17

Algorithms for supercomputers

17.1 Introduction

The previous chapters concentrated on translating physical problems into practical simulations. Computational efficiency, beyond the use of cells and neighbor lists (as well as hierarchical subdivision when appropriate), received little attention. For 'conventional' computers, there is not a great deal more that the average user can do in this respect, assuming that a reasonably effective programming style has been adopted. This attitude is no longer adequate when modern, high performance, multiprocessor machines are to serve as the platforms for large-scale simulation.

In this chapter we focus on ways of adapting the basic MD approach to take advantage of advanced computer architectures; since enhanced performance comes not only from a faster processor clock cycle, but also from a number of fundamental changes in the way computers process data, this is a subject that cannot be ignored. The subject is also a relatively complex one and, at best, only peripheral to the goals of the practicing simulator. We will therefore not delve too deeply into the issues involved, but will merely focus on three examples, all of which can be valuable for large-scale MD simulation; the first employs message-passing parallelism, the second involves parallelism achieved by the use of computational threads and shared memory, and the third demonstrates how to rearrange data to achieve effective vector processing†.

17.2 The quest for performance

It comes as no surprise to learn that spreading a computational task over several processors is a way to complete the job sooner. Multiprocessor systems benefit from an economy of scale, and high performance computers now almost always

† The vectorizeable program can be run on a nonvector machine in its present form (although this would only be done only for development purposes), and the parallel programs can also be run on a uniprocessor system (also just for testing) if the necessary supporting software is available.

consist of at least a few processing units, if not more; the number of processors per machine begins at two and extends into the thousands, and there is apparently no limit in sight. The ideal kind of problem for such a machine – assuming that each processor, or each small group of processors, has its own private memory – is one that can be partitioned into a number of smaller computations that are carried out in parallel on all the processors, without too much data having to be shared between them. Molecular dynamics systems with short-range forces fall comfortably into this category.

Another means of extracting higher performance from computer hardware is to resort to vector processing. This entails pipelining the computations in assembly-line fashion. The constraint placed on a computation, if it is to be effectively vectorized, is that data items should be organized into relatively long vectors in such a way that all items can be processed independently, without fear that the processing of one item will affect a later one. This condition is not always easily satisfied.

Needless to say, both these architectural features are increasingly likely to be encountered†. The price of utilizing them effectively is increased algorithm and software complexity. Unlike 'simple' computers, where optimizing compilers can take a typical program and massage it to achieve reasonable performance, the needs of parallel and vector processing cannot always be resolved in this way, because it is not always obvious – assuming that it is possible at all – how to automate the process of optimally mapping an intrinsically serial computation onto the more 'complex' computer architectures. In addition to actually producing a working parallel and/or vectorized program, the efficiency of the end result must be considered: a parallel computation with large interprocessor communication overheads is doomed to failure, as is a vectorized computation that either uses the vector capability inefficiently because the vectors are too short, or spends a disproportionate amount of time rearranging data into vectorizeable form.

17.3 Techniques for parallel processing

Living with multiprocessor computers

The taxonomy of multiprocessing is far from simple. Among the factors to be taken into account are whether individual processors all carry out the same operation during each cycle, or whether they are able to act independently; whether each

† Beyond these two most visible features of many modern computers, there are a number of more subtle and less adequately documented processor features that can have a significant impact on performance. Just to name some of them: the internal processor registers, primary and secondary (and sometimes even tertiary) caches, address space mapping and memory interleaving. An algorithm that in some way conflicts with certain engineering design assumptions, for example, in its pattern of memory accesses, can experience a drastic performance drop. Beyond drawing the attention of the reader to the existence of such potential pitfalls, there is little more that can be said without addressing each computer model individually.

processor has its own private memory, or all share a common memory, or both; whether processors communicate with one another by passing messages across a communication network, or through common memory; the nature and topology of the communication network, if any. Some of these features can influence the way in which software ought to be organized.

We will avoid becoming involved in these issues here by assuming a particular generic architecture, one that is in fact widespread because it is the simplest to implement. The assumption is that there are several independent processors, each with private memory, communicating over a network – the message-passing approach. Systems of this kind are easily assembled by simply linking modest personal computers using standard (or above standard) network hardware; depending on the nature of the computation, the performance may or may not be satisfactory, but the same general approach is often found embedded in more customized hardware designs.

In order for the message-passing scheme to work efficiently, it is vital that the communication overheads be kept low compared with the time a processor spends computing; the ideal application consists of a lot of calculation with small amounts of data being transferred from time to time. The communication overhead is composed of two parts: the time to initiate a message transfer, typically a constant value, and a transfer time that is roughly proportional to the message length. There is also the issue of load balancing; obviously if all processors can be kept busy doing useful computing the overall system utilization will be optimal, but if some processors have more work to do than others, overall effectiveness is reduced.

Algorithm organization

There are different ways to partition an MD computation among multiple processors. The act of partitioning can focus on the computations, on the atoms involved, or on the simulation region. While all three elements are part of every scheme, the emphasis differs; this has a considerable impact on the memory and communication requirements of each method.

If it is just the computations that are partitioned, then all information about the system resides in the memory of each processor, but each only carries out the interaction computations for certain atom pairs. The information about the forces on each atom is then combined. This approach is extremely wasteful in terms of memory and is best suited to small computations only (and, perhaps, shared-memory computers).

The second partitioning scheme is based on the atoms themselves, and assigns each atom to a particular processor for the duration of the simulation, irrespective of its spatial location [rai89]. While conceptually simple, large amounts of

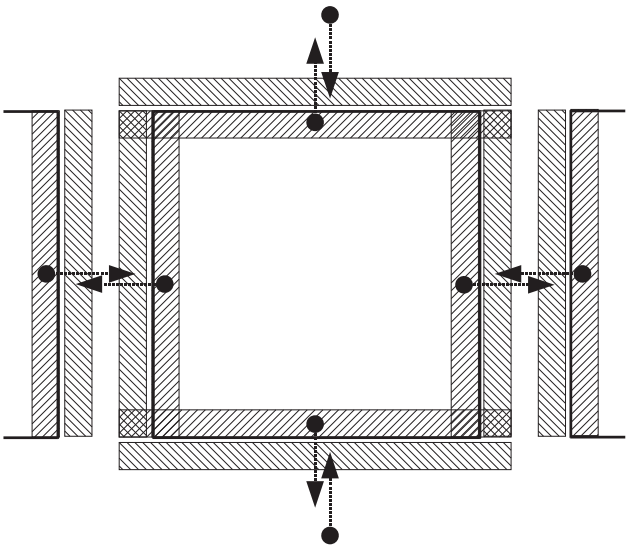

Fig. 17.1. The portion of the simulation region (for a two-dimensional subdivision) represented by the square outline is handled by a single processor; it contains shaded areas denoting subregions whose atoms interact with atoms in adjacent processors, and is surrounded by shaded areas denoting subregions from adjacent processors whose interactions must be taken into account; arrows indicate the flow of data between processors.

communication are required to handle interactions between atoms assigned to different processors. For long-range interactions this may not be a problem, but for the short-range case the third choice turns out to be far more efficient.

The third scheme subdivides space and assigns each processor a particular subregion [rap91b]. All the atoms that are in a given subregion at some moment in time reside in the processor responsible and when an atom moves between subregions all the associated variables are explicitly transferred from one processor to another. Thus there is economy insofar as memory is concerned, and also in the communication required to allow atoms to transfer between processors, since comparatively few atoms make such a move during a single timestep. More importantly, assuming there are some 10^4 or more atoms per subregion (in three dimensions) and a relatively short-ranged potential, most of the interactions will occur among atoms in the subregion and relatively few between atoms in adjacent subregions – see Figure 17.1. In order to accommodate the latter, copies of the coordinates of atoms close to any subregion boundary are transferred to the processor handling the adjacent subregion prior to the interaction computation. This transfer also involves only a small fraction of the atoms.

It is this third scheme that will be described here. The only requirement is that communication be reasonably efficient, with the associated system overheads –

both hardware and software – kept to a low level in comparison with the amount of computation involved. Under these circumstances, both computation speed and memory requirements scale in the expected linear way with the numbers of atoms and processors†.

17.4 Distributed computation

Overview

The program described here is based on the three-dimensional soft-sphere computation described in §3.4 that uses both cells and neighbor lists. Some parts of the original program can be used unchanged, but wherever atoms become aware of the existence of subregions the computations must take this into account. While conceptually straightforward, the distributed computation involves numerous details that must be treated with care.

The functions handling tasks related to communications are referenced in a generic form that does not assume any particular message-passing software package. The text describes what each of these functions is supposed to do and, in practice, since all such software must provide similar functionality, a simple series of macro substitutions may be all that is required to produce a working program; an actual implementation concludes the section.

In addition to the interaction calculations, integration of the equations of motion, initialization and measurements, each of which is modified to a lesser or greater degree for the distributed implementation, it is also necessary to specify which processor is responsible for each spatial subregion, to identify the atoms participating in each data transfer and to carry out the transfers. Several kinds of data transfer are used:

- for interaction calculations it is necessary to copy information about the coordinates of atoms close to subregion boundaries;
- when atoms move between subregions their entire descriptions are transferred;
- while making measurements, the values computed separately in each processor must be combined to produce an overall result.

Basic computations

The program description♠ begins with the neighbor-list construction, on the assumption that the atoms are already in the correct processors and that copies of the coordinates of atoms in adjacent subregions have been made available. These tasks

† In practice, the performance depends on the details of the processor architecture, communication infrastructure and operating system; performance that grows linearly with the number of processors is not always achievable.

♠ `pr_17_1`

will be described later. Two new variables play an important role here; *nMolMe* is the number of atoms currently in the subregion and *nMolCopy* is the number of additional atoms from adjacent subregions (processors) whose coordinates have been copied to this processor because they are potential interaction candidates. Given the additional information, this version of *BuildNebrList* (and the other functions that follow) can be executed independently on each processor.

```
#define OFFSET_LIST                                                  \
   {{7}, {7,8,9}, {9}, {6,7,13}, {5,6,7,8,9,10,11,12,13},           \
    {9,10,11}, {13}, {11,12,13}, {11}, {2,7},                        \
    {2,3,4,7,8,9}, {4,9}, {1,2,6,7,13},                              \
    {0,1,2,3,4,5,6,7,8,9,10,11,12,13}, {4,9,10,11}, {13},            \
    {11,12,13}, {11}}
#define OFFSET_LEN                                                   \
   {1,3,1,3,9,3,1,3,1,2,6,2,5,14,4,1,3,1}

void BuildNebrList ()
{
  VecR cellBase, dr, invWid, rs, t1, t2;
  VecI cc, m1v, m2v, vOff[] = OFFSET_VALS;
  real rrNebr;
  int c, indx, j1, j2, m1, m1x, m1y, m1z, m2, n, offset, tOffset,
     vOffList[][N_OFFSET] = OFFSET_LIST, vOffTableLen[] = OFFSET_LEN;

  VAddCon (t1, cells, -2.);
  VSub (t2, subRegionHi, subRegionLo);
  VDiv (invWid, t1, t2);
  VSetAll (t1, 1.);
  VDiv (t1, t1, invWid);
  VSub (cellBase, subRegionLo, t1);
  rrNebr = Sqr (rCut + rNebrShell);
  for (n = nMolMe + nMolCopy; n < nMolMe + nMolCopy + VProd (cells);
     n ++) cellList[n] = -1;
  for (n = 0; n < nMolMe + nMolCopy; n ++) {
    VSub (rs, mol[n].r, cellBase);
    VMul (cc, rs, invWid);
    c = VLinear (cc, cells) + nMolMe + nMolCopy;
    cellList[n] = cellList[c];
    cellList[c] = n;
  }
  nebrTabLen = 0;
  for (m1z = 0; m1z < cells.z - 1; m1z ++) {
    for (m1y = 0; m1y < cells.y; m1y ++) {
      for (m1x = 0; m1x < cells.x; m1x ++) {
        VSet (m1v, m1x, m1y, m1z);
        tOffset = 13;
        if (m1z == 0) tOffset -= 9;
        if (m1y == 0) tOffset -= 3;
        else if (m1y == cells.y - 1) tOffset += 3;
```

```
      if (m1x == 0) tOffset -= 1;
      else if (m1x == cells.x - 1) tOffset += 1;
      m1 = VLinear (m1v, cells) + nMolMe + nMolCopy;
      for (offset = 0; offset < vOffTableLen[tOffset]; offset ++) {
        indx = vOffList[tOffset][offset];
        VAdd (m2v, m1v, vOff[indx]);
        m2 = VLinear (m2v, cells) + nMolMe + nMolCopy;
        DO_CELL (j1, m1) {
          DO_CELL (j2, m2) {
            if (m1 != m2 || j2 < j1) {
              VSub (dr, mol[j1].r, mol[j2].r);
              ... (identical to standard version) ...
}
```

Several points should be noted†. The cell array is defined separately for each subregion and includes an additional layer of cells that completely surrounds the subregion; it is here that all the atoms copied from adjacent subregions are located. Periodic boundaries are not mentioned at this stage of the computation because, as will be shown later, they are treated during the copying operation. The vector variables *subRegionLo* and *subRegionHi* contain the spatial limits of the subregion handled by each processor. Because periodic boundaries are handled by other means, the range of adjacent cells scanned during neighbor-list construction depends on the cell location; for cells that are located on a subregion face, edge or corner, fewer adjacent cells need be examined – the data in *vOffList* makes provision for all 18 distinct cases and the way it is used together with *vOffTableLen* should be apparent from the listing.

Only minor changes are needed in the force calculation. The principal reason for the changes is in order to evaluate accumulated properties such as the potential energy and virial sum. The force computation does not distinguish between atoms that really belong to the subregion and those that are merely copies from an adjacent subregion, since the force contributions associated with the latter are simply discarded afterwards. Energy and virial sums are treated in a manner that ensures the correct contributions from atom pairs that interact across a subregion boundary. Once again, no mention of periodic wraparound is needed.

```
void ComputeForces ()
{
  ...
  for (n = 0; n < nMolMe + nMolCopy; n ++) VZero (mol[n].ra);
  for (n = 0; n < nebrTabLen; n ++) {
    j1 = nebrTab[2 * n];
    j2 = nebrTab[2 * n + 1];
    VSub (dr, mol[j1].r, mol[j2].r);
```

† The reader new to distributed processing should bear in mind that all variables are local to each processor; the concept of a global variable does not exist in a message-passing environment.

```
    rr = VLenSq (dr);
    if (rr < rrCut) {
      ... (identical to standard version) ...
      if (j1 < nMolMe) {
        uSum += uVal;
        virSum += fcVal * rr;
      }
      if (j2 < nMolMe) {
        uSum += uVal;
        virSum += fcVal * rr;
      }
    }
  }
  uSum *= 0.5;
  virSum *= 0.5;
}
```

Integration uses the leapfrog method. The only change to *LeapfrogStep* is the use of *nMolMe* for the number of atoms to be processed. The atoms whose coordinates were copied from adjacent subregions are readily excluded from this calculation since they appear in the *mol* array after the first *nMolMe* entries corresponding to atoms in the subregion.

In establishing the initial state we encounter communication† for the first time, albeit still in a very minor role. All the initialization has been combined into a single function *InitState* that is executed concurrently on all processors, with each processor determining which atoms belong to its subregion. Since a particular atom is no longer associated with a fixed memory location, unlike the uniprocessor version of the program, each atom is labeled with a unique identifier *mol[].id*. Although not really needed in this particular example, it is sometimes necessary to be able to distinguish individual atoms (a polymer fluid would be one such example). A simple cubic lattice is used for the initial state. The macros *VGe* and *VLt* (§18.2) compare vector components.

```
void InitState ()
{
  VecR vSumL;
  VecR c, gap;
  int n, np, nx, ny, nz;
  ValList msg1[] = {
    ValR (vSumL)
  };
  ValList msg2[] = {
```

† The distributed environment makes its presence felt both during algorithm development and at the programming stage; any features that are added to the simulation must take this extra software overhead into account. Much of the communication processing is hidden away from the application, as indeed it should be. It is a pity that even the few details included here have to be mentioned at all, but this situation will persist until some standardized method (or language) for programming parallel computers achieves widespread acceptance.

```
    ValR (vSum)
  };

  VDiv (gap, region, initUcell);
  VZero (vSumL);
  nMol = 0;
  nMolMe = 0;
  for (nz = 0; nz < initUcell.z; nz ++) {
    for (ny = 0; ny < initUcell.y; ny ++) {
      for (nx = 0; nx < initUcell.x; nx ++) {
        VSet (c, nx + 0.25, ny + 0.25, nz + 0.25);
        VMul (c, c, gap);
        VVSAdd (c, -0.5, region);
        VRand (&mol[nMolMe].rv);
        if (VGe (c, subRegionLo) && VLt (c, subRegionHi)) {
          mol[nMolMe].r = c;
          VScale (mol[nMolMe].rv, velMag);
          VVAdd (vSumL, mol[nMolMe].rv);
          mol[nMolMe].id = nMol;
          ++ nMolMe;
          if (nMolMe > nMolMeMax) {
            errCode = ERR_TOO_MANY_MOLS;
            -- nMolMe;
          }
        }
        ++ nMol;
      }
    }
  }
  if (ME_BOSS) {
    vSum = vSumL;
    DO_SLAVES {
      MsgRecvUnpack (np, 121, msg1);
      VVAdd (vSum, vSumL);
    }
    VScale (vSum, 1. / nMol);
    MsgBcPackSend (122, msg2);
  } else {
    MsgPackSend (0, 121, msg1);
    MsgBcRecvUnpack (122, msg2);
  }
  DO_MOL_ME {
    VVSub (mol[n].rv, vSum);
    VZero (mol[n].ra);
  }
}
```

The latter part of `InitState` includes several communication operations and a summary of what occurs here is as follows. There is a macro `ME_BOSS` that is able to distinguish one processor from all the others; the mission of the processor

having the status of 'boss' is to collect the values of the array *vSum* from all the other 'slave' processors after each has computed its local values, evaluate the total sums, compute the values that each processor must subtract from all its atoms' velocities to ensure a zero center of mass velocity and, finally, broadcast these values to each of the slave processors. The slaves perform the complementary task; they accumulate *vSum*, send it to the boss and wait for the necessary values to be returned. Although one particular processor has been designated the boss and the remainder slaves, this should not detract from the fact that, with a couple of minor exceptions, all processors perform completely equivalent tasks and are mutually synchronized by the data transfers that occur throughout the calculation.

The function *MsgRecvUnpack* waits for, and accepts, the message specified in the *ValList* argument (see §18.5) from a designated processor; the numerical argument appearing here and in other communication functions is an arbitrary value used by the message software to distinguish between different kinds of message†. This function also processes the received message, by storing its contents sequentially in the indicated locations. *MsgPackSend* performs the complementary operation, by collecting the specified values into the message body and then sending it to the designated processor. There is also a broadcast capability; the function *MsgBcPackSend* is used by the boss processor to broadcast an identical message to all the slaves, and each slave processor will use *MsgBcRecvUnpack* to receive this message. Other communication functions will appear in due course as needed.

Another function that must retrieve a small amount of information from each processor is *EvalProps*. The copy of *EvalProps* running on the boss processor produces the same final results as the uniprocessor version, but it must first collect the partial results from the slaves.

```
void EvalProps ()
{
  VecR vSumL;
  real uSumL, virSumL, vv, vvMax, vvMaxL, vvSumL;
  int errCodeL, n, np;
  ValList msg[] = {
    ValI (errCodeL),
    ValR (uSumL),
    ValR (virSumL),
    ValR (vSumL),
    ValR (vvMaxL),
    ValR (vvSumL)
  };

  VZero (vSumL);
```

† This feature is not used here, but it is especially helpful during development for ensuring that messages sent and received correspond to one another.

```
  vvSumL = 0.;
  vvMaxL = 0.;
  DO_MOL_ME {
    VVAdd (vSumL, mol[n].rv);
    vv = VLenSq (mol[n].rv);
    vvSumL += vv;
    vvMaxL = Max (vvMaxL, vv);
  }
  if (ME_BOSS) {
    vSum = vSumL;
    vvSum = vvSumL;
    vvMax = vvMaxL;
    DO_SLAVES {
      MsgRecvUnpack (np, 161, msg);
      if (errCodeL != ERR_NONE) errCode = errCodeL;
      vvMax = Max (vvMax, vvMaxL);
      vvSum += vvSumL;
      VVAdd (vSum, vSumL);
      uSum += uSumL;
      virSum += virSumL;
    }
    dispHi += sqrt (vvMax) * deltaT;
    if (dispHi > 0.5 * rNebrShell) nebrNow = 1;
    kinEnergy.val = 0.5 * vvSum / nMol;
    totEnergy.val = kinEnergy.val + uSum / nMol;
    pressure.val = density * (vvSum + virSum) / (nMol * NDIM);
  } else {
    errCodeL = errCode;
    uSumL = uSum;
    virSumL = virSum;
    MsgPackSend (0, 161, msg);
  }
}
```

Finally, the appropriately modified version of the function `SingleStep` is

```
void SingleStep ()
{
  ValList msg[] = {
    ValI (moreCycles),
    ValI (nebrNow)
  };

  ++ stepCount;
  timeNow = stepCount * deltaT;
  LeapfrogStep (1);
  if (nebrNow > 0) DoParlMove ();
  DoParlCopy ();
  if (nebrNow > 0) {
    nebrNow = 0;
    if (ME_BOSS) dispHi = 0.;
```

```
    BuildNebrList ();
  }
  ComputeForces ();
  LeapfrogStep (2);
  EvalProps ();
  if (ME_BOSS) {
    MsgBcPackSend (151, msg);
  } else MsgBcRecvUnpack (151, msg);
  if (ME_BOSS) {
    if (stepCount >= stepEquil) {
      AccumProps (1);
      if (stepCount > stepEquil &&
          (stepCount - stepEquil) % stepAvg == 0) {
        AccumProps (2);
        PrintSummary (stdout);
        AccumProps (0);
      }
    }
  }
}
```

The boss processor is responsible for collecting the results and handling the output. The communication operations appearing here provide each processor with the current value of `nebrNow` informing it whether an update of the neighbor list is due. The functions `DoParlMove` and `DoParlCopy`, described below, deal with the interprocessor data transfers needed for interaction calculations and atom movements.

An important detail should be apparent from the way communications in the above functions are organized. Obviously, there must be a one-to-one correspondence between messages sent and messages received. Equally significant, however, is the fact that these message transfer operations are used to synchronize the processors†. When writing parallel software based on a message-passing paradigm, it is important to plan the communications carefully, otherwise deadlock and race conditions can occur that are difficult to diagnose.

Message-passing operations

We now turn to the functions where the majority of the interprocessor communication occurs, namely, the movement of atoms between subregions and the copying of coordinate data prior to the interaction calculations.

The functions responsible for deciding which atoms should be moved, and then actually doing the work, including coordinate adjustment for periodic wraparound, are as follows. There are six directions (in three dimensions) to be considered,

† None of the additional capabilities found in parallel software systems for explicit synchronization, such as creating a barrier that no processor can pass until it receives permission, are required here.

and each is treated in turn; atoms can of course participate in more than one such transfer. The special case that a processor is its own neighbor, which occurs when no subdivision of the region is made in a particular direction, is also taken into account. Once all the moves are complete, each processor compresses its own data to eliminate gaps in the arrays.

```
#define OutsideProc(b)                                          \
   (sDir == 0 && VComp (mol[n].r, dir) <                        \
   VComp (subRegionLo, dir) + b ||                              \
   sDir == 1 && VComp (mol[n].r, dir) >=                        \
   VComp (subRegionHi, dir) - b)                                        5
#define NWORD_MOVE  (2 * NDIM + 1)

void DoParlMove ()
{
  int dir, n, nIn, nt, sDir;                                            10

  for (dir = 0; dir < NDIM; dir ++) {
    for (sDir = 0; sDir < 2; sDir ++) {
      nt = 0;
      DO_MOL_ME {                                                       15
        if (mol[n].id >= 0) {
          if (OutsideProc (0.)) {
            trPtr[sDir][trBuffMax * dir + nt] = n;
            ++ nt;
            if (NWORD_MOVE * nt > NDIM * trBuffMax) {                   20
              errCode = ERR_COPY_BUFF_FULL;
              -- nt;
            }
          }
        }                                                               25
      }
      nOut[sDir][dir] = nt;
    }
    for (sDir = 0; sDir < 2; sDir ++) {
      nt = nOut[sDir][dir];                                             30
      PackMovedData (dir, sDir, &trPtr[sDir][trBuffMax * dir], nt);
      if (VComp (procArraySize, dir) > 1) {
        MsgSendInit ();
        MsgPackI (&nt, 1);
        MsgPackR (trBuff, NWORD_MOVE * nt);                             35
        if (sDir == 1) MsgSendRecv (VComp (procNebrLo, dir),
           VComp (procNebrHi, dir), 140 + 2 * dir + 1);
        else MsgSendRecv (VComp (procNebrHi, dir),
           VComp (procNebrLo, dir), 140 + 2 * dir);
        MsgRecvInit ();                                                 40
        MsgUnpackI (&nIn, 1);
        MsgUnpackR (trBuff, NWORD_MOVE * nIn);
      } else nIn = nt;
      if (nMolMe + nIn > nMolMeMax) {
```

```
        errCode = ERR_TOO_MANY_MOVES;
        nIn = 0;
      }
      UnpackMovedData (nIn);
    }
  }
  RepackMolArray ();
}
```

The only new communication function appearing here is *MsgSendRecv*, which both transmits data to a neighboring processor and receives data from the opposite neighbor; in some message-passing systems a function of this kind can be used to achieve overlapped data transfers. Several functions for packing and unpacking messages also make an appearance here (such as *MsgPackR* and *MsgUnpackI*) and these will be discussed later. The size of the multiprocessor configuration and the way the simulation region is subdivided – for example, into slices spanning the entire region, or into smaller boxes as in Figure 17.1 – are specified by *procArraySize*, while the location of each processor in the (up to three-dimensional) multiprocessor array is specified by *procArrayMe*†.

Message packing and unpacking are two-stage processes; the work that is specific to the application data (shown below) is kept separate from the actual filling or emptying of message buffers (in *DoParlMove* above). The periodic boundaries are addressed at this stage.

```
void PackMovedData (int dir, int sDir, int *trPtr, int nt)
{
  real rShift;
  int j;

  rShift = 0.;
  if (sDir == 1 &&
     VComp (procArrayMe, dir) == VComp (procArraySize, dir) - 1)
     rShift = - VComp (region, dir);
  else if (sDir == 0 && VComp (procArrayMe, dir) == 0)
     rShift = VComp (region, dir);
  for (j = 0; j < nt; j ++) {
    VToLin (trBuff, NWORD_MOVE * j, mol[trPtr[j]].r);
    trBuff[NWORD_MOVE * j + dir] += rShift;
    VToLin (trBuff, NWORD_MOVE * j + NDIM, mol[trPtr[j]].rv);
    trBuff[NWORD_MOVE * j + 2 * NDIM] = mol[trPtr[j]].id;
    mol[trPtr[j]].id = -1;
  }
}
```

† Note the checks – here and subsequently – to ensure storage arrays are not overfilled; such safety measures should always be present whenever unpredictable amounts of data are involved.

```
void UnpackMovedData (int nIn)
{
  int j;

  for (j = 0; j < nIn; j ++) {
    VFromLin (mol[nMolMe + j].r, trBuff, NWORD_MOVE * j);
    VFromLin (mol[nMolMe + j].rv, trBuff, NWORD_MOVE * j + NDIM);
    mol[nMolMe + j].id = trBuff[NWORD_MOVE * j + 2 * NDIM];
  }
  nMolMe += nIn;
}
```

Repacking is required to remove gaps due to atoms that have moved out,

```
void RepackMolArray ()
{
  int j, n;

  j = 0;
  DO_MOL_ME {
    if (mol[n].id >= 0) {
      mol[j] = mol[n];
      ++ j;
    }
  }
  nMolMe = j;
}
```

The functions for copying the coordinates of atoms close to subregion boundaries to adjacent processors (prior to the interaction calculations) are very similar; the sets of atoms involved are updated only when the neighbor list is about to be rebuilt.

```
#define NWORD_COPY  (NDIM + 1)

void DoParlCopy ()
{
  real rCutExt;
  int dir, n, nIn, nt, sDir;

  rCutExt = rCut + rNebrShell;
  nMolCopy = 0;
  for (dir = 0; dir < NDIM; dir ++) {
    if (nebrNow > 0) {
      for (sDir = 0; sDir < 2; sDir ++) {
        nt = 0;
        for (n = 0; n < nMolMe + nMolCopy; n ++) {
          if (OutsideProc (rCutExt)) {
            trPtr[sDir][trBuffMax * dir + nt] = n;
```

```
            ++ nt;
            if (NWORD_COPY * nt > NDIM * trBuffMax) {
              errCode = ERR_COPY_BUFF_FULL;
              -- nt;
            }
          }
        }
        nOut[sDir][dir] = nt;
      }
    }
    for (sDir = 0; sDir < 2; sDir ++) {
      nt = nOut[sDir][dir];
      PackCopiedData (dir, sDir, &trPtr[sDir][trBuffMax * dir], nt);
      if (VComp (procArraySize, dir) > 1) {
        MsgSendInit ();
        MsgPackI (&nt, 1);
        MsgPackR (trBuff, NWORD_COPY * nt);
        if (sDir == 1) MsgSendRecv (VComp (procNebrLo, dir),
           VComp (procNebrHi, dir), 130 + 2 * dir + 1);
        else MsgSendRecv (VComp (procNebrHi, dir),
           VComp (procNebrLo, dir), 130 + 2 * dir);
        MsgRecvInit ();
        MsgUnpackI (&nIn, 1);
        MsgUnpackR (trBuff, NWORD_COPY * nIn);
      } else nIn = nt;
      if (nMolMe + nMolCopy + nIn > nMolMeMax) {
        errCode = ERR_TOO_MANY_COPIES;
        nIn = 0;
      }
      UnpackCopiedData (nIn);
    }
  }
}

void PackCopiedData (int dir, int sDir, int *trPtr, int nt)
{
  real rShift;
  int j;

  rShift = 0.;
  if (sDir == 1 &&
     VComp (procArrayMe, dir) == VComp (procArraySize, dir) - 1)
     rShift = - VComp (region, dir);
  else if (sDir == 0 && VComp (procArrayMe, dir) == 0)
     rShift = VComp (region, dir);
  for (j = 0; j < nt; j ++) {
    VToLin (trBuff, NWORD_COPY * j, mol[trPtr[j]].r);
    trBuff[NWORD_COPY * j + dir] += rShift;
    trBuff[NWORD_COPY * j + NDIM] = mol[trPtr[j]].id;
  }
}
```

```
void UnpackCopiedData (int nIn)
{
  int j;

  for (j = 0; j < nIn; j ++) {
    VFromLin (mol[nMolMe + nMolCopy + j].r, trBuff, NWORD_COPY * j);
    mol[nMolMe + nMolCopy + j].id = trBuff[NWORD_COPY * j + NDIM];
  }
  nMolCopy += nIn;
}
```

The main program and initialization function for the distributed computation are as follows.

```
int main (int argc, char **argv)
{
  MsgStartup ();
  if (ME_BOSS) {
    GetNameList (argc, argv);
    PrintNameList (stdout);
  }
  InitSlaves ();
  NebrParlProcs ();
  SetParams ();
  SetupJob ();
  moreCycles = 1;
  while (moreCycles) {
    SingleStep ();
    if (stepCount == stepLimit) moreCycles = 0;
  }
  MsgExit ();
}

void SetupJob ()
{
  AllocArrays ();
  InitRand (randSeed);
  stepCount = 0;
  InitState ();
  nebrNow = 1;
  if (ME_BOSS) AccumProps (0);
}
```

We have glossed over two details that are intimately associated with the message-passing software, namely, how to ensure that all the processors run copies of the same program and how each processor obtains its distinct identity `procMe` (which determines, among other things, who is the boss). The former may be automatic,

or require some user action either before or while running the program; the latter may be as simple as a function call†.

New variables introduced in this program are

```
VecR subRegionHi, subRegionLo;
VecI procArrayMe, procArraySize, procNebrHi, procNebrLo;
real *trBuff;
int **trPtr, nOut[2][NDIM], errCode, nMolCopy, nMolMe, nMolMeMax,
   nProc, procMe, trBuffMax;
```

There are new input data items specifying the maximum number of atoms a processor can hold (including copies), the number of processors and the way the simulation region is subdivided, and the size of the buffers used for collecting data to be transferred,

```
  NameI (nMolMeMax),
  NameI (procArraySize),
  NameI (trBuffMax),
```

In `SetParams`, the subregion limits are established and the cell array size (with an extra cell at each end) is determined from the size of the subregion,

```
  VecR w;
  ...
  nebrTabMax = nebrTabFac * nMolMeMax;
  VDiv (w, region, procArraySize);
  VMul (subRegionLo, procArrayMe, w);
  VVSAdd (subRegionLo, -0.5, region);
  VAdd (subRegionHi, subRegionLo, w);
  VScale (w, 1. / (rCut + rNebrShell));
  VAddCon (cells, w, 2.);
```

Memory allocation differs from the standard case,

```
void AllocArrays ()
{
  int k;

  AllocMem (mol, nMolMeMax, Mol);
  AllocMem (cellList, VProd (cells) + nMolMeMax, int);
  AllocMem (nebrTab, 2 * nebrTabMax, int);
  AllocMem (trBuff, NDIM * trBuffMax, real);
  AllocMem2 (trPtr, 2, NDIM * trBuffMax, int);
}
```

† Examination of the software documentation will resolve these questions.

Finally, each processor discovers who its neighbors are, based on the value of its own individual copy of `procMe`,

```
void NebrParlProcs ()
{
  VecI t;
  int k;

  nProc = VProd (procArraySize);
  procArrayMe.x = procMe % procArraySize.x;
  procArrayMe.y = (procMe / procArraySize.x) % procArraySize.y;
  procArrayMe.z = procMe / (procArraySize.x * procArraySize.y);
  for (k = 0; k < NDIM; k ++) {
    t = procArrayMe;
    VComp (t, k) = (VComp (t, k) +
       VComp (procArraySize, k) - 1) % VComp (procArraySize, k);
    VComp (procNebrLo, k) = VLinear (t, procArraySize);
    t = procArrayMe;
    VComp (t, k) = (VComp (t, k) + 1) % VComp (procArraySize, k);
    VComp (procNebrHi, k) = VLinear (t, procArraySize);
  }
}
```

and the boss processor, the only one with access to the input data file, distributes its contents to all the other processors,

```
void InitSlaves ()
{
  ValList initVals[] = {
    ValR (deltaT),
    ValR (density),
    ValI (initUcell),
    ValI (nebrTabFac),
    ValI (nMolMeMax),
    ValI (procArraySize),
    ValI (randSeed),
    ValR (rNebrShell),
    ValI (stepEquil),
    ValI (stepLimit),
    ValR (temperature),
    ValI (trBuffMax)
  };

  if (ME_BOSS) {
    MsgBcPackSend (111, initVals);
  } else MsgBcRecvUnpack (111, initVals);
}
```

Additional details

In order to complete the distributed MD demonstration, we provide the extra information needed to produce a working program for use with MPI, a message passing software standard [gro96] on which several widely available software packages are based. The two missing details are:

- initialize the computation and let each processor know its identity;
- express the generic communication functions as actual MPI functions.

The following is required by `main` to initialize the MPI system, allocate buffer storage, and obtain the values of `nProc` and `procMe`. We assume the reader is familiar with the MPI functions (all prefixed with `MPI_`) used here†.

```
#define MsgStartup()                                    \
   MPI_Init (&argc, &argv),                             \
   MPI_Comm_size (MPI_COMM_WORLD, &nProc),              \
   MPI_Comm_rank (MPI_COMM_WORLD, &procMe),             \
   AllocMem (buffSend, BUFF_LEN, real),                 \
   AllocMem (buffRecv,BUFF_LEN, real)
```

We also require

```
#define BUFF_LEN  64000

MPI_Status mpiStatus;
real *buffRecv, *buffSend;
int buffWords, mpiNp;
```

The replacement of the generic communication functions, either by calls to their MPI equivalents, or by other function calls or operations, is accomplished with the following macro definitions:

```
#define MsgSend(to, id)                                 \
   MPI_Send (buffSend, buffWords, MPI_MY_REAL, to, id,  \
   MPI_COMM_WORLD)
#define MsgRecv(from, id)                               \
   MPI_Recv (buffRecv, BUFF_LEN, MPI_MY_REAL, from, id, \
   MPI_COMM_WORLD, &mpiStatus)
#define MsgSendRecv(from, to, id)                       \
   MPI_Sendrecv (buffSend, buffWords, MPI_MY_REAL, to, id,  \
   buffRecv, BUFF_LEN, MPI_MY_REAL, from, id,           \
   MPI_COMM_WORLD, &mpiStatus)
```

† MPI typically requires the value of `nProc` to be entered on the command line, so it must agree with the value obtained from `procArraySize` which specifies the 'shape' of the subdivision. The appropriate MPI runtime software, with the necessary include files and libraries, must of course be installed on the computer in order to use this program. Information on how to compile and run programs based on MPI is to be found in the appropriate user documentation. (The first edition used the alternative PVM [gei94] package.)

```
#define MsgBcSend(id)                                          \
   for (mpiNp = 1; mpiNp < nProc; mpiNp ++)                    \
      MsgSend (mpiNp, id)
#define MsgBcRecv(id)      MsgRecv (0, id)
#define MsgExit()          MPI_Finalize (); exit (0)
#define MsgSendInit()      buffWords = 0
#define MsgRecvInit()      buffWords = 0
#define MsgPackR(v, nv)    DoPackReal (v, nv)
#define MsgPackI(v, nv)    DoPackInt (v, nv)
#define MsgUnpackR(v, nv)  DoUnpackReal (v, nv)
#define MsgUnpackI(v, nv)  DoUnpackInt (v, nv)
```

Further useful definitions are

```
#define MsgRecvUnpack(from, id, ms)                            \
   {MsgRecv (from, id);                                        \
    MsgRecvInit ();                                            \
    UnpackValList (ms, sizeof (ms));}
#define MsgPackSend(to, id, ms)                                \
   {MsgSendInit ();                                            \
    PackValList (ms, sizeof (ms));                             \
    MsgSend (to, id);}
#define MsgBcRecvUnpack(id, ms)                                \
   {MsgBcRecv (id);                                            \
    MsgRecvInit ();                                            \
    UnpackValList (ms, sizeof (ms));}
#define MsgBcPackSend(id, ms)                                  \
   if (nProc > 1) {                                            \
     MsgSendInit ();                                           \
     PackValList (ms, sizeof (ms));                            \
     MsgBcSend (id);                                           \
   }
#define ME_BOSS      (procMe == 0)
#define MPI_MY_REAL  MPI_DOUBLE
#define DO_MOL_ME    for (n = 0; n < nMolMe; n ++)
#define DO_SLAVES    for (np = 1; np < nProc; np ++)
```

The packing and unpacking operations for blocks of data are

```
void DoPackReal (real *w, int nw)
{
  int n;

  if (buffWords + nw >= BUFF_LEN) errCode = ERR_MSG_BUFF_FULL;
  else {
    for (n = 0; n < nw; n ++) buffSend[buffWords + n] = w[n];
    buffWords += nw;
  }
}
```

```
void DoUnpackReal (real *w, int nw)
{
  int n;

  for (n = 0; n < nw; n ++) w[n] = buffRecv[buffWords + n];
  buffWords += nw;
}
```

together with analogous functions `DoPackInt` and `DoUnpackInt`. Finally, packing of `ValList` arrays is carried out by

```
void PackValList (ValList *list, int size)
{
  int k;

  for (k = 0; k < size / sizeof (ValList); k ++) {
    switch (list[k].vType) {
      case N_I:
        MsgPackI (list[k].vPtr, list[k].vLen);
        break;
      case N_R:
        MsgPackR (list[k].vPtr, list[k].vLen);
        break;
    }
  }
}
```

and there is a complementary function `UnpackValList`.

This approach is preferable to embedding actual MPI calls in the body of the program from the point of view of portability; note also that only a subset of the available MPI functionality is utilized, for the same reason.

17.5 Shared-memory parallelism

Overview

In the case of several processors sharing access to a common memory there are two approaches available. One is to use the same scheme for message passing described previously; depending on the computer and operating system, it is possible that this will run faster than if the processors have to communicate across a network – even a fast one – because of specialized functions for internal communication. The alternative is to use an approach based on threads (or 'lightweight' processes); these are basically replica copies of the computational process that access a common region of memory and are supposed to have a relatively low computational overhead associated with their use. This turns out to be a much simpler approach than message passing; it does, however, require care to ensure that data are accessed in a consistent manner by the different threads and, of course, it cannot be used

when processors do not share common memory. The overall efficiency depends on the processor architecture and the nature of the problem, and it is quite possible that performance will not scale as efficiently with the number of processors as the message-passing approach†.

Use of computational threads

The example used here♠ is, once again, a simple soft-sphere MD simulation. We will show how threads can be introduced into two key parts of the program. The first is the leapfrog integrator, included on account of its simplicity rather than because of its heavy computational requirements. The second involves the neighbor-list construction and force evaluation procedures, which is where most of the computation time is spent and which, therefore, stand to benefit most from thread usage. Several macro definitions are used to conceal programming details.

The leapfrog integration function is changed so that now it calls another function *LeapfrogStepT*. There will be one such call for each thread and it is this new function that does the real work.

```
void LeapfrogStep (int part)
{
  int ip;

  THREAD_PROC_LOOP (LeapfrogStepT, part);
}

void *LeapfrogStepT (void *tr)
{
  int ip, n;

  QUERY_THREAD ();
  switch (QUERY_STAGE) {
    case 1:
      THREAD_SPLIT_LOOP (n, nMol) {
        VVSAdd (mol[n].rv, 0.5 * deltaT, mol[n].ra);
        VVSAdd (mol[n].r, deltaT, mol[n].rv);
      }
      break;
    case 2:
      THREAD_SPLIT_LOOP (n, nMol)
         VVSAdd (mol[n].rv, 0.5 * deltaT, mol[n].ra);
      break;
  }
  return (NULL);
}
```

† Parallel compilers can, in principle, produce the kind of code described here, although it is often simpler and more efficient (and sometimes essential) to introduce the changes into the MD code manually.

♠ *pr_17_2*

The above functions need explanation. When *LeapfrogStep* is called, it spawns several threads that can execute concurrently on separate processors. It does this by running through a loop that starts each one of *nThread* processes separately. This is carried out by *THREAD_PROC_LOOP*. The first argument is the name of the function (here *LeapfrogStepT*) to be executed by the thread, the second (*part*) is a value to be passed across to the thread. The function *LeapfrogStep* returns only after all the threads have completed their work.

The function *LeapfrogStepT* is executed by each of the threads and, since the threads execute in parallel and in a totally unsynchronized manner, care is required to ensure that data are handled properly; in particular, any data written by one thread should not be accessed (either for reading or for writing) by another concurrent thread. The form of the function header and the *return* statement are a requirement of the thread functions.

The reference to the macro *QUERY_THREAD* produces the serial number of the thread, a value between 0 and *nThread-1*, which is placed in *ip*. It is used in *THREAD_SPLIT_LOOP*, which is a loop over a subset of the atoms defined as

```
#define THREAD_SPLIT_LOOP(j, jMax)                              \
   for (j = ip * jMax / nThread;                                \
      j < (ip + 1) * jMax / nThread; j ++)
```

The value of the argument *part* is supplied by the macro *QUERY_STAGE*. The outcome of executing all the threads is that the leapfrog update is applied to all atoms. A similar approach can be used, for example, in *ApplyBoundaryCond*, although functions that are responsible for only a small fraction of the overall workload may not warrant conversion to use threads.

The force computation is a little more intricate. It requires multiple neighbor lists, one for each thread, containing distinct subsets of atom pairs; individual atoms will generally appear in more than one of these lists. It also requires additional storage for acceleration and potential energy values that are computed separately for each of the lists and subsequently combined to produce the correct values. Note that allowing the different threads to update a common array of acceleration values would violate the restrictions on what threads are permitted to do with data and would be guaranteed to produce incorrect results.

Replacing the function *BuildNebrList* of §3.4 involves the following alterations:

```
void BuildNebrList ()
{
  int ip, n;

  for (n = nMol; n < nMol + VProd (cells); n ++) cellList[n] = -1;
  THREAD_PROC_LOOP (BuildNebrListT, 1);
```

```
  THREAD_PROC_LOOP (BuildNebrListT, 2);
}

void *BuildNebrListT (void *tr)
{
  ...
  int ip;

  QUERY_THREAD ();
  switch (QUERY_STAGE) {
    case 1:
      VDiv (invWid, cells, region);
      DO_MOL {
        VSAdd (rs, mol[n].r, 0.5, region);
        VMul (cc, rs, invWid);
        if (cc.z % nThread == ip) {
          c = VLinear (cc, cells) + nMol;
          ...
        }
      }
      break;
    case 2:
      rrNebr = Sqr (rCut + rNebrShell);
      nebrTabLenP[ip] = 0;
      THREAD_SPLIT_LOOP (m1z, cells.z) {
        for (m1y = 0 ...
          ...
            if (VLenSq (dr) < rrNebr) {
              if (nebrTabLenP[ip] * nThread >= nebrTabMax)
                 ErrExit (ERR_TOO_MANY_NEBRS);
              nebrTabP[ip][2 * nebrTabLenP[ip]] = j1;
              nebrTabP[ip][2 * nebrTabLenP[ip] + 1] = j2;
              ++ nebrTabLenP[ip];
            }
          ...
      }
      break;
  }
  return (NULL);
}
```

Here, during the first call to `BuildNebrListT` (the first stage), each thread is responsible for constructing the linked lists of cell occupants for only a fraction of the cells, based on their z coordinates. During the second call, each thread produces a separate portion of the neighbor list, stored in the array `nebrTabP[ip][]`, involving pairs of atoms for which the first member of the pair lies in a cell being handled by that thread. Thus, while the atoms themselves typically appear in more than one of the neighbor lists, the pairs themselves appear only once.

The corresponding function *ComputeForces* becomes

```
void ComputeForces ()
{
  int ip, iq;

  THREAD_PROC_LOOP (ComputeForcesT, 1);
  THREAD_PROC_LOOP (ComputeForcesT, 2);
  uSum = 0.;
  THREAD_LOOP uSum += uSumP[iq];
}

void *ComputeForcesT (void *tr)
{
  ...
  int ip, iq;

  QUERY_THREAD ();
  switch (QUERY_STAGE) {
    case 1:
      rrCut = Sqr (rCut);
      DO_MOL VZero (raP[ip][n]);
      uSumP[ip] = 0.;
      for (n = 0; n < nebrTabLenP[ip]; n ++) {
        j1 = nebrTabP[ip][2 * n];
        j2 = nebrTabP[ip][2 * n + 1];
        ...
        if (rr < rrCut) {
          ...
          VVSAdd (raP[ip][j1], fcVal, dr);
          VVSAdd (raP[ip][j2], - fcVal, dr);
          uSumP[ip] += uVal;
        }
      }
      break;
    case 2:
      THREAD_SPLIT_LOOP (n, nMol) {
        VZero (mol[n].ra);
        THREAD_LOOP VVAdd (mol[n].ra, raP[iq][n]);
      }
      break;
  }
  return (NULL);
}
```

In the first call to *ComputeForcesT*, the particular neighbor list associated with thread `ip` is processed, with acceleration values being stored in array `raP[ip][]` and potential energies in `uSumP[ip]`. The second call accumulates these separate

contributions, where, for brevity,

```
#define THREAD_LOOP                                          \
   for (iq = 0; iq < nThread; iq ++)
```

Additional quantities declared here are

```
pthread_t *pThread;
VecR **raP;
real *uSumP;
int **nebrTabP, *nebrTabLenP, funcStage, nThread;
```

where the elements *pThread* are required by the thread processing; the necessary array allocations (in *AllocArrays* – the usual *nebrTab* is not required, having been replaced by corresponding arrays *nebrTabP* private to each thread) and input data item are

```
  AllocMem (pThread, nThread, pthread_t);
  AllocMem (uSumP, nThread, real);
  AllocMem (nebrTabLenP, nThread, int);
  AllocMem2 (raP, nThread, nMol, VecR);
  AllocMem2 (nebrTabP, nThread, 2 * nebrTabMax / nThread, int);          5

  NameI (nThread),
```

Finally, the remaining definitions used in the program are

```
#define QUERY_THREAD()  ip = (int) tr
#define QUERY_STAGE     funcStage
#define THREAD_PROC_LOOP(tProc, fStage)                      \
   funcStage = fStage;                                       \
   for (ip = 1; ip < nThread; ip ++)                         \          5
      pthread_create (&pThread[ip], NULL, tProc,             \
      (void *) ip);                                          \
   tProc ((void *) 0);                                       \
   for (ip = 1; ip < nThread; ip ++)                         \
      pthread_join (pThread[ip], NULL);                                  10
```

Further information about the functions prefixed with *pthread_* can be found in the documentation of the thread library functions. It is left to the reader to explore the performance benefits that can be obtained by this approach (a prerequisite being a computer with multiple processors and shared memory).

17.6 Techniques for vector processing

Pipelined computation

When supercomputers first appeared, their performance derived from two principal features: a fast processor clock and the use of pipelined vector processing. With the appearance of comparatively cheap microprocessors, and their employment in parallel computers, the role of vector processing temporarily diminished. However, the reader may rest assured that it is returning once again to boost microprocessor performance. Some pipelining is already to be found in these chips (as well as multiple instruction units), but the need for application software to take its existence into account is less of an issue than when 'serious' vector processing is crucial to performance. There are still mainframe supercomputers that rely heavily on vector processing.

We will again concentrate on the soft-sphere simulation; this turns out to be the most difficult to vectorize effectively. For longer-range forces the number of interacting neighbors of each atom is large, and since a slightly modified form of the all-pairs version of the interaction function can then be used (provided there are not too many atoms) the compiler is able to handle the vectorization automatically [bro86]. Likewise for small systems with only short-range forces, where the loss of efficiency due to using the all-pairs method is adequately compensated by the vectorization speedup. But for bigger systems (beyond several hundred atoms) with short-range forces, for which cells are essential and neighbor lists advisable if the memory is available, compilers are unable to rearrange the conventional MD program so that it may be vectorized effectively. The problem must be solved the hard way – by rearranging the computation into a more suitable form [rap91a].

A specific computer architecture will not be invoked here. Instead, the computations will be expressed in a form that any processor with certain common hardware features – to be enumerated later – and an effective compiler will be able to execute relatively efficiently. The approach differs from previous sections, where we spelled out the communication and thread functions to be used; a similar approach could have been used here as well, but since the result of reformulating the algorithm is a program that can be vectorized automatically, there is little point in doing the work manually (by calling an assortment of vector functions directly).

Effective vector processing requires long sequences of data items (the vectors) that can be processed independently in pipeline fashion. Because of the overhead in filling these pipes initially, there is an advantage to using long vectors, although the minimum useful length depends on the specific machine. Vector computers emphasize floating-point performance, but in recognition of the fact that most problems are not organized in precisely the form needed, the machines are also able to carry out certain kinds of data rearrangement at high speed. More precisely, the

capability exists to gather randomly distributed data items speedily into a single array based on a vector of addresses, as well as the converse scatter operation; the MD code below relies heavily on such operations.

Layer method

The problem with the cell method, which makes it impossible to vectorize effectively, is its use of linked lists to identify the atoms in each cell. The method we describe here takes the same cell occupancy data and rearranges it into a form more suitable for vector processing. The approach can also be extended to produce neighbor lists [gre89b] organized in a way that can be automatically vectorized.

In the original, cell-based version of `ComputeForces` in §3.4 there are several nested loops; the outermost is over cells and within this there is a loop over the offsets to neighboring cells; two further inner loops consider all pairs of cell occupants. We will now proceed to rearrange these loops. If the cell occupants are assumed to be ordered in some (arbitrary) way, then the role of the two outermost loops in the revised version is to produce all valid pairings of the ith and jth occupants of whatever pair of cells happens to be under consideration by the inner loops (the case where both cells are the same is also covered). Within these two outer loops there is a loop over possible relative offsets between cell pairs. Finally, the innermost loop is over all cells, with the second cell of the pair obtained using the known cell offset. If the number of cells is close to the number of atoms, the innermost loop will fulfill the principal requirement of effective vectorization – a large repetition count. Of course this is not the whole story, because it is the details of the processing carried out by the innermost loop that determine performance and, in addition, the work needed to rearrange the data for this computation must be taken into account.

The reordered cell occupancy data are stored in 'layers' [rap91a], as shown in Figure 17.2. Each layer consists of one storage element per cell, and there are as many layers as the maximum cell occupancy. The first atom in a cell (the order has no special significance) is placed in the first layer, and so on; the first layer will be practically full, but later layers will be less densely populated. Since the two outer loops scan all pairs of layers and the number of cycles of the inner loops is independent of cell occupancy, it follows that computation time varies quadratically with the number of layers. Thus the method is most effective for relatively dense systems, where cell occupancy does not vary too widely.

The presence of periodic boundaries complicates the algorithm, so they are eliminated by a process of replication (§3.4), reminiscent of the copying operation used earlier in the distributed approach. Before beginning the interaction computation (`ComputeForces`), all atoms close to any boundary are duplicated so that they appear just beyond the opposite boundary. These replicas are used for all

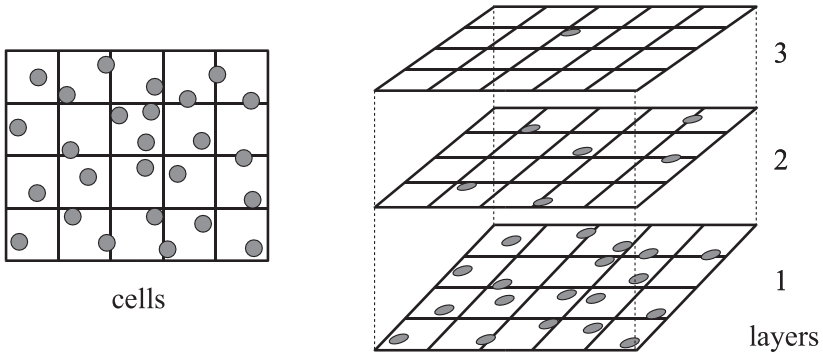

Fig. 17.2. The reorganization of the cell contents into layers for efficient vector processing.

interactions that would otherwise extend across one or more boundaries. The cell array must be enlarged to provide a shell of thickness r_c (the interaction cutoff range) surrounding the region.

The replication function♠ follows. Each spatial dimension is treated in turn and atoms near edges or corners may be replicated more than once. A check for array overflow is included†.

```
void ReplicateMols ()
{
  int k, n, na;

  nMolRep = nMol;
  for (k = 0; k < NDIM; k ++) {
    na = nMolRep;
    for (n = 0; n < nMolRep; n ++) {
      if (fabs (VComp (mol[n].r, k)) >= 0.5 * VComp (region, k) -
          rCut) {
        mol[na].r = mol[n].r;
        if (VComp (mol[na].r, k) > 0.)
          VComp (mol[na].r, k) -= VComp (region, k);
        else VComp (mol[na].r, k) += VComp (region, k);
        ++ na;
      }
    }
    nMolRep = na;
    if (nMolRep >= nMolMax) ErrExit (ERR_TOO_MANY_REPLICAS);
  }
}
```

♠ *pr_17_3*

† To vectorize the loops in this function it may be necessary to divide each loop over atoms into two parts: the first loop identifies the atoms to be replicated and stores their indices in a separate array; the second does the replication using this index array.

The interactions are computed in several stages. The first stage determines the cell occupied by each atom. Next the layers are constructed by a method to be discussed below. Then come the multiply-nested loops: the outer two loops select all possible layer pairs, within them is the loop over cell offsets and inside this are four successive loops where all the interactions are computed. The first of the innermost loops considers all cell pairings involving the chosen layers and the specified offset, determines whether a valid pair of atoms is to be found there and, if so, adds this information to a list. The second loop processes the listed atom pairs and computes their interactions. The final two loops add these newly computed terms to the accumulated interactions of the respective atoms.

The structure *Mol* used here has the form

```
typedef struct {
  VecR r, rv, ra;
  real u;
  int inCell;
} Mol;
```

where *u* is the potential energy of the individual atom and is required in order to achieve vectorization; the new variables appearing in the program are

```
Mol *mol;
VecR *raL;
real *uL;
int **layerMol, **molPtr, *inside, *molId, bdyOffset, nLayerMax,
   nMolMax, nMolRep;
```

The arrays are allocated by

```
void AllocArrays ()
{
  int k;

  AllocMem (mol, nMolMax, Mol);
  AllocMem (molId, nMolMax, int);
  AllocMem (raL, VProd (cells), VecR);
  AllocMem (uL, VProd (cells), real);
  AllocMem (inside, 2 * bdyOffset + VProd (cells), int);
  AllocMem2 (molPtr, 2, VProd (cells), int);
  AllocMem2 (layerMol, nLayerMax, 2 * bdyOffset + VProd (cells), int);
}
```

The variable *nMolMax* is the maximum number of atoms, including replicas, for which storage is available, *nLayerMax* is the number of available layers and the constant *bdyOffset* is used to avoid any problems with negative array indices

when shifted layers are paired (see later). The meaning of the other variables will become clear in due course. If more than a few layers are needed the array `layerMol` will dominate the storage requirements of the program.

The interaction calculation follows.

```
#define OFFSET_VALS                                                  \
   {{-1,-1,-1}, {0,-1,-1}, {1,-1,-1}, {-1,0,-1}, {0,0,-1},  \
    {1,0,-1}, {-1,1,-1}, {0,1,-1}, {1,1,-1}, {-1,-1,0},     \
    {0,-1,0}, {1,-1,0}, {-1,0,0}, {0,0,0}, {1,0,0},         \
    {-1,1,0}, {0,1,0}, {1,1,0}, {-1,-1,1}, {0,-1,1},        \
    {1,-1,1}, {-1,0,1}, {0,0,1}, {1,0,1}, {-1,1,1},         \
    {0,1,1}, {1,1,1}}

void ComputeForces ()
{
  VecR dr, invWid, regionEx, rs, t;
  VecI cc, vOff[] = OFFSET_VALS;
  real fcVal, rr, rrCut, rri, rri3;
  int ic, layer1, layer2, m1, m2, n, na, nat, nLayer, nPair,
     offset, offsetLo;

  VCopy (t, cells);
  VAddCon (t, t, -2.);
  VDiv (invWid, t, region);
  VSetAll (t, 2.);
  VDiv (t, t, invWid);
  VAdd (regionEx, region, t);
  for (n = 0; n < nMolRep; n ++) {
    VSAdd (rs, mol[n].r, 0.5, regionEx);
    VMul (cc, rs, invWid);
    mol[n].inCell = VLinear (cc, cells);
    molId[n] = n;
  }
  nLayer = 0;
  for (na = nMolRep; na > 0; na = nat) {
    if (nLayer >= nLayerMax) ErrExit (ERR_TOO_MANY_LAYERS);
    for (ic = 0; ic < VProd (cells); ic ++)
       layerMol[nLayer][bdyOffset + ic] = -1;
    for (n = 0; n < na; n ++)
       layerMol[nLayer][bdyOffset + mol[molId[n]].inCell] = molId[n];
    for (ic = 0; ic < VProd (cells); ic ++) {
      n = layerMol[nLayer][bdyOffset + ic];
      if (n >= 0) mol[n].inCell = -1;
    }
    nat = 0;
    for (n = 0; n < na; n ++) {
      if (mol[molId[n]].inCell >= 0) molId[nat ++] = molId[n];
    }
    ++ nLayer;
  }
```

```
  for (n = 0; n < nMolRep; n ++) {
    VZero (mol[n].ra);
    mol[n].u = 0.;
  }
  rrCut = Sqr (rCut);
  for (layer1 = 0; layer1 < nLayer; layer1 ++) {
    for (layer2 = layer1; layer2 < nLayer; layer2 ++) {
      offsetLo = (layer2 == layer1) ? 14 : 0;
      for (offset = offsetLo; offset < 27; offset ++) {
        nPair = 0;
        m2 = bdyOffset + VLinear (vOff[offset], cells);
        for (m1 = bdyOffset; m1 < bdyOffset + VProd (cells); m1 ++) {
          if ((inside[m1] || inside[m2]) &&
              layerMol[layer1][m1] >= 0 && layerMol[layer2][m2] >= 0) {
            molPtr[0][nPair] = layerMol[layer1][m1];
            molPtr[1][nPair] = layerMol[layer2][m2];
            ++ nPair;
          }
          ++ m2;
        }
        for (n = 0; n < nPair; n ++) {
          VSub (dr, mol[molPtr[0][n]].r, mol[molPtr[1][n]].r);
          rr = VLenSq (dr);
          if (rr < rrCut) {
            rri = 1. / rr;
            rri3 = Cube (rri);
            fcVal = 48. * rri3 * (rri3 - 0.5) * rri;
            VSCopy (raL[n], fcVal, dr);
            uL[n] = 4. * rri3 * (rri3 - 1.) + 1.;
          } else {
            VZero (raL[n]);
            uL[n] = 0.;
          }
        }
        for (n = 0; n < nPair; n ++) {
          VVAdd (mol[molPtr[0][n]].ra, raL[n]);
          mol[molPtr[0][n]].u += uL[n];
        }
        for (n = 0; n < nPair; n ++) {
          VVSub (mol[molPtr[1][n]].ra, raL[n]);
          mol[molPtr[1][n]].u += uL[n];
        }
      }
    }
  }
  uSum = 0.;
  DO_MOL uSum += mol[n].u;
  uSum *= 0.5;
}
```

Each of the innermost loops can be shown to satisfy the basic requirement of vectorization, namely, that each item processed is independent of all others†. The other characteristic of the loops is that, at least for the earliest layers, the vectors processed are of length proportional to N_m.

The one slightly subtle detail in this computation is the assignment of atoms to layers. The way this is done is to make several iterations over the set of atoms, one such cycle for each layer, each iteration assigning whichever atoms happen to be in each cell to the layer being filled. Thus several atoms may be assigned to the same position in a layer (in the array *layerMol*), but all atoms except the last one encountered in each cell will be overwritten. A check is made after each cycle to see which atoms were recorded in the layer; these are removed from the set before proceeding to the next layer, by zeroing the values of *inCell* for these atoms and compressing the identities of the remaining atoms held in *molId*. This process is repeated until no unassigned atoms remain. Clearly such a scheme would be wasteful in terms of computation, were it not for the fact that it can be vectorized.

Other comments about the above function are the following. Computation of the cell size ignores the outermost cells, because these are outside the simulation region. The array *inside* is used to distinguish quickly cells that are within the simulation region from those adjoining the boundary; this is preferable to a test based on the three indices needed to specify cell position. When a layer is paired with itself, only positive cell offsets need be considered.

Changes to *SetParams* are

```
VecI t;
...
VAddCon (cells, cells, 2);
t = initUcell;
VAddCon (t, t, 2);
nMolMax = 4 * VProd (t);
bdyOffset = cells.x * (cells.y + 1) + 1;
```

and *nLayerMax* must be added to the input data. Initialization also requires

```
void SetupLayers ()
{
  int n, nLayer, nx, ny, nz;

  for (n = 0; n < 2 * bdyOffset + VProd (cells); n ++) inside[n] = 0;
  for (nz = 1; nz < cells.z - 1; nz ++) {
    for (ny = 1; ny < cells.y - 1; ny ++) {
```

† Supplementary, system-specific compiler directives may have to be used to inform the compiler of this fact, since it is not at all obvious from just reading the program without understanding the underlying organization.

```
        for (nx = 1; nx < cells.x - 1; nx ++) {
          inside[bdyOffset + (nz * cells.y + ny) * cells.x + nx] = 1;
        }
      }
    }
    for (nLayer = 0; nLayer < nLayerMax; nLayer ++) {
      for (n = 0; n < 2 * bdyOffset + VProd (cells); n ++)
        layerMol[nLayer][n] = -1;
    }
  }
```

The rest of the program is unchanged.

17.7 Further study

17.1 Implement the distributed MD computation on your favorite multiprocessor computer and measure the communication overheads. How does performance vary with system size and number of processors?

17.2 Examine the performance of the threaded approach on a shared-memory computer.

17.3 Extend the vectorized layer method to include neighbor lists.