Appendix F

Saving CPU Time

The energy or force calculation is the most time-consuming part of almost all Molecular Dynamics and Monte Carlo simulations. If we consider a model system with pairwise additive interactions (as is done in many molecular simulations), we have to consider the contribution to the force on particle i, by all its neighbors. If we do not truncate the interactions, this implies that, for a system of N particles, we must evaluate $N(N-1)/2$ pair interactions. And even if we do truncate the potential, we still would have to compute all $N(N-1)/2$ pair distances to describe which pairs can interact. This implies that, if we use no tricks, the time needed for the evaluation of the energy scales as N^2. There exist efficient techniques for speeding up the evaluation of both short-range and long-range interactions in such a way that the computing time scales as $N^{3/2}$, rather than N^2. The techniques for the long-range interactions were discussed in Chapter 12.1; here, we discuss some of the techniques used for the short-range interactions. These techniques are:

1. Verlet list
2. Cell (or linked) list
3. Combination of Verlet and cell lists

F.1 Verlet List

If we simulate a large system and use a cutoff that is smaller than the simulation box, many particles do not contribute to the energy of a particle i. It is advantageous therefore to exclude the particles that do not interact from the expensive energy calculation. Verlet [13] developed a bookkeeping technique, commonly referred to as the Verlet list or neighbor list, which is illustrated in Figure F.1. In this method a second cutoff radius $r_v > r_c$ is in-

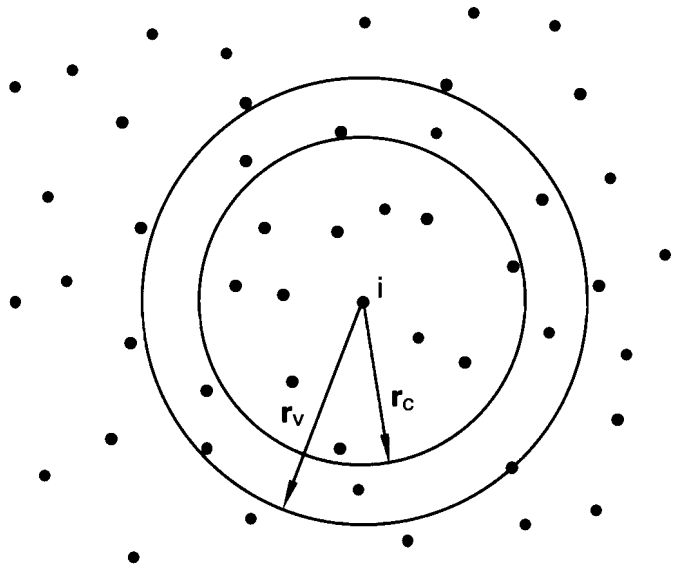

Figure F.1: The Verlet list: a particle i interacts with those particles within the cutoff radius r_c; the Verlet list contains all the particles within a sphere with radius $r_v > r_c$.

troduced, and before we calculate the interactions, a list is made (the Verlet list) of all particles within a radius r_v of particle i. In the subsequent calculation of the interactions, only those particles in this list have to be considered. Until now we have not saved any CPU time. We gain such time when we next calculate the interactions; if the maximum displacement of the particles is less than $r_v - r_c$, then we have to consider only the particles in the Verlet list of particle i. This is a calculation of order N. As soon as one of the particles is displaced more than $r_v - r_c$, we have to update the Verlet list. The latter operation is of order N^2, and although this step is not performed each time an interaction is calculated, it will dominate for a very large number of particles.

The Verlet list can be used for both Molecular Dynamics and Monte Carlo simulations. However, there are some small differences in the implementation. For example, in a Molecular Dynamics simulation, the force on all particles is calculated at the same time. It is sufficient therefore to have a Verlet list with half the number of particles for each particle as long as the interaction i-j is accounted for in either the list of particle i or that of j. In a Monte Carlo simulation each particle is considered separately, therefore it is convenient to have for each particle the complete list. Algorithm 33 shows the use of the Verlet list in a Monte Carlo simulation.

Bekker *et al.* have developed an elegant extension of the Verlet list for systems with periodic boundary conditions [530]. To calculate the force or potential energy of particle i one has to locate the nearest image of the particles in the Verlet list of particle j (see, Algorithm 34). Bekker *et al.* have

Algorithm 33 (Use of Verlet List in a Monte Carlo Move)

```
  SUBROUTINE mcmove_verlet          attempts to displace a particle
                                    using a Verlet list
  o=int(ranf()*npart)+1             select a particle at random
  if (abs(x(o)-xv(o)).gt.(rv-rc)    check to make a new list
 +  /2) call new_vlist
  call en_vlist(o,x(o),eno)         energy old configuration
  xn=x(o)+(ranf()-0.5)*delx         random displacement
  if (abs(xn-xv(o)).gt.(rv-rc)/2)   check to make a new list
 +  call new_vlist
  call en_vlist(o,xn,enn)           energy new configuration
  arg=exp(-beta*(enn-eno))
  if (ranf().lt.arg)
 +  x(o)=xn                         accepted: replace x(o) by xn
  return
  end
```

Comments to this algorithm:

1. *The algorithm is based on Algorithm 2.*
2. *Subroutine* `newvlist` *makes the Verlet list (see Algorithm 34) and subroutine* `en_vlist` *calculates the energy of a particle at the given position using the Verlet list (see Algorithm 35).*

shown that this nearest image calculation in the inner loop of a MD or MC simulation can be avoided.

In a periodic system, the total force on particle i can be written as

$$\mathbf{F}_i = \sum_{j=1}^{N} {\sum_{k=-13}^{13}}' \mathbf{F}_{i(j.k)},$$

where the prime denotes that the summation is performed over the nearest image of particle j in the central box ($k = 0$) or in one of its 26 periodic images. Here, (j.k) denotes the periodic image of particle j in box k. Box k is defined by the integer numbers n_x, n_y, n_z:

$$k = 9n_x + 3n_y + n_z$$

and

$$\mathbf{t}_k = n_x\mathbf{L}_x + n_y\mathbf{L}_y + n_z\mathbf{L}_z,$$

Algorithm 34 (Making a Verlet List)

```
SUBROUTINE new_vlist                    makes a new Verlet list
do i=1,npart                            initialize list
  nlist(i)=0
  xv(i)=x(i)                            store position of particles
enddo
do i=1,npart-1
  do j=i+1,npart
    xr=x(i)-x(j)
    if (xr.gt.hbox) then                nearest image
      xr=xr-box
    else if (xr.lt.-hbox) then
      xr=xr+box
    endif
    if (abs(xr).lt.rv) then             add to the lists
      nlist(i)=nlist(i)+1
      nlist(j)=nlist(j)+1
      list(i,nlist(i))=j
      list(j,nlist(j))=i
    endif
  enddo
enddo
return
end
```

Comments to this algorithm:

1. *Array* `list(i,itel)` *is the Verlet list of particle* `i`*, the total number of particles in the Verlet list of particle* `i` *is given by* `nlist(i)`*, and the array* `xv(i)` *contains the position of the particles at the moment the list is made (is used to see when a new list has to be made).*
2. *Note that in this algorithm we assume* all *particles are in the simulation box; hence* `x(i)` $\in$ *[0,*box*].*

where $\mathbf{t}_k$ is the translation vector of the central box to its periodic image k. A particle in the central box is denoted by (i.0) = i. Using this notation, we can write, for the interaction between particles i and j,

$$\mathbf{F}_{i(j.k)} = \mathbf{F}_{(i.-k)j} = -\mathbf{F}_{(j.k)i} = -\mathbf{F}_{j(i.-k)}.$$

Algorithm 35 (Calculating the Energy Using a Verlet List)

```
SUBROUTINE en_vlist(i,xi,en)      calculates energy using
                                  the Verlet list
en=0
do jj=1,nlist(i)                  loop over the particles in the list
  j=list(i,jj)                    next particle in the list
  en=en+enij(i,xi,j,x(j))
enddo
return
end
```

Comment to this algorithm:

1. *Array* `list(i,itel)` *and* `nlist` *are made in Algorithm 34 and* `enij` *gives the energy between particles* `i` *and* `j` *at the given positions.*

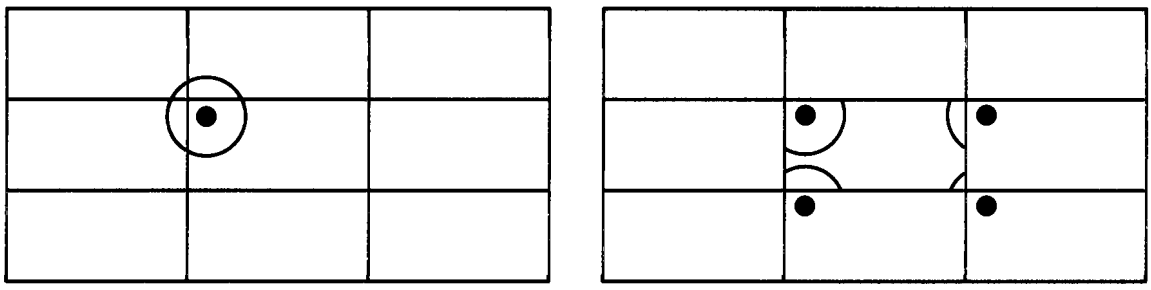

Figure F.2: Verlet lists: (left) conventional approach in which each particle has a Verlet list; (right) the approach of Bekker *et al.* in which each periodic image of a particle has its own Verlet list that contains only those particles in the central box.

We can write

$$\mathbf{F}_i = \sum_{j=1}^{N} \sum_{k=-13}^{13} {}'\mathbf{F}_{(i.k)j}.$$

The importance of this seemingly trivial result is that the summation is over all particles j in the *central* box with the nearest image of particle i. The difference between the two approaches is shown in Figure F.2.

This method is implemented using different Verlet lists for each periodic image of particle i. These lists contain only those particles that interact with particle i *and* are in the central box. If these lists are used, it is not necessary to

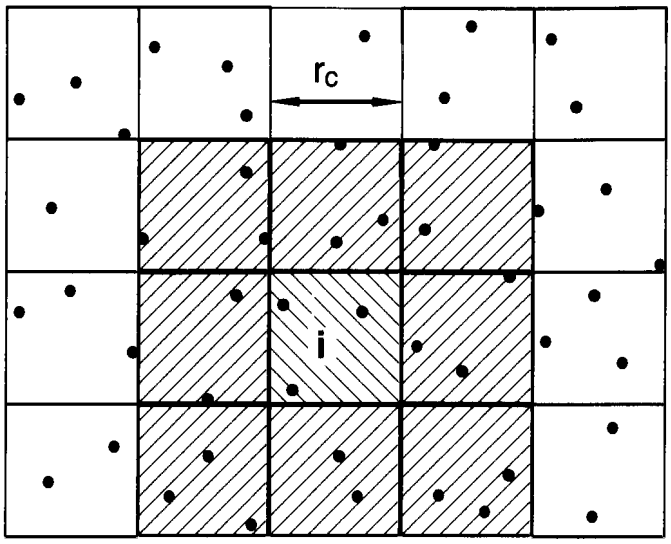

Figure F.3: The cell list: the simulation cell is divided into cells of size $r_c \times r_c$; a particle i interacts with those particles in the same cell or neighboring cells (in 2D there are 9 cells; and in 3D, 27 cells).

use the nearest image operation during the calculation of the force or energy. In [530] the use of these lists is shown to speed up an MD simulation by a factor 1.5. In addition, Bekker *et al.* have shown that a similar trick can be used to take the calculation of the virial (pressure) out of the inner loop.

F.2 Cell Lists

An algorithm that scales with N is the cell list or linked-list method [24]. The idea of the cell list is illustrated in Figure F.3. The simulation box is divided into cells with a size equal to or slightly larger than the cutoff radius r_c; each particle in a given cell interacts with only those particles in the same or neighboring cells. Since the allocation of a particle to a cell is an operation that scales with N and the total number of cells that needs to be considered for the calculation of the interaction is independent of the system size, the cell list method scales as N. Algorithm 36 shows how a cell list can be used in a Monte Carlo simulation.

F.3 Combining the Verlet and Cell Lists

It is instructive to compare the efficiency of the Verlet list and cell list in more detail. In the Verlet list the number of particles for which the distance needs to be calculated is in three dimensions, given by

$$n_v = \frac{4}{3}\pi\rho r_v^3;$$

Algorithm 36 (Use of Cell List in a Monte Carlo Move)

```
   SUBROUTINE mcmove_neigh        attempts to displace a particle
                                  using a cell list
   call newnlist(rc)              make the cell list
   o=int(ranf()*npart)+1          select a particle at random
   call en_nlist(o,x(o),eno)      calculate energy old configuration
   xn=x(o)+(ranf()-0.5)*delx      give particle random displacement
   call en_nlist(o,xn,enn)        calculate energy new configuration
   arg=exp(-beta*(enn-eno))
   if (ranf().lt.arg)
  +  x(o)=xn                      accepted: replace x(o) by xn
   return
   end
```

Comments to this algorithm:

1. *This algorithm is based on Algorithm 2.*
2. *Subroutine* `new_nlist` *makes the cell list (see Algorithm 37) and subroutine* `en_nlist` *calculates the energy of a particle at the given position using the cell list (see Algorithm 38). Note that it is possible, at the expense of some extra bookkeeping, to update the list once a move is accepted instead of making a new list every move.*

for the cell list the corresponding number is

$$n_l = 27\rho r_c^3.$$

If we use typical values for the parameters in these equations (Lennard-Jones potential with $r_c = 2.5\sigma$ and $r_v = 2.7\sigma$), we find that n_l is five times larger than n_v. As a consequence, in the Verlet scheme, the number of pair distances that needs to be calculated is 16 times less than in the cell list.

The observation that the Verlet scheme is more efficient in evaluating the interactions motivated Auerbach *et al.* [531] to use a combination of the two lists: use a cell list to construct a Verlet list. The use of the cell list removes the main disadvantage of the Verlet list for a large number of particles—scales as N^2—but keeps the advantage of an efficient energy calculation. An implementation of this method in a Monte Carlo simulation is shown in Algorithm 39.

Algorithm 37 (Making a Cell List)

```
SUBROUTINE new_nlist(rc)      makes a new cell list with cell size r_c
                              using a linked-list algorithm
rn=box/int(box/rc)            determine size of cells r_n ≥ r_c
do icel=0,ncel-1
  hoc(icel)=0                 set head of chain to 0 for each cell
enddo
do i=1,npart                  loop over the particles
  icel=int(x(i)/rn)           determine cell number
  ll(i)=hoc(icel)             link list the head of chain of cell icel
  hoc(icel)=i                 make particle i the head of chain
enddo
return
end
```

Comment to this algorithm:

1. *This algorithm uses the linked-list method. To each cell a particle* `i` *is named head of chain and stored in the array* `hoc(icel)`. *To this particle the next particle in the cell (chain) is linked via the linked-list array* `ll(i)`. *If the value of the* `ll(i)` *is 0 no more particles are in the cell (chain). The desired (optimum) cell size is* `rc`, *and* `rn` *is the closest size that fits in the box.*

F.4 Efficiency

The first question that arises is when to use which method. This depends very strongly on the details of the systems. In any event, we always start with a scheme as simple as possible, hence no tricks at all. Although the algorithm scales as N^2, it is straightforward to implement and therefore the probability of programming errors is relatively small. In addition we should take into account how often the program will be used.

The use of the Verlet list becomes advantageous if the number of particles in the list is significantly less than the total number of particles; in three dimensions this means

$$n_v = \frac{4}{3}\pi r_v^3 \rho \ll N.$$

If we substitute some typical values for a Lennard-Jones potential ($r_v = 2.7\sigma$ and $\rho = 0.8\sigma^{-3}$), we find $n_v \approx 66$, which means that only if the number of particles in the box is more than 100 does it make sense to use a Verlet list.

To see when to use one of the other techniques, we have to analyze the algorithms in somewhat more detail. If we use no tricks, the amount of CPU

Algorithm 38 (Calculating the Energy Using a Cell List)

```
      SUBROUTINE ennlist(i,xi,en)        calculates energy using
                                         the cell list
      en=0
      icel=int(xi/rn)                    determine the cell number
      do ncel=1,neigh                    loop over the neighbor cells
        jcel=neigh(icel,ncel)            number of the neighbor
        j=hoc(jcel)                      head of chain of cell jcel
        do while (j.ne.0)
          if (i.ne.j)
     +     en=en+enij(i,xi,j,x(j))
          j=ll(j)                        next particle in the list
        enddo
      enddo
      return
      end
```

Comment to this algorithm:

1. *Array* `ll(i)` *and* `hoc(icel)` *are constructed in Algorithm 37;* `enij` *is a function that gives the energy between particles* `i` *and* `j` *at the given positions.* `neigh(icel,ncel)` *gives the location of the* `ncel`*th neighbor of cell* `icel`.

time to calculate the total energy is given by

$$\tau = cN(N-1)/2.$$

The constant gives the required CPU time for an energy calculation between a pair of particles. If we use the Verlet list, the CPU time is

$$\tau_v = cn_vN + \frac{c_v}{n_u}N^2,$$

where the first term arises from the calculation of the interactions and the second term from the update of the Verlet list, which is done every n_u^{th} cycle.

The cell list scales with N and the CPU time can be split into two contributions: one that accounts for the calculation of the energy and the other for the making of the list,

$$\tau_l = cn_lN + c_lN.$$

If we use a combination of the two lists, the total CPU time becomes

$$\tau_c = cn_vN + \frac{c_l}{n_u}N.$$

Algorithm 39 (Combination of Verlet and Cell Lists)

```
 SUBROUTINE mcmove_clist          displace a particle
                                  using a combined list
 o=int(ranf()*npart)+1            select a particle at random
 if (abs(x(o)-xv(o)).gt.rv-rc)    check to make a new list
+  call new_clist
 call en_vlist(o,x(o),eno)        energy old configuration
 xn=x(o)+(ranf()-0.5)*delx        random displacement
 if (abs(xn-xv(o)).gt.rv-rc)      check to make a new list
+  call new_clist
 call en_vlist(o,xn,enn)          energy new configuration
 arg=exp(-beta*(enn-eno))
 if (ranf().lt.arg)
+  x(o)=xn                        accepted: replace x(o) by xn
 return
 end
```

Comments to this algorithm:

1. *The algorithm is based on Algorithm 33.*
2. *Subroutine* `newclist` *makes the Verlet list using a cell list (see Algorithm 40) and subroutine* `en_vlist` *calculates the energy of a particle at the given position using the Verlet list (see Algorithm 35).*

The way to proceed is to perform some test simulations to estimate the various constants, and from the equations, it will become clear which technique is preferred. In Case Study 26, we have made such an estimate for a simulation of the Lennard-Jones fluid.

Case Study 26 (Comparison of Schemes for the Lennard-Jones Fluid)
It is instructive to make a detailed comparison of the various schemes to save CPU time for the Lennard-Jones fluid. We compare the following schemes:

1. Verlet list
2. Cell list
3. Combination of Verlet and cell lists
4. Simple N^2 algorithm

We have used the program of Case Study 1 as a starting point. At this point it is important to note that we have not tried to optimize the parameters (such

Algorithm 40 (Making a Verlet List Using a Cell List)

```
SUBROUTINE new_clist                    makes a new Verlet list
                                        using a cell list
call new_nlist(rv)                      make the cell lists
do i=1,npart                            initialize list
  nlist(i)=0
  xv(i)=x(i)                            store position of particles
enddo
do i=1,npart
 icel=int(x(i)/rn)                      determine cell number
 do ncel=1,neigh                        loop over the neighbor cells
   jcel=neigh(icel,ncel)                number of the neighbor
   j=hoc(jcel)                          head of chain of cell jcel
   do while (j.ne.0)
     if (i.ne.j) then
      xr=x(i)-x(j)
      if (xr.gt.hbox) then              nearest image
       xr=xr-box
      else if(xr.lt.-hbox)then
       xr=xr+box
      endif
      if (abs(xr).lt.rv) then           add to the Verlet lists
       nlist(i)=nlist(i)+1
       nlist(j)=nlist(j)+1
       list(i,nlist(i))=j
       list(j,nlist(j))=i
      endif
     endif
    j=ll(j)                             next particle in the cell list
   enddo
 enddo
enddo
return
end
```

Comments to this algorithm:

1. *Array* `list(i,itel)` *is the Verlet list of particle* `i`*, the number of particles in the Verlet list of particle* `i` *is given by* `nlist(i)`*, and the array* `xv(i)` *contains the position of the particles at the moment the list is made (is used to see when a new list has to be made). We assume that* all *particles are in the simulation box; hence* `x(i)` $\in$ *[0,*`box`*].*
2. *Subroutine* `new_nlist(rv,rn)` *makes a cell list (Algorithm 37). The desired cell size is* `rv` *and the actual cell size is* `rn`.

as the Verlet radius) for the various methods; we have simply taken some reasonable values.

For the Verlet list (and for the combination of Verlet and cell lists) it is important that the maximum displacement be smaller than twice the difference between the Verlet radius and cutoff radius. For the cutoff radius we have used $r_c = 2.5\sigma$, and for the Verlet radius $r_v = 3.0\sigma$. This limits the maximum displacement to $\Delta_x = 0.25\sigma$ and implies for the Lennard-Jones fluid that, if we want to use a optimum acceptance of 50%, we can use the Verlet method only for densities larger than $\rho > 0.6\sigma^{-3}$. For smaller densities, the optimum displacement is larger than 0.25. Note that this density dependence does not exist in a Molecular Dynamics simulation. In a Molecular Dynamics simulation, the maximum displacement is determined by the integration scheme and therefore is independent of density. This makes the Verlet method much more appropriate for a Molecular Dynamics simulation than for a Monte Carlo simulation. Only at high densities does it make sense to use the Verlet list.

The cell list method is advantageous only if the number of cells is larger than 3 in at least one direction. For the Lennard-Jones fluid this means that, if the number of particles is 400, the density should be lower than $\rho < 0.5\sigma^{-3}$. An important advantage of the cell list over the Verlet list is that this list can also be used for moves in which a particle is given a random position.

From these arguments it is clear that, if the number of particles is smaller than 200–500, the simple N^2 algorithm is the best choice. If the number of particles is significantly larger and the density is low, the cell list method is probably more efficient. At high density, all methods can be efficient and we have to make a detailed comparison.

To test these conclusions about the N dependence of the CPU time of the various methods, we have performed several simulations with a fixed number of Monte Carlo cycles. For the simple N^2 algorithm the CPU time per attempt is

$$\tau_{N^2} = cN,$$

where c is the CPU time required to calculate one interaction. This implies that the total amount of CPU time is independent of the density. For a calculation of the total energy, we have to do this calculation N times, which gives the scaling of N^2. Figure F.4 shows that indeed for the Lennard-Jones fluid, the τ_{N^2} increases linearly with the number of particles.

If we use the cell list, the CPU time will be

$$\tau_n = cV_l\rho + c_l p_l N,$$

where V_l is the total volume of the cells that contribute to the interaction (in three dimensions, $V_l = 27r_c^3$), c_n is the amount of CPU time required to

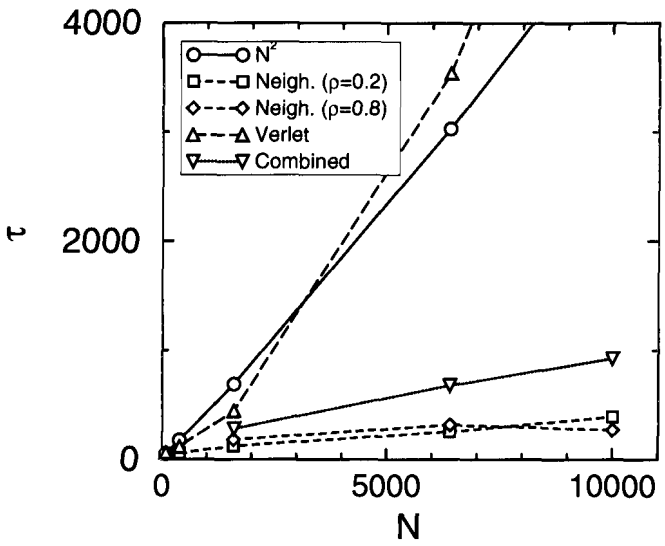

Figure F.4: Comparison of various schemes to calculate the energy: τ is in arbitrary units and N is the number of particles. As a test case the Lennard-Jones fluid is used. The temperature was $T^* = 2$ and per cycle the number of attempts to displace a particle was set to 100 for all systems. The lines serve to guide the eye.

make a cell list, and p_n is the probability that a new list has to be made. Figure F.4 shows that the use of a cell list reduces the CPU time for 10,000 particles with a factor 18. Interestingly, the CPU time does not increase with increasing density. We would expect an increase since the number of particles that contribute to the interaction of a particle i increases with density. However, the second contribution to τ_{Neigh} (p_n) is the probability that a new list has to be made, depends on the maximum displacement, which decreases when the density increases. Therefore, this last term will contribute less at higher densities.

For the Verlet scheme the CPU time is

$$\tau_v = cV_v\rho + c_v p_v N^2,$$

where V_v is the volume of the Verlet sphere (in three dimensions, $V_v = 4\pi r_v^3/3$), c_v is the amount of CPU time required to make the Verlet-list, and p_v is the probability that a new list has to be made. Figure F.4 shows that this scheme is not very efficient. The N^2 operation dominates the calculation. Note that we use a program in which a new list for all particles has to be made as soon as one of the particles has moved more than $(r_v - r_c)/2$; with some more bookkeeping it is possible to make a much more efficient program, in which a new list is made for only the particle that has moved out of the list.

The combination of the cell and Verlet lists removes the N^2 dependence of the simple Verlet algorithm. The CPU time is given by

$$\tau_c = cV_v\rho + c_v p_v c_n N.$$

Figure F.4 shows that indeed the N^2 dependence is removed, but the resulting scheme is not more efficient than the cell list alone.

This case study demonstrates that it is not simple to give a general recipe for which method to use. Depending on the conditions and number of particles, different algorithms are optimal. It is important to note that for a Molecular Dynamics simulation the conclusions may be different.