INFR11167

Performance Programming

Coursework Report

Author:
B066600

April 7, 2019

Contents

1 Introduction

The aim of this coursework was to perform performance optimisation for a relatively simple application code carrying out a molecular dynamics (MD) simulation using exclusively the back-end compute nodes of Cirrus. Due to previous experience of the student with the C programming language, the performance activities were carried out using the C version of the code.

2 Contents of scripts for MD simulation

The programme reads an initial state from a file (i.e. *input.dat*) and then performs 5 blocks of 100 timesteps worth of simulation writing an output file after each block. The code automatically reports timing information for each block of 100 timesteps as well as the total time for running the entire simulation.

The *MD.c* file contains the computations carried out in order to simulate the collisions. The *util.c* file contains the declaration of some helper functions that are used in the *MD.c* file. The *coord.h* defines some pointers to some dynamically allocated arrays contained in the *control.c* files. The main variables and arrays used in the code and their associated physical concept are listed in Table 1. The *control.c* file controls the update of the MD simulation and is responsible for reading the input file and writing out the output to disk.

Variable or Array Name	**Physical concept**	**Size**
`Nbody`	number of particles	4096
`Npair`	number of particle pairs	`(Nbody*(Nbody-1))/2 = 8,386,560`
`Ndim`	number of dimensions	3
`pos`	position of particles	`[Ndim][Nbody]`
`r`	distance of particle from central mass	`[Nbody]`
`velo`	velocity of particles	`[Ndim][Nbody]`
`f`	forces acting on each particle	`[Ndim][Nbody]`
`vis`	viscosity coefficient for each particle	`[Nbody]`
`mass`	mass of each particle	`[Nbody]`
`delta_pos`	separation vector for each particle pair	`[Ndim][Npair]`
`delta_r`	separation vector for each particle pair	`[Nbody][Nbody]`

Table 1: List of variables and arrays used by the programme, with the associated physical concept and size in terms of integer value for variables or number of elements in each dimension for arrays.

3 Methodology

The activities performed during the coursework can be divided into profiling and performance experiments. For profiling code, Gprof, Cachegrind and Intel's loop profiler tools were used. Gprof (version 2.25.1-22.base.el7 used in the coursework) is a commonly used Unix tool that measures the frequency and duration of function calls [1]. Cachegrind (version 3.11.0 used in the coursework) is part of the Valgrind distribution and it simulates how the program interacts with a machine's cache hierarchy [5]. In particular, Cachegrind simulates the first-level and last-level caches. All the performance experiments were carried out using the Intel's C/C++ Compiler (version 17.0.2). This compiler version allows to compile code with flags (i.e. -profile-loops=all, -profile-loops-report=2) that insert instrumentation calls before and after instrumentable loops.

Although it is generally not recommended to use compiler-based profilers (such as Gprof) in combination with compiler optimisation [4], for practical reasons all Gprof experiments were conducted in combination with -O3 compiler flag on 50 MD simulation timesteps. On the other hand, Cachegrind profiling experiments were performed with different optimisation flags, but always over 10 MD simulation timesteps. This was again due to practical reasons, as some of the low level optimisation flags led to very long runtimes when Cachegrind profiling was enabled during execution.

Individual loops were profiled by compiling code using the loop profiling flags mentioned above and the -O1 flag, and by subsequently running the simulation on 500 timesteps. The first two flags enable the generation of a report about loop execution time in the application. The reason the -O1 flag was used is because the loop profiling flags are automatically disabled when the -O0 flag is present.

4 First program compilation and execution

In order to ensure that the provided program was working as expected and obtain an idea of the initial performance of the program, the original *Makefile* (shown in Listing 1) provided as part of the coursework was used to compile the program.

Although it is not written in the *Makefile* itself, the "module load intel-compilers-17" command must be run before using the *Makefile* for compilation, otherwise an error will be given. The "CFLAGS" line contains the user defined list of flags to be used as part of code compilation prior to running the executable for the simulation and these can be modified as part of performance experiments. The runtime for the

```
SRC=MD.c control.c util.c
OBJ=$(SRC:.c=.o)
CC=icc
CFLAGS= -g -O0 -check=uninit -check-pointers:rw -no-vec


all: MD

MD: $(OBJ)
  $(CC) $(CFLAGS)  -o $@  $(OBJ) -lm


output.dat: MD input.dat
  ./MD


clean:
  rm -f MD $(OBJ)

$(OBJ) : coord.h Makefile
```

Listing 1: Original Makefile provided as part of the coursework.

program after being compiled with the flags shown in the *Makefile* in Listing 1 was 5172.90 seconds (≈1.44 h).

5 Understanding of the performance issues

In order to understand the pre-existing issues in the code from a performance point of view, code profiling was carried out. Code profiling allows to understand the behaviour of the program using information gained as the program executes. It is worth noting that careful consideration needs to be given to how the program is run during profiling, as this may affect the information that shows up in the profile data. In particular, all the features currently implemented by the programme between input and output need to be used. The program is currently divided into 5 `Nsave` (i.e. number of programme output file saves) and each `Nsave` is associated with 100 `Nstep` (i.e. number of timesteps), for a total of 500 timesteps. In order to reduce the number of timesteps, `Nstep` can be set to any value between 1 and 100.

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  s/call   s/call  name
47.16     110.24   110.24 25165824000   0.00     0.00  force
39.30     202.10    91.87        5    18.37    45.57  evolve
11.01     227.84    25.74     3000     0.01     0.01  add_norm
 2.54     233.78     5.94                   __intel_avx_rep_memset
 0.00     233.79     0.01     1500     0.00     0.00  visc_force
 0.00     233.80     0.01     1500     0.00     0.00  wind_force
```

Listing 2: Gprof interpreted profile data generated from 50 steps of MD simulation when original code was compiled with -g -O3 -pg. Functions are sorted first by decreasing runtime spent in them, then by decreasing number of calls, then alphabetically by name. First column indicates the percentage of total runtime of the program used by a given function. The fourth column indicates the number of times each function was invoked. The fourth row presents data for Intel's memset function used to fill blocks of memory with values.

5.1 Gprof

The program code was compiled with the following flags: -g -O3 -pg. In the *control.c* file, `Nstep` was specified to be 10, therefore giving a total of 50 timesteps. The program was then executed. The resulting profile data file (i.e. *gmon.out*) was then interpreted using the Gprof tool. The results of code profiling (see Listing 2) clearly show that the function `force` is responsible for the majority (47.16%) of the current runtime. In particular, ≈25 million calls to this function were made during execution.

5.2 Cachegrind

All the compiler flags present in the original *Makefile* were removed apart from -g and -O0. The original code was run with `Nstep` value set to 2, giving a total of 10 steps. The result of profiling the code during execution using Cachegrind (shown in Listing 3) indicate that the current number of level 1 instruction cache misses exhibited by the programme is insignificant (I1 and LLi miss rates are 0%), suggesting that the entire program fits inside the level 1 instruction cache. The programme however exhibits some degree of level 1 and last level data cache misses (0.5 and 0.3%, respectively). Ideally, these miss rates should be decreased as a result of performance optimisations.

```
I    refs:        118,169,434,058
I1   misses:      2,161
LLi misses:      2,122
I1   miss rate:  0.00%
LLi miss rate:  0.00%

D    refs:     49,670,530,997 (40,961,622,841 rd   + 8,708,908,156 wr)
D1   misses:      255,622,007 (213,564,177 rd      +    42,057,830 wr)
LLd misses:      157,395,345 (115,403,899 rd      +    41,991,446 wr)
D1   miss rate:  0.5%         (0.5%               +          0.5%   )
LLd miss rate:  0.3%         (0.3%               +          0.5%   )

LL refs:         255,624,168 (213,566,338 rd      +    42,057,830 wr)
LL misses:       157,397,467 (115,406,021 rd      +    41,991,446 wr)
LL miss rate:    0.1%         (0.1%               +          0.5%   )
```

Listing 3: Cachegrind interpreted profile data generated from 10 steps of MD simulation when original code was compiled with -g -O0. Information about the number of references, misses and miss rates for the level 1 instruction cache is displayed in the first 5 lines. In the following 5 lines information about data caches is displayed. In the following 5 lines after that information about last level cache is displayed.

5.3 Loop profiling

Loop profiling was performed on the original code and the results are shown in Table 2. If the loop profiling report entries are filtered such that loops that have a negligible contribution (i.e. % self time value smaller than 1.00) to the total application time are disregarded, it can be seen that the application exhibits several loops with very large loop entries and/or iteration counts. In particular, the innermost loop of the triple loop (*MD.c*:59) that calculates the pairwise separation of particles (i.e. updating the values of `delta_pos` array) has 4,193,280,000 loop entries. Moreover, each of the 3 individual loops (*MD.c*:68, *util.c*:20 inlined at *MD.c*:71 and *MD.c*:74) that perform the combined function of calculating the norm of the separation vector has a loop iteration count of 8,386,560. Finally, the loop used in order to add pairwise forces (*MD.c*:82) has 2,047,500 loop entries. It can be expected that reducing the total number of loop entries and/or iteration counts should lead to performance improvements.

6 Overview of proposed optimisations

As part of the performance programming work, different types of optimisations were introduced in a sequential order to the code. It is worth noting that regardless of

Loop	# Loop entries	Avg. iterations count	Max. iterations count
MD.c:83	2,047,500	2,048	4,095
util.c:20	3,000	4,195,328	8,386,560
MD.c:74	500	8,386,560	8,386,560
MD.c:59	4,193,280,000	3	3
MD.c:68	500	8,386,560	8,386,560
Totals	4,195,331,500	20,970,499	25,163,778

Table 2: Results of loops profiling on 500 timesteps when original code was compiled with -profile-loops=all, -profile-loops-report=2 and -O1 flags. Number of loop entries indicates the number of times the loop was entered. The average (Avg.) iterations count is the average recorded number of iterations for a single invocation of the loop. The maximum (Max.) iterations count is the largest recorded number of iterations for a single invocation of the loop.

the type of performance optimisation that is introduced in the code, the program correctness must remain unaffected. Hence, in order to check for correctness, a specifically designed program called *diff-output* (provided before the start of the coursework) was used. After being compiled, this program can be run in order to compare the output file generated from a given experiment against the one generated as part of the original implementation with the untouched *Makefile*. Another quality control measure is checking that the number of recorded collisions as a result of an optimisation is within an acceptable range (i.e. 68540) to the one generated by the original implementation. All the optimisations described below successfully passed these correctness tests.

6.1 Initial implementation and compiler flags optimisation part 1

As mentioned above, the original implementation recorded a time of 5172.90 seconds. The first step of the optimisation work was therefore to examine the compiler flags present in the *Makefile* one by one:

- **-g**: this option tells the compiler to generate symbolic debugging information in the object file.
- **-O0**: this means optimisation level 0, which disables all possible optimisations.
- **-check=uninit**: this option enables checking for uninitialized variables.
- **-check-pointers:rw**: this option enables checking of all indirect accesses through pointers and accesses to arrays.
- **-no-vec**: this option disables vectorisation.

Single flag removed	Runtime (sec.)
no flags removed	5172.90
-no-vec	5171.08
-g	5170.75
-check=uninit	5138.69
-check-pointers:rw	1023.18

Table 3: Recorded runtime on 500 timesteps for experiments where original code was compiled with each compiler flag from original Makefile removed in turn, while keeping all the remaining flags.

After consideration, it appears that all of the flags that are currently present in the *Makefile* are not beneficial to performance. In order to confirm this hypothesis, each flag was removed one at a time (while keeping all of the remaining flags) before compiling the code. The runtime of each version of the program was then recorded (see Table 3).

The clear conclusion is that −check−pointers:rw is the most detrimental flag to performance of the current version of the code. This may be related to the fact that the program heavily relies on pointers as the majority of the function arguments are passed by pointers. As pointers could point anywhere in memory, leading to the potential problem of pointer aliasing or violating arrays/buffer overflows, when this option is enabled the compiler will perform bounds checking at the expense of performance [3]. In fact, enabling the flag causes an increase in runtime of 4149.72 seconds (80.22% of total runtime). While it could be understood that checking pointers could be a useful debugging aid, since the current version of the code is not giving any segmentation faults or errors even when the flag is enabled, this compiler flag can be removed.

Removing the $-check{=}uninit$ flag appears to be a sensible idea as well, as this can be easily replaced by manually ensuring that all the variables are initialized at the top of the *MD.c* file. Moreover, when this option is enabled, a 34.22 seconds increase in runtime is registered, which further supports the argument of removing it.

The *-g* flag generates debugging information that is helpful for profiling code (e.g. with Gprof or Cachegrind), however, it can be removed as it does not improve performance.

Finally, the *-no-vec* flag should also be removed as vectorisation is a very common way of increasing performance by leveraging the fact that most modern processors (including the ones found on Cirrus with AVX2 instructions) can perform single instruction, multiple data (SIMD) operations.

Optimisation flag	Runtime (sec.)
-O0	1013.04
-O1	670.54
-O2	137.79
-O3	169.82
-Ofast	169.58

Table 4: Recorded runtime on 500 timesteps for experiments where original code was compiled with a single optimisation flags, where an optimisation flag is either -O0, -O1,-O2, -O3 or -Ofast.

6.2 Compiler flags optimisation part 2

After having removed all the unnecessary compiler flags, the next step was to tweak the compiler optimisation flag. Table 4 shows the recorded runtime for all timesteps with different optimisation flags (i.e. -O0, -O1, -O2, -O3 and Ofast). When the -O1 flag is enabled, the compiler tries to reduce code size and execution time, without performing any optimisations that employ a lot of compilation time. It is worth noting that -O2 flag is the generally recommended optimisation flag to be used and it enables compiler vectorisation as well as some basic loop optimisations, inlining and single file interprocedural optimisation. The -O3 flag performs all of the optimisations that are already done by the -O2 flag and it additionally enables more aggressive loop transformations such as fusion, unroll-and-jam and collapsing if statements. The -O3 option is also recommended for applications that contain loops that make heavy use of floating-point calculations and process large data sets [6]. Finally, the -Ofast flag enables the optimisations already carried out by the -O3 flag and also enables optimisations that are not valid for all standard-compliant programs.

The lowest runtime was recorded with the -O2 flag (Table 4). Although the runtime associated with using the -O2 flag is smaller than -O3, using -O3 is still expected to be more advantageous for later optimisation steps, as it is a more aggressive optimisation than -O2 while still being standard-compliant. Also, as the code includes a large amount of floating point calculations and processes large datasets (e.g. array `delta_pos` is of size [3][8,386,560]), it may also benefit more from using the -O3 flag rather than -O2.

6.3 Compiler flag optimisation part 3

As the -O3 flag is expected to perform some optimisations that could otherwise potentially be done to the code by hand (e.g. loop fusions, loop interchange, inlining),

additional compiler flags were reviewed (shown below). Different combinations of these were then used for compiling the code and the respective program runtime was recorded. The combination of -no-prec-div, -xCORE-AVX2 and -ipo registered the highest runtime (see Table 5) and therefore was used in all successive optimisations.

- **-no-prec-sqrt**: this flag decreases precision of square root implementations while having a slight impact on speed.
- **-no-prec-div**: this flag decreases precision of floating point divides while having a slight impact on speed.
- **-xCORE-AVX2**: this flag optimises execution for Intel Advanced Vector Extensions 2 (Intel AVX2) processors.
- **-ipo**: this flag enables multi-file interprocedural optimisation.

Compiler flag(s)	**Runtime (sec.)**
-no-prec-sqrt	170.08
-no-prec-div	169.65
-xCORE-AVX2	143.75
-no-prec-div -xCORE-AVX2 -ipo	81.36

Table 5: Recorded runtime for experiments where original code was compiled with one or combination of the following flags: -no-prec-sqrt, -no-prec-div, -xCORE-AVX2, -ipo.

6.4 Memory structures optimisation

Currently, the program performs dynamic memory allocation for several arrays by making use of `calloc` function calls (lines 1-10 in Listing 4). In particular, a loop over `Ndim` is then used in order to define the number of elements for each dimension of the multidimensional arrays (lines 11-16 in listing 4). Finally, pointers to all of the dynamically allocated arrays are declared in the *coord.h* file.

It is generally known that compilers have a better chance to optimise when dimension sizes of arrays are known at compile time [2]. Hence, the next step was to turn the dynamically allocated arrays into static arrays (see Listing 5) and remove the pointers declarations. Introducing these changes on top of the best set of compiler flags identified in section 6.3 allowed to achieve a runtime of 75.42.

```
//control.c
r = calloc(Nbody,sizeof(double));
delta_r = calloc(Nbody*Nbody,sizeof(double));
mass = calloc(Nbody,sizeof(double));
radius = calloc(Nbody,sizeof(double));
vis = calloc(Nbody,sizeof(double));
f[0] = calloc(Ndim*Nbody,sizeof(double));
pos[0] = calloc(Ndim*Nbody,sizeof(double));
velo[0] = calloc(Ndim*Nbody,sizeof(double));
delta_pos[0] = calloc(Ndim*Nbody*Nbody,sizeof(double));
for(i=1;i<Ndim;i++){
f[i] = f[0] + i * Nbody;
pos[i] = pos[0] + i * Nbody;
velo[i] = velo[0] + i * Nbody;
delta_pos[i] = delta_pos[0] + i*Nbody*Nbody;
}

//coord.h
DEF double *pos[Ndim], *velo[Ndim];
DEF double *f[Ndim], *vis, *mass, *radius;
DEF double *delta_pos[3];
DEF double *r;
DEF double *delta_r;
DEF double wind[Ndim];
DEF int collisions;
```

Listing 4: Before memory structures optimisation.

```
//coord.h
DEF double pos[Ndim][Nbody];
DEF double velo[Ndim][Nbody];
DEF double f[Ndim][Nbody];
DEF double vis[Nbody];
DEF double mass[Nbody];
DEF double radius[Nbody];
DEF double delta_pos[Ndim][Npair];
DEF double r[Nbody];
DEF double delta_r[Npair];
DEF double wind[Ndim];
DEF int collisions;
```

Listing 5: After memory structures optimisation.

6.5 Data arrays alignment

SIMD and vector store operations usually work best when writing well-aligned contiguous blocks of data [3]. Moreover, although the -O3 flag allows vectorisation, it is still part of the programmer's job to make sure that the data arrays are aligned. This was tackled by adding the keyword `__attribute__` followed by the `aligned` specification after each static array declaration, which causes the compiler to allocate the array on a 64-byte boundary (see Listings 6 and 7). A 64-byte boundary was chosen as that is the cache line size of the processors used on back-end nodes on Cirrus.

```
1  //coord.h
2  DEF double pos[Ndim][Nbody];
3  DEF double velo[Ndim][Nbody];
4  DEF double f[Ndim][Nbody];
5  DEF double vis[Nbody];
6  DEF double mass[Nbody];
7  DEF double radius[Nbody];
8  DEF double delta_pos[Ndim][
       Npair];
9  DEF double r[Nbody];
10 DEF double delta_r[Npair];
11 DEF double wind[Ndim];
12 DEF int collisions;
```

Listing 6: Before data arrays alignment optimisation.

```
1  //coord.h
2  DEF double pos[Ndim][Nbody]
       __attribute__((aligned(64)))
       ;
3  DEF double velo[Ndim][Nbody]
       __attribute__((aligned(64)))
       ;
4  DEF double f[Ndim][Nbody]
       __attribute__((aligned(64)))
       ;
5  DEF double vis[Nbody]
       __attribute__((aligned(64)))
       ;
6  DEF double mass[Nbody]
       __attribute__((aligned(64)))
       ;
7  DEF double radius[Nbody]
       __attribute__((aligned(64)))
       ;
8  DEF double delta_pos[Ndim][
       Npair] __attribute__((
       aligned(64)));
9  DEF double r[Nbody]
       __attribute__((aligned(64)))
       ;
10 DEF double delta_r[Npair]
       __attribute__((aligned(64)))
       ;
11 DEF double wind[Ndim]
       __attribute__((aligned(64)))
       ;
12 DEF int collisions;
```

Listing 7: After data arrays alignment optimisation.

6.6 Hand optimisation part 1

After inspecting the compiler report (generated by compiling the optimised code with -qopt-report 5) it was apparent that several loops could not be automatically vectorised by the compiler due to the presence of some dependencies. The next step was therefore to remove such dependencies by hand. One of these could be the presence of `k` (see lines 4-10 in Listing 8), which is used as index for accessing the

second dimension of `delta_pos` and is updated after every `j`-th iteration.

Moreover, in the loop between lines 17 and 19 in Listing 8, `delta_pos` is updated using the `add_norm` function. Although this may not be apparent from reading the code, the iteration space of this update is equivalent to the iteration space of the 3 nested loops between lines 4-10, in which `delta_pos` is updated for the first time. As a matter of fact, if the `add_norm` function was to be inlined in line 18 in Listing 8, it would be replaced by a loop traversing the `delta_r` and `delta_pos` arrays for `Npair` number of times. In turn, `Npair` is defined in the *coord.h* file as being `(Nbody*(Nbody-1))/2,`which can be understood as two nested loops between 0 and `Nbody-1,`and between 1 and `Nbody-1.`

The proposed hand optimisation to the code shown in Listing 8 therefore involves:

1. removal of the redundant `delta_pos` array from the *coord.h* file;
2. re-definition of `delta_r` as a 2D array of size `[Nbody][Nbody]`;
3. removing the update of `k` in loop used to calculate the pairwise separation of particles (lines 4-10 in Listing 8);
4. inlining the `add_norm` function call (line 12 in Listing 8);
5. restructuring of the loop in which the `add_norm` function is called to three nested loops iterating between 0 and `Nbody-1,`1 and `Nbody-1,`and 0 and `Ndim-1`;
6. since the loops in lines 4-10 and 13-15 in Listing 8 are iterating through different arrays, the loops in 4-10 and the rewritten version of loop in lines 17-19 containing inlined function `add_norm` can be fused;
7. re-writing the update of `delta_r` inside the fused loops from steps 6 in terms of difference between `pos` arrays.

As part of this optimisation, `delta_pos` in the nested loops used for adding the pairwise forces must be expressed in terms of the difference of `pos` arrays (see lines 13-17 in Listing 10 and lines 11-15 in Listing 11). The resulting runtime after introducing the optimisation shown in Listings 9 and 11 was 55.42.

```
//MD.c
/* calculate pairwise
   separation of particles */
k = 0;
for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
        for(l=0;l<Ndim;l++){
        delta_pos[l][k] = pos[
    l][i] - pos[l][j];
                }
            k = k + 1;
          }}

/* calculate norm of
   separation vector */
for(k=0;k<Npair;k++){
    delta_r[k] = 0.0;
}

for(i=0;i<Ndim;i++){
  add_norm(Npair,delta_r,
    delta_pos[i]);
}

for(k=0;k<Npair;k++){
    delta_r[k] = sqrt(delta_r[
    k]);
}
//util.c
void add_norm(int N,double *r,
    double *delta)
{
  int k;
        for(k=0;k<N;k++){
          r[k] += (delta[k] *
    delta[k]);
        }
}
//coord.h
DEF double delta_r[Npair]
    __attribute__((aligned(64))
    );
DEF double delta_pos[Ndim][
    Npair] __attribute__((
    aligned(64)));
```

Listing 8: Before hand optimisation part 1.

```
//MD.c
/* calculate norm of
   separation vector */
for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
    delta_r[i][j] = 0.0;
}}

/* calculate pairwise
   separation of particles AND
    update delta_r at the same
    time*/
for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
        for(l=0;l<Ndim;l++){
            delta_r[i][j] += (
    pos[l][i]-pos[l][j]) * (pos
    [l][i]-pos[l][j]);
}}}

for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
        delta_r[i][j] = sqrt(
    delta_r[i][j]);
        }}

//coord.h
DEF double delta_r[Nbody][
    Nbody] __attribute__((
    aligned(64)));
```

Listing 9: After hand optimisation part 1.

```
//MD.c
//add pairwise forces.
k = 0;
for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
        Size = radius[i] +
    radius[j];
        collided=0;
        for(l=0;l<Ndim;l++){
/*  flip force if close in */
            if( delta_r[k] >=
    Size ){
                f[l][i] = f[l
    ][i] - force(G*mass[i]*mass
    [j],delta_pos[l][k],delta_r
    [k]);
                f[l][j] = f[l
    ][j] + force(G*mass[i]*mass
    [j],delta_pos[l][k],delta_r
    [k]);
            }else{
                f[l][i] = f[l
    ][i] + force(G*mass[i]*mass
    [j],delta_pos[l][k],delta_r
    [k]);
                f[l][j] = f[l
    ][j] - force(G*mass[i]*mass
    [j],delta_pos[l][k],delta_r
    [k]);
    collided=1;
            }
        }
      if( collided == 1 ){
        collisions++;
      }
        k = k + 1;
      }
    }
```

Listing 10: Before hand optimisation part 1.

```
//MD.c
//add pairwise forces.
 k = 0;
 for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
        Size = radius[i] +
    radius[j];
        collided=0;
        for(l=0;l<Ndim;l++){
/*  flip force if close in */
            if( delta_r[i][j
    ] >= Size ){
                f[l][i] = f[l
    ][i] - force(G*mass[i]*mass
    [j],(pos[l][i] - pos[l][j])
    ,delta_r[i][j] );
                f[l][j] = f[l
    ][j] + force(G*mass[i]*mass
    [j],(pos[l][i] - pos[l][j])
    ,delta_r[i][j] );
            }else{
                f[l][i] = f[l
    ][i] + force(G*mass[i]*mass
    [j],(pos[l][i] - pos[l][j])
    ,delta_r[i][j] );
                f[l][j] = f[l
    ][j] - force(G*mass[i]*mass
    [j],(pos[l][i] - pos[l][j])
    ,delta_r[i][j] );
    collided=1;
            }
        }
      if( collided == 1 ){
        collisions++;
      }
        k = k + 1;
      }
    }
```

Listing 11: After hand optimisation part 1.

6.7 Hand optimisation part 2

The next step was to perform additional hand optimisations to the loop used for adding pairwise forces (shown in Listing 12):

1. removing the update of `k` (line 22 in Listing 12) altogether as it is no longer used as looping index;
2. taking out the conditional statements that are present inside the innermost loop over `Ndim` (lines 10 and 13 in Listing 12), such that each branch will now contain a loop over `Ndim`;
3. pre-calculating values for masses and forces, and storing these as temporary variables in order to reduce the number of `force` function calls (lines 7, 11-13 and 16-18 in Listing 13) and ultimately reduce the overall number of computations made in the innermost loops.

The resulting runtime after introducing the optimisation shown in Listing 13 was 45.66.

```
//MD.c
// add pairwise forces.
 k = 0;
 for(i=0;i<Nbody;i++){
     for(j=i+1;j<Nbody;j++){
          Size = radius[i] +
    radius[j];
          collided=0;
          for(l=0;l<Ndim;l++){
/*   flip force if close in */
                  if( delta_r[i][j
    ] >= Size ){
                    f[l][i] = f[l
    ][i] - force(G*mass[i]*mass
    [j],(pos[l][i] - pos[l][j])
    ,delta_r[i][j] );
                    f[l][j] = f[l
    ][j] + force(G*mass[i]*mass
    [j],(pos[l][i] - pos[l][j])
    ,delta_r[i][j] );
                  }else{
                    f[l][i] = f[l
    ][i] + force(G*mass[i]*mass
    [j],(pos[l][i] - pos[l][j])
    ,delta_r[i][j] );
                    f[l][j] = f[l
    ][j] - force(G*mass[i]*mass
    [j],(pos[l][i] - pos[l][j])
    ,delta_r[i][j] );
     collided=1;
     }
   }
        if( collided == 1 ){
          collisions++;
          }
              k = k + 1;
          }
     }
```

Listing 12: Before hand optimisation part 2.

```
//MD.c
//add pairwise forces.
for(i=0;i<Nbody;i++){
     for(j=i+1;j<Nbody;j++){
          Size = radius[i] +
    radius[j];
          collided=0;
          precalculated_masses =
     G*mass[i]*mass[j];
/*   flip force if close in */
               if( delta_r[i][j]
    >= Size ){
                    for(l=0;l<Ndim
    ;l++){

    precalculated_force= force(
    precalculated_masses,(pos[l
    ][i] - pos[l][j]),delta_r[i
    ][j]);
                      f[l][i] =
    f[l][i] -
    precalculated_force;
                      f[l][j] =
    f[l][j] +
    precalculated_force;
                    }} else{
                      for(l=0;l<
    Ndim;l++){

    precalculated_force= force(
    precalculated_masses,(pos[l
    ][i] - pos[l][j]),delta_r[i
    ][j]);
                        f[l][i
    ] = f[l][i] +
    precalculated_force;
                        f[l][j
    ] = f[l][j] -
    precalculated_force;

    collided=1; }}
     if( collided == 1 ){
               collisions++;
               }
          }
     }
```

Listing 13: After hand optimisation part 2.

6.8 Hand optimisation part 3

The last hand optimisation involved fusing all the loops responsible for calculating the norm of separation vectors and the pairwise separation of particles (see Listing 14) into 3 nested loops, in order to reduce redundant loop entries. The resulting runtime after introducing the optimisation shown in Listings 15 was 35.24. It appears that this modification is beneficial to code quality as it removes repeated loops and displays all the operations into a single place.

```
//MD.c
/* calculate norm of
   separation vector */
for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
    delta_r[i][j] = 0.0;
}}

/* calculate pairwise
   separation of particles AND
    update delta_r at the same
    time*/
for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
        for(l=0;l<Ndim;l++){
            delta_r[i][j] += (
   pos[l][i]-pos[l][j]) * (pos
   [l][i]-pos[l][j]);
}}}

for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
        delta_r[i][j] = sqrt(
   delta_r[i][j]);
        }}
```

Listing 14: Before hand optimisation part 3.

```
//MD.c
/* calculate pairwise
   separation of particles*/
/* calculate norm of
   separation vector */
for(i=0;i<Nbody;i++){
    for(j=i+1;j<Nbody;j++){
        delta_r[i][j] = 0.0;
        for(l=0;l<Ndim;l++){
            delta_r[i][j] += (
   pos[l][i]-pos[l][j]) * (pos
   [l][i]-pos[l][j]);
                    }
                    delta_r[i
   ][j] = sqrt(delta_r[i][j]);
            }
        }
```

Listing 15: After hand optimisation part 3.

7 Discussion of proposed optimisations

7.1 Analysis of performance data

Compiler flags optimisations had the largest measurable impact on performance, resulting in runtime decreases in the region between 42 and 86% compared to each respective version of the code without the specific compiler flag optimisation. Although the memory structures optimisation had a relatively small effect on performance (7.30% decrease relative to previous optimisation), it can be argued that this optimisation was necessary in order to allow the compiler to actually increase performance through vectorisation as a result of the subsequent hand optimisations. Although it is a recommended measure for optimal performance, aligning the data arrays using the keyword `__attribute__` had a negligible impact on performance (0.39% decrease relative to previous optimisation). Since virtually all the data array elements of the programme that are to be fit into cache are of the same type (i.e. `double`), the program already exhibits minimal memory wastage, resulting in no performance improvement using alignment/padding techniques.

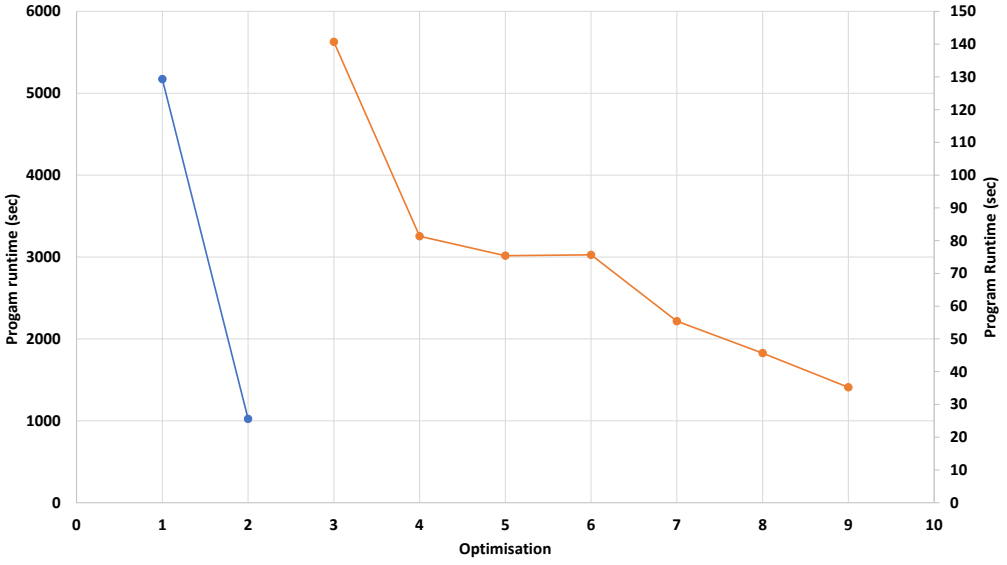

Figure 1: Plot of program runtime recorded for the 8 proposed optimisations that were successively introduced to the initial implementation (marked as optimisation 1 in the plot). The left (primary) axis shows runtime for optimisations 1 and 2 (shown in blue) in a scale from 1000 to 6000 in time increments of 1000 seconds. The right (secondary) axis shows runtime for optimisations 3-9 (shown in orange) in a scale from 0 to 150 in time increments of 10 seconds.

Opt. ID	Optimisation	Runtime	% decrease initial	% decrease opt. 3	% decrease previous
1	Initial implementation	5172.90	0	N/A	N/A
2	Compiler flags optimisation part 1	1023.18	80.22	N/A	80.22
3	Compiler flags optimisation part 2	140.71	97.28	0	86.25
4	Compiler flags optimisation part 3	81.36	98.43	42.18	42.18
5	Memory structures optimisation	75.42	98.54	46.40	7.30
6	Data arrays alignment	75.12	98.55	46.61	0.39
7	Hand optimisation part 1	55.42	98.93	60.61	26.23
8	Hand optimisation part 2	45.66	99.12	67.55	17.60
9	Hand optimisation part 3	35.24	99.32	74.96	22.83

Table 6: Program runtime in seconds recorded for the 8 proposed optimisations, along with percentage decrease relative to initial implementation (fourth column), proposed optimisation 3 (fifth column) and previous optimisation with respect to a given optimisation (sixth column).

7.2 Gprof on final optimised code

The results of code profiling with Gprof (shown in Listing 16) clearly show that although the percentage of runtime attributed to `force` function calls has not changed significantly (i.e. 47.16 % in original code vs. 44.49 in final optimised version), the total number of function calls has decreased from ≈25 billion to ≈1 billion calls. It is also worth noting that, consistent with inlining the `add_norm` function inside the loop used for calculating pairwise separation of particles, the percentage of runtime and number of calls made by this function has largely decreased from 11.01 to 1.98%, and from 3000 calls to 150, respectively.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 53.45      5.13     5.13        5     1.03     1.92  evolve
 44.49      9.40     4.27 1258598400     0.00     0.00  force
  1.98      9.59     0.19      150     0.00     0.00  add_norm
  0.10      9.60     0.01                             main
  0.00      9.60     0.00      150     0.00     0.00  visc_force
  0.00      9.60     0.00      150     0.00     0.00  wind_force
```

Listing 16: Gprof interpreted profile data generated from 50 steps of MD simulation when final optimised code was compiled with -g -03 -pg.

7.3 Cachegrind on final optimised code

The Cachegrind analysis performed on the final version of the code (compiled with -g and -O0 flags) indicate that while the instruction (I1 and LLi) cache miss rate and the level 1 data cache (D1) miss rate have remain unchanged (0.0 and 0.5%, respectively) between the original and the final optimised code, the last level data cache (LLd) miss rate has decreased from 0.3 to 0.1%. Moreover, the last level cache (LL) overall miss rate has decreased from 0.1 to 0.0% as a result of the introduced optimisations.

```
1  I    refs:      81,017,590,087
2  I1   misses:             1,962
3  LLi  misses:             1,731
4  I1   miss rate:           0.00%
5  LLi  miss rate:           0.00%
6
7  D    refs:      28,618,439,658  (23,180,301,917 rd   + 5,438,137,741 wr)
8  D1   misses:       148,149,235  (   137,574,615 rd   +    10,574,620 wr)
9  LLd  misses:        18,451,497  (     9,145,924 rd   +     9,305,573 wr)
10 D1   miss rate:            0.5% (           0.6%     +           0.2%   )
11 LLd  miss rate:            0.1% (           0.0%     +           0.2%   )
12
13 LL   refs:         148,151,197  (   137,576,577 rd   +    10,574,620 wr)
14 LL   misses:        18,453,228  (     9,147,655 rd   +     9,305,573 wr)
15 LL   miss rate:            0.0% (           0.0%     +           0.2%   )
```

Listing 17: Cachegrind interpreted profile data generated from 10 steps of MD simulation when final optimised code was compiled with -g -O0.

Loop	**# Loop entries**	**Avg. iterations count**	**Max. iterations count**
MD.c:74	2,047,500	2,048	4,095
MD.c:61	2,046,000	512	1,023
Totals	4,093,500	2,560	5,118

Table 7: Results of loops profiling on 500 timesteps when final optimised code was compiled with -O3 -no-prec-div -xCORE-AVX2 -ipo -profile-functions -profile-loops=all -profile-loops-report=2 flags.

7.4 Loops profiling on final optimised code

The final optimised code was compiled with the best set of compiler flags used throughout the coursework (i.e. -O3 -no-prec-div -xCORE-AVX2 -ipo) and the -profile-loops=all -profile-loops-report=2 flags for generating the loops profiling report. As a result of all the optimisations introduced, the only two loops with self time greater than 1% are the loops iterating over 1 and `Nbody-1` within the triple loops used to independently calculate pairwise separation of particles (line 5 in Listing 15) and adding pairwise forces (line 4 in Listing 13). It is worth noting that the combined total loop entries for these loops is 4,093,500, which is ≈1000 times smaller than the total loops entries for the original program (i.e. 4,195,331,500 loop entries). This is consistent with the 99.32 % decrease in runtime achieved as a result of all the introduced optimisations.

7.5 Compiler report on final optimised code

After inspection of the compiler report generated using the final optimised implementation, it appears that there are still a few loops that could not be fully vectorised by the compiler, leading to "remainder" or "peeled" loops (e.g. line 5 in Listing 15). The common pattern for these loops is that the outer loop is unrolled while the inner loop(s) is already vectorized. One approach for this would be to unroll the outer loop in different ways to achieve further performance improvement, potentially by making use of a function. That being said, this approach would require some complex modifications that would largely affect code readability and was therefore not carried out as part of the coursework.

8 Conclusion

During this coursework, the performance of a simple application code used for carrying out a MD simulation was improved such that the total application runtime was decreased to 35.34 seconds, which is equivalent to a 99.32% decrease relative to the original one. The optimised code performs approximately 25 times less calls to the function required for calculating forces, carries out approximately 1000 times less loop entries and also exhibits reduced data cache miss rates while still generating the correct simulation output. Due to time constraints, it was not possible to explore alternative approaches for addressing performance portability such as autotuning.

9 Appendix

9.1 Cachegrind raw data - original implementation

Function	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw
program totals	118,169,434,058	2,161	2,122	40,961,622,841	213,564,177	115,403,899	8,708,908,156	42,057,830	41,991,446
libm pow l9	51,841,597,440	38	38	15,099,494,400	7,478,954	1,113	1,509,949,440	0	0
evolve	46,818,266,815	98	97	18,122,671,005	143,034,347	52,484,638	3,440,128,495	41,981,480	41,937,477
force	9,059,696,640	1	1	3,523,215,360	20	10	3,019,898,880	20	10
add norm	7,551,591,180	2	2	3,524,075,760	62,930,020	62,899,260	503,439,660	0	0
pow	1,509,949,440	2	2	503,316,480	380	40	0	0	0
libm sqrt ex	839,065,600	2	2	83,906,560	0	0	167,813,120	0	0
printf fp	177,957,689	141	137	36,917,435	806	55	26,793,623	40,996	40,986

Table 8: Cachegrind raw data generated after running initial program with Nstep set to 10, and code compiled with -g and -O0 flags. Ir = instruction cache reads (which equals the number of instructions executed); I1mr = first-level instruction cache read misses ; ILmr = last-level instruction cache read misses; Dr = data cache reads (which equals the number of memory reads); D1mr = first-level cache data read misses; DLmr = last level cache data read misses; Dw = data cache writes (which equals the number of memory writes); D1mw = first level cache data write misses; DLmw = last level data cache data write misses.

9.2 Cachegrind raw data - compiler flag optimisation part 2

9.2.1 -O1 flag

Function	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw
Program Totals	76,464,214,534	1,768	1,739	26,611,399,504	213,390,870	115,403,842	5,268,746,488	42,058,603	41,991,482
???:libm pow l9	51,841,597,440	19	19	15,099,494,400	7,444,415	1,062	1,509,949,440	0	0
MD.c:evolve	13,928,080,780	31	31	8,305,643,935	142,900,800	52,484,610	1,761,915,115	41,981,425	41,937,487
util.c:force	6,039,797,760	0	0	2,013,265,920	20	10	1,509,949,440	0	0
util.c:add norm	1,762,038,060	1	1	503,439,420	62,930,080	62,899,290	251,719,680	0	0
???:pow	1,509,949,440	1	1	503,316,480	20	20	0	0	0
???:libm sqrt ex	839,065,600	1	1	83,906,560	0	0	167,813,120	0	0
???: printf fp	177,958,050	123	120	36,917,554	806	55	26,793,685	41,000	40,985
???:sqrt	83,906,560	1	1	0	0	0	0	0	0

Table 9: Cachegrind raw data generated after running initial program with Nstep set to 10, and code compiled with -g and -O1 flags.

9.2.2 -O2 flag

Function	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw
Program Totals	15,487,567,367	2,134	2,096	7,610,447,137	204,689,843	115,402,844	2,248,658,544	42,057,189	41,991,418
MD.c:evolve	10,740,036,140	94	94	6,753,413,530	141,644,610	52,484,640	1,384,294,620	31,492,810	31,449,669
util.c:force	3,019,898,880	0	0	503,316,480	0	0	0	0	0
???:intel avx rep memset	671,253,020	24	24	60	60	49	671,252,180	10,488,340	10,487,818
util.c:add norm	597,836,760	11	11	251,719,740	62,930,080	62,899,290	125,859,840	0	0
???:hack digit.13549	57,682,070	7	7	19,139,735	0	0	8,917,080	5	5
???: printf fp	177,957,346	134	130	36,917,350	636	55	26,793,580	41,000	40,990
???:hack digit.13549	57,682,070	7	7	19,139,735	0	0	8,917,080	5	5
???:vfprintf	43,122,322	90	81	9,854,363	1,129	170	10,017,414	270	90
???: mpn mul 1	41,604,612	6	6	6,127,147	0	0	4,132,075	5	0
???: IO vfscanf	26,100,773	67	67	7,720,964	9,446	9,290	2,584,580	8,133	960
???: mpn divrem	19,695,106	5	5	5,266,634	0	0	4,858,379	15	10
???: strtod l internal	18,740,564	54	54	2,628,836	4	4	1,380,234	4	0

Table 10: Cachegrind raw data generated after running initial program with Nstep set to 10, and code compiled with -g and -O2 flags.

9.2.3 -O3 flag

Function	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw
Program Totals	16,326,510,166	2,076	2,039	7,610,529,071	204,685,363	115,402,844	2,248,740,476	42,057,204	41,991,408
MD.c:evolve	11,578,978,870	87	87	6,753,495,450	141,640,470	52,484,640	1,384,376,540	31,492,810	31,449,669
util.c:force	3,019,898,880	0	0	503,316,480	0	0	0	0	0
???:intel avx rep memset	671,253,020	20	20	60	60	49	671,252,180	10,488,340	10,487,818
util.c:add norm	597,836,760	7	7	251,719,740	62,930,080	62,899,290	125,859,840	0	0
???:printf fp	177,957,376	138	134	36,917,355	476	55	26,793,583	41,000	40,990
???:hack digit.13549	57,682,085	7	7	19,139,741	0	0	8,917,086	5	5
???:vfprintf	43,122,322	91	82	9,854,363	1,124	170	10,017,414	260	80
???: mpn mul 1	41,604,612	5	5	6,127,147	0	0	4,132,075	5	0
???: IO vfscanf	26,100,773	67	67	7,720,964	9,461	9,290	2,584,580	8,068	960
???:mpn divrem	19,695,130	5	5	5,266,637	0	0	4,858,382	15	10
???:strtod l internal	18,740,564	54	54	2,628,836	4	4	1,380,234	4	0

Table 11: Cachegrind raw data generated after running initial program with Nstep set to 10, and code compiled with -g and -O3 flags.

9.3 Cachegrind raw data - final optimised implementation

Function	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw
Program Totals	81,017,590,087	1,962	1,731	23,180,301,917	137,574,615	9,145,924	5,438,137,741	10,574,620	9,305,573
MD.c:evolve	48,412,244,825	74	40	13,424,599,605	126,828,998	9,129,600	2,936,934,815	10,506,325	9,268,719
???:__libm_pow_l9	25,927,127,040	25	25	7,551,590,400	10,616,855	1,072	755,159,040	0	0
util.c:force	4,530,954,240	2	1	1,762,037,760	20	10	1,510,318,080	20	10
???:__libm_sqrt_ex	839,065,600	1	1	83,906,560	0	0	167,813,120	0	0
???:pow	755,159,040	0	0	251,719,680	20	15	0	0	0
???:__printf_fp	177,958,013	136	124	36,917,549	61	28	26,793,676	40,985	26,347
???:sqrt	83,906,560	1	1	0	0	0	0	0	0

Table 12: Cachegrind raw data generated after running final optimised program with Nstep set to 10, and code compiled with -g and -O0 flags.

References

[1] Free Software Foundation Inc. *GNU gprof: Top*. URL: https://sourceware.org/binutils/docs/gprof/ (visited on 04/07/2019).

[2] A. Jackson. *Optimising Memory Structures I Lecture*. University of Edinburgh Performance Programming course. (Visited on 03/15/2019).

[3] A. Jackson. *Optimising Memory Structures III Lecture*. University of Edinburgh Performance Programming course. (Visited on 03/15/2019).

[4] C. C. McGeoch. *A Guide to Experimental Algorithmics*. en. Google-Books-ID: gGOU5YL4U64C. Cambridge University Press, Jan. 2012. ISBN: 978-1-107-00173-2.

[5] Valgrind. *Valgrind*. URL: http://valgrind.org/docs/manual/cg-manual.html (visited on 04/07/2019).

[6] Y. Wang. *Step by Step Performance Optimization with IntelC++ Compiler*. en. URL: https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler (visited on 04/05/2019).