Parallel Computing - WA1

1st José Carvalho
Departamento de Informática
Universidade do Minho
Braga, Portugal
PG53975@alunos.uminho.pt

2nd Miguel Silva
Departamento de Informática
Universidade do Minho
Braga, Portugal
PG54097@alunos.uminho.pt

***Abstract*—In this report we will explain the decisions we made for the first phase of the Parallel Computing group assignment. We will go in depth into the optimisations we managed to implement and we will also explicit some others that we could not implement because they ended up reducing the performance of the program.**

***Index Terms*—Parallel Computing, C, Vectorization, Cache, Pipeline, SIMD, Memory Hierarchy**

I. Introduction

This document serves as a means to document and explain our decisions throughout the first phase of the group assignment. The code provided is a program that simulates simple molecular dynamics applied to atoms of argon gas (the original code is from FoleyLab/MolecularDynamics: Simple Molecular Dynamics).

We tried multiple techniques to see if our code ran better on the SeARCH cluster, but not all of them worked, therefor we will try and explain why that was the case, going in depth into our optimisations.

We will also mention the GCC flags we used, explore a roofline model of our code and in the end we will compare results and finish the report with a section dedicated to conclusions about what we learned during this project.

Throughout the report we will be using multiple concepts learned throughout our academic life. Most noticeably the equation to calculate the execution time of a piece of code: $t_{\text{exec}} = \#I \cdot (\#CPI_{\text{CPU}} + \#CPI_{\text{MEM}}) \cdot T_{\text{cc}}$

This report is divided into four more sections on which we will write about the critical zones, the optimisations we implemented and conclusions and critics about what we learned during this project.

II. Critical Zones

Before we started optimising the code we had to discover which sections of it had the most potential for optimisation. That was explored using *gprof*'s capabilities.

To use *gprof* we had to run the commands shown on the assignment. And after doing so we wound up with the graph present on figure 1.

After observing the graph we can clearly see that the functions which cause the biggest issues to our performance are *Potential()* and *VelocityVerlet(double,int,_IO_FILE*)* because it evokes the *computeAccelerations()* function.

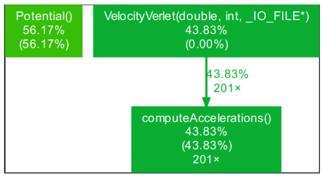

Fig. 1. *Gprof* graph that represents the bottlenecks in the code's original version

III. Optimisations

Since this phase is just a single thread program, we will not work with multi-threading, so we focused on other aspects of the code.

After analysing the most expensive functions (*Potential* and *computeAccelerations*), we saw that we could improve performance on both by looking at:

- Memory Hierarchy
- Loop unrolling
- Simplifying the math
- Pipeline
- Vectorization

A. Loop Unrolling

Loop Unrolling was one of the first techniques we used. Loop Unrolling affects directly the $\#CPI_{\text{CPU}}$ and $\#I$ components on the equation mentioned in section I.

We noticed that in some parts of the code, we could make loop unrolling manually on some basic static loops. The goal was to copy the code inside the loop, and paste it the number of iteration of the associate loop.

For example on the *computeAcceleration* function we had this loop.

```
for(k = 0; k < 3; k++) {
   a[i][k] = 0;
}
```

It is a nested loop, but it is responsible for setting all values from the accelerations matrix to 0.

So, we can adapt it to:

```
a[i][0] = 0;
a[i][1] = 0;
a[i][2] = 0;
```

We did this to many loops all over the code, reducing the number of instructions, because now we do not have to make an increment and a condition in every iteration. This also improves the code in terms of the pipeline, since there are no conditions to validate and each instruction is totally independent.

B. Memory Hierarchy

Memory Hierarchy will reduce the $\#CPI_{\text{MEM}}$ component on the equation referred in section I.

There were not many changes to do, because the code itself was already navigating through adjacent positions of the arrays. Still, in the function *computeAccelerations* we transformed the original array *rij* into 3 different variables.This was done to read the value of registers faster not needing to read it from the L1 cache.

We then tried to make some more improvements, but the results became worse, so they do not appear on our final version. We will explain them nevertheless.

Firstly we created new variables, or in other words, new registers to handle some of the work done by arrays. On our machines the performance of the program improved, but on SeARCH it got worse. One possible reason could be that our machines have more space to store registers than on SeARCH.

Another optimisation attempt was about using the cache more. We tried to make the loops on *computeAccelerations* and *Potential* execute in blocks. Unfortunately, we faced issues, because the program was either returning wrong results, or the performance worsened.

C. Pipeline

Pipelining reduces the $\#CPI_{\text{CPU}}$ component of the equation referred in I.

We noticed that the function *Potential* had one important element to improve, an if condition that could be removed.

This condition is inside in the nested loop and it affects the performance of the pipeline, because in the runtime the CPU does not know if the condition on it will be true or false. This could stop the pipeline, depending on how the CPU works, so we removed it.

We understood that the condition was false only when the variables i and j were equal.

To remove the condition, we created two different loops, one that makes j to iterate from 0 to i, and the other one that makes j iterate from i+1 to the end.

So this improvements will change the code from this:

```
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      if(j!=i)
          //
```

To this:

```
for (i=0; i<N; i++) {
   for (j=0; j<i; j++) // ...
   for (j=i++; j < N; j++) // ...
}
```

D. Simplifying the Maths

Math Simplifications help decrease both $\#I$ and $\#CPI_{\text{CPU}}$.

The math on *Potential* and *computeAcceleration* could be improved just by simplifying the operations.

Starting with the *Potential* function, the non simplified version of its equation is something like:

```
double PotentialAux(double a1, double a2,
    double a3)
{
   double r2 = a1 * a1 + a2 * a2 + a3 * a3;
   double rnorm=sqrt(r2);
   double quot=sigma/rnorm;
   double term1 = pow(quot,12.);
   double term2 = pow(quot,6.);
   return 4*epsilon*(term1 - term2);
}
```

The variables, a1, a2 and a3 are the subtractions of r[i][0] - r[j][0], r[i][1] - r[j][2], r[i][2] - r[j][2] respectively.

We started by removing the $4 \cdot epsilon$. Based on math equation properties

$$constant \cdot expression1 + constant \cdot expression \iff$$

$$constant \cdot (expression1 + expression2)$$

. With this we can remove the constant $4 * epsilon$ from *PotentialAux* function, placing it instead later when we already sum all the expressions.

Let's now take a look at the expression itself and simplify it:

$$(\frac{sigma}{\sqrt{a1^2 + a2^2 + a3^2}})^{12} - (\frac{sigma}{\sqrt{a1^2 + a2^2 + a3^2}})^6$$

$$\iff$$

$$\frac{sigma^{12}}{(a1^2 + a2^2 + a3^2)^6} - \frac{sigma^6}{(a1^2 + a2^2 + a3^2)^3}$$

$$\iff$$

$$(\frac{sigma^2}{a1^2 + a2^2 + a3^2})^6 - (\frac{sigma^2}{a1^2 + a2^2 + a3^2})^3$$

With this, we can store in the variable quot the result from:

$$\frac{sigma^2}{a1^2 + a2^2 + a3^2}$$

So the final expression will contain:

$$quot^6 - quot^3$$

With this simplifications we also concluded that the function *sqrt* is not needed.

Another improvement was not using the function pow, to multiply a number n by itself times. For example $pow(quot, 3)$ can be transformed into $quot \cdot quot \cdot quot$.

So assuming that $quot6 = quot^3$, the final expression will be $quot6 \cdot quot6 - quot6$

So the final version of the equation became:

```
double PotentialAux(double a1, double a2,
    double a3)
{
   double quot = sigma*sigma/(a1 * a1 + a2 *
       a2 + a3 * a3);
   double quot6 = quot * quot * quot;
   return quot6 * quot6 - quot6;
}
```

Now on *computeAccelerations*, which already has the optimizations mentioned previously, we have this:

```
rij0 = r[i][0] - r[j][0];
rij1 = r[i][1] - r[j][1];
rij2 = r[i][2] - r[j][2];
rSqd  = rij0*rij0 + rij1*rij1 + rij2*rij2;
f = 24 * (2 * pow(rSqd, -7) - pow(rSqd, -4));
a[i][0] += rij0 * f;
a[i][1] += rij1 * f;
a[i][2] += rij2 * f;
a[j][0] -= rij0 * f;
a[j][1] -= rij1 * f;
a[j][2] -= rij2 * f;
```

Here we also have $constant \cdot expression1 + constant \cdot expression2$, so we can take the constant out and apply it on the sum of all expressions. The only difference from *computeAccelerations* to *Potential*, is that the values are in an array and not on a local variable, so theee will be a loop iterating over the results array and multiplying every value by 24.

Now on the equation:

$$2 \cdot rSqd^{-7} - rSqd^{-4} \iff 2 \cdot (\frac{1}{rSqd})^7 - (\frac{1}{rSqd})^4$$

$$\iff (\frac{1}{rSqd})^4 \cdot (2 \cdot (\frac{1}{rSqd})^3 - 1)$$

We can simply store the value of $\frac{1}{rSqd}$ and use it 2 times. With this we only make 1 division on each iteration, instead of 2 or more.

Another thing we noticed was that we have $rij0 \cdot f$, $rij1 \cdot f$ and $rij2 \cdot f$ 2 times each, so we can update the values $rij0$, $rij1$ and $rij2$, multiplying them by f, and making the multiplication only 1 time.

So the final version of *computeAccelerations* became:

```
rij0 = r[i][0] - r[j][0];
rij1 = r[i][1] - r[j][1];
rij2 = r[i][2] - r[j][2];
rSqd = 1/(rij0*rij0 + rij1*rij1 + rij2*rij2);
rSqd3 = rSqd * rSqd * rSqd;
f    = rSqd3 * rSqd * (2 * rSqd3 - 1);
rij0 = rij0 * f;
rij1 = rij1 * f;
rij2 = rij2 * f;
a[i][0] += rij0;
a[i][1] += rij1;
a[i][2] += rij2;
a[j][0] -= rij0;
a[j][1] -= rij1;
a[j][2] -= rij2;
```

On both final versions we reduced the number of instructions executed and to sum up the decisions we made were:

- Not using the functions *sqrt* and *pow*
- Reduce the number of multiplications executed in each iteration
- Simplifying the equations, using math properties

We can still improve the code using vectorization, which we will explain in section III-E.

E. Vectorization

This was the main optimisation of this phase and it's main goal is to reduce the $\#I$ component.

We opted to not write any vectorized code, because the *gcc* compiler already makes that work for us and we chose to make the code as legible and simple as possible. With this in mind we only have to give *gcc* code that can be vectorized.

We are using 32 bytes SIMD instructions, in order to reduce the number of instruction as much as possible.

To use SIMD instruction, we had to re-organise how the data is declared on the r, v, a and F matrix.

We started by transforming the matrix into arrays, so we assure that the positions of the matrix are adjacent on memory positions. But this is not enough to apply 32 bytes SIMD instructions. To reach the goal, we have to assure that the index of the first position on memory of each arrays must be a multiple of 32. So, the first position of each array can be something between 0,32,64,96,128, (...), until the last element multiple 32 that can fit on memory.

In the code, we only need to add the attribute ((aligned (32))) to the array declarations. So from:

```
double r[MAXPART][3];
double v[MAXPART][3];
double a[MAXPART][3];
double F[MAXPART][3];
```

We made this:

```
double r[MAXPART*3] __attribute__((aligned
    (32)));
double v[MAXPART*3] __attribute__((aligned
    (32)));
double a[MAXPART*3] __attribute__((aligned
    (32)));
double F[MAXPART*3] __attribute__((aligned
    (32)));
```

Due to this change, we now have to update every loop that worked with this arrays, to iterate over 3 positions each iterations, instead of 1 position each iteration.

To use all this we must activate a flag on gcc, that is explained on III-F.

Since the arrays have doubles, that occupy 8 bytes, we can assume theoretically that our code will reduce 4 times the number of instructions. This prediction will not be achieved since there are always pieces of code that cannot be vectorized.

The number of instructions of vectorized code went from 35 944 910 807 to 24 506 094 976, so we reduced the number of instructions by 11 438 815 831, which means 1.467 times less instructions. These metrics were measured after adding and removing the *gcc* flag -mavx.

F. GCC flags

To run our code in the most optimal way we decided to use the following flags:

- -O3
- -Ofast
- -pg
- -funroll-loops
- -mavx

-O3 sets the optimisation level 3, this enables aggressive compiler optimisations such as inlining functions and other features such as loop unswitching, and some vectorization.

-Ofast enables all O3 optimisation and disregards safety standards maximising execution speed, this may cause floating point errors. In our case we only used it because it did not cause a significant error even though some errors where still caused.

The flag -pg allows the generation of profiling information which is needed to use tools such as *gprof*.

The flag -funroll-loops applies loop unrolling throughout the code.

Lastly the flag -mavx is used to allow for usage of Advanced Vector Extensions (AVX) instructions, in other words, vectorization for x86 processors, a helpful tool for operations with complex data structures and/or calculations.

G. Roofline

Roofline is a technique to see if our program is slow due to the memory bandwidth or not.

Figure 2 contains the architecture of SeARCH cores.

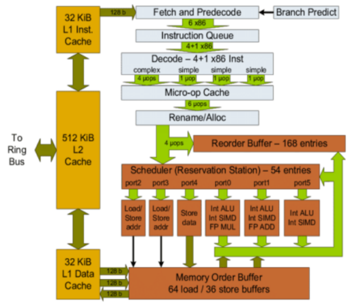

Fig. 2. SeARCH cores optimization

W can execute at the same time 1 sum and 1 multiplication on doubles. We also know that the frequency of our processing unit is 2.5GHZ. We got this value by running the command lscpu on the SeARCH cluster. So the peak performance in GFlops/s is $2.5 \cdot 2 = 5$GFlops/s. With level 4 vectorization (32 bytes SIMD Instructions / 8 bytes doubles), this number will be $5 \cdot 4 = 20$GFlops/s.

Since the functions that make our program slower are *Potential* and *computeAccelerations*, we focused on those.

In *Potential*, we can see that in each iteration of the nested cycles we make 7 multiplications, 4 subtraction, 2 sums and 1 division. On *computeAccelerations*, on each iteration of nested cycle we have 7 subtractions, 5 sums, 11 multiplications and 1 division.

We cannot execute all operations at the same time on both functions, because we do not have sufficient cores, so the program will never reach the peak performance. Also, having a division in each iteration of both functions, will negatively impact performance, even if we were able to remove some other operations. As the divisions take more clock cycles to complete that the other operations. As we know, there are no processing unit capable of dividing in a single instruction.

H. Results

Previously we presented a *gprof* graph for the original code. Which as mentioned had two bottlenecks we tried to solve. With that in mind after optimizing our code we ran *gprof* once again, ending up with the graph from figure 3.

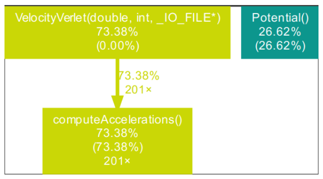

Fig. 3. *Gprof* graph that represents the bottlenecks in the code's optimized version

The graph shows that there was a significant decrease in the percentage of time needed to run the *Potential()* function, this in turn means that this function was improved. On the other hand, *computeAccelerations()*'s percentage went up, this is because it is still the most complex function in the code and it ended up taking most of the runtime of our code, that now runs noticeably faster.

In terms of pure metrics, we started out with this performance:

TABLE I
STATISTICS BEFORE OPTIMIZATIONS

Metric	Value
L1-Misses	2365426182
#I	986779125734
#CPI	0.7
#CC	649168044023
Time	196,97

Whilst at the end of the project this were our results:

It is important to note that the original results were measured with the flags activated. The optimised results ended up being quite impressive with a huge reduction in cache misses, instructions and clock cycles and a massive difference in execution times.

All this ended up causing the following gains:

TABLE II
STATISTICS AFTER OPTIMIZATIONS

Metric	Value
L1-Misses	671653428
#I	24506749832
#CPI	0,8
#CC	20280170222
Time	6,34

$$\frac{2365426182}{671653428} = 3.521795741$$
$$\frac{986779125734}{24506749832} = 40.2656057$$
$$\frac{649168044023}{20280170222} = 32.0099899$$
$$\frac{196,971986828}{6,343183461} = 31.0525445211$$

Which means we managed to reduce misses on cache to 3.5 times less, 40 times less instructions executed, 32 times less clock cycles and most impressively 31 times faster code execution.

The CPI grew 0.1, because the instructions became more dependent from each other.

IV. CONCLUSION

With the conclusion of this phase of the project, we can say that we managed to apply most of the techniques learned in the lessons.

After analysing our results we can state that we managed to do a good amount of optimisation and with that we managed to confirm just how useful this curricular unit's teachings can be.

Still, we would love to improve results even further and we hope we can do so in the next phase with the help of threads.

Surprisingly the biggest obstacle we faced was just how complicated it was to run the code in the SeARCH cluster due to how overcrowded it is. That lead to wrong performance measurements multiple times, but we managed to pull through and considering all our work we are proud with what we accomplished.