Introduction into High Performance Computing

Dr. M.F. Somers, May 2020.
Theoretical Chemistry group, Leiden University.
m.somers@chem.leidenuniv.nl

Table of content

Introduction into SMDEP

In this course a basic introduction into the field of High Performance Computing (HPC) is given. HPC is a field of science in which challenging scientific problems are addressed through computing. These challenging problems often work on the boundary of what is still just computable and what is not. Examples of those scientific challenging areas are i.e. protein folding or molecular dynamics or the dynamics of large galaxies. In these problems, it is the computer that limits the number of particles or stars that can actually be simulated.

In this course the basics of HPC are discovered by actually doing a challenging example molecular dynamics simulation and by going through the example step by step, you will learn how to make possible what seems to be impossible at first. The tricks and techniques you learn in this course can be applied to many other challenging computations.

This course makes heavily use of a Simple Molecular Dynamics Example Program (SMDEP). This code is only about 450 lines of C, but does contain all the basic stuff to be a true Molecular Dynamics (MD) program. You can find the sources of the program in the SMDEP sub-directory. The main code can be found in the file 'SMDEP.c' and essentially only contains two things, a function to setup an initial test system, and a loop for integrating the equation of motion, which in this case is just the Newton law (http://en.wikipedia.org/wiki/Newton's_laws_of_motion). The integration of the equation of motion is done step by step in time by calling the integrator located in the file 'Integrator.c', which implements a straightforward Velocity-Verlet type integrator (http://en.wikipedia.org/wiki/Verlet_integration). It is in this integrator that the forces acting on all the particles need to be computed and that's why the integrator has a special parameter to specify the force field function of the example system:

```
int IntegrateATimeStep( int N, Particle P[], Particle Pdt[], double dt, ForceRoutineType
TheForceRoutine )
{
 int i;

 // Perform integration of position from P into Pdt... Assume that the forces are already
computed correctly...
 for( i = 0; i < N; ++i )
 {
  Pdt[ i ].m = P[ i ].m;
  Pdt[ i ].r = AddVectors( P[ i ].r, AddVectors( ScalarVector( dt / P[ i ].m, P[ i ].p ),
ScalarVector( 0.5 * dt * dt / P[ i ].m, P[ i ].F ) ) );
 }

 // Now calculate forces at t=dt into Pdt...
 if( i = (*TheForceRoutine)( N, Pdt ) ) return( i );

 // And finally update momenta too into Pdt...
 for( i = 0; i < N; ++i )
  Pdt[ i ].p = AddVectors( P[ i ].p, ScalarVector( 0.5 * dt, AddVectors( P[ i ].F,
Pdt[ i ].F ) ) );

 return( 0 );
}
```

The integrator and the force routine all use vectors and the basic vector arithmetic is implemented into the 'Vector.c' code. Particles, and basic functions for handling them, are defined in the 'Particles.c' file. The example Lennard-Jones force field (http://en.wikipedia.org/wiki/Lennard-Jones_potential) has been implemented into the 'LennardJones.c' file.

Please study the code well and try to answer the following questions before going to the next chapter.

Questions and Exercises:

1) *How many components does a vector in SMDEP have and how is the length of such a vector determined?*

2) *What fields does a particle structure have in SMDEP and what do they represent? How is the total kinetic energy of a collection of particles computed and what is the center of mass?*

3) *Where is the number of particles set for the simulation and the total time to be simulated?*

4) *Given the integrator routine above, how many force calculations are needed per time step?*

5) *Below is given the basic Lennard-Jones force routine. Given N particles, how many pairs are there in total? How come the loops over i and j are as shown? Why was the routine implemented by looping over pairs rather than looping directly over all particles? What part of this routine do you think is the computationally most expensive part?*

```
int LennardJonesForces( int N, Particle P[] )
{
 Vector r;
 int i, j;
 double k;

 for( i = 0; i < N; ++i )
  P[ i ].F.X = P[ i ].F.Y = P[ i ].F.Z = 0.0;

 for( i = 0; i < N; ++i )
  for( j = i + 1; j < N; ++j )
  {
   r = SubVectors( P[ j ].r, P[ i ].r );
   k = 4.0 * LENNARDJONES_EPSILON * ( 12.0 * pow( LENNARDJONES_SIGMA / AbsVector( r ), 12.0 ) -
6.0 * pow( LENNARDJONES_SIGMA / AbsVector( r ), 6.0 ) ) / AbsVector( r );
   P[ i ].F = AddVectors( P[ i ].F, ScalarVector( k, r ) );
   P[ j ].F = SubVectors( P[ j ].F, ScalarVector( k, r ) );
  }

 return( 0 );
}
```

Makefiles and compiling SMDEP

Before you can actually start and run your first simulation with SMDEP, you will need to compile the program. Compiling scientific codes is often done by using the 'make' tool. The 'make' tool takes a so called 'Makefile', which contains all the information on how to make the program. 'make' will use that information and invoke the necessary compilers or commands. In the SMDEP subdirectory you can find the 'Makefile' for SMDEP:

```
# Basic makefile for SMDEP code...

# The flags to use in compiling and loading...
CC = gcc
CFLAGS =
LDFLAGS =

# All the .o needed in this program...
OBJS = SMDEP.o Vector.o Particles.o Integrator.o LennardJones.o

# Rule to make .o from .c files...
%.o : %.c
        ${CC} ${CFLAGS} -c $<

# Rule to make SMDEP.x from all .o files...
SMDEP.x : ${OBJS}
        ${CC} -o SMDEP.x ${LDFLAGS} ${OBJS} -lm

# Rule to clean up crap...
clean :
        rm -rf ${OBJS} SMDEP.x gmon.out
```

In this 'Makefile' two targets have been specified; 'SMDEP.x' and 'clean'. If you invoke 'make' with the 'clean' target (by invoking 'make clean'), the 'rm -rf ...' command is actually executed. If you invoke 'make' without any targets, the first target (in this case 'SMDEP.x') is being made. It will then invoke the command 'gcc -o SMDEP.x SMDEP.o Vector.o Particles.o Integrator.o LennardJones.o' after having made these individual .o files because as was specified for the target 'SMDEP.x', SMDEP.x depends on it. Each of the needed .o files are automatically created from the corresponding .c file with a command similar to i.e. 'gcc -c Vector.c'. 'make' knows how to do this because it was specified in the 'Makefile' in the part containing '%.o : %.c'. This line actually means, any .o file depends on its corresponding .c file and can be generated form the .c file by invoking the GNU gcc compiler. So, to compile the code, just run the 'make' command in the SMDEP subdirectory. 'make' will take care of all dependencies for you and invoke the correct compiler commands for you. You can even add extra compiler options by setting the 'CFLAGS' and 'LDFLAGS' parameters in the beginning of the 'Makefile'.

Questions and Exercises:

1) *Go into the SMDEP source directory, invoke 'make', make sure you understand how the SMDEP code is compiled and run the program by issuing the command 'time ./SMDEP.x'. How many seconds did this example simulation take?*

2) *Change the number of particles from 100 to 50 and determine how long this simulation takes on your computer. Also time the program with a 150 and a 25 particle simulation. How does the program scale with the number of particles? How long would it take to run a 1000 particle*

simulation? What about a 10000 particle simulation?

3) *Which part of the program is very likely to cause the above observed scaling behavior?*

Basic computer architecture

In the previous chapter you actually compiled the SMDEP code and ran a few simulations. You also found out that, although the code works pretty fast for system sizes of about 100 particles, the scaling of the code, with respect to the number of particles, is quadratic, and simulations on systems of sizes of 1000 or more particles are probably less tangible on your desktop computer. Now you could obviously refrain yourself from simulating such large systems, however, sometimes the science you are doing does not give you any choice in that and somehow, programs often need to be optimized and perhaps parallelized too, to be able to run bigger simulations. With the SMDEP code, when going beyond 1000 particles, your are entering in the true realm of High Performance Computing (HPC). However, before indulging yourself even more, a basic introduction in computer architecture is needed. This chapter will deal with the knowledge you need to safely navigate through HPC-land.

Particularly important is the book 'Art of Assembly Language' written by Randall Hyde. You can find the book on-line at http://www.phatcode.net/res/223/files/html/toc.html. Although the book covers far more on the subject of actual assembly programming the x86 Intel architectures, chapter 3 covers the basic computer architecture often encountered in Supercomputers, Beowulf's, workstations and desktops nowadays. As is nicely demonstrated in the well-known Top-500 list of supercomputers (http://www.top500.org), the x86 Intel EM64T architecture takes up more than 70% of the 500 fastest computers of today. Chapter 3 of this book is therefore compulsory and should be read before answering or doing the following exercises.

Questions and Exercises:

1) *Read chapter 3 of the 'Art of Assembly' of Randall Hyde and answer the following questions:*

   - What is a 'Von Nuemann architecture' and what components does it have?
   - What is a 'data-bus' and what size is it on a Pentium processor?
   - Is the x86 architecture 'little-endian' or 'big-endian'?
   - How does the layout of data structures in memory affect the performance of the x86 architecture?
   - Why is there a system-clock and what is a 'wait-state'?
   - What is a 'cache', a 'cache-hit' and a 'cache-miss' and how can 'caching' improve the performance of a computer?
   - What is a 'register'?
   - Where are the 'instructions' of a program stored, what part of the CPU decodes them and what parts actually executes things in the CPU?
   - Why is there a 'prefetch' unit?
   - What is a 'pipe-line' and what purpose does it serve?
   - How can you get a CPU to be 'super-scalar'?
   - What can cause a 'pipe-line stall' or a 'pipe-line hazard'?
   - How come the ordering of the CPU instructions is important for performance?

2) *The following two functions both setup a matrix and compute the trace of the matrix. Which version is faster and why?*

```
double version_1( int N, int M, double Matrix[ N ][ M ] )
{
 double R = 0.0;
 int n, m;

 for( m = 0; m < M; ++m )
  for( n = 0; n < N; ++n )
  {
   Matrix[ n ][ m ] = n * m;
   if( n == m ) R += Matrix[ n ][ m ];
  }

 return( R );
}


double version_2( int N, int M, double Matrix[ N ][ M ] )
{
 double R = 0.0;
 int n, m;

 for( n = 0; n < N; ++n )
  for( m = 0; m < M; ++m )
  {
   Matrix[ n ][ m ] = n * m;
   if( n == m ) R += Matrix[ n ][ m ];
  }

 return( R );
}
```

3) *The speed difference between version 1 and version 2 of the above function is clearly noticeable. Version 2 however, can be even further sped up. Why is the version 3 below again faster than version 2 for general values of N and M? Does the ordering of the loops still matter for the performance or for the algorithm?*

```
double version_3( int N, int M, double Matrix[ N ][ M ] )
{
 double *P = (double *)(Matrix);
 double R = 0.0;
 int n, m;

 for( n = 0; n < N; ++n )
  for( m = 0; m < M; ++m, ++P )
  {
   (*P) = n * m;
   if( n == m ) R += (*P);
  }

 return( R );
}
```

4) *Version 3 of the function, which is already significantly faster than the first version of the function, still contains an 'if'-statement in the inner-loop that can be removed at the expense of adding in a possible cache-miss into the outer-loop:*

```
double version_4( int N, int M, double Matrix[ N ][ M ] )
{
 double *P = (double *)(Matrix);
 double R = 0.0;
 int n, m;

 if( N > M )
 {

  for( n = 0; n < M; ++n )
  {
   for( m = 0; m < M; ++m, ++P )
```

```
    (*P) = n * m;
   R += Matrix[ n ][ n ];        /* if N > M this would segfault and */
  }                              /* therefore we broke the n-loop into */
                                 /* two parts... */
  for( ; n < N; ++n )
   for( m = 0; m < M; ++m, ++P )
    (*P) = n * m;

 }
 else
  for( n = 0; n < N; ++n )
  {
   for( m = 0; m < M; ++m, ++P )
    (*P) = n * m;
   R += Matrix[ n ][ n ];   /* if M > N there is no segfault possible... */
  }

 return( R );
}
```

What line of code introduces the cache-miss? Read about 'branch-prediction' at http://en.wikipedia.org/wiki/Branch_predictor and http://en.wikipedia.org/wiki/Loop_unrolling. Compare the speedup obtained by using version 4 of the function. Why is this version still faster than version 3?

5) *Is it possible to rewrite version 4 of the function such that the cache-misses introduced are removed and at the same time still get rid of the unwanted conditional branch from the inner-loop from version 3? Hint: perhaps a loop can be split into three parts. What version of the function would you use in general?*

6) *The following pointers can be used as a guideline to write fast codes on the x86 architectures. Try to explain why you should pay attention to each of them:*

    1. *if-statements should be avoided in inner-loops and if they cannot, they should be made as predictable as possible.*
    2. *memory accesses should be done linearly with increasing addresses and not randomly or with big strides.*
    3. *multi-dimensional arrays should be accessed with a single index or pointer.*

Profiling SMDEP and optimizing the algorithm

In the previous chapter you learned the basics of the x86 computer architecture and discovered some rules you should take into account when writing fast code. In this chapter we will actually try to optimize the SMDEP algorithm. This will be done in three steps; first we will locate the 'hot-spot' of the code by 'profiling' it, then some tricks will be used to optimize the algorithm, taking the knowledge of the previous chapter into account, finally we will try to use the correct compiler and compiler options to let the compiler generate an even more optimized code.

Usually when dealing with a code that does not show the right level of performance, optimizations are required only at specific places in the performance critical parts of the code. Finding the performance critical parts of a code, however, can be quite challenging if the code is big and complex. It would be very nice if there was a tool that would show exactly which parts of the code take most of the time during a typical run. This can be achieved by recompiling the code with so-called 'profiling support' enabled. For the GNU gcc compiler, this is achieved by adding the option '-pg' into the compiling and loading stages of the SMDEP program. This extra option will automatically include specific code into the executable to automatically 'sample' the program when it runs. Once the code has been completely recompiled with the option included, profiling is now done by just running the code. With the extra profiling support directly embedded into the executable, the run of the program is interrupted regularly. During each such an interrupt, the embedded profiling code determines in what subroutine the program was. Simple counters are used to determine what subroutine got interrupted the most. It is the most performance critical subroutine, in which the program spends most of the time, that will obviously show the highest counters at the end. After having performed a typical run with the profiler enabled executable, a special file 'gmon.out' is being created in which the profiling data is stored. The 'gmon.out' file cannot be read directly, it contains the binary data of the counters used in sampling the run. You can use the 'gprof' command to extract the data from the 'gmon.out' file:

```
# make
gcc -pg -c SMDEP.c
gcc -pg -c Vector.c
gcc -pg -c Particles.c
gcc -pg -c Integrator.c
gcc -pg -c LennardJones.c
gcc -o SMDEP.x -pg  SMDEP.o Vector.o Particles.o Integrator.o LennardJones.o -lm
# ./SMDEP.x
...
... usual output of program
...
# gprof SMDEP.x gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 54.72      4.89     4.89    10001   488.63   874.06  LennardJonesForces
 21.14      6.77     1.89 149504950     0.01     0.02  AbsVector
  7.51      7.45     0.67 149524950     0.00     0.00  DotVectors
  6.17      8.00     0.55 99999900     0.01     0.01  SubVectors
  5.44      8.48     0.49 102010000     0.00     0.00  ScalarVector
  3.59      8.80     0.32 53504950     0.01     0.01  AddVectors
  0.67      8.86     0.06    10000     6.01   883.89  IntegrateATimeStep
  0.45      8.90     0.04      200   200.28   312.25  LennardJonesPotential
  0.45      8.94     0.04                             main
  0.00      8.94     0.00    20000     0.00     0.00  KineticEnergy
  0.00      8.94     0.00      200     0.00     0.45  TotalKineticEnergy
  0.00      8.94     0.00      100     0.00     0.00  PrintParticle
  0.00      8.94     0.00        1     0.00   876.24  SetupParticles
 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
...
... even more stuff
...
```

Immediately clear from the profiling data is that most of the time (roughly 97%) is spent in the 'LennardJonesForces' subroutine. This was also expected beforehand because it is this part of the algorithm that does scale quadratic with the system size. Also clear is that the 'AbsVector' function is also called rather frequently. Probably because 'AbsVector' is called a lot from the 'LennardJonesForces' subroutine:

```
int LennardJonesForces( int N, Particle P[] )
{
 Vector r;
 int i, j;
 double k;

 for( i = 0; i < N; ++i )
  P[ i ].F.X = P[ i ].F.Y = P[ i ].F.Z = 0.0;

 for( i = 0; i < N; ++i )
  for( j = i + 1; j < N; ++j )
  {
   r = SubVectors( P[ j ].r, P[ i ].r );
   k = 4.0 * LENNARDJONES_EPSILON * ( 12.0 * pow( LENNARDJONES_SIGMA / AbsVector( r ), 12.0 ) -
6.0 * pow( LENNARDJONES_SIGMA / AbsVector( r ), 6.0 ) ) / AbsVector( r );
   P[ i ].F = AddVectors( P[ i ].F, ScalarVector( k, r ) );
   P[ j ].F = SubVectors( P[ j ].F, ScalarVector( k, r ) );
  }

 return( 0 );
}
```

Questions and Exercises:

1) *Considering the rules learned in the previous chapter, do you expect any problems with the cache, prefetch queue or the pipe-line in the inner-loop of the 'LennardJonesForces' subroutine?*

2) *Is it possible to remove some of the calls to 'AbsVector' from the 'LennardJonesForces', perhaps by using an extra variable, so that the subroutine performs better?*

3) *Try to profile SMDEP yourself and compare with the following version of 'LennardJonesForces':*

```
int LennardJonesForces( int N, Particle P[] )
{
 Vector r;
 int i, j;
 double k, kk;

 for( i = 0; i < N; ++i )
  P[ i ].F.X = P[ i ].F.Y = P[ i ].F.Z = 0.0;

 for( i = 0; i < N; ++i )
  for( j = i + 1; j < N; ++j )
  {
   r = SubVectors( P[ j ].r, P[ i ].r );
   kk = 1.0 / AbsVector( r );
   k = 4.0 * LENNARDJONES_EPSILON * kk *( 12.0 * pow( LENNARDJONES_SIGMA * kk, 12.0 ) - 6.0 *
pow(LENNARDJONES_SIGMA * kk, 6.0 ) );
   P[ i ].F = AddVectors( P[ i ].F, ScalarVector( k, r ) );
   P[ j ].F = SubVectors( P[ j ].F, ScalarVector( k, r ) );
  }

 return( 0 );
}
```

What trick has been used to optimize the routine? What is the speedup? Did it help significantly? Did the number of calls to 'AbsVector' and the time spent on 'AbsVector' change? What line in the above version is probably the most time consuming and why?

4) *If you run the SMDEP program, with the above optimized version of 'LennardJonesForces', but also with the 'time' command you will notice a big difference in the actual time spent by the program and how much is accounted for by the profiler. Why is that? Is most of the time spent in 'LennardJonesForces' or something else?*

5) *In the 'LennardJonesForces' subroutine, the 6th and12th power of 'LENNARDJONES_SIGMA*kk' is needed. The 12th power can be easily computed from the 6th power by squaring it. The 6th power is easily obtained by squaring the 3rd power of the value. The 'pow' function is actually not needed! This trick is used in the 'LennardJonesForces' version below:*

```
int LennardJonesForces( int N, Particle P[] )
{
 Vector r;
 int i, j;
 double k, kk;

 for( i = 0; i < N; ++i )
  P[ i ].F.X = P[ i ].F.Y = P[ i ].F.Z = 0.0;

 for( i = 0; i < N; ++i )
  for( j = i + 1; j < N; ++j )
  {
   r = SubVectors( P[ j ].r, P[ i ].r );
   k = 1.0 / AbsVector( r );
   kk = LENNARDJONES_SIGMA * k;
   kk = kk * kk * kk;
   kk = kk * kk;
   k = 4.0 * LENNARDJONES_EPSILON * k * ( 12.0 * kk * kk - 6.0 * kk );
   P[ i ].F = AddVectors( P[ i ].F, ScalarVector( k, r ) );
   P[ j ].F = SubVectors( P[ j ].F, ScalarVector( k, r ) );
  }

 return( 0 );
}
```

Profile SMDEP with this new version. Why does this code perform significantly better? Now remove the '-pg' profiling switches, recompile SMDEP and use the 'time' command to run SMDEP again. Did the performance improve? Would you keep the profiling code generated by the '-pg' switch in the program for big simulations once you are done profiling?

In the above exercises you learned that profiling helps you to find the 'hot-spot' of the code. When having found the 'hot-spot' you can perhaps optimize it. Experience shows that if you can device a smarter or faster algorithm to your problem, you will gain the most, whatever other optimization tricks you can do. The first step in HPC is thus always to think about the algorithms you use! If, however, you cannot change the algorithm like was demonstrated in the 'LennardJonesForces' routine, you might still be able to use some of the rules from the previous chapter. In the SMDEP code this 'hand-optimizing' already lead to a measured speedup of about 3.6 for a simulation of a system with 100 particles on an 3.0GHz Intel EM64T Nocona Xeon CPU using GNU gcc 3.4.6. A speedup of 2.3 was found using gcc 4.8.5 on a 2.4 GHz AMD Epyc 7351 (the simulation only took 6.9 seconds).

After having picked the best algorithm and having performed the 'hand-optimization' of the hot-spot, the next step in speeding-up the code is to use correct compilers and flags during compiling. By default the GNU gcc compiler compiles with no optimization flags turned on and a significant extra speedup can be obtained by careful use of the available compiler switches. From experience, form all the possible options gcc offers, the most important ones are: -O2 and -O3, -march=xxx, -funroll-loops and -fprofile-generate and -fprofile-use. More options can be found in the manual of GNU gcc. For the SMDEP code, use of the options '-O3 -march=nocona -fprofile-use' after having performed a test run with the options '-O3 -march=nocona -fprofile-generate', an extra speedup of about 2 was obtained (on the 3.0GHz Intel EM64T Nocona Xeon CPU using GNU gcc 3.4.6) making the code about 8 times faster in total! Tests on the 2.4 GHz AMD Epyc 7351 using ‘-O3 -march=native’ with gcc 4.8.5 showed a simulation time of 2.5 seconds, so another speedup of 2.8 was found, making the total speedup so far about a factor of 6.3.

The use of the correct compiler switches can make a significant difference. Another significant speedup can often be obtained by using vendor specific compilers. On the Intel architectures, the Intel C/C++ and Fortran compilers are known generate faster codes than the GNU compiler suite. When using the Intel C/C++ compiler to compile SMDEP with the compiler flag '-fast', the 100 particle simulation takes only 2.6 seconds instead of the 7.1 seconds previously with the GNU gcc compiler (again on the 3.0 GHz EM64T Nocona Xeon using Intel icc 9.1)! This would make the total speedup of SMDEP of about 20!

If the original non-hand-optimized code is compiled with the Intel C/C++ 9.1 compiler (without any optimization flags whatsoever) on the same Nocona box, the 100 particle simulation takes about 22.3

seconds instead of the 49.2 seconds obtained for the original unoptimized GNU gcc compiled version. When compiling the original non-hand-optimized code with the Intel compiler, but now with using the '-fast' option, the 100 particle simulation takes only about 2.9 seconds in total! The mere use of the Intel compiler already shows a speedup of about 2 but the '-fast' flag really makes the code FAST! When using the profiled and hand-optimized routine with the Intel compiler but without the '-fast' flag or any other optimization flags, the SMDEP run takes about 15.2s on the Nocona box. The astonishing speedup of 20 in total is only obtained when using the Intel compiler with the '-fast' option and using the hand-optimized version of the algorithm. The hand-optimization still accounts for a 35% to 10% speedup when using the Intel compiler.

On the 2.4 GHz AMD Epyc 7351 using the Intel C/C++ 19.0.2.187 compiler, the 100 particle simulation took only 0.687 seconds when the '-fast -xHost' options were used. Compared to the use of gcc 4.8.5 with no options this is thus a speedup of ~10. Comparing to the gcc 4.8.5 '-O3 -march=native', the Intel C/C++ 19.0.187 compiler generates ~3.5 times faster code on the AMD Epyc machine. Disabling the hand optimized code, but using the '-fast -xHost' with the Intel C/C++ 19.0.2.187 compiler on the Epyc, the simulation took 0.786 seconds. Clearly, already using more modern hardware together with decent compilers pays off significantly!

Questions and Exercises:

1) *Why would the Intel compiler be so much more efficient in optimizing than the GNU compiler?*

2) *Read the manual pages of the GNU compiler suite. What (general) options are available for profiling and optimizing codes?*

3) *Although for the Intel compilers the profiling and hand-optimizing did not lead to a similar significant speedup as was obtained with GNUs gcc, but it did not hurt either. Why? In general, would you still profile and hand-optimize your code?*

4) *Run the SMDEP code on your box with your GNU gcc compiler and see what speedups you get when changing compiler flags. Make a table that shows the execution time of SMDEP when the option is used and when no options are used. Try to use the '-fprofile-generate' and '-fprofile-use' compiles together with a test run. Did that help?*

5) *What sort of optimizations can you leave to your compiler in general and why? Explain this with the knowledge you obtained from the previous chapter about the x86 computer architecture.*

Parallelizing SMDEP with OpenMP

In the previous chapter you learned that by profiling, hand-optimizing and carefully compiling the code with the correct compiler and flags, you can get a significant improvement in performance. The SMDEP program became 20 times faster on the Nocona system. The 20 times speedup means that the number of particles of the system to be simulated can be taken about 5 times bigger already. The 5000 particle simulation is now probably within reach on your desktop! The 10000 number of particles simulation will take 4 times the time of the 5000 particle simulation and probably is still a bit to much for just a desktop (although if you run it overnight, you might be able to finish it already). Using more modern server hardware and compilers already pushes your limits further. In this chapter you will learn how to be able to actually do 10000 or more particle simulations by taking the next step in HPC and by parallelizing the SMDEP algorithm. However, before indulging into the technicalities of parallelizing SMDEP, first some general notes about parallel programming.

When it comes to running a program in parallel, one has to realize that it is not possible to completely parallelize a program up to 100%. This is easily understood; OpenMP programs start as a serial program, as will be explained below. Also note that if a program is assumed to be completely parallelized, the individual tasks performed in parallel should be completely independent and can be treated as several smaller independent serial tasks. Parallel programs that do consists of independent tasks are often called 'embarrassingly parallel'. In general, parallel programs cannot be divided into completely independent tasks and therefore cannot be 100% parallel by nature. Even more important, a direct consequence of this fact is expressed through the so-called “Amdahl's Law” (http://en.wikipedia.org/wiki/Amdahls_Law).

Amdahl's law states that for a program that has been parallelized for P%, the maximum speedup by using more CPUs is theoretically limited to 1/(1-P). It also states that if you use N CPUs, you'll obtain a total speedup of 1/[(1-P)+P/N]. At http://en.wikipedia.org/wiki/Amdahls_Law a nice picture is shown for several different values of P and N. Clearly, even when P=0.95, the maximum speedup is only 20 and this is only more or less achieved by using 512 CPUs! Keep also in mind, Amdahl's Law assumes that the parallel part of the program is perfectly parallel and no time is lost in communicating or synchronization between the individual CPUs. In practice this is not the case and deviations from Amdahl's Law are not uncommon often making things worse than already stipulated by the law itself.

In the previous chapter you profiled the SMDEP code and noticed that about 97% of the time is spend in the 'LennardJonesForces' routine when simulating a system of 100 particles. If we were to perfectly parallelize the 'LennardJonesForces' subroutine, the use of 4 CPUs would give us a speedup of about 3.7. To be able to do the 10000 particle simulation, we would thus need 4 CPUs. We would be able to run such simulations just by parallelizing, however, we also know beforehand that the maximum speedup will be 33 and no more. It is not feasible to go yet another order of magnitude further in system size. Probably, systems of about 25000 particles are still possible by using more and more CPUs, however the 100000 particle simulations probably remain out of scope. Only by parallelizing even more of SMDEP and by using faster hardware, such systems can be tackled in practice, but simulations will take days!

When it comes to parallelizing codes, two methods exists. The first method parallelizes by running the same program on different computers and allow the program to communicate between the different instances of it. Parallelizing through this method entails the use of a communication library like MPI (Message Passing Interface, http://en.wikipedia.org/wiki/Message_Passing_Interface). As a programmer you have to specifically program calls to the MPI library. Calls to routines like 'MPI_Send' and 'MPI_Receive' allow the program to send messages to instances of itself and to synchronize those instances. As is nicely demonstrated by the 'hello world' example at http://en.wikipedia.org/wiki/Message_Passing_Interface, the inclusion of MPI calls and to parallelize the program means that the program changes significantly. Calls have to be directly coded into the algorithms to pass messages and often it will be the case that parallelizing with MPI boils down to an 'all or nothing' situation. Meaning, you either completely parallelize the complete code using MPI, or you don't, but it is often impossible to just parallelize a single routine like 'LennardJonesForces' within SMDEP.

Exercises and Questions:

1) *Inspect the 'Hello world' example at http://en.wikipedia.org/wiki/Message_Passing_Interface. Try to install OpenMPI on your linux workstation, use the command 'mpicc' to compile the 'Hello world' example program and try to run it with 'mpirun' using two processes.*

Parallelizing SMDEP with MPI is not impossible, but it does introduce some extra complexity. Also, MPI introduces communication between the instances of the program and depending on the type of communication fabric between the CPUs, time will be spend on communicating. For a standard gigabit ethernet networks (http://en.wikipedia.org/wiki/Ethernet), the fabric does allow for a full gigabit of bandwidth, but at the same time, introduces a latency of more than 10 microseconds (http://en.wikipedia.org/wiki/Latency_%28engineering%29). For HPC codes that need to communicate or synchronize a lot between different instances, the communication overhead and latency will actually become more important than the actual bandwidth of the network connecting the CPUs or nodes of a Supercomputer. Remember, latency cannot be lowered by parallelization and is therefore more of a component in the serial fraction of a code!

For parallelizing SMDEP, another parallelization method is actually preferred to begin with. The method we will use within SMDEP will be far less intrusive to the code, is especially suitable for running SMDEP on multiple cores and CPUs within a single computer (http://en.wikipedia.org/wiki/Shared_memory) and facilitates an SMP (Symmetric Multi Processing http://en.wikipedia.org/wiki/Symmetric_multiprocessing) approach. The AMD Epyc machine SMDEP has been tested on provides 32 real cores in a single server. Nowadays it is not uncommon to have 24 to 64 cores in a single high-end HPC compute node!

The OpenMP (http://en.wikipedia.org/wiki/Openmp) method of parallelizing allows for gradually parallelizing SMDEP routine by routine and our first attempt at it will not alter the original code except for only two lines! Even more, after having changed these two lines and thus having parallelized SMDEP, the resulting code can still be run serially if needed. With MPI codes this has to be specifically taken care of within the algorithm. A good introduction into OpenMP can be found at http://hpc.llnl.gov/tuts/openMP. Further resources on OpenMP can be obtained at http://www.openmp.org. Please check these resources before continuing with the next exercises.

Exercises and Questions:

1) *Read the OpenMP tutorial at http://hpc.llnl.gov/tuts/openMP/. What line needs to be included into 'LennardJonesForces' to let multiple threads share the load of calculating the needed forces.*

2) *Currently the loop in 'LennardJonesForces' loops over pairs of particles and for each pair it computes the contribution of the force to each of the two particles involved. Could there be a problem with this when running the outer-loop with multiple threads?*

3) *Read about 'race-conditions' at http://en.wikipedia.org/wiki/Race_condition. Just adding the '#pragma omp parallel for' line above the outer-loop in 'LennardJonesForces' as it is now, will result into a race condition. Clearly if thread n deals with pair m, the pair between particles i and j, thread n+1 could be dealing with a pair k that also involves either particle j or i. This can happen because the outer-loop updates the force of two particles within a single iteration. How could this be solved so that the hidden race-condition is removed?*

4) *Below is given a version of the 'LennardJonesForces' routine without any race-conditions. Adjust the SMDEP code to include this version, recompile SMDEP using the GNU gcc compiler and include OpenMP support by adding the compiler flag '-fopenmp' at the compiling and loading stages. Set the number of threads to be used by SMDEP to 2 by using the OMP_NUM_THREADS environment variable. Determine the speedup. If you do not have a quad / dual core or a hyper-threading enabled desktop yet, just use a single thread and determine if the OpenMP enabled code is slower than the original serial code. Also inspect the timings in 'scaling_results.txt'.*

```
int LennardJonesForces( int N, Particle P[] )
{
 Vector r;
 int i, j;
 double k, kk;

#pragma omp parallel for private(i,j,r,k,kk)
 for( i = 0; i < N; ++i )
 {
  P[ i ].F.X = P[ i ].F.Y = P[ i ].F.Z = 0.0;
 for( j = 0; j < N; ++j )
   if( i != j ) {
    r = SubVectors(P[j].r,P[i].r); k = 1.0/AbsVector(r); kk = LENNARDJONES_SIGMA*k;
    kk = kk*kk*kk; kk = kk*kk; k = 4.0*LENNARDJONES_EPSILON*k*(12.0*kk*kk - 6.0*kk);
    P[i].F = AddVectors(P[i].F,ScalarVector(k,r));
   }
 }

 return( 0 );
}
```

Running SMDEP on Beowulf clusters

Having parallelized SMDEP with OpenMP in the previous chapter, the 10000 particle simulations came within reach, though at least 5 CPUs will be needed. With the current implementation, SMDEP can easily run on a single quad-core desktop, but it will take some time and probably the desktop cannot be used for anything else at the time SMDEP is running on it. Having a collection of desktops with multiple cores would easily allow for running several simulations at once, but it would be really nice if you had a bunch of dual processor servers with multiple cores, plenty of memory and ample disk-space especially reserved for doing the SMDEP calculations. Like the AMD Epycs. Now, if these servers are also connected with each-other by a local network and a central job scheduler is used, the concept of a Beowulf cluster (http://en.wikipedia.org/wiki/Beowulf_%28computing%29) is obtained.

Beowulf clusters have been built by many and from many different types of hardware. Currently, if you inspect the already mentioned Top500 list, most supercomputers are actually clusters built from commodity hardware. Beefy Intel or AMD servers, equipped with 16 or more cores are nowadays affordable and manageable for scientific groups. Often one uses 1 or 10 gigabit Ethernet or perhaps even Infiniband (http://en.wikipedia.org/wiki/Infiniband) as the interconnect between the nodes, facilitating the possibility to run MPI programs across the nodes. Using such a Beowulf is not complicated once you know how to work with a job scheduler or queue manager (http://en.wikipedia.org/wiki/Job_scheduler). Clearly, you will not be the only user of the cluster and some coordination will need to take place to manage the jobs of users . The queue manager takes care of that and using it is the subject of this chapter.

In this course the SLURM (https://slurm.schedmd.com/documentation.html) batch-manager will be investigated. Although many other batch-managers exists, they all work more or less the same way and offer similar functionalities. Using SLURM is rather easy and is being used more and more often (also on the Dutch supercomputer Cartesius: https://userinfo.surfsara.nl/systems/cartesius/usage/batch-usage). The basic steps involved are writing a script to perform your calculation, submitting the script with 'sbatch', checking the queues with 'squeue' or ‘sinfo’ and perhaps killing jobs with 'scancel':

```
[mark@epyc ~]$ cd SMDEP/
[mark@epyc SMDEP]$ cat Makefile
# Basic makefile for SMDEP code...

# The flags to use in compiling and loading...
CC = gcc
CFLAGS = -O3 -fopenmp -march=native -D__FAST__ -D__OMP__
LDFLAGS = -fopenmp

# All the .o needed in this program...
OBJS = SMDEP.o Vector.o Particles.o Integrator.o LennardJones.o

# Rule to make .o from .c files...
%.o : %.c
	${CC} ${CFLAGS} -c $<

# Rule to make SMDEP.x from all .o files...
SMDEP.x : ${OBJS}
	${CC} -o SMDEP.x ${LDFLAGS} ${OBJS} -lm

# Rule to clean up crap...
clean :
	rm -rf ${OBJS} SMDEP.x gmon.out
[mark@epyc SMDEP]$ make
gcc  -O3 -fopenmp -D__FAST__ -c SMDEP.c
gcc  -O3 -fopenmp -D__FAST__ -c Vector.c
gcc  -O3 -fopenmp -D__FAST__ -c Particles.c
gcc  -O3 -fopenmp -D__FAST__ -c Integrator.c
gcc  -O3 -fopenmp -D__FAST__ -c LennardJones.c
LennardJones.c: In function ‘LennardJonesForces’:
LennardJones.c:53:4: warning: #warning Using profiled code... [-Wcpp]
 #  warning Using profiled code...
    ^
gcc  -o SMDEP.x  SMDEP.o Vector.o Particles.o Integrator.o LennardJones.o -lm
[mark@epyc SMDEP]$ cat SMDEP.job
#!/bin/bash
export OMP_NUM_THREADS=8
${HOME}/SMDEP/SMDEP.x > output
[mark@epyc SMDEP]$ sbatch -t 2:00:00 -N 1 -n 1 -c 8 ./SMDEP.job
Submitted batch job 96622
[mark@epyc SMDEP]$ ls
Integrator.c  Integrator.o    LennardJones.h  Makefile  Particles.c  Particles.o          slurm-
96622.out  SMDEP.job  SMDEP.x   Vector.h
Integrator.h  LennardJones.c  LennardJones.o  output    Particles.h  scaling_results.txt  SMDEP.c
SMDEP.o    Vector.c  Vector.o
[mark@epyc SMDEP]$ cat slurm-96622.out
[mark@epyc SMDEP]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
normal*      up 7-00:00:00      1   resv Z5R1U11
normal*      up 7-00:00:00      1    mix Z5R1U12
normal*      up 7-00:00:00     24  alloc Z5R1U[5-10,13-30]
[mark@epyc SMDEP]$ squeue -u mark
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             96543    normal VASP544_     mark PD       0:00      1 (Priority)
             96542    normal VASP544_     mark PD       0:00      1 (Priority)
             96541    normal VASP544_     mark PD       0:00      1 (Priority)
             96513    normal VASP535_     mark  R    3:13:52      1 Z5R1U10
             96512    normal VASP535_     mark  R    3:26:55      1 Z5R1U18
             96511    normal VASP535_     mark  R    3:29:35      1 Z5R1U10
             96510    normal VASP535_     mark  R    3:29:41      1 Z5R1U10
             96509    normal VASP535_     mark  R    3:33:40      1 Z5R1U10
             96508    normal VASP535_     mark  R    3:46:16      1 Z5R1U5
             96507    normal VASP535_     mark  R    3:47:02      1 Z5R1U18
             96506    normal VASP535_     mark  R    4:29:19      1 Z5R1U13
             96505    normal VASP535_     mark  R    4:41:24      1 Z5R1U17
             96504    normal VASP535_     mark  R    4:48:26      1 Z5R1U26
             96503    normal VASP535_     mark  R    4:57:50      1 Z5R1U26
             96502    normal VASP535_     mark  R    5:11:30      1 Z5R1U6
             96501    normal VASP535_     mark  R    5:18:38      1 Z5R1U13
             96500    normal VASP535_     mark  R    5:36:47      1 Z5R1U13
             96499    normal VASP535_     mark  R    5:44:27      1 Z5R1U6
             96498    normal VASP535_     mark  R    5:51:23      1 Z5R1U6
             96497    normal VASP535_     mark  R    5:51:51      1 Z5R1U6
             96496    normal VASP535_     mark  R    6:25:32      1 Z5R1U12
             96540    normal VASP544_     mark  R       8:15      1 Z5R1U13
             96539    normal VASP544_     mark  R      29:52      1 Z5R1U17
             96538    normal VASP544_     mark  R      52:00      1 Z5R1U17
             96534    normal VASP544_     mark  R    1:33:12      1 Z5R1U5
             96535    normal VASP544_     mark  R    1:33:12      1 Z5R1U5
             96536    normal VASP544_     mark  R    1:33:12      1 Z5R1U18
             96537    normal VASP544_     mark  R    1:33:12      1 Z5R1U18
             96533    normal VASP544_     mark  R    1:54:50      1 Z5R1U26
             96532    normal VASP544_     mark  R    2:59:45      1 Z5R1U26
             96531    normal VASP544_     mark  R    3:09:18      1 Z5R1U17
             96530    normal VASP544_     mark  R    3:10:48      1 Z5R1U5
[mark@epyc SMDEP]$
```

In the above example a simple job script is created which will set the OMP_NUM_THREADS correctly to 8. When submitting the job, the parameters given to 'sbatch' specifies the maximum amount of 'wall time' this job may take (2 hours) and the number of cores, tasks and nodes it will use (8 cores per task, 1 task on 1 node). After successfully finishing the job, the output and errors are collected in the files 'output' and 'slurm-96622.out' respectively.

Exercises and Questions:

1) *Get access to a cluster and setup the SMDEP code yourself for a simulation of 500 particles. Create a special job script for the run and use 'sbatch' to run the script. Estimate the needed resources and specify them to 'sbatch' whilst submitting. If you have no access to a cluster with SLURM, you can use the on-line SLURM documentation to determine the correct 'sbatch' parameters to use.*

2) *Study the (on-line) SLURM manual pages of 'sbatch' and 'srun' and list the most important resource specifiers you can give to 'sbatch' and 'srun'.*

3) *Adapt your job script to include the resource specifiers as '#SBATCH' attributes in the beginning of the script. Resubmit it. If you have no access to a cluster with SLURM, please use to the on-line SLURM documentation to write your new job script.*

4) *Make two job scripts, one that runs a 1000 particle simulation and one that runs a 500 particle simulation. Submit both jobs and make sure everything works. When the 500 particle simulation has finished, stop the 1000 particle simulation by using 'scancel'. If you have no access to a cluster with SLURM, you can skip this exercise.*

5) *Run 'sinfo' to see what partitions are defined on the cluster. Read the manual page of 'squeue' and try to run 'squeue' and 'scontrol show job <id>' on a running job. If you have no access to a cluster with SLURM, you can use the on-line SLURM documentation and examples to answer this question.*