i));
    if (issame) { icount = icount + 1; }
    else {
      if (icount > 0) { /* flush out */
        /* call storeibuf( IGncount, 1, ibin, icount ); */

        mcount = mcount + 1;
        ilist(mcount) = ibin;
        ivalue(mcount) = icount;
      };
      ibin = list(i); icount = 1;
    };
  }; /* end doloop */

  if (icount > 0) { /* flush out last remaining part */
    /* call storeibuf( IGncount, 1, ibin, icount ); */

    mcount = mcount + 1;
    ilist(mcount) = ibin;
    ivalue(mcount) = icount;
  };

  if (mcount > 0) {
    call storeibuf( IGncount, mcount, ilist, ivalue );
  };
}
else {

  /* by pass intermediate buffering */

  call doiexpby( IGncount, isize, list, ivalue, 1,1 );
};
```

```
}; /* end do ipage */

if (usewcache) { call closeibuf( IGncount ); };

call dogsync();

#ifdef USE_PROFILE
call profend('hashxyz');
#endif

return;
}
end
```

#### 4.4. subroutine setxlist

```
#include "stdinc.h"

#define logdev 16

/* given the ncount global array is complete, compute
the xlist pointer array */

subroutine setxlist()
#include "globals.h"
#include "parallel.h"

{
intrinsic max,min;

#define ndim (8*pagesize)

integer istrt,iend, isize;
integer i,ip, nbins;
integer ncount(ndim),xlist(ndim);

#define maxproc (2*1024)
integer ipstart(maxproc+1);
integer mycount(maxproc), icount;

integer iproc, ibstrt,ibend,ibsize;

/* for simplicity, let processor 0 handle it,
may not scale well for very large problems */

/* ===== start execution ===== */

#ifdef USE_PROFILE
call profstart('setxlist')
#endif
#endif
```

```
/* each processor compute part of the total count */

nbinz = nbinx*nbiny*nbinz;
call partition( nbinz, nproc,myid, ibstrt,ibend);
ibsize = ibend - ibstrt + 1;

doloop(iproc,1,nproc) { mycount(iproc) = 0; };

call doenable( IGncount );
iproc = myid + 1;

doloop4(istrt,ibstrt,ibend,ndim) {
    iend = min(ibend, istrt+ndim-1);
    isize = (iend - istrt + 1);
    if (isize <= 0) { break; };

    call dowait( dobigather( IGncount,isize,istrt,ncount ));
    icount = 0;
    doloop(i,1,isize) {

        icount = icount + ncount(i);
    };
    mycount(iproc) = mycount(iproc) + icount;
    ASSERT( mycount(iproc) > 0,
        ' ** setxlist: possible integer overflow ', mycount(iproc));
}; /* end do istrt */

call dogsync();
GISUM( mycount, nproc );

/* double check */
if (IDEBUG >= 2) {

    if (myid == 0) {
        write(logdev,*) 'mycount(:)';
        write(logdev,'(5(1x,i8))') (mycount(iproc),iproc=1,nproc);
    };

    doloop(iproc,1,nproc) {
        call partition( nbinz, nproc,(iproc-1), ibstrt,ibend );

        icount = 0;
        doloop4(istrt,ibstrt,ibend,ndim) {
            iend = min(ibend, istrt+ndim-1);
            isize = (iend - istrt + 1);
            if (isize <= 0) { break; };

            call dowait( dobigather( IGncount, isize, istrt, ncount ));

            doloop(i,1,isize) {
                ASSERT( ncount(i) > 0,
                    ' ** setxlist: ncount(i) <= 0',ncount(i) );
                icount = icount + ncount(i);
            };
        };

        ASSERT( icount == mycount(iproc),
            ' ** setxlist: icount != mycount(iproc)',mycount(iproc));

    }; /* end do iproc */
};
```



```
/*
   set up xlist so that

   xlist(1) == 1,
   xlist(2) = xlist(1) + ncount(1),...,
   xlist(k+1) = xlist(k) + ncount(k)...

   at beginning of call to copyxyz

*/

ipstart(1) = 1;
doloop(iproc,1,nproc) {
    ipstart(iproc+1) = ipstart(iproc) + mycount(iproc);
};

call dodisable( IGxlist );

call partition( nbins,  nproc,myid,  ibstrt,ibend );

iproc = myid+1;
ip = ipstart(iproc);
doloop4(istrt,ibstrt,ibend,ndim) {
    iend = min(ibend,  istrt+ndim-1);
    isize = iend - istrt + 1;
    if (isize <= 0) { break; };

    call dowait( dobigather( IGncount, isize,
                           (istrt), ncount));

    doloop(i,1,isize) {
        xlist(i) = ip;
        ip = ip + ncount(i);
    };

    call dobiscatter( IGxlist, isize, istrt, xlist );
}; /* end do istrt */

call dogsync();

if (IDEBUG >= 2) {
    if (myid == 0) {
        ip = 1;
        doloop4(istrt,1,nbins,ndim) {
            iend = min(nbins, istrt+ndim-1);
            isize = (iend - istrt + 1);
            if (isize <= 0) { break; };

            call dowait( dobigather( IGncount, isize, istrt, ncount ));
            call dowait( dobigather( IGxlist, isize, istrt, xlist ));

            if (IDEBUG >= 3) {
                write(logdev,*) 'ncount(:) ';
```

```
        write(logdev,9001) (ncount(i),istrt+(i-1),i=1, isize);

        write(logdev,*) 'xlist(:) ';
        write(logdev,9001) (xlist(i),istrt+(i-1),i=1, isize);
9001      format(4( i6,'(',i6,')' ));
        };

        doloop(i,1, isize) {
            if (xlist(i) != ip) {
                write(logdev,*) 'ni, ip, xlist ',
                    (istrt+(i-1)),ip,xlist(i);
            };
            ip = ip + ncount(i);
        };

        }; /* end do istrt */

        }; /* end if (myid) */
    }; /* end if (IDEBUG) */

    call dogsync();

#ifdef USE_PROFILE
    call profend('setxlist')
#endif

    return;
}

end
```

#### 4.5. subroutine setuplist

```
#include "stdinc.h"

#define SETLIST_OPTION 4

subroutine setuplist()
#include "globals.h"
#include "parallel.h"
{

    if (SETLIST_OPTION == 1) {

        /* use long vector gathers */

        call set1list();
    }
    else if (SETLIST_OPTION == 2) {

        /* use short vector gathers, take advantage of pagesize */

        call set2list();
    }
    else if (SETLIST_OPTION == 3) {
```

```
        /* take advantage of xlist(:) */
        call set3list();
    }
else {
    /* default action */

    /* perform compression */

    call set4list();
};

}
end
```

#### 4.6. subroutine set4list

```
#include "stdinc.h"

    /* use a long vector gather */

subroutine set4list()
#include "globals.h"
#include "parallel.h"
{

    integer ndim;
    parameter(ndim=8*(pagesize));

    integer list(ndim), ipos(ndim), numlist(ndim);

    integer istr, iend, isize;
    integer ibstart, ibend, ibsize, gsize;

    integer maxmcount; parameter(maxmcount=ndim);
    integer icount, mcount, ibin, ni, ip, i, j;

    integer binlist(maxmcount), countlist(maxmcount);
    integer idest(ndim), nilist(ndim);

    logical issame;

    /* save storage */
    equivalence (nilist, binlist);

    /* ===== start execution ===== */

#ifdef USE_PROFILE
    call profstart('set4list');
#endif

    gsize = natoms;

    call dodisable( IGxlist );
```

```
call partition( gsize, nproc,myid, ibstart,ibend );
doloop4( istrtr, ibstart,ibend,ndim) {
  iend = min(ibend, istrtr + ndim-1);
  isize = (iend - istrtr + 1);
  if (isize <= 0) { break; };

  call dowait(dobigather( IGimap, isize, idximap(istrtr), list ));

  /* attempt compression, instead of adding one k times,
     try to add k only once */

  mcount = 0; icount = 0; ibin = -1;
  doloop(i,1,isize) {
    issame = (ibin == list(i));
    if (issame) { icount = icount + 1; }
    else {
      if (icount > 0) { /* store these updates */
        mcount = mcount + 1;
        ASSERT( mcount <= maxmcount,
          ' ** set4list: mcount > maxmcount ', mcount );

        binlist(mcount) = ibin;
        countlist(mcount) = icount;
      };
      ibin = list(i); icount = 1;
    };
  };

  /* handle last remaining case */

  if (icount > 0) { /* store these updates */
    mcount = mcount + 1;
    ASSERT( mcount <= maxmcount,
      ' ** set4list: mcount > maxmcount ', mcount );

    binlist(mcount) = ibin;
    countlist(mcount) = icount;
  };

  /* set up ipos(:) for axpbyz operation */

  ASSERT( mcount >= 1,
    ' ** set4list: invalid mcount ', mcount );

  /* add by how much, note axpbyz operation overwrites ipos(:) */
  call icopy( mcount, countlist, 1, ipos, 1 );

  call doiexpbyz( IGxlist, mcount, binlist, ipos, 1,1 );

  /* prepare for scatter */

  ni = istrtr; ip = 1;
  doloop(i,1,mcount) {

    doloop(j,1,countlist(i)) {
      idest(ip) = (ipos(i)-1) + j;
      nilist(ip) = ni;

      ni = ni + 1;
      ip = ip + 1;
    };
  };
};
```

```
        };
    };

    call doiscatter( IGNUMLIST, ISIZE, IDEST, NILIST );

};

    call dogsync();

#ifdef USE_PROFILE
    call profend('set4list');
#endif
return;
}
end
```

#### 4.7. subroutine copyxyz

```
#include "stdinc.h"

#define logdev 16
    /* perform hashing of xyz and copy coordinates to new array */

#define ISWAP( ia,ib ) { itemp = ia; ia = ib; ib = itemp; };

subroutine copyxyz()
#include "globals.h"
#include "parallel.h"
{

    logical usedfxyz;
#ifdef USE_DFXYZ
    parameter(usedfxyz=TRUE);
#else
    parameter(usedfxyz=FALSE);
#endif

    STRING name; STRING ctype;
    integer psize,gsiz, blocksize;

    integer ibstart,ibend;
```

```
intrinsic int,max,min,abs,sqrt;

integer ipage,npages,ipagestr;
integer i,istrt,iend,ysize, ip,jp, itemp,ni;
integer ix,iy,iz;

/* simulation in box [-0.5,0.5] */

real xmin,xmax,xadjust;
real ymin,ymax,yadjust;
real zmin,zmax,zadjust;

parameter(xmin=-0.5,xmax=0.5,xadjust=(xmax-xmin));
parameter(ymin=-0.5,ymax=0.5,yadjust=(ymax-ymin));
parameter(zmin=-0.5,zmax=0.5,zadjust=(zmax-zmin));

logical useminspace;
#ifdef USE_MINSPACE
parameter(useminspace=TRUE);
#else
parameter(useminspace=FALSE);
#endif

#define ndim pagesize

integer numlist(ndim);
integer iposxyz(3,ndim);

real xyz(3,ndim); real vxyz(3,ndim);
equivalence (xyz,vxyz);

#define rx(i) xyz(1,i)
#define ry(i) xyz(2,i)
#define rz(i) xyz(3,i)

/* ===== start execution ===== */

#ifdef USE_PROFILE
call profstart('copyxyz')
#endif

/* disable everything */
call dodisable( 0 );

if (useminspace) {
/* free storage for imap and
recreate fxyz */

/* disable everything, extra safe, and
```

```
        encourage memory compaction */

call dodisable(0);

call dogsync();
call dodestroy( IGimap );

call dogsync();

ctype = 'real' // char(0);
gsize = 3*natoms; psize = 3*pagesize; blocksize = 1;

name = 'fxyz(3,natoms)' // char(0);
call dodeclare( IGfxyz, name, gsize, ctype, psize, blocksize );

call dogsync();
};

/* use a "pull" strategy to copy the data */

#ifdef USE_PROFILE
    call profstart('copyxyz:cp xyz');
#endif

/* should be local, no need to search cache */
call dodisable( IGnumlist );

call doenable( IGxyz );
call dodisable( IGfxyz );

/* permuted copy xyz to fxyz, then fxyz to xyzlist */

gsize = natoms;
npages = int( gsize/ pagesize ) + 1;
ipagestrt=myid;

doloop4(ipage,ipagestrt,npages,nproc) {
    istr = 1 + ipage*pagesize;
    iend = min(gsize, istr+pagesize-1);
    isize = iend - istr + 1;
    if (isize <= 0) { break; };

    call dowait( dobigather( IGnumlist, isize,
        idxnumlist(istr), numlist ));

    doloop(i,1,isize) {
        ni = numlist(i);
        jp = idxyz(1,ni);

        /* assume continugous storage */

        iposxyz(1,i) = jp;
        iposxyz(2,i) = jp+1;
        iposxyz(3,i) = jp+2;
    };

    call dorgather( IGxyz, 3*isize, iposxyz, xyz );
    call dobrscatter( IGfxyz, 3*isize, idxfxyz(1,istr), xyz );
}
```

```
}; /* end do ipage */

#ifdef USE_PROFILE
  call profend('copyxyz:cp xyz');
#endif

  call dogsync();
  /*
   * safer to copy vectors than renaming them, will simply swap
   * global array descriptors to rename them if this is really costly.
   */
  call dodisable( IGxyzlist );
  gsize = 3*natoms; psize = 3*pagesize;
  call grveccopy( IGfxyz, gsize, psize, IGxyzlist );

  call dogsync();

  /* permuted copy vxyz to fxyz, then fxyz to vxyz */

  call doenable( IGvxyz );
  call dodisable( IGfxyz );

#ifdef USE_PROFILE
  call profstart('copyxyz:cp vel');
#endif

  gsize = natoms;
  npages = int( gsize/ pagesize ) + 1;
  ipagestrt=myid;

  doloop4(ipage,ipagestrt,npages,nproc) {
    istr = 1 + ipage*pagesize;
    iend = min(gsize, istr+pagesize-1);
    isize = iend - istr + 1;
    if (isize <= 0) { break; };

    call dowait( dobigather( IGnumlist, isize,
      idxnumlist(istr), numlist ));

    doloop(i,1,isize) {
      ni = numlist(i);
      jp = idxyz(1,ni);

      /* assume continugous storage */

      iposxyz(1,i) = jp;
      iposxyz(2,i) = jp+1;
      iposxyz(3,i) = jp+2;
    };

    call dorgather( IGvxyz, 3*isize, iposxyz, vxyz );
    call dobrscatter( IGfxyz, 3*isize, idxfxyz(1,istr), vxyz );

  }; /* end do ipage */

#ifdef USE_PROFILE
  call profend('copyxyz:cp vel');
#endif

  call dogsync();
```



```
/*
  safer to copy vectors than renaming them, will simply swap
  global array descriptors to rename them if this is really costly.
*/

call dodisable( IGvxyz );
gsize = 3*natoms; psize = 3*pagesize;
call grveccopy( IGfxyz, gsize, psize, IGvxyz );

call dogsync();

if (useminspace & usedfxyz) {

  /* disable everything */

  call dodisable( 0 );

  call dogsync();
  call dodestroy( IGfxyz );

  /* recreate real*8 global fxyz array */

  ctype = 'real*8' // char(0);
  gsize = 3*natoms; psize = 3*pagesize; blocksize = 1;

  name = 'fxyz(3,natoms)' // char(0);
  call dodeclare( IGfxyz, name, gsize, ctype, psize, blocksize );

  call dogsync();
};

#ifdef USE_PROFILE
  call profend('copyxyz')
#endif

return;
}
end
```

#### 4.8. subroutine force

```
#include "stdinc.h"

subroutine force()
#include "globals.h"
#include "parallel.h"

{

#define maxproc (2*1024)
  integer mystart(maxproc+1);
```

```
integer istr, iend, ncols,  binsize;
logical islast;

integer gsize, psize;
real one, zero;
parameter(one=1.0, zero=0.0);

real8 oned, zerod;
parameter(oned=1.0d0, zerod=0.0d0);

real rvalue;
real8 dvalue;

/* perform dynamic load balancing and all force computations */

/* ===== start execution ===== */

#ifdef USE_PROFILE
    call profstart('force')
#endif

ASSERT( (1 <= nproc) & (nproc <= maxproc),
        ' ** force: invalid nproc ', nproc );

#ifdef NO_BALANCE
    /* no need to call "calwork" */

    call lbalance( mystart );
#else
    call profstart('force:calwork');
    call calwork();

    call lbalance( mystart );
    call profend('force:calwork');
#endif

istr = mystart(myid+1);

/* iend is the last column actually computed */
islast = ((myid + 1) == nproc );
if (islast) {
    binsize = nbiny*nbinz;
    iend = binsize;
}
else {
    iend = mystart( (myid+1) + 1) - 1;
};

ncols = (iend - istr + 1);

/* clear out array for accumulation */

gsize = 3*natoms; psize = 3*pagesize;
#ifdef USE_DFXYZ
    dvalue = 0.0;
    call gdvecfill( IGfxyz, dvalue, gsize, psize );
#else
    rvalue = 0.0;
    call grvecfill( IGfxyz, rvalue, gsize, psize );
#endif
```

```
v = 0.0; w = 0.0;

if (ncols > 0) {
    call forceall( istrtr, ncols );
};
call dogsync();

/* incorporate energy factors */
v = 4.0*v;
w = 48.0*w/3.0;

gsize = 3*natoms; psize = 3*pagesize;

#ifdef USE_DFXYZ
    dvalue = 48.0;
    call gdvecscale( IGfxyz, dvalue, gsize, psize );
#else
    rvalue = 48.0;
    call grvecscale( IGfxyz, rvalue, gsize, psize );
#endif

    call dogsync();

#ifdef USE_PROFILE
    call profend('force')
#endif

    return;
}

end
```

#### 4.9. subroutine forceall

```
#include "stdinc.h"

#ifdef USE_SYMMETRY

#ifdef nneighbor
#define nneighbor 14
#endif

#ifdef ncols
#define ncols 5
#endif

#else

#ifdef nneighbor
#define nneighbor 27
#endif

#ifdef ncols
#define ncols 9
#endif

#endif
```

```
#endif /* USE_SYMMETRY */

subroutine forceall( ipstart, ncolumns )
#include "globals.h"
#include "parallel.h"

    integer ipstart,ncolumns;

{

    real one,zero; parameter(one=1.0,zero=0.0);

    integer jj,ipos,ix,iy,iz,  ijob;

    integer icol, i,ntotal;

    integer gsize, psize;

    logical isnewcol, isfirst;
    integer iyold;

    /* ===== start execution ===== */
#ifdef USE_PROFILE
    call profstart('forceall')
#endif

    call dodisable( IGfxyz );

    call doenable( IGxyzlist );

    call doenable( IGncount );
    call doenable( IGxlist );

    iyold = -1; isfirst = TRUE;

    doloop(jj,1,ncolumns) {
        ipos = ipstart + (jj-1);

        /* convert ipos back to (iy,iz) */

        call ni2ij( ipos, nbiny, nbinz, iy,iz );

        isnewcol = (iyold + 1 != iy);
        if (isfirst) { ijob = 1; isfirst = FALSE; }
        else if (isnewcol) { ijob = 2; } else { ijob = 3; };

        call forcecol( iy,iz,  ijob );

        iyold = iy;
    };

    /* final flushing of results */

    ijob = 4;
    call forcecol( iy,iz, ijob );
```

```
#ifndef USE_PROFILE
    call profend('forceall')
#endif

    return;
}
    end
```

#### 4.10. subroutine forcecol

```
#include "stdinc.h"

#undef USE_PROFILE

#define logdev 16

#define eps 1.0e-5
#define isapprox(r1,r2) ( abs((r1)-(r2)) <= eps*max(one,max(abs(r1),abs(r2))) )

#define RETURN_LABEL 999

#ifdef USE_SYMMETRY

#ifndef nneighbor
#define nneighbor 14
#endif

#ifndef ncols
#define ncols 5
#endif

#else

#ifndef nneighbor
#define nneighbor 27
#endif

#ifndef ncols
#define ncols 9
#endif

#endif

/* USE_SYMMETRY */

#define CHECKXYZ( icol ) { \
    doloop(jx,1,nbinx) { \
        \
        call doigather( IGncount, 1,idxncount(jx,jy,jz),icount); \
        ASSERT( icount == ncount(jx,icol), \
            ' ** forcecol: icount != ncount(jx,icol) ',icount); \
    } \
}
```

```
\
    jp = xlist(icol); \
    doloop(j,1,jx-1) { jp = jp + ncount(j,icol); }; \
\
    call doigather( IGxlist, 1, idxxlist(jx,jy,jz), ip ); \
    ASSERT( ip == jp, \
        ' ** forcecol: error in xlist calculation ', jp ); \
\
    if (icount > 0) { \
        call dowait( dobrgather( IGxyzlist, 3*icount, \
            idxxyzlist(1,ip), txyz )); \
\
        doloop(j,1,icount) { \
            jp = gxlist(jx,icol) + (j-1); \
\
            ASSERT( isapprox(tx(j),rx(jp)), \
                ' ** forcecol: tx(j) != rx(jp), icol=',icol); \
            ASSERT( isapprox(ty(j),ry(jp)), \
                ' ** forcecol: ty(j) != ry(jp), icol=',icol); \
            ASSERT( isapprox(tz(j),rz(jp)), \
                ' ** forcecol: tz(j) != rz(jp), icol=',icol); \
            }; /* end do j */ \
        }; /* end if (icount) */ \
\
    }; /* end do jx */ \
};

#define FOLDBC( icol ) { \
    /* fold boundary contribution back */ \
\
    isrc = gxlist(0,icol); idest = gxlist(nbinx,icol); \
    icount = ncount(0,icol); \
    doloop(i,1,icount) { \
        fx(idest) = fx(idest) + fx(isrc); \
        fy(idest) = fy(idest) + fy(isrc); \
        fz(idest) = fz(idest) + fz(isrc); \
\
        idest = idest + 1; isrc = isrc + 1; \
    }; \
\
    isrc = gxlist(nbinx+1,icol); idest = gxlist(1,icol); \
    icount = ncount(nbinx+1,icol); \
\
    doloop(i,1,icount) { \
        fx(idest) = fx(idest) + fx(isrc); \
        fy(idest) = fy(idest) + fy(isrc); \
        fz(idest) = fz(idest) + fz(isrc); \
\
        idest = idest + 1; isrc = isrc + 1; \
    }; \
};

#define FLUSHOUT( icol ) { \
    if (ntotal(icol) > 0) { \
        FOLDBC( icol ); \
\
        if (usedfxyz) { \
            call dobdaxpby(IGfxyz, 3 * ntotal(icol), \
                idxfxyz(1, xlist(icol)), \
                fxyz(1, gxlist(1, icol)), oned, oned); \
        } \
    } else { \

```

```
        call dobraxpby(IGfxyz, 3 * ntotal(icol), \
                    idxfxyz(1, xlist(icol)), \
                    fxyz(1, gxlist(1, icol)), one, one); \
    }; \
}; \

#define SHIFTCOPY( isrc, idest ) { \
    isvalid = (1 <= isrc) & (isrc <= ncols); \
    ASSERT( isvalid, \
            ' ** forcecol: invalid isrc ', isrc ); \
    isvalid = (1 <= idest) & (idest <= ncols); \
    ASSERT( isvalid, \
            ' ** forcecol: invalid idest ', idest ); \
    \
    ASSERT( isrc != idest, \
            ' ** forcecol: isrc == idest ', isrc ); \
    \
    nlist(1, idest) = nlist(1, isrc); \
    nlist(2, idest) = nlist(2, isrc); \
    \
    ntotal(idest) = ntotal(isrc); \
    xlist(idest) = xlist(isrc); \
    \
    ipdest = gxlist(0, idest); \
    ASSERT( ipdest == ipstart(idest), \
            ' ** forcecol: ipdest != ipstart(idest) ', idest); \
    \
    ipsrc = gxlist(0, isrc); \
    ASSERT( ipsrc == ipstart(isrc), \
            ' ** forcecol: ipsrc != ipstart(isrc) ', isrc); \
    \
    \
    gntotal = ntotal(idest) + ncount(0, idest) + ncount(nbinx+1, idest); \
    \
    call icopy(nbinx + 2, gxlist(0, isrc), 1, gxlist(0, idest), 1); \
    call icopy(nbinx + 2, ncount(0, isrc), 1, ncount(0, idest), 1); \
    \
    /* reset ipstart(:) to reuse storage in xyz(:) and fxyz(:) */ \
    \
    ipstart(idest) = ipsrc; \
    ipstart(isrc) = ipdest; \
    \
    gxlist(0, isrc) = ipstart(isrc); \
    gxlist(0, idest) = ipstart(idest); \
    \
    if (IDEBUG >= 2) { \
        call rfill( 3*ndim, zero, xyz(1, ipstart(isrc)), 1 ); \
        if (usedfxyz) { \
            call dfill( 3*ndim, zerod, fxyz(1, ipstart(isrc)), 1 ); \
        } \
        else { \
            call rfill( 3*ndim, zero, fxyz(1, ipstart(isrc)), 1 ); \
        } \
    }; \
}; /* if (IDEBUG) */ \
};
```

```
/*
 * compute forces associated with a column (ix=1..nbinx) for a particular
 * (iy,iz)
 */

#define gndim (ncols*ndim)

subroutine forcecol(iy, iz, ijob)
#include "globals.h"
#include "parallel.h"

integer      iy, iz;
integer ijob;

{
#ifdef MP
/* #define nthread 6 */
/* integer numcpus; external numcpus; */
/* #define nthread (numcpus()) */
#define nthread 2
#else
#define nthread 1
#endif

integer ixstart,ixend,istep;

logical usesymmetry;
#ifdef USE_SYMMETRY
parameter(usesymmetry=TRUE);
#else
parameter(usesymmetry=FALSE);
#endif

integer      maxbinx, ndim;
#ifdef RX || I860
/* RX has less memory, unlikely to solver extremely huge problems */
parameter(maxbinx=64,ndim=700);
#else
parameter(maxbinx=700,ndim=20*maxbinx);
#endif

integer      ncount(0:(maxbinx + 1), ncols);
integer      gxlist(0:(maxbinx + 1), ncols);
integer      xlist(ncols);

#ifdef USE_DFXYZ
#define REAL real8
#else
#define REAL real
#endif

REAL vvpos,vpos(maxbinx),vvneg,vneg(maxbinx);
REAL wwpos,wpos(maxbinx),wwneg,wneg(maxbinx);

real txyz(3,ndim);
#define tx(i) txyz(1,i)
```



```
#define ty(i) txyz(2,i)
#define tz(i) txyz(3,i)

    logical usedfxyz;

#ifdef USE_DFXYZ
    parameter(usedfxyz=TRUE);
    real8      fxyz(3, gndim);
#else
    parameter(usedfxyz=FALSE);
    real      fxyz(3, gndim);
#endif

    real      xyz(3, gndim);

#define rx(ip) xyz(1,ip)
#define ry(ip) xyz(2,ip)
#define rz(ip) xyz(3,ip)

#define fx(ip) fxyz(1,ip)
#define fy(ip) fxyz(2,ip)
#define fz(ip) fxyz(3,ip)

    real8      oned,zerod;
    parameter(oned=1.0d0,zerod=1.0d0);

    real      one, zero;
    parameter(one=1.0,zero=0.0);

    integer      maxcols;
    parameter(maxcols = 9);
    integer      list(maxcols), nlist(2, maxcols);

    integer      lcount(nneighbor,maxbinx), lxlist(nneighbor,maxbinx);

    integer      gntotal, ntotal(maxcols);
    real      xadj,yadj, zadj, yadjust(maxcols), zadjust(maxcols);

    integer      i, j, ip, jp, istr, iend;
    integer      jcol,icol, icolstr, icolend, icolsize;
    integer      ix, jx, jy, jz, idx, idy, idz, jdx, jdy, jdz;
    integer      isrc, idest, icount;

    logical      allok,isvalid, spaceok;
    logical      isfirst,isnewcol,issshift,islst;

    integer      ipstart(maxcols), ipsrc,ipdest;

    save ipstart,ncount,xlist,gxlist,nlist,ntotal;
    save xyz,vxyz,fxyz;

#ifdef USE_SYMMETRY
#define ncopies 3
#else
```

```
#define ncopies 6
#endif
    integer nclist(2,ncopies); save nclist;

#include "mapping.h"

    data          isfirst / TRUE /;

#ifdef USE_SYMMETRY

    /* copy sequence 4->1, 3->2, then 5->3 */

    /* based on the ordering:
        [ *, *, 2]
        [ *, 1, 3]
        [ *, 4, 5]
    */

    data nclist / 4,1, 3,2, 5,3 /;
#else
    /* copy sequence 4->1, 5->2, 6->3, then
        7->4, 8->5, 9->6 */

    /* based on the ordering:
        [ 1, 2, 3]
        [ 4, 5, 6]
        [ 7, 8, 9]
    */

    data nclist / 4,1, 5,2, 6,3,
        7,4, 8,5, 9,6 /;

#endif /* USE_SYMMETRY */

    /* ===== start execution ===== */

    #dir$r noconcur

#ifdef USE_PROFILE
    call profstart('forcecol');
#endif

    spaceok = (1 <= nbinx) & (nbinx <= maxbinx);
    ASSERT(spaceok,
        ' ** forcecol: invalid nbinx ', nbinx);

    isvalid = (1 <= ijob) & (ijob <= 4);
    ASSERT( isvalid,
        ' ** forcecol: invalid ijob ', ijob );

    isfirst = (ijob == 1);
    isnewcol = (ijob == 2);
    isshift = (ijob == 3);
    islast = (ijob == 4);
```

```
if (isfirst) {

    /* initialize data structure */

    doloop(icol,1,ncols) {
        ip = (icol-1)*ndim + 1;
        ipstart(icol) = ip;
    };
};

if (!isshift) {
    /* get everthing from scratch */
    icolstrt = 1;
}
else {
    /* reuse previous data */

    if (usesymmetry) { icolstrt = ncols - 1; }
    else { icolstrt = ncols - 2; };
};

icolend = ncols;
icolsize = (icolend - icolstrt + 1);

/* flush previous entries ? Nothing to flush out if it isfirst */

if (isshift) {
    if (usesymmetry) {
        icol = 1; FLUSHOUT( icol );
        icol = 2; FLUSHOUT( icol );
    }
    else {
        /* fxyz of neighbours are not updated, flush only myown */

        icol = 5; FLUSHOUT( icol );
    };
}
else if (isnewcol | islast) {
    if (usesymmetry) {
        /* flush out everything */
        doloop(icol,1,ncols) {
            FLUSHOUT( icol );
        };
    }
    else {

        /* fxyz of neighbours are not updated, flush only myown */

        icol = 5; FLUSHOUT( icol );
    };
};

/* perform shifts */

if (isshift) {
```

```
doloop(i,1,ncopies) {
    isrc = nclist(1,i);
    idest = nclist(2,i);

    SHIFTCOPY( isrc, idest );
};

/* bring in xyz coordinats, and clear out fxyz */

if (islast) { goto RETURN_LABEL; /* don't do any more work */ };

doloop(icol, 1, ncols) {
    yadjust(icol) = 0.0;
    zadjust(icol) = 0.0;
};

/* determine neighbors */

doloop(icol, icolstrt, icolend) {

    idy = colmap(1, icol);
    idz = colmap(2, icol);

    jy = iy + idy;
    jz = iz + idz;

    if (jy > nbiny) { jy = jy - nbiny; yadjust(icol) = one; };
    if (jy < 1) { jy = jy + nbiny; yadjust(icol) = -one; };

    if (jz > nbinz) { jz = jz - nbinz; zadjust(icol) = one; };
    if (jz < 1) { jz = jz + nbinz; zadjust(icol) = -one; };

    nlist(1, icol) = jy;
    nlist(2, icol) = jz;
};

/* get xlist(:) */
jx = 1;
doloop(icol, icolstrt, icolend) {
    jy = nlist(1, icol);
    jz = nlist(2, icol);
    list(icol) = idxxlist(jx, jy, jz);
};

call doenable(IGxlist);
call doigather(IGxlist, icolsize,
               list(icolstrt), xlist(icolstrt));

/* get ncount(:) for columns */

call doenable( IGncount );

doloop(icol, icolstrt, icolend) {
```

```
jx = 1;
jy = nlist(1, icol);
jz = nlist(2, icol);
call dowait(dobigather(IGncount, nbinx,
                    idxncount(jx, jy, jz), ncount(1, icol)));

if (IDEBUG >= 2) {
    /* double check */
    doloop(ix,1,nbinx) {
        call doigather( IGncount, 1, idxncount(ix,jy,jz), icount);
        ASSERT( icount == ncount(ix,icol),
            ' ** forcecol: icount != ncount(ix,icol)',ix );
    };
};

/* wrap around boundary condition */

ncount(0, icol) = ncount(nbinx, icol);
ncount(nbinx + 1, icol) = ncount(1, icol);

/* calculate total storage needed */

gntotal = 0;
doloop(i, 1, nbinx) {
    gntotal = gntotal + ncount(i, icol);
};
ntotal(icol) = gntotal;

gntotal = ntotal(icol) + ncount(0, icol) + ncount(nbinx + 1, icol);
spaceok = (0 <= gntotal) & (gntotal <= ndim);
ASSERT(spaceok,
    ' ** forcecol: insufficient temporary space, need ', gntotal);
};

/* set up gxlist(:) */

doloop(icol, icolstrt, icolend) {
    gxlist(0,icol) = ipstart(icol);

    /* not tightly packed, make room for shifting columns */

    doloop(i, 0, nbinx) {
        gxlist(i + 1, icol) = gxlist(i, icol) + ncount(i, icol);
    };
};

/* double check ipstart(:) and gxlist(:) */
if (IDEBUG >= 2) {
    doloop(icol,1,ncols) {
        ASSERT( ipstart(icol) == gxlist(0,icol),
            ' ** forcecol: invalid ipstart(icol)',icol );
    };
    doloop(icol,1,ncols) {
        doloop(jcol,icol+1,ncols) {
            ASSERT( ipstart(icol) != ipstart(jcol),
                ' ** forcecol: duplicate entry in ipstart(:)',icol );
        };
    };
};

/* get xyz coordinates */

doloop(icol, icolstrt, icolend) {
```

```

jy = nlist(1, icol);
jz = nlist(2, icol);

if (ntotal(icol) > 0) {
  call dowait(dobrgather(IGxyzlist, 3 * ntotal(icol),
    idxyzlist(1, xlist(icol)), xyz(1, gxlist(1, icol))));

  if (IDEBUG >= 2) {

    /* double check xyz coordinates */
    CHECKXYZ( icol );

    }; /* if (IDEBUG) */
};

/* fix up boundary */

isrc = gxlist(nbinx, icol);
idest = gxlist(0, icol);
icount = ncount(0, icol);

istrt = idest;
iend = istrt + (icount - 1);

doloop(idest, istrt, iend) {

  rx(idest) = rx(isrc) - one;
  ry(idest) = ry(isrc);
  rz(idest) = rz(isrc);

  isrc = isrc + 1;
};

isrc = gxlist(1, icol);
idest = gxlist(nbinx + 1, icol);
icount = ncount(nbinx + 1, icol);

istrt = idest;
iend = istrt + (icount - 1);

doloop(idest, istrt, iend) {

  rx(idest) = rx(isrc) + one;
  ry(idest) = ry(isrc);
  rz(idest) = rz(isrc);

  isrc = isrc + 1;
};

};

/* adjust y,z coordinates */

doloop(icol, icolstrt, icolend) {
  gntotal = ntotal(icol) + ncount(0, icol) + ncount(nbinx + 1, icol);
  istrt = gxlist(0, icol);
  iend = istrt + gntotal - 1;

  if (yadjust(icol) != zero) {
    yadj = yadjust(icol);
    doloop(i, istrt, iend) {
      ry(i) = ry(i) + yadj;
    };
  };
};
```

```
};

if (zadjust(icol) != zero) {
    zadj = zadjust(icol);
    doloop(i, istr, iend) {
        rz(i) = rz(i) + zadj;
    };
};
};

/* zero out fxyz */

if (usesymmetry) {
    doloop(icol,icolstr,icolend) {
        gntotal = ntotal(icol) + ncount(0, icol) + ncount(nbinx + 1, icol);
        istr = gxlist(0, icol);
        iend = istr + gntotal - 1;

        doloop(i,istr,iend) {
            fx(i) = 0.0;
            fy(i) = 0.0;
            fz(i) = 0.0;
        };

    }; /* end do icol */
}
else {
    /* zero out only 1 column of fxyz */

    icol = 5;
    gntotal = ntotal(icol) + ncount(0, icol) + ncount(nbinx + 1, icol);
    istr = gxlist(0, icol);
    iend = istr + gntotal - 1;

    doloop(i,istr,iend) {
        fx(i) = 0.0;
        fy(i) = 0.0;
        fz(i) = 0.0;
    };

};

/* prepare lcount and lxlist */
doloop(ix,1,nbinx) {
    doloop(i, 1, nneighbor) {

        idx = mapping(1, i);
        idy = mapping(2, i);
        idz = mapping(3, i);

        jx = ix + idx;
        jy = iy + idy;
        jz = iz + idz;

        icol = icolmap(i);

        isvalid = (0 <= jx) & (jx <= nbinx + 1);
    };
};
};
```

```
if (IDEBUG >= 2) {
    ASSERT(isvalid,
        ' ** forcecol: invalid jx ', jx);

    ASSERT((1 <= icol) & (icol <= ncols),
        ' ** forcecol: invalid icol ', icol);
}; /* if (IDEBUG) */

jdy = colmap(1, icol);
jdz = colmap(2, icol);
if (IDEBUG >= 2) {
    ASSERT((jdy == idy) & (jdz == idz),
        ' ** forcecol: invalid icol ', icol);
}; /* if (IDEBUG) */

lcount(i,ix) = ncount(jx, icol);
lxlist(i,ix) = gxlist(jx, icol);

}; /* end do i */

}; /* end do ix */

/* perform computation in routine 'forcecal' */

if (mod(nbinx,nthread) == 0) {
    /* evenly divide */
    istep = nbinx/nthread;
}
else {
    istep = 1 + (nbinx/nthread);
};
ixend = istep;
doloop(ixstart,1,ixend) {

    call docheck();

    #dir$l concur
    #dir$l cncall
    doloop4(ix,ixstart,nbinx,istep) {

        call forcecal(ix, iy, iz,
            lcount(1,ix), lxlist(1,ix),
            gndim, fxyz, xyz,
            vpos(ix),vneg(ix),wpos(ix),wneg(ix));

    }; /* end do ix */

}; /* end do ixstart */

vvpos = 0.0; vvneg = 0.0; wwpos = 0.0; wwneg = 0.0;
doloop(ix,1,nbinx) {
    vvpos = vvpos + vpos(ix);
    vvneg = vvneg + vneg(ix);
    wwpos = wwpos + wpos(ix);
    wwneg = wwneg + wneg(ix);
};

if (w > 0.0) { w = (w + wwpos) + wwneg; } else { w = (w + wwneg) + wwpos; };
if (v > 0.0) { v = (v + vvpos) + vvneg; } else { v = (v + vvneg) + vvpos; };
```



```
RETURN_LABEL {  
  
#ifdef USE_PROFILE  
    call profend('forcecol');  
#endif  
  
    return;  
};  
}  
end
```

#### 4.11. subroutine forcecal

```
#include "stdinc.h"  
  
#undef ORIGINAL  
#define USE_ABS  
  
#ifdef MP  
#undef USE_PROFILE  
#endif  
  
#define RETURN_LABEL 9999  
  
#ifdef USE_SYMMETRY  
  
#ifndef nneibor  
#define nneibor 14  
#endif  
  
#else  
#ifndef nneibor  
#define nneibor 27  
#endif  
  
#endif /* USE_SYMMETRY */  
  
#define maxneibor 27  
  
subroutine forcecal(ix,iy,iz,  
                  ncount,lxlist,  
                  ndim,fxyz,xyz,  
                  vpos,vneg,wpos,wneg )  
  
#include "globals.h"  
  
    integer ix,iy,iz;  
  
    integer ncount(nneibor),lxlist(nneibor);  
    integer ndim;  
    real xyz(3,ndim);
```

```
#ifndef USE_DXYZ
    real8 fxyz(3,ndim);
#define REAL real8
#else
    real fxyz(3,ndim);
#define REAL real
#endif

{
#define cdim 1024
    integer ii,ccount, ilist(cdim), jlist(cdim);
    REAL rsq(cdim);

    intrinsic sqrt,abs,max,min,mod,sign;

    integer ni, i,j, ip,jp, jstrt,jend, idxval;
    logical isempty;

    integer ip1,ipj, jneibor,jcount;
    integer ninside;

    logical isxok,isyok,iszok, isallok;

    integer idx,idy,idz, jx,jy,jz;

    REAL one, zero;
    parameter(one=1.0,zero=0.0);

    REAL rxi,ryi,rzi, rxj,ryj,rzj;
    REAL rxij,ryij,rzij, rij,rijsq;

    REAL sr2,sr6;
    REAL sigsq,rcutsq;

    REAL fij, fxij,fyij,fzij;
    REAL fxi,fyi,fzi;

    REAL wij,wpos,wneg, vij,vpos,vneg;

    /* routine to compute forces and potential related to
    a single bin cell (ix,iy,iz) */

#include "dolib.h"

#define rx(i) xyz(1,i)
#define ry(i) xyz(2,i)
#define rz(i) xyz(3,i)

#define fx(i) fxyz(1,i)
#define fy(i) fxyz(2,i)
#define fz(i) fxyz(3,i)

    /* ===== */

    #dir$r noconcur

```

```
#ifdef USE_PROFILE
#exe$ CALL MP_BCS
    call profstart('forcecal')
#exe$ CALL MP_ECS
#endif

isxok = (1 <= ix) & (ix <= nbinx);
isyok = (1 <= iy) & (iy <= nbiny);
iszok = (1 <= iz) & (iz <= nbinz);
isallok = isxok & isyok & iszok;
if (!isallok) {
    write(*,*) ' ** forcecal: invalid (ix,iy,iz) ',
        ix,iy,iz;
    stop ' ** ERROR in forcecal ';
};

ninside = ncount(1);
isempty = (ninside == 0);
if (isempty) { goto RETURN_LABEL; };

/* Assume fx(:),fy(:),fz(:), already initialized */

vpos = zero; vneg = zero;
wpos = zero; wneg = zero;

sigsq = sigma*sigma;
rcutsq = rcut*rcut;

/* note minus one for offset calculations */

ip1 = lxlist(1)-1;

/* handle interactions within the same cell */
ASSERT( cdim >= ninside*ninside,
    ' ** forcecal: cdim too small, need ', ninside*ninside );
ip = 1;
doloop(i,1,ninside) {
    doloop(j,i+1,ninside) {

        rsq(ip) =      (rx(ip1+i) - rx(ip1+j))**2 +
                      (ry(ip1+i) - ry(ip1+j))**2 +
                      (rz(ip1+i) - rz(ip1+j))**2;

        ip = ip + 1;
    };
}; /* end do i */

ccount = 0; ip = 1;
doloop(i,1,ninside) {
    doloop(j,i+1,ninside) {
        rijsq = rsq(ip);
        if (rijsq <= rcutsq) {
            ccount = ccount + 1;
        }
    }
}
```

```
        ildist(ccount) = i; jldist(ccount) = j;
        rsq(ccount) = rijsq;
    };
    ip = ip + 1;
}; /* end do i */

/* perform force and potential calculations */

doloop(ii,1,ccount) {
    i = ildist(ii); j = jldist(ii); rijsq = rsq(ii);

    rij = sqrt(rijsq);

    rxij = rx(ip1+i) - rx(ip1+j);
    ryij = ry(ip1+i) - ry(ip1+j);
    rzij = rz(ip1+i) - rz(ip1+j);

    sr2 = sigsq/rijsq;
    sr6 = sr2*sr2*sr2;

    /* v is the total potential energy */
    /* w is the total virial energy */

    vij = sr6*(sr6-1.0)-vrcut-dvrc12*(rij-rcut);
    wij = sr6*(sr6-0.5)+dvrcut*rij;
    fij = wij/rijsq;

    fxij = fij*rxij;
    fyij = fij*ryij;
    fzij = fij*rzij;

    fx(ip1+i) = fx(ip1+i) + fxij;
    fy(ip1+i) = fy(ip1+i) + fyij;
    fz(ip1+i) = fz(ip1+i) + fzij;

    fx(ip1+j) = fx(ip1+j) - fxij;
    fy(ip1+j) = fy(ip1+j) - fyij;
    fz(ip1+j) = fz(ip1+j) - fzij;

#ifdef USE_ABS
    vpos = vpos + (abs(vij) + vij);
    vneg = vneg + (vij-abs(vij));

    wpos = wpos + (abs(wij)+wij);
    wneg = wneg + (wij-abs(wij));
#else
    if (vij > 0) { vpos = vpos + vij;} else {vneg = vneg + vij;};
    if (wij > 0) { wpos = wpos + wij;} else {wneg = wneg + wij;};
#endif /* USE_ABS */

}; /* endo do ii */

/* handle interactions with neighbor cells */
```

```
doloop(jneibor,2,nneibor) {

    ipj = lxlist(jneibor)-1;
    jcount = ncount(jneibor);

    ASSERT( cdim >= jcount*ninside,
        ' ** forcecal: cdim too small, need ', jcount*ninside );

    ip = 1;
    doloop(j,1,jcount) {
    doloop(i,1,ninside) {

        rsq(ip) =      (rx(ip1+i) - rx(ipj+j))**2 +
                      (ry(ip1+i) - ry(ipj+j))**2 +
                      (rz(ip1+i) - rz(ipj+j))**2 ;

        ip = ip + 1;
    };
    };

    ccount = 0; ip = 1;
    doloop(j,1,jcount) {
    doloop(i,1,ninside) {
        rijsq = rsq(ip);
        if (rijsq <= rcutsq) {
            ccount = ccount + 1;
            ilist(ccount) = i; jlist(ccount) = j;
            rsq(ccount) = rijsq;
        };
        ip = ip + 1;
    };
    };

    doloop(ii,1,ccount) {
        i = ilist(ii); j = jlist(ii); rijsq = rsq(ii);

        rij = sqrt( rijsq );

        rxij = rx(ip1+i) - rx(ipj+j);
        ryij = ry(ip1+i) - ry(ipj+j);
        rzij = rz(ip1+i) - rz(ipj+j);

        sr2 = sigsq/rijsq;
        sr6 = sr2*sr2*sr2;

        vij = (sr6*(sr6-1.0)-vrcut-dvrc12*(rij-rcut))
        wij = sr6*(sr6-0.5)+dvrcut*rij;
        fij = wij/rijsq;

        fxij = fij*rxij;
        fyij = fij*ryij;
        fzij = fij*rzij;

        fx(ip1+i) = fx(ip1+i) + fxij;
        fy(ip1+i) = fy(ip1+i) + fyij;
        fz(ip1+i) = fz(ip1+i) + fzij;

#ifdef USE_SYMMETRY

        fx(ipj+j) = fx(ipj+j) - fxij;
        fy(ipj+j) = fy(ipj+j) - fyij;
```

```
        fz(ipj+j) = fz(ipj+j) - fzij;

#else /* USE_SYMMETRY */

        vij = 0.5*vij;
        wij = 0.5*wij;

#endif /* USE_SYMMETRY */

#ifdef USE_ABS
        vpos = vpos + (abs(vij) + vij);
        vneg = vneg + (vij - abs(vij));

        wpos = wpos + (abs(wij) + wij);
        wneg = wneg + (wij - abs(wij));
#else
        if (vij > 0.0) {vpos = vpos + vij;} else {vneg = vneg + vij;};
        if (wij > 0.0) {wpos = wpos + wij;} else {wneg = wneg + wij;};
#endif /* USE_ABS */

}; /* end do ii */

}; /* end do jneighbor */

#ifdef USE_ABS
        vpos = 0.5*vpos; vneg = 0.5*vneg;
        wpos = 0.5*wpos; wneg = 0.5*wneg;
#endif /* USE_ABS */

        /* summation into global v and w performed in forcecol */

RETURN_LABEL {

#ifdef USE_PROFILE
#exe$ CALL MP_BCS
        call profend('forcecol')
#exe$ CALL MP_ECS
#endif

        return;
};

}

end
```

## 5. References

- [1] M. P. ALLEN AND D. TILDESLEY, *Computer Simulation of Liquids*, Clarendon Press, Oxford, 1987.
- [2] D. M. BEASLEY AND P. S. LOMDAHL, *Message passing multi-cell molecular dynamics on the Connection Machine 5*, Parall. Comp., 20 (1994), pp. 173–196.
- [3] E. F. D’AZEVEDO AND C. H. ROMINE, *DOLIB: Distributed Object Library*, Tech. Report ORNL/TM-12744, Oak Ridge National Laboratory, 1994.
- [4] R. W. HOCKNEY, S. P. GOEL, AND J. W. EASTWOOD, *Quiet high-resolution computer models of a plasma*, J. Comp. Phys, 14 (1974), pp. 148–158.
- [5] P. S. LOMDAHL, P. TAMAYO, N. GRONBECH-JENSEN, AND D. M. BEASLEY, *50 GFlops molecular dynamics on the Connection Machine 5*, in Proceedings Supercomputing ’93, November 15-19, Portland, Oregon, Association for Computing Machinery and The Institute of Electrical and Electronics Engineers, Inc., 1993, pp. 520–527.
- [6] B. MARR, R. PEIERLS, AND J. PASCIAK, *IPX – Preemptive remote procedure execution for concurrent applications*, tech. report, Brookhaven National Laboratory, 1994.
- [7] M. PINCHES, D. TILDESLEY, AND W. SMITH, *Large scale molecular dynamics on parallel computers using the link-cell algorithm.*, Molecular Simulation, 6 (1991), pp. 51–87. see also [ftp://ftp.dl.ac.uk/ccp5/SOTON\\_PAR/README](ftp://ftp.dl.ac.uk/ccp5/SOTON_PAR/README).
- [8] S. PLIMPTON, *Fast parallel algorithms for short-range molecular dynamics*, Tech. Report SAND91-1144, UC-705, Sandia National Laboratories, Printed May 1993.

- [9] J. G. POWLES, W. A. B. EVANS, AND N. QUIRKE, *Non-destructive molecular dynamics simulation of the chemical potential of a fluid*, Mol. Phys, 46 (1982), pp. 1347–1370.
- [10] L. VERLET, *Computer experiments on classical fluids I. Thermodynamical properties of Lennard-Jones molecules*, Phys. Rev., 159 (1967), pp. 98–103.