CHAPTER

MOLECULAR DYNAMICS EXAMPLE

13

13.1 INTRODUCTION

A few applications dealing with specific computational problems are discussed in this and the following two chapters. Our purpose is to examine in detail how they are structured and organized, understanding when, why, and how threads or tasks must be synchronized. Two major classical parallel patterns are addressed: *data parallelism*, where the same computational workload is concurrently executed by different threads on different sectors of the data set, and *control parallelism*, where worker threads overlap in time the execution of different successive tasks on the complete data set. Data parallelism is the subject of this and the following chapter. Control parallelism is discussed in Chapter 15.

This chapter focuses on a rather simple molecular dynamics application, involving the computation of the trajectories of a large number of interacting particles, which illustrates a number of basic multithreading issues. This example belongs to a class of problems where the amount of potential parallelism grows with the size of the data set (in this case, the number of interacting particles). Parallel work-sharing strategies are implemented by having different threads compute the trajectories of different groups of particles.

The example that follows is simple and compact, but its parallel implementation requires a number of synchronizations among the worker threads. It therefore provides a very adequate context for assessing the performance of some of the synchronization tools discussed in Chapter 9. This example will also be used to examine vectorization at the single-core level, capable of enhancing the initial performance boost induced by multithreading.

The computational problem to be solved is described first. Even if it looks simple at first sight, it is not, however, an artificial problem. In fact, this example is the most important component of the algorithm used by the `GaussGen` utility, introduced in the last example of Chapter 12 to generate vectors of correlated Gaussian random numbers.

13.2 MOLECULAR DYNAMICS PROBLEM

Consider a simple mechanical system made out of N point particles of equal mass (set equal to 1), *moving in one-dimension along the real axis* and subject to the action of forces that will soon be defined. Let

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00013-7

$$\vec{q} = (q_1, q_2, \ldots, q_N)$$

$$\vec{p} = (p_1, p_2, \ldots, p_N)$$

be the set of coordinates and the momenta—the velocities—of these N particles. We assume that the force acting on particle i is

$$F_i = -\sum_{i,j} D_{ij} q_j \tag{13.1}$$

where D is a real, symmetric, positive-definite $N \times N$ matrix. Positive-definite means the matrix has positive eigenvalues. The total energy of this mechanical system, equal to the sum of the kinetic and the potential energies, is given by

$$E = T + V = \frac{1}{2}\sum_{i} p_i^2 + \frac{1}{2}\sum_{i,j} q_i D_{ij} q_j \tag{13.2}$$

Readers with a physics background may have recognized that we are dealing with a collection of N coupled harmonic oscillators. The D matrix, being real and symmetric, can be brought to diagonal form by an orthogonal matrix O (a matrix whose inverse is just its transposed matrix):

$$O\,D\,O^T = Dd$$

where Dd is a diagonal matrix of the form

$$Dd_{ij} = \omega_i^2\,\delta_{ij}$$

Since the eigenvalues of D are positive, they can be written as the square of a real quantity ω_i. Now, *knowing the matrix O*, it is possible to define a new set of coordinates and momenta:

$$P_i = \sum_{i,j} O_{ij}\,p_j$$

$$Q_i = \sum_{i,j} O_{ij}\,q_j$$

in terms of which the initial problem reduces to a set of *uncoupled, independent harmonic oscillators*. Each coordinate Q_i performs a harmonic oscillatory motion around the origin with frequency ω_i:

$$Q_i(t) = A_i\,\cos(\omega_i\,t + \phi_i)$$

The amplitudes A_i and phases ϕ_i are determined by the initial conditions for the coordinates and momenta. Therefore, the problem of the computation of the particle trajectories is solved if the D matrix is diagonalized, namely, if the matrix O and the eigenvalues ω_i are known.

Note that we are here in the presence of attractive restoring forces (they push the particles toward the origin of coordinates). Their strength grows as particles move away from the origin. The particle trajectories $q_i(t)$ will therefore remain bounded in a finite domain near the origin of coordinates, because they are linear combinations of the $Q_i(t)$ given by the equation above.

The diagonalization of the D matrix can be performed with well-known linear algebra algorithms, whose computational cost grows like N^3 with the matrix size. This is not by itself a major problem because, having paid this computational cost once, it is possible to compute as many trajectories as one wants or needs. The real problem is, however, that this mechanical system will be used in contexts in which the D matrix evolves with time, which forces repeated updates of the computation of its diagonal form. In this case, it is more efficient to perform directly a numerical integration of the initial system of coupled harmonic oscillators, using an algorithm whose computational cost grows like N^2. This is what will be done in the rest of the chapter.

13.2.1 RELEVANCE OF THE MECHANICAL MODEL

The simple mechanical model described in the previous section is the main component of an algorithm for generating vectors whose components are *correlated* Gaussian random numbers used in Chapter 12. Consider a vector of rank N, whose components are Gaussian random numbers produced by one of the standard generators used in previous chapters (e.g., Chapter 4).

$$\vec{\xi} = (\xi_1, \xi_2, \ldots, \xi_N)$$

The vector components ξ_n are real (positive and negative) numbers, distributed following a Gaussian law. Loosely speaking, this means they represent independent events that know absolutely nothing of one another. The mathematical formulation of this idea is that *the mean value of the product of two different components is zero*. This is indeed very reasonable: vector components are positive or negative with equal probability, and in fact their mean value is zero. The product of two *different* components is not positive-definite. If they are indeed different and totally unrelated, the mean value of this product must also be zero. This can be checked by taking a large number of samples of the product of two different Gaussian random numbers. The fact that the components of our initial vector are uncorrelated can be expressed in a compact way as follows:

$$< \xi_i \, \xi_j >= \delta_{ij}$$

where δ_{ij} is just the identity matrix: $\delta_{ii} = 1$, and $\delta_{ij} = 0$ if $i \neq j$.

The problem we need to consider is the generation of vectors with *correlated* components. Individual vector components must be real numbers distributed around the origin. They are correlated in the sense that the mean value of the product of two components is now

$$< \xi_i \, \xi_j >= D_{ij} \tag{13.3}$$

where D is a predefined *positive-definite* matrix whose diagonal elements are still $D_{ii} = 1$, but the off diagonal elements are not zero $D_{ij} \neq 0$ if $i \neq j$. This statement, plus the requirement that an ordinary Gaussian generator is recovered when the correlations are switched off—namely, when $D_{ij} = \delta_{ij}$—define our problem. It turns out that a substantial part of a Monte-Carlo algorithm designed to generate the correlated Gaussian vectors can be reinterpreted in terms of the molecular dynamics problem formulated above. The `D` matrix that occurs in the potential energy of the mechanical system is the same `D` matrix that defines the required correlations in the equation above.

Correlated Gaussian fluctuations are needed, for example, in phenomenological models of protein folding. Macromolecules are represented by worm-like chains of beads connected by straight elastic

segments with torsion. Beads represent molecular aggregates where some specific activity is located, and straight segments represent chains of DNA base pairs. These phenomenological models of supercoiled DNA need to take into account the interaction of the macro-molecule with the environment in which it evolves. Indeed, the macro-molecule is subject to collisions with the atoms or molecules present in its environment. This manifests itself in the form of a Brownian motion of the macro-molecule, namely, erratic movements that are superimposed to the displacements computed from the deterministic mechanical model itself. These Brownian motions of the worm-like chain are therefore simulated by additional random displacements in the positions of the beads. For an isolated bead—that is, a heavy molecule in an environment—the Brownian motion is well described by uncorrelated Gaussian fluctuations in the three directions of space, with a variance proportional to the temperature of the environment. This is called *white noise*. However, in our case, these displacements cannot be just a white noise: *because of the mechanical constraints operating on the beads, they are all correlated in a way dictated by the specific protein configuration*. A symmetric D matrix determining the way in which the Brownian displacements of the vertices are correlated is provided by the underlying mechanical model of the macro-molecule. This matrix is a function of the bead positions. Obviously, this correlation matrix changes as the macro-molecule configuration evolves.

13.3 INTEGRATION OF THE EQUATIONS OF MOTION

The numerical integration of Newton's equations of motion is performed in our case by using the *leapfrog method*. Time is discretized by introducing a finite time step δ.

Let $(q_i^{(n)}, p_i^{(n)})$ denote the coordinates at time $n\delta$ and the momenta at time $(n-1/2)\delta$. Indeed, *in the leapfrog scheme, coordinates and momenta are not computed at equal times*: they are defined in two different inter-penetrating time grids. The integration procedure over one time step proceeds as follows (see Figure 13.1):

Step 1: When the coordinates $q_i^{(n)}$ are known, the accelerations $\overrightarrow{a}^{(n)}$ at the same time can be computed. Since the masses are 1, the acceleration is simply equal to the force as given by Equation (13.1):

$$a_i = -\sum_{j \neq i} D_{ij}\, q_j \tag{13.4}$$

Step 2: Once the particle accelerations at time $n\delta$ are known, the velocities at time $(n+1/2)\delta$ can be computed:

$$p_i^{(n+1)} = p_i^{(n)} + \delta\, a_i^{(n)}$$

Step 3: Once the velocities at time $(n+1/2)\delta$ are known, the new positions at time $(n+1)\delta$ can be computed:

$$q_i^{(n+1)} = q_i^{(n)} + \delta\, p_i^{(n+1)}$$

This completes the calculation of a leapfrog time step. Steps 2 and 3 involve simple vector operations, and the computational cost is therefore proportional to the vector size N. But the

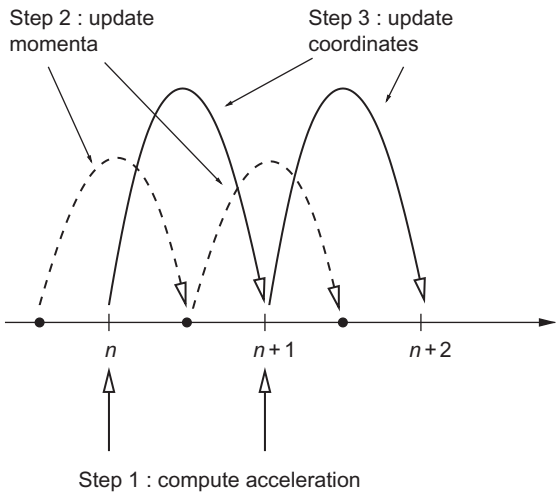

FIGURE 13.1

Illustration of the leapfrog integration scheme.

computation of the acceleration for each particle requires a sum over all the others particles in the system. Therefore, the total computational cost of a leapfrog time step is N^2.

In this leapfrog scheme, the finite time step δ introduces systematic errors of order δ^3. We know the total energy is conserved in the system evolution. Energy conservation violations are therefore of the order δ^3. Monitoring the total energy values is a way of controlling *a posteriori* the precision of the trajectory computation.

Besides following the time evolution of our system, the total energy, given by Equation (13.2), needs to be computed from time to time. This equation involves coordinates and momenta *at the same time* t, and in the leapfrog method, if the coordinates are known at time t, the momenta are known at time $t - 0.5\delta$. Therefore, the values of the momenta at time t need to be computed as follows:

- Compute again the acceleration a_i at time t, using Equation (13.4).
- The momenta at time t are then given by $p_i + 0.5\delta\, a_i$.

The total energy is then:

$$E = \frac{1}{2}\sum_i (p_i + 0.5\delta\, a_i)^2 - \frac{1}{2}\sum_i a_i\, q_i \tag{13.5}$$

13.3.1 PARALLEL NATURE OF THE ALGORITHM

The overall strategy for performing a parallel computation of the time evolution of our mechanical system is rather evident. Each leapfrog time step requires the following three operations, which are indeed vector operations on the particle indices:

$$a_i = -\sum_j D_{ij} q_j \quad i = 1, \ldots, N$$
$$pi \ += \delta a_i \quad i = 1, \ldots, N$$
$$qi \ += \delta p_i \quad i = 1, \ldots, N$$

In a shared memory framework, the computational workload can therefore be shared by several worker threads by assigning different sets of consecutive particles (namely, particles labeled by consecutive vector indices) to different threads. Each worker thread follows the trajectory of a set of particles whose indices are in a predefined subrange.

Most of the computational effort is spent in the computation of the acceleration of a given particle. Since the forces among particles grow linearly with distance, the computation of the acceleration a_i of the ith particle requires the knowledge of the positions of all the other particles in the system. A shared memory programming model, where all threads share the global data set, is ideally suited to this problem.

The same strategy could be implemented in a distributed memory environment, by distributing the workload in such a way that different sets of particles are followed by different MPI processes. However, each MPI process must have at each time step a copy of the whole position dataset in order to be able to compute the relevant accelerations. Once the positions of particles are updated, each MPI process must communicate to all others the new positions of its proprietary particles at each time step. This amounts to a large number of persistent messages equal to the square of the number of MPI processes. Parallel efficiency and scalability for this problem are very difficult to achieve in a distributed memory context.

13.4 Md.C SEQUENTIAL CODE

The Md.C application proposed next computes a very long molecular dynamics trajectory, given a time step δ, monitoring the values of the total and the kinetic energies of the system as the time evolution of the system proceeds. The computation proceeds as follows:

- A long trajectory involving nSteps time steps δ is computed.
- Then, the total and kinetic energies are computed and printed to stdout. Values of the total energy—a constant of the motion—validate the precision of the computation. Values of the kinetic energy are stored for future analysis.
- The trajectory computation continues, repeating the previous steps nSample times.
- At the end of the process, we dispose of nSample samples of the kinetic energy values. Their mean value and variance are computed and printed to stdout.

As stated before, this molecular dynamics code is really a part of a larger algorithm designed to compute correlated Gaussian fluctuations. There is, however, something that can be learned from this simplified computation. The kinetic energy is not by itself a constant of the motion, but it can be observed that its values are not wildly scattered. Instead, they appear concentrated inside a rather narrow range. Besides testing the scalability of the parallel algorithms involved in the application for solving the equations of motion, this code can be used to verify that, as the system size N grows, the

kinetic energy fluctuations decrease (the variance gets smaller and smaller). In the thermodynamic limit $N \to \infty$, the mean value of the kinetic energy per particle is practically a constant, and it is this physical observable that defines the temperature of the system.

13.4.1 INPUT AND POSTPROCESSING

Besides the sequential version Md.C of this application, there are several other parallel versions using different programming environments and optimization options. All the details of the code not related to parallel processing (input, preprocessing, memory allocations) have been factored away in an independent module, MdAux.C, which must be linked with every one of the different code versions.

The listing below shows the common data set that is defined in MdAux.C.

```
// Global variables accessed in "MdAux.C" module
// - - - - - - - - - - - - - - - - - - - - - - - -

int    Nb;              // input, number of beads
int    nTh;             // input, number of threads
int    nSteps;          // input, number of time steps in trajectory
int    nSamples;        // input, number of successive trajectories
float  delta;           // input, time step
float  dA, dB, dC;      // input, D matrix parameters

double  **D;            // correlation matrix
double  *q, *p, *a;     // position, momenta, acceleration
```

LISTING 13.1

Common data set in MdAux.C

The input parameters are read from the file md.dat. Besides the obvious parameters needed for the simulation, other parameters are needed to construct the correlation matrix D. Any symmetric, positive-definite matrix will do for the purposes of our computation here, but we have chosen a correlation matrix coming from a specific macro-molecule model. The three parameters, dA, dB, and dC, are used by an auxiliary function BuildMatrix() in the MdAux.C module to build the D matrix. This is also the reason the number of beads Nb is passed as input parameter, instead of the real matrix size N, which is N=3*Nb

The MdAux.C module defines two public functions that are called by main():

- **InitJob() function**
  - Reads the input data from file md.dat.
  - Allocates D, q, p, and a.
  - Constructs an initial state for the trajectory computation, by giving initial values to the q and p vectors. These initial values are provided by a Gaussian random generator.
  - Calls the BuildMatrix() function to construct D.
- **CloseJob() function**
  - Deallocates D, q, p, and a.

For postprocessing, the main() function uses the DMonitor class from the vath library to compute mean values and variances of a collection of doubles (in this case, the mean values and variances of the kinetic energy samples). This class is used as follows:

- DMonitor M declares the object M that provides the service.
- M.AccumData(x) increases an internal counter, and accumulates internally x—for the mean value computation—and x.x—for the variance computation.
- M.Reset() computes and stores the mean value and variance of the previously collected data, and resets the internal data to prepare for reuse in another computation. The previously computed results are retrieved by calling M.Average() and M.Variance().

13.4.2 SEQUENTIAL CODE, Md.C

Here is the sequential code listing.

```
// Auxiliary functions declarations
// – – – – – – – – – – – – – – – – –
void   InitJob();      // in module MdAux.C
void   CloseJob();     // in module MdAux.C
void   ComputeTrajectory();

// Global variables in this module
// – – – – – – – – – – – – – – – – ––
double Ekin, Etot;        // kinetic and potential energies
DMonitor EK;              // to compute kinetic energy mean values
int N;

void ComputeTrajectory()
   {
   // – – – – – – – – – – – – – – – – – – – – – – – – – – –
   // This function computes first the nStep time steps,
   // and then the kinetic and potential energies
   // – – – – – – – – – – – – – – – – – – – – – – – – – – ––
   int n, j, count;
   double scr;

   for(count=0; count < nSteps; ++count)
      {
      for(n=0; n<N; n++)     // computation of acceleration
         {
         a[n] = 0.0;
         for(j=0; j<N; j++) a[n] += D[n][j] * q[j];
         }

      for( n=0; n<N; n++)   //integration of equations of motion
         {
         p[n] –= (delta * a[n]);
         q[n] += (delta * p[n]);
```

```
          }
       }

    Ekin = 0.0;              // energy computations
    Etot = 0.0
    for(n=0; n<N; n++)
       {
       a[n] = 0.0;
       for(j=0; j<N; j++) a[n] += D[n][j] * q[j];
       scr = p[n] – 0.5 * a[n] * delta;
       Ekin += 0.5*scr*scr:
       Etot += 0.5*q[n]*a[n];
       }
     Etot += Ekin;
     }

// – – – – – – – – – ––
// The main() function
// – – – – – – – – – ––
int main(int argc, char **argv)
     {
     int sample;
     CpuTimer TR;

     InitJob();
     if(argc==2) nTh = atoi(argv[1]);
     N = 3*Nb;
     delta = 0.02;

     TR.Start();
     for(sample=1; sample<=nSamples; ++sample)
         {
     ComputeTrajectory();
         std::cout << "\n Ekin = " << Ekin << "       Etot = "
                   << Etot << std::endl;
         EK.AccumData(Ekin);
     }
      TR.Stop();
      EK.Reset();
      std::cout << "\nAverage kinetic energy " << EK.Average()
                << "\nVariance:              " << EK.Variance();
      TR.ReportTimes();
      CloseJob();
      }
```

LISTING 13.2

Sequential Md.C code.

After performing all the required initializations, main() enters into the computation region and performs a loop over samples, calling the ComputeTrajectory() function that integrates the equations of motion for nSteps time steps, and returns the values of the total and kinetic energies in two global variables, Etot and Ekin.

The ComputeTrajectory() function executes the loop over the nSteps time steps, and then computes the total and kinetic energies. This function shows all the potential parallelism in this problem, and the main modifications in the parallel versions of the code will be performed here.

Example 1: Md.C

To compile, run make md. Sequential version. Input data is read from file md.dat. The number of threads, read initially from this file, can be changed from the command line.

13.5 PARALLEL IMPLEMENTATIONS OF Md.C

There are several different parallel implementations of the molecular dynamics application, discussed next.

13.5.1 GENERAL STRUCTURE OF DATA PARALLEL PROGRAMS

A typical data parallel code has the structure shown in Figure 13.2. The main() function starts and performs all the input, memory allocations, and initialization operations needed to set up the program context. Then, there is in general a series of parallel regions where the workload is distributed across a

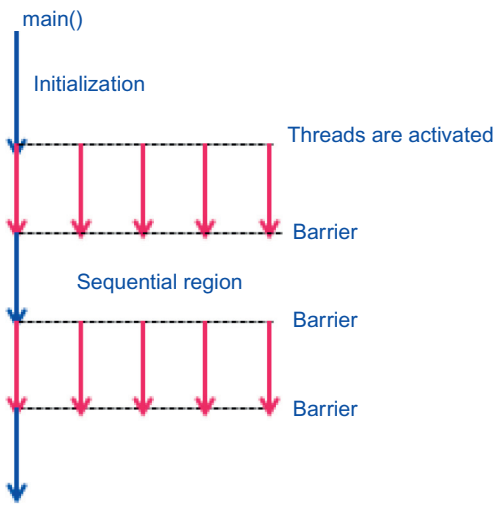

FIGURE 13.2

General structure of a data parallel code.

team of worker threads, followed by sequential sections where the workload is carried out by only one thread. Remember that in OpenMP the master thread suspends the ongoing task and joins the workers team, while in the case of the vath library pools the master thread is either doing something else or executing an idle wait.

There are several ways to organize the parallel treatment of Figure 13.2. In some simple cases, like in our molecular dynamics code, the parallel regions are restricted to loops that can be computed in parallel because the loop iterations are totally independent. In this case, a *microtasking* approach can be adopted. The OpenMP parallel for directive or the TBB parallel_for or parallel_reduce algorithms can be used to automatically distribute the loop workload across the worker threads. In this case, the code remains all the time in sequential mode with the exception of the work-sharing parallel sections related to the loops.

If the successive parallel regions in the code are all similar, in the sense that they involve the same number of threads acting on the same data sets, a *macrotasking* approach can be adopted. The complete parallel code consists of a unique parallel section executing a unique parallel task. The worker thread team is active all along, the code being by default in parallel mode. Eventual transitions to sequential regions are handled either by using the master or single directives in OpenMP, or by selecting only one active thread between two barriers (see the figure) with a conditional if(rank==n) statement.

13.5.2 DIFFERENT CODE VERSIONS

There are several different versions of the code that take the same input and produce the same output:

- *Md.C*: Sequential version.
- *MdOmp.C*: Parallel, OpenMp version using a macrotasking programming style (big parallel tasks with explicit synchronizations).
- *MdOmpF.C*: Parallel, OpenMP version using a microtasking programming style in which the loops involved in the leapfrog time step are parallelized using the parallel for directive.
- *MdTh.C*: Parallel version based on the NPool thread pool, using a macrotasking programming style. There are several barrier synchronization points in this application, and this version of the code allows us to select, using the preprocessor, either the standard Barrier class based on a idle wait, or the alternative SpBarrier class based on a busy wait, in order to compare their performance.
- *MdTbb.C*: Parallel TBB version using a microtasking style where the loops involved in the leapfrog time step are parallelized using the parallel_for algorithm.

These different code versions, corresponding to different multithreading environments and strategies, are portable as explained in Annex A. Additional optimization options, not directly related to multithreading, are also explored, such as vectorization or optimized memory accesses. This will been done at the end of the chapter for the OpenMP version of the code, using the capabilities of the Intel C-C++ compiler, because the OpenMP 4.0 simd vectorization directive was not yet implemented in the GNU compiler at the time of the preparation of this book. In well-adapted data parallel applications—and Md.C is one of them—the combination of multithreading and vectorization can provide significant performance enhancements.

13.5.3 OpenMP MICROTASKING IMPLEMENTATION, MdOmpF.C

This first OpenMP version of the code only requires a couple of minor changes to the sequential version:

- Add the parallel for directive in the loops involved in the leapfrog time step computation, as indicated in the listing below.
- Add also, in the code block where the parallel_for directive appears, a call to omp_set_num_threads() fixing the number of threads in the next parallel regions. Or, instead, add a num_threads clause in each parallel directive.
- Note that there is no change in the main function. Main just calls a function with internal, OpenMP-driven, parallel execution.

```
void ComputeTrajectory()
   {
   ...
   omp_set_num_threads(nTh);
   for(count=0; count < nSteps; ++count)
      {
      #pragma omp parallel for
      for(n=0; n<N; n++)     // computation of acceleration
         {
         a[n] = 0.0;
         for(j=0; j<N; j++) a[n] += D[n][j] * q[j];
         }

      #pragma omp parallel for
      for( n=0; n<N; n++)   //integration of equations of motion
         {
         p[n] -= (delta * a[n]);
         q[n] += (delta * p[n]);
         }
      }
   ...
```

LISTING 13.3

MdOmpF.C code (partial listing).

In the listing above, there is no collapse(2) clause in the parallel for directive acting on the double acceleration loop, asking for the fusion of the two nested loops into a huge single loop. In fact, the collapse clause here is refused by the compiler, politely explaining that because of the a[n]=0.0 initialization statement the two loops are not strictly nested. It is possible to separate in another previous loop the a[n] initialization, and this works, but the whole modification does not buy us substantial performance improvement. As will be discussed later on, *vectorizing the inner loop* is a much better option than using loop fusion.

The ComputeTrajectory() function includes also a loop involved in the energy computation that can also be the target of a parallel_for directive. This is not critical for performance, because energy computations are indeed very occasional. Nevertheless, the parallel for directive has been added in

the energy computation loops in order to have the same level of parallel execution as in other implementations of the code.

Example 2: MdOmpF.C

To compile, run make mdompf. The input data is read from file md.dat. The number of threads, read initially from this file, can be changed from the command line.

13.5.4 OpenMP MACROTASKING IMPLEMENTATION, MdOmp.C

Here is the parallel OpenMP version of the code, in a macrotasking programming style. The most important changes are made in the ComputeTrajectory() function, which is transformed into a task function executed by the worker threads.

```
#include <ThreadRangeOmp.h>
...
// – – – – – – – – – – – – – – – – – – – – – – – – – –
// Task function for the computation of the molecular
// dynamics trajectory
// – – – – – – – – – – – – – – – – – – – – – – – – – –
void ComputeTrajectory()
   {
   int n, j, count, rank;
   double ekin, epot, scr;
   int nl, nh;

   nl = 0;     // set thread range
   nh = N;
   ThreadRangeOmp(nl, nh);

   for(count=0; count < nSteps; ++count)
      {
      for(n=nl; n<nh; n++)      // computation of accelerations
     {
     a[n] = 0.0;
     for(j=0; j<SZ; j++) a[n] += D[n][j] * q[j];
     }

      #pragma omp barrier       // barrier 1

      for( n=nl; n<nh; n++)     // equations of motion
     {
     p[n] –= (delta * a[n]);
     q[n] += (delta * p[n]);
     }
      #pragma omp barrier     // barrier 2
      }
```

Continued

```
    // Trajectory done. Compute energies
    // - - - - - - - - - - - - - - - - - -
    ekin = 0.0;
    epot = 0.0
    for(n=nl; n<nh; n++)
       {
       a[n] = 0.0;
       for(j=0; j<SZ; j++) a[n] += D[n][j] * q[j];
       // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
       // Notice: we do not need a barrier here, because we are
       // not going to use the acceleration to update coordinates.
       // See the text below.
       // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
       scr = p[n] - 0.5 * a[n] * delta;
       ekin += 0.5*scr*scr;
       epot += 0.5*q[n]*a[n];
       }

    #pragma omp critical      // accumulate in global variables
       {
       Ek += ekin;
       Et += (ekin+epot);
       }
    }


// The main() function (partial listing)
// - - - - - - - - - - - - - - - - - - -
int main(int argc, char **argv)
    {
    ...

    TR.Start();
    for(sample=1; sample<=nSamples; ++sample)
        {
        Ek = 0.0; Et = 0.0;
        #pragma omp parallel
            {
            ComputeTrajectory();
            }
        // Print kinetic and total energies
        // - - - - - - - - - - - - - - - - -
        std::cout << "\n Ekin = " << Ek << "       Etot = "
                  << Et << std::endl;
        M.AccumData(Ek);     // accumulate kinetic energy values
    }
     TR.Stop();
     ...
     }
```

LISTING 13.4

Partial listing of MdOmp.C code.

Work-sharing among threads

In order to implement the work distribution among threads by hand, an auxiliary function ThreadRangeOmp() has been introduced that operates in the same way as the ThreadRange() member function of the SPool utility. It receives as arguments two references to the global index range $[0, N)$ and returns in these arguments the subrange appropriate for the caller thread. This function takes into account the fact that the OpenMP thread ranks are in [0, nThreads-1] instead of [1, nThreads] as is the case for the SPool utility. The function is defined in the include file ThreadRangeOmp.h.

After completing all the scheduled leapfrog time steps, each thread computes etot and ekin, their partial contributions to the total and kinetic energies. A critical section is then used to accumulate these partial results in the global variables Et and Ek, read by main().

Barrier synchronizations

The task function listed above involves two barrier synchronization points at each time step, and it is important to clearly understand the nature of the race conditions that enforce the presence of these two barriers. It is intuitively obvious that a barrier must be placed at points where worker threads must wait for results coming from other worker threads to proceed. But this is not all: worker threads must also avoid corrupting the work performed by the other partners.

Let us focus our attention on what a particular worker thread is doing. In examining the need of synchronization with other worker threads, two questions must be asked:

- *Criterium A*: How can my partner threads induce race conditions that invalidate my calculation?
- *Criterium B*: How can my computation induce race conditions that invalidate my partner's calculations?

Let us examine, on the basis of these two criteria, the role played by the two barriers encountered in the listing above:

- Barrier 1: forces a wait after the computation of the acceleration. Here, criterion B is at work. It may appear at first glance that, once the accelerations are known, each worker is free to go ahead and update the velocities and positions of the particles under its control. This looks very innocent, and one may conclude at first sight that no barrier is needed. However, it may happen that other threads are late in computing *their* accelerations, and they still need to read the current thread-owned coordinates at time step n, not at time step $n + 1$. Therefore, the current thread owned positions cannot be updated before all the other threads are finished computing their accelerations.
- Barrier 2: forces a wait at the end of a time step in the solution of the equations of motion, before starting a new loop iteration. Here, criterion A is at work: wait until all positions are updated before starting again with a new computation of the accelerations.

Example 2: MdOmp.C

To compile, run make mdomp. OpenMP version. Input data is read from file md.dat. The number of threads, read initially from this file, can be changed from the command line.

13.5.5 SPool IMPLEMENTATION, MdTh.C

Given the simplicity of the parallel algorithm, this version of the code uses the SPool thread pool utility. The thread function to be executed by each thread is very similar to the OpenMP function presented in the listing above. In the main function, the OpenMP parallel section is replaced by a job dispatching to the pool.

Idle-wait versus spin-wait barriers

The two barrier synchronization points that are repeatedly accessed inside a loop provide an excellent opportunity for testing the performance of the different barrier classes introduced in Chapter 9, corresponding to different wait strategies. The MdTh.C module uses the C preprocessor to select the different barrier interfaces: idle wait (Barrier class) or spin wait (SpBarrier class). In this version of the molecular dynamics code, B is a pointer to a barrier object, initialized as follows:

```
// In the global variable declarations:
// - - - - - - - - - - - - - - - - - - -
#ifdef SPIN_BARRIER
SpBarrier *B;
#else
Barrier *B;
#endif

// In the main() initialization of B
// - - - - - - - - - - - - - - - - - - -
#ifdef SPIN_BARRIER
B = new SpBarrier(nTh);
#else
B = new Barrier(nTh);
#endif
```

LISTING 13.5

Selecting barrier type with preprocessor.

The rest of the code is unchanged, because both barrier classes have the same public interfaces.

Example 3: MdTh.C

To compile, run make mdth. SPool version. Input data is read from file md.dat. The number of threads, read initially from this file, can be changed from the command line.

Comparison of code performance does confirm that the spin-wait barriers moderately outperform the idle-wait barriers when the synchronization contention begins to be important. Consider first the tests performed running a medium-sized configuration in a four-core laptop. We take the case of Ns=660 beads (the vector size being therefore 1980), and 20 kinetic energy samples are computed, with 1000 trajectory time steps per sample. We are naturally using four threads. The other parameters in the run

Table 13.1 Wall Execution Times in Seconds, for 18,000 Particles, 10 Samples, 10 Time Steps per Sample (Using GNU 4.8 Compiler)

Idle vs Spin Barriers			
Barrier Type	**Wall (s)**	**User (s)**	**System (s)**
Idle	57	202.9	0.98
Spin	55.1	218.9	0.03
OpenMP	53.9	214.5	0.08

are the ones defined in the md.dat file that comes with the codes. Table 13.1 shows the measured wall, user and system times.

A data set of 1980 particles shared among four threads does not represent an enormous computational workload per thread, and we have in this configuration a total of 40,000 barrier calls per thread. Spin barriers systematically provide in this case a 3-4% better performance than idle-wait barriers. Performance improvements are expected to be more important when the number of worker threads (and the barrier contention) is increased. The barrier contention grows faster than linear with the number of threads.

The OpenMP performance reported in Table 13.1 does not change if the OpenMP spin-wait policy is forced by setting OMP_WAIT_POLICY to active. This probably means the OpenMP implementation is using a spin-wait policy by default.

13.5.6 TBB IMPLEMENTATION, MdTbb.C

The TBB implementation is again a microtasking style code, which remains in sequential mode. At each time step the main function just calls parallel_for twice, first to compute the accelerations, and next to update momenta and coordinates. Two very simple appropriate Body classes are then provided.

Occasionally, the code computes the total and kinetic energies, and it is not critical to perform this computation in parallel. However, for comparison with other implementations, a third Body class for the parallel energy computation using parallel_reduce is provided.

Example 10: MdTBB.C

To compile, run make mdtbb. Input data is read from file md.dat.

13.6 PARALLEL PERFORMANCE ISSUES

The performance of the different implementations of the molecular dynamics code are examined next, the target platform being a Linux SMP node composed of two Intel Sandy Bridge sockets, each socket integrating eight cores running two hardware threads each. In the Sandy Bridge socket, each core has

its own, private L1 and L2 caches. This configuration can run up to 32 hardware threads, but hyper-threading has been excluded in our tests, and only one thread per core is run. In general-purpose Intel processors, hyper-threading is often useful, but it is not a *sine qua-non* condition for performance, as is the case for the Intel Xeon Phi co-processor.

A relatively large data set (18,000 particles) is chosen, and the runs are limited to short trajectories (10 time steps each) and only 10 energy samples. There are therefore only 100 time steps altogether. The motivation is to have short execution times, while giving each thread a large number of particles to work upon so as to maximize as much as possible the ratio of computation to synchronization overhead. Table 13.2 gives the execution times for the sequential code `md` and for the different parallel implementations, for 4, 8, 12, and 16 threads. In running the TBB implementation, the granularity in the `parallel_for` and `parallel_reduce` algorithms has been adjusted to fit the domain decomposition pattern to the number of worker threads. The Intel C-C++ compiler, version 13.0.1, has been used, because of its additional vectorization options, which will be discussed in the next section.

Two obvious comments follow from the results of Table 13.2:

1. *Comparable performance of all programming environments*, as well as the fact that the automatic work-sharing facility in OpenMP has the same performance as the macrotasking version with the work-sharing done "by hand." This is not, however, very surprising after all. The parallel pattern in this problem is rather trivial—a straightforward work-sharing data parallel pattern—and, given the limited number of time steps, there is a limited amount of synchronization overhead.
2. *Excellent scaling up to eight threads*. Up to eight threads, nice parallel behavior is observed, with a speedup proportional to the number of threads. Beyond eight threads, these nice scaling properties start to fade away. This is a rather common feature for memory bound applications running on general-purpose computational engines, not specially designed for high-performance computing. On-chip communications inside a socket are much more efficient than off-chip communications across sockets.

What features of this application contribute to the scalability limitations? First, there is the growing impact of synchronization overhead. In the macrotasking versions (SPool and OpenMP), there are

Table 13.2 Wall Execution Times in Seconds, for 18,000 Particles, 10 Samples, 10 Time Steps per Sample, With Intel C-C++ Compiler, Version 13.0.1

	Performance				
nb Threads	**md**	**mdth**	**mdtbb**	**mdomp**	**mdompf**
1	118.7				
4		27.3	27.2	27.2	27.3
8		14.4	13.9	14.3	14.1
12		10.3	9.9	9.9	9.9
16		8.27	7.94	8.2	8.2

two barriers per time step. In the microtasking versions (TBB and parallel for in OpenMP), there are no explicit barriers, but implicit barriers are present at the transition points from a parallel to a sequential region. Perfect scaling is only possible if synchronization costs are negligible with respect to computation. This is the case for a small number of threads. However, when profiling the mdth code execution for 16 threads, we can observe that barrier wait calls account for about 30% of the execution time. It is clear that, when the same fixed amount of work is shared among too many workers, they are forced to spend most of their time in synchronizing their activity.

Another issue worth thinking about is the efficiency of memory accesses. The hierarchical cache-based memory system is designed to take advantage of space and time locality: space locality, by moving a contiguous memory block to a cache line, and time locality, by keeping in the cache the most frequently accessed cache lines. When looking at the pattern of memory accesses to the $D[ij]$, $q[i]$, $p[i]$) arrays in our application, we observe that:

- The $D[ij]$ matrix is constructed in the initialization part of the code, and from there on it is only accessed in read mode, in the computation of the acceleration. Moreover, in this computation, *each thread repeatedly accesses the same exclusive set of contiguous rows*. This means these rows are likely to persist in the owner-thread L2 cache, if the cache is big enough. This memory access pattern is satisfactory.
- The same observation applies to the momentum vector $p[i]$. A given thread only accesses the contiguous momentum components corresponding to the particles under its control. Therefore, these momentum components are likely to persist in each thread L2 cache. It may happen, if subranges sizes are not an exact multiple of the cache line size, that threads A and B have a cache line with momentum values corresponding, for example, to the end of the thread A range and the start of the thread B range. In this case, the *false-sharing* phenomenon may occur: when A updates its momenta, the cache line is invalidated and thread B is forced to recover its momentum values from main memory in spite of the fact that they have not been updated (yet). However, for a big data set, this phenomenon is infrequent.
- The situation is completely different for the coordinates $q[i]$. In the acceleration calculation, all threads read the complete data set, and there is absolutely no locality of any kind in this case. Indeed, all threads upload the complete $q[i]$ vector into their L2 cache. Then, each one updates its exclusive part. This means all cache lines in all L2 caches are invalidated, and the complete data set must be recovered from main memory at the next time step.

The $q[i]$ memory accesses are not optimal, but they result from the nature of the problem at hand (long-range forces connecting all pairs of particles), and there is nothing we can do about it.

13.7 ACTIVATING VECTOR PROCESSING

Once a satisfactory multithreaded implementation of the application is obtained, additional performance improvement can be obtained if threads are executing loops that can be vectorized, which is true in our case. Vectorization, as explained in Chapter 1, is an SIMD (single instruction, multiple data) intrinsic parallel facility inside a core that enables the simultaneous execution of several loop

iterations. The Sandy Bridge cores being used here have vector registers 256 bytes wide that can accommodate four SIMD lanes for doubles or eight for floats. In our case, where arrays of doubles are used, vectorization should provide a theoretical speedup of 4 if the code execution is strongly dominated by the execution of vectorizable loops.

Section 13.8 at the end of this chapter provides a short review of the Intel compiler vectorization programming interfaces. Readers are strongly encouraged to consult the official documentation [36]. Chapter 7 of J. Jeffers and J. Reinders book on Intel Xeon Phi programming [8] is also an excellent vectorization tutorial that applies to all Intel processors. Loop vectorization is the default behavior of the compiler, but its vectorization analysis is very conservative (as it should be), and in most cases it must be helped with directives to achieve a satisfactory vectorization. These basic directives are reviewed in Section 13.8 at the end of this chapter. We will concentrate on the OpenMP macrotasking `mdomp` implementation of our code, but vectorization is strictly a single-core optimization affair, having nothing to do with multithreading, and the same analysis can be implemented for `vath` of `TBB` implementations.

The `MdOmpV.C` file is the same as `MdOmp.C`, with added vectorization directives and a simple reformulations of the code needed to facilitate vectorization. The listing below shows the `ComputeTrajetory()` task function, where the vectorization directives are inserted.

```
void ComputeTrajectory()
   {
   int n, j, count, rank;
   double ekin, epot, scr;
   int nl, nh;

   double *V = D[0];     // new
   ...
   for(count=0; count < nSteps; ++count)
      {
      for(n=nl; n<nh; n++)      // computation of accelerations
         {
         scr = 0.0;
         #pragma ivdep
         for(j=0; j<SZ; j++) scr += V[n*N+j] * q[j];
         a[n] = scr;
         }

      #pragma omp barrier       // barrier 1

      #pragma ivdep
      for( n=nl; n<nh; n++)     // equations of motion
         {
         p[n] -= (delta * a[n]);
         q[n] += (delta * p[n]);
         }
      #pragma omp barrier     // barrier 2
      }
```

```
// Trajectory done. Compute energies
// Similar directives are applied in the loops
// that follow
// - - - - - - - - - - - - - - - - - - - - - -
...
}
```

LISTING 13.6

Vectorized trajectory computation (partial listing).

Vectorization is performed by using the mild #pragma ivdep, telling the compiler that the pointers that follow point to nonoverlapping regions in memory (see Section 13.8). The loops that update momenta and coordinates are directly vectorized with this directive. The double loop for the acceleration computation is slightly more subtle. The correct procedure in this case is to vectorize the huge inner loop that runs over the complete particle set. This part of the code has been rewritten in a way that helps the compiler better understand what is going on.

Indeed, when the strong #pragma simd directive was applied to force the vectorization of the initial version using the D[i][j] matrix elements, the vectorization failed with the *dereference too complex* error message (see Section 13.8). In fact, the compiler was reluctant to vectorize a construct with nested indirections, due to the way the D matrix was allocated. Section 13.7 later in this chapter shows the matrix allocation procedure: D is a pointer to an array of pointers to vectors of doubles. However, internally, D[0] is just a vector array of size NM where the D matrix rows are ranged one after the other, and this simpler data structure is better controlled by the vectorization tests performed by the compiler. The code was therefore reformulated to simplify the compiler job by using a vector array V=D[0] in the inner loop, instead of the D matrix elements themselves. With this modification, the loop was vectorized. Finally, the strong #pragma simd directive was replaced by the milder #pragma ivdep directive.

This episode shows that vectorization is not an affair that reduces to throwing the right directive at the right place. Compiler feedback is often needed for success. The #pragma omp simd directive introduced by OpenMP 4.0 has been discussed in Chapter 10. Having standard vectorization interfaces in OpenMP is a huge step forward, but keep in mind that the compiler feedback is not part of the standard, and that efficient usage of the OpenMP vectorization directive may require some familiarity with the specific implementation.

Another single-core optimization that can be explored is memory alignment. This code works with arrays of doubles coded over 64 bytes. In all platforms, L2 cache line sizes are a multiple of 64 bytes. The Sandy Bridge L2 cache lines, for example, are 512 bytes (8 doubles) wide. If the starting memory address of an array is 64 bytes aligned, namely, if this address is a multiple of 64 bytes, then array elements will never spread across a cache line boundary when array chunks are mapped to the cache memory. Memory alignment facilitates data transfers in the hierarchical memory system. As explained in Section 13.8, aligned memory allocation can be implemented by using an Intel replacement of malloc and free. Table 13.3 compares the performance of the mdomp code reported already in Table 13.2, with the vectorized version mdompv. The preprocessor has been used to select the memory alignment option. If the ALIGNED_MEM_ALLOC variable is defined (both in MdOmpV.C and MdAux.C), the Intel allocation routines are selected. Otherwise, the standard allocation with malloc() applies. In Table 13.3,

Table 13.3 Wall Execution Times in Seconds, for 18,000 Particles, 10 Samples, and 10 Time Steps per Sample

Vectorization Performance			
nb Threads	**mdomp**	**mdompv**	**mdompv(a)**
4	27.2	10.5	11
8	14.3	8.5	8.6
12	9.9	8.3	8.4
16	8.2	8.2	8.25

the mdompv column corresponds to vectorization only, and mdompv_a to vectorization plus aligned memory allocation.

We can observe that, for four and eight threads, when the code has good scaling properties because computation dominates synchronization overhead, vectorization provides significant performance improvement: a speedup close to three for four threads and slightly above two for eight threads. We can also see that aligned memory allocation does not induce here a significant change in performance.

13.8 ANNEX: MATRIX ALLOCATIONS

Consider a M×N matrix D[m][n], where 0<=m<M and 0<=n<N. This corresponds to *matrix indices having offset 0*, as is customary in C-C++. This matrix will be allocated using *new* and *delete*, but the same thing can be done with *malloc* and *free*.

The allocation and deallocation procedures are as follows:

```
double **D;     //  is a pointer to an array of pointers to doubles

// Allocate an array of M pointers to (double *).

D = new (double *)[M];

// Allocate D[0] — the first usable pointer in the array above — as
// a huge array of doubles of size (N*M), where all the matrix
// elements go, one row after the other.

D[0] = new double[N*M];

// D[0][0] is pointing to the first element of the first row, and all
// the matrix elements follow. We will now initialize the pointers D[i]
// so that D[i][0] points to the first element of the ith row:

for(int n=1; n<N; n++) D[n] = D[n-1] + N;
```

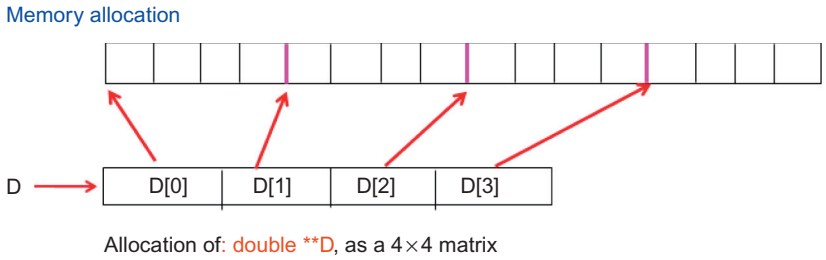

FIGURE 13.3

Matrix allocation procedure.

```
// Deallocation:
//  - - - - - -
delete [] D[0];   // get rid of the data
delete [] D;      // get rid of the pointers
```

LISTING 13.7

Allocation of matrix of doubles.

Figure 13.3 shows clearly what happens for a 4*4 matrix, called D. Note that one of the advantages of this allocation procedure is being able to enable *loop fusion*: a double loop over rows and columns can be replaced by a single loop over $D[0][n]$ with $0 <= n < (N * M)$.

13.9 ANNEX: COMMENTS ON VECTORIZATION WITH INTEL COMPILERS

This section summarizes the compiler vectorization directives, as well as the Intel interfaces for aligned memory allocation [8, 36]. Automatic vectorization is enabled by default, and a typical command line for the compiler is (see the makefiles of this chapter):

```
icpc -O3 -vec-report=3 -omdomp MdOmp.C MdAux.C
```

The vec-report=3 switch demands a detailed report of the compiler vectorization activity. Every loop in the code is carefully examined, and the compiler produces a very detailed report for each one of them explaining why vectorization failed (the most common situation). The compiler has substantial built-in intelligence. It performs lots of checks and tests, and takes a very conservative attitude. At this point start the programmer efforts to help the compiler achieve an efficient vectorization of the critical loops that control the overall performance, by adding directives. These directives are hints to the compiler to remove certain restrictions it may be enforcing by default.

The Intel compiler allows programmers to guide the vectorizer activity at three different levels:

- *#pragma ivdep*: This means *ignore vector dependencies*. This directive instructs the compiler to ignore the checks concerning possible vector dependencies. A typical use case are loops with pointers to arrays: the compiler will refuse vectorization unless it is absolutely certain there is no aliasing; namely, that different pointers are not pointing to overlapping memory buffers. This

directive provides the assurance that this is not the case. But the compiler remains very conservative on all other vectorization checks.

- *#pragma vector* [clauses]: This directive provides further hints for ignoring some other checks, but still leaves room for conservative behavior of the vectorizer. For example, #pragma vector always tells the compiler to ignore efficiency heuristics, but still only works if the loop can be vectorized. And #pragma vector aligned tells the compiler that all memory references in the loop are aligned on 64-byte boundaries.
- *#pragma simd* [clauses] is the most aggressive clause: It is a *vectorize or die* directive, asking the compiler to relax all requirements and make all possible efforts to vectorize the loop. This directive is roughly equivalent to #pragma vector always and #pragma ivdep, but more powerful because it does not perform any dependency analysis.
- The simd directive is very close to the same OpenMP directive. It accepts similar clauses: reduction, private, firstprivate, and lastprivate.
- There are, however, a number of restrictions for SIMD vectorization:
  - Loops must be countable: the number of iterations must be known before the loop starts to execute.
  - Some operations are not supported: the compiler produces an *operation not supported* error message.
  - Very complex array subscripts or pointer arithmetic may not be vectorized: the compiler produces a *dereference too complex* error message.
  - Loops with a very low number of iterations (low trip count) may not be vectorized: the compiler produces a *low trip count* error message.
  - Extremely large loop bodies (with many lines and symbols) may not be vectorized.
  - The SIMD directive cannot be applied to a loop containing exception handling code in C++.

For a more detailed discussion, the extensive Intel documentation on this subject can be consulted. We strongly recommend reading Chapter 4 of James Reinder's book on Intel Xeon Phi [8], which provides a clear and complete overview of all the programmer's vectorization options, as well as memory access optimizations like the data alignment issue discussed below.

13.9.1 DATA ALIGNMENT

Imposing data alignment on specific memory boundaries may help the runsystem to optimize memory accesses. The way to proceed is to align the data when memory is allocated, and then use directives in places where the data is used to inform the compiler that data arguments are aligned.

- Align the data.
- Use pragmas/directives in places where the data is used to tell the compiler that arguments are aligned.

Aligned dynamic memory allocation based on replacements of malloc() and free() provided by Intel C++ Composer XE. The malloc replacement, called _mm_malloc(), has the following signature:

```
void *_mm_malloc(int size, int base)
```

which is the malloc() signature with an extra argument base that defines the alignment. Pointers allocated in this way must be released with _mm_free(). The listing below shows some examples of aligned allocation of static and dynamic arrays.

```
// Aligned allocation of an static array
// – – – – – – – – – – – – – – – – – – – –
float A[1000] __attribute__((aligned(64))));

// Aligned declaration of pointers
// – – – – – – – – – – – – – – – – – – – –
_ _assume_aligned(ptr, 64); // ptr is 64B aligned

// Aligned dynamic allocation of an array of floats, starting on
// a 64 byte boundary
// – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
float *V = (float *)_mm_malloc(1000*sizeof(float), 64);
...
_mm_free(V);

// Informing the compiler of the alignment:
// – – – – – – – – – – – – – – – – – – – –
void myfunc(double p[])
   {
   __asume_aligned(p, 64);
   for(int n=0; n<N; n++) p[i]++;
   }

// Telling the compiler that ALL references are nicely aligned
// – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – – –
#pragma vector aligned
for(n=0; n<N; n++)
  A[n] = B[n] + C[n];
```

LISTING 13.8

Data alignment examples.