Computação Paralela

Phase 1

Ricardo Lopes Lucena
Departamento de Informática
Universidade do Minho
Braga, Portugal

Xavier Santos Mota
Departamento de Informática
Universidade do Minho
Braga, Portugal

***Abstract*—This document delves into our group's efforts to enhance the provided code from our professors. We analyze various metrics and draw conclusions from our optimizations.**

***Index Terms*—Optimization, clock cycles, execution time, code readability**

I. Introduction

For this assignment, our goal was to optimize existing C++ code without compromising its readability.

II. A priori assessment of the code

Upon initial examination and utilizing the 'perf' tool, our group promptly pinpointed the two primary functions within the code responsible for nearly the entire execution time. These functions are 'potential' and 'computeAccelerations.' In contrast, the 'velocityVerlet' function incurred minimal overhead, while all other functions had negligible impact on performance.

The thought of the group, after analising the entire code was to start making adjustments to one of the two functions refered above that took most of the execution time. Since the "Potential" function took more time than the other we decided we would start there.

III. Changes in the code

In this section we will be showcasing the strategies that we applied so that our code would transform into something more efficient with less execution time. It should be noted that the time that the program took to run was 240 seconds (around 4 minutes) in the SeARCH engine.

The following optimizations, shown in the next subchapters, were implemented in the 'Potential', 'computeAccelerations' and velocityVerlet functions

A. "Pow" and "Sqrt" function removal

The first thing we noticed was that there were many calls made to the *pow* and *sqrt* functions. Since they were inside two loops that went from 0 to 2160, it meant that they were called many times, which meant they could be one of the main sources of the heavy execution time in the program since calling these functions involves a function call and additional overhead.

The main change was made in this snippet of code:

```
rnorm=sqrt(r2);
quot=sigma/rnorm;
term1 = pow(quot,12.);
term2 = pow(quot,6.);
```

The 'pow' function was replaced with manual multiplication, and to maintain the same result while removing the "sqrt" function, all we needed to do is divide the exponent by 2. This is because the square root of a value raised to the 12th power is equivalent to that value raised to the 6th power. After making these changes, the code execution time was significantly improved.

```
quot = sigma/(r2);
term2 = quot *quot * quot;
term1 = term2 *term2;
```

B. Loop Unrolling

The loop unrolling optimization strategy capitalizes on maximizing the use of the L1 cache. Given that the cluster's CPU has 64 bytes per row, and each double takes up 8 bytes, we can infer that each memory access loads 8 doubles into the cache. This strategy was primarily employed in code blocks like the following:

```
for(k=0; k<3; k++){
r2+=(r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
```

By implementing this optimization in this context, we can efficiently access 6 doubles without relying on main memory, making optimal use of the cache. Ultimately, this results in a reduction in the program's execution time.

The optimized version of the code unrolls the loop as follows:

```
r2+=(r[i][0]-r[j][0])*(r[i][0]-r[j][0]);
r2+=(r[i][1]-r[j][1])*(r[i][1]-r[j][1]);
r2+=(r[i][2]-r[j][2])*(r[i][2]-r[j][2]);
```

This unrolled version minimizes cache misses and contributes to improved execution time.

C. Improved array access patterns

In both the 'potential' and 'computeAccelerations' functions, we optimized the array access patterns in two key ways:

Firstly, we harnessed the memory hierarchy by transposing the primary arrays, 'a,' 'r,' and 'v,' originally structured as

```
array[N][3] Into array[3][N]
```

This transformation significantly reduced cache misses during repeated array accesses since we now take advantage of the L1 cache. The original code didn't take full advantage of the caching mechanism, as every new access to an array value caused a cache miss, necessitating the retrieval of the required data line into the cache. To enhance this aspect of performance, we made an improvement by transposing the array structure. We reconfigured it into a 3-line by N-column format. This adjustment increased the likelihood that the subsequent item needed for array access was already loaded in the cache, ultimately resulting in improved overall performance.

Secondly, we substantially minimized the number of array accesses. This was achieved by storing values corresponding to array accesses dependent solely on the 'i' variable of the outer loop in local variables. These local variables were updated within the inner loop, thereby eliminating the need to access those positions within the inner loop on each iteration. Upon completion of the inner loop, the array values were updated based on the local variables. This approach greatly enhanced efficiency in array access.

```
for (i = 0; i < N-1; i++) {
    for (j = i+1; j < N; j++) {
        y = array[i];
```

Would change to

```
for (i = 0; i < N-1; i++) {
    x = array[i];
    for (j = i+1; j < N; j++) {
        y = x;
```

D. Optimization of mathematical equations

We enhanced the code's performance by substituting certain mathematical formulas with equivalent expressions that significantly improved computational speed.

As an example, consider the original formula:

```
f = 24 * (2*pow(rSqd, -7) - pow(rSqd, -4));
```

Additionally, following the modifications in subsection B, we streamlined the process by reducing the number of divisions, which can be computationally expensive, to just one division per iteration, we did this by first calculating the inverse of rSqd and using it to calculate pow(rSqd, -7) and pow(rSqd, -4). The revised formula looks as follows:

```
double invSqd = 1/rSqd;
double invHelp = invSqd * invSqd;
double invHelp2 = invHelp * invHelp;


f=24*(2*invHelp2*invHelp*invSqd-invHelp2);
```

This significantly improved performance.

E. Loop optimizations

In the 'potential' function, we eliminated the 'if' statement to maximize CPU pipelining resources utilization. To ensure that the logic of the 'if' statement placement remains consistent with

```
if(j != i)
```

we divided the original 'for' loop (ranging from 0 to N) into two identical loops. One loop iterates from 0 to 'i' while the other spans from 'i+1' to 'N.' Afterwards, we discovered that the 'r' matrix exhibited symmetry, eliminating the necessity for both loops. As a result, the 'potential' function now only performs half the number of cycles compared to its previous implementation.

This reconfiguration effectively preserves the 'if' statement's intended logic while improving code performance.

F. Vectorization and overall flags

To leverage vectorization for optimizing our code, we incorporate several compilation flags, including -mavx and -ftree-vectorize. These flags enhance performance by making efficient use of vector operations. In addition, we apply -Ofast, -funroll-all-loops, and -march=native. These combined flags help reduce execution time and improve the overall efficiency of our code.

G. Creating auxiliary functions

To enhance the overall code readability, we introduced auxiliary functions at various points in our code. An example of this can be found in the 'Potential' function, where we delegate the intensive mathematical computations to a separate function called 'contaPotential.' However, it's worth noting that this modification is not meant to have an impact on code's execution time.

IV. Final Execution Time

Reaching the end of the project, these were the obtained results, where the first line corresponds to the initial run, without any code optimizations, while the second line represents the final results after all the optimizations have been applied.

#I	#CC	#CM
1247809985705	1061626728804	1392086
6633867623	7597833881	2168

Texec Inicial: 348,56s **Texec Final:** 2,45s