Parallel Computing
Work Assignment Phase 1

Ana Rita Poças (*pg53645)* e Orlando Palmeira (*pg54123)*

***Abstract*—The program that we were supplied for this assignment phase is part of a simple molecular dynamics' simulation code applied to atoms of argon gas. The main goal in this phase is for us to analyse and optimise the program in order to reduce it's execution time, by mainly using optimisation techniques that impact the program's performance and also techniques that make the code more legible to prepare it for upcoming phases in this assignment.**

***Index Terms*—performance, optimisations, execution time, ILP, memory hierarchy, data structures, vectorisation, legibility**

I. Introduction

THE main aim of this first assignment phase was to apply optimisation techniques into a single threaded program, by using tools for code analysis/profiling in order to improve the final performance of the program.

We started by analysing the given code, first by estimating execution performance and then with the help of tools like *gprof* we were able to exactly identify the functions and areas of the code that were weighting on the execution time of the program.

After, we went on to apply optimizations that we considered to be meaningful into our code in order to improve the program's execution time.

II. Source Code Analysis

We started by running the original code provided to us and we noticed, with the help of the *perf* tool, how long it took for it to run (approximately 236 seconds). .

A. Estimated Execution Performance

After running the code, we proceeded to analyze the given code without the help of any tool, just by examining the code. We calculated the complexity of the functions in the code (the complete complexity analysis is in Appendix B):

- VelocityVerlet: $O(550N + \frac{273(N^2-N)}{2})$ ($O(508N)$ without computeAccelerations)
- computeAccelerations: $O(42N + \frac{273(N^2-N)}{2})$
- Potential: $O(174N^2 - 172N)$
- MeanSquaredVelocity: $O(67N)$
- Kinetic: $O(85N)$

As we can see, the most expensive functions in the code are computeAccelerations and Potential because they're the only ones with quadratic complexity (note that velocityVerlet also has quadratic complexity because it uses computeAccelerations). Thus, these two functions will probably be our main focus when it comes to optimizations.

B. Code Profiling

In order to find out exactly which functions were weighting the most on the execution time, we used the tool *gprof*. We've realized that the functions that contributed the most to this execution time were Potential and computeAccelerations, as shown in the image below generated by *gprof2dot*.

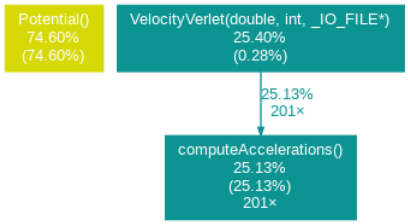

Fig. 1. Results from gprof2dot for the non-optimised code

These results provided by gprof corroborated the predictions we made through the complexity analysis of the code's functions.

C. Optimisation Targets

It became clear that the functions that need more attention in terms of optimisation are definitely **Potential** and **computeAccelerations**.

III. Optimisations Implemented

Once we've found the found our optimisation targets, we started focusing on applying optimisations to each of these functions:

A. Potential

The original code for the Potential function has parts that we should highlight in terms of performance. We were able to improve it as it follows:

1) There's a **for** loop (regulated by the **k** variable that goes from 0 up to 3) whose size is always fixed. This loop can be removed, allowing the reduction of the #instructions, since there's no need for: the initial assignment of the value 0 to **k**, the increment of the **k** variable and therefore the checking if k is less than 3.
2) We've also noticed that the operation $4 \times epsilon$ doesn't depend on the loops controlled by the variables i and j. Furthermore, the value of this operation is being applied to a sum that is being accumulated in the variable **Pot** (as Shown in Appendix A, section Potential, subsection 1).

   So, mathematically, we can factor out the value of $4 \times epsilon$, which in terms of code means that we can remove the operation from the loop and only placing

it the moment we're calculating the return value of the function, by multiplying it to the value of the **Pot** variable. This helps to reduce the number of multiplications (heavier than sums and subtractions), which may help to reduce CPU's pipeline stalls.

3) Within the cycle controlled by the **k** variable (that was removed), the subtraction $r[i][k] - r[j][k]$ is done twice. We can reduce the number of times this operation is performed by keeping its value in a variable. This optimization contributed to reducing the #instructions (by reducing subtractions) and the number of memory accesses (since we now have 2 accesses to matrix r instead of 4).
4) We've also verified that the variables **term1** and **term2**, elevate the value of the variable **quot** to even numbers. This lead to a possible elimination of the square root of **r2** given to the variable **rnorm**, since it's used to calculate the value of **quot** that is then elevated to 6 and 12. So, we proceeded to the calculations, shown in Appendix A, section Potential, subsection 2, in order to try to simplify the mathematical operations in this part of the function.
   So, by applying this mathematical simplification, we're able to avoid the use of heavy operations, such as $sqrt$ and pow. These heavier operations ($sqrt$ and pow) can contribute to increasing the number of stalls in the CPU's pipeline and unnecessary function calls.
5) We've noticed that the code inside the loops *i* and *j* ($i, j \in [0, N[$) runs only when $i \neq j$. Thus, we can see the indices **i** and **j** in the following table:

i,j	0	1	2	...
0	×	A	B	C
1	A	×	D	E
2	B	D	×	F
...	C	E	F	×

Legend:
- Cells with ×: Code does not run.
- Cells with letters: Code runs

   Each cell in the table above represents an iteration in the function for specific i's and j's. The cells with equal letters have the same value of $r2 = \sum_{i=0}^{N-1} \sum_{j=0, j\neq i}^{N-1} \sum_{k=0}^{2} r[i][k] - r[j][k]$. Hence, we can perform the execution of the j loop from $i + 1$ to N, which reduces the #instructions significantly. However, in the end of the function, we need to calculate $8 \times epsilon$ instead of $4 \times epsilon$ (see point 2), because we perform the half of the calculations (half of the sum in the variable **Pot**).

B. computeAccelerations

Just like the previous function, this one has numerous parts that we agree that could be improved by the implementation of the following optimisations:

- There's three **for** loops, controlled by the variable **k**, with a fixed size (3) that can disappear (just like *Potential*).
- We've noticed the operation $f = 24 * (2 * pow(rSqd, -7) - pow(rSqd, -4))$ is heavy (due to the use of the function pow and can be simplified by modifying it to the expression shown in Appendix A, section computeAccelerations, subsection 1.
  With this new approach, we were able to simplify the calculations of the value of **f** by creating the two variables shown in the Appendix A, section computeAccelerations, subsection 2 (which values do not require the use of the function pow).
  And therefore, in the code, the calculations made for f are now:$f = (48 - 24 * rSqd3)/rSqd7$. As with the Potential function, avoiding the use of heavy operations can help to reduce the number of stalls in the CPU pipeline.
- Within the **for** loop controlled by the **j** variable, there are repetitive accesses (reading and writing) to $a[i][0]$, $a[i][1]$ and $a[i][2]$ which cause unwanted memory accesses. This problem was minimized with the creation of three variables that temporarily keep this values and at the end of the execution of the loop controlled by **j**, can be added to the values of $a[i][0]$, $a[i][1]$ and $a[i][2]$. This helps to take advantage of the memory hierarchy by reducing read and write accesses to matrix **a**.

C. The merging of Potential and ComputeAccelerations

We later noticed, particularly after optimization number 5 in the Potential function, that there's an operation that was being performed by both functions that was the same (the calculation of **r2** in the Potential and **rsqd** in the computeAccelerations). By merging them into just one function, we would be able to reduce the number of times that this calculation is performed and thus optimising our code.

So, we joined them, by adding to the computeAccelerations the instructions that were being performed within the Potential function, and now everything is executed within the computeAccelerations function.

D. Makefile

To further optimise our program we also used the following compiler flags:

- **-O3** : compiler optimisation flag used to instruct the compiler to activate a set of optimization options that generate faster and more efficient machine code.
- **-funroll-loops**: compiler optimisation flag that directs the compiler to perform loop unrolling, which helps reducing the overhead of loop control structures.
- **-ftree-vectorize** : compiler optimisation flag that enables tree vectorization, optimises the code by transforming scalar operations into vector operations.
- **-msse4** : flag that takes advantage of SSE4 instructions for improved performance on processors that support SSE4.

IV. Results Comparison

Comparing to the original program that was taking approximately 236s to run ($\approx 1.3 \times 10^{12}$ instructions), our final program takes approximately 3.4s to run ($\approx 1.93 \times 10^{10}$ instructions). Not only was the execution time greatly improved, but also the number of instructions and the number of cycles.

V. Conclusion

In this first assignment we were able to explore numerous optimisation techniques and analyse their impact into the final performance of our program, specially in terms of execution time.

APPENDIX A
SIMPLIFICATIONS OF THE MATHEMATICAL OPERATIONS

A. Potential

1) : $Pot = \sum_{i=0}^{N} \sum_{j=0, j\neq i}^{N} 4 \times epsilon \times (term1 - term2)$

$= \sum_{i=0}^{N} 4 \times epsilon \times \sum_{j=0, j\neq i}^{N} (term1 - term2)$

$= 4 \times epsilon \times \sum_{i=0}^{N} \sum_{j=0, j\neq i}^{N} (term1 - term2)$

2) : $term1 = quot^{12} = ((quot^3)^2)^2 = (((\frac{sigma}{rnorm})^3)^2)^2 = (((\frac{sigma}{\sqrt{r2}})^3)^2)^2 = \frac{(sigma^6)^2}{(r2^3)^2} = (\frac{sigma^6}{r2^3})^2 = term2 \times term2$

$term2 = quot^6 = (quot^3)^2 = ((\frac{sigma}{rnorm})^3)^2 = ((\frac{sigma}{\sqrt{r2}})^3)^2 = \frac{(sigma^3)^2}{r2^3} = \frac{sigma^6}{r2^3}$

B. computeAccelerations

1) : $f = 24 \times (2 \times rSqd^{-7} - rSqd^{-4}) = 48 \times rSqd^{-7} - 24 \times rSqd^{-4} = \frac{48}{rSqd^7} - \frac{24}{rSqd^4} = \frac{48}{rSqd^7} - \frac{24 \times rSqd^3}{rSqd^7} = \frac{48 - 24 \times rSqd^3}{rSqd^7}$

2) : $rSqd3 = rSqd \times rSqd \times rSqd$ $(rsqd3 = rSqd^3)$
$rSqd7 = rSqd3 \times rSqd3 \times rSqd$ $(rsqd7 = rSqd^7)$

APPENDIX B
CODE ANALYSIS - COMPLEXITY

The entire code analysis was based on the original code provided by the teaching team.

A. Function VelocityVerlet

- The first loop `i` has a for loop `j` which runs N times and has the following assembly ($79 \times 3 = 237$ instructions).

```
.L76:
    cmpl $2, -8(%rbp)
    jg .L75
    movl -8(%rbp), %eax
    movslq %eax, %rcx
    ...
    addq %rcx, %rax
    movsd %xmm0, v(,%rax,8)
    addl $1, -8(%rbp)
    jmp .L76
```

- The function `computeAccelerations()` has a complexity of $O(42N + \frac{273(N^2-N)}{2})$
- The second loop `i` has a for loop `j` which runs N times and has the following assembly ($35 \times 3 = 105$ instructions):

```
.L80:
    cmpl $2, -8(%rbp)
    jg .L79
    movl -8(%rbp), %eax
    movslq %eax, %rcx
    ...
    addq %rcx, %rax
    movsd %xmm0, v(,%rax,8)
    addl $1, -8(%rbp)
    jmp .L80
```

- The third loop `i` has a for loop `j` which runs N times and has the following assembly (approximately, because of the if's,, $53 \times 3 = 159$ instructions):

```
.L88:
    cmpl $2, -8(%rbp)
    jg .L83
    movl -8(%rbp), %eax
    ...
    movsd %xmm0, -16(%rbp)
.L86:
    addl $1, -8(%rbp)
    jmp .L88
```

- The calculation of `psum/(6*L*L)` has the following assembly (7 instructions):

```
movsd L(%rip), %xmm1
movsd .LC89(%rip), %xmm0
mulsd %xmm0, %xmm1
movsd L(%rip), %xmm0
mulsd %xmm0, %xmm1
movsd -16(%rbp), %xmm0
divsd %xmm1, %xmm0
```

- Now we can conclude that the complexity of `VelocityVerlet` is $O((237 + 105 + 159 + 7)N + 42N + \frac{273(N^2-N)}{2}) = O(508N + 42N + \frac{273(N^2-N)}{2}) = O(550N + \frac{273(N^2-N)}{2})$
- The complexity of `VelocityVerlet` is high because it uses `computeAccelerations`. Without `computeAccelerations`, this function has a complexity of $O(508N)$

B. Function computeAccelerations

- The code inside of the first loop `i` has a for loop `k` which runs N times and has the following assembly the following assembly ($3 \times 14 = 42$ instructions):

```
.L62:
    cmpl $2, -12(%rbp)
    jg .L61
    movl -12(%rbp), %eax
    movslq %eax, %rcx
    movl -4(%rbp), %eax
    movslq %eax, %rdx
    movq %rdx, %rax
    addq %rax, %rax
    addq %rdx, %rax
    addq %rcx, %rax
    pxor %xmm0, %xmm0
    movsd %xmm0, a(,%rax,8)
    addl $1, -12(%rbp)
    jmp .L62
```

- The second loop i runs $N - 1$ times and loop j inside of it runs $N - i - 1$ times. Since we have this information, we can calculate how many times the code inside of the loop j will run: $\sum_{i=0}^{N-2} (\sum_{j=i+1}^{N-1} 1) = \frac{N^2-N}{2}$
  So, the code inside the loop j runs $\frac{N^2-N}{2}$ times.
- The first loop k inside the loop j has the following assembly code ($36 \times 3 = 108$ instructions):

```
.L67:
    cmpl $2, -12(%rbp)
    jg .L66
    movl -12(%rbp), %eax
    movslq %eax, %rcx
    ...
    addsd %xmm1, %xmm0
    movsd %xmm0, -24(%rbp)
    addl $1, -12(%rbp)
    jmp .L67
```

- Calculating the value of `f` has the following assembly (15 instructions):

```
.L66:
```

```
    movq -24(%rbp), %rax
    movl $-7, %edi
    movq %rax, %xmm0
    call pow
    addsd %xmm0, %xmm0
    movsd %xmm0, -72(%rbp)
    movq -24(%rbp), %rax
    movl $-4, %edi
    movq %rax, %xmm0
    call pow
    movsd -72(%rbp), %xmm1
    subsd %xmm0, %xmm1
    movsd .LC91(%rip), %xmm0
    mulsd %xmm1, %xmm0
    movsd %xmm0, -32(%rbp)
```

- The second loop k inside the second loop j has the following assembly code ($50 \times 3 = 150$ instructions):

```
.L69:
    cmpl $2, -12(%rbp)
    jg .L68
    movl -12(%rbp), %eax
    movslq %eax, %rcx
    ...
    addq %rcx, %rax
    movsd %xmm0, a(,%rax,8)
    addl $1, -12(%rbp)
    jmp .L69
```

- We can conclude that the complexity of the second loop i is $O((108 + 15 + 150) \times \frac{N^2-N}{2}) = O(\frac{273(N^2-N)}{2})$
- Now, we have the complexities of the both loops controlled by i ($O(42N)$ and $O(\frac{273(N^2-N)}{2})$). So we can conclude that the complexity of `computeAccelerations` is $O(42N + \frac{273(N^2-N)}{2})$

C. Function Potential

- The code inside of the loop j runs $N(N-1)$ times. However, the `if(j!=i)` runs N^2 times.
- The `if(j!=i)` has the following assembly (2 instructions):

```
movl -24(%rbp), %eax
cmpl -20(%rbp), %eax
```

- The loop k has the following assembly ($47 \times 3 = 141$ instructions):

```
.L55:
    cmpl $2, -28(%rbp)
    jg .L54
    movl -28(%rbp), %eax
    ...
    addsd %xmm1, %xmm0
    movsd %xmm0, -8(%rbp)
    addl $1, -28(%rbp)
    jmp .L55
```

- The rest of the calculations inside de loop j (`rnorm=sqrt(r2);...`) has the following assembly code (31 instructions):

```
.L54:
    movq -8(%rbp), %rax
    movq %rax, %xmm0
    call sqrt
    ...
    movsd -16(%rbp), %xmm1
    addsd %xmm1, %xmm0
    movsd %xmm0, -16(%rbp)
```

- Now we can conclude that the complexity of the `Potential` is $O(2N^2+(141+31)N(N-1)) = O(2N^2+172N(N-1)) = O(174N^2 - 172N)$.

D. Function MeanSquaredVelocity

- The code inside loop i runs N times.
- The whole code inside loop i has the following assembly (60 instructions):

```
movl -28(%rbp), %eax
movslq %eax, %rdx
movq %rdx, %rax
...
movsd -24(%rbp), %xmm1
addsd %xmm1, %xmm0
movsd %xmm0, -24(%rbp)
```

- The calculation `v2 = (vx2+vy2+vz2)/N` has the following assembly (7 instructions):

```
movsd -8(%rbp), %xmm0
addsd -16(%rbp), %xmm0
addsd -24(%rbp), %xmm0
movl N(%rip), %eax
cvtsi2sdl %eax, %xmm1
divsd %xmm1, %xmm0
movsd %xmm0, -40(%rbp)
```

- Now we can conclude that the complexity of `MeanSquaredVelocity` is $O((60 + 7)N) = O(67N)$

E. Function Kinetic

- The code inside the loop i runs N times.
- The loop j has the following assembly ($26 \times 3 = 78$ instructions):

```
.L47:
    cmpl $2, -24(%rbp)
    jg .L46
    movl -24(%rbp), %eax
    movslq %eax, %rcx
    ...
    addsd %xmm1, %xmm0
    movsd %xmm0, -8(%rbp)
    addl $1, -24(%rbp)
    jmp .L47
```

- The calculation `kin += m*v2/2.;` has the following assembly (7 instructions):

```
movsd m(%rip), %xmm0
mulsd -8(%rbp), %xmm0
movsd .LC87(%rip), %xmm1
divsd %xmm1, %xmm0
movsd -16(%rbp), %xmm1
addsd %xmm1, %xmm0
movsd %xmm0, -16(%rbp)
```

- Now we can conclude that the complexity of `Kinetic()` is $O((78 + 7)N) = O(85N)$