Parallel Computing

Work Assignment Phase 1

Francisca Barros
Universidade do Minho
Braga, Portugal
PG53816

Rafael Correia
Universidade do Minho
Braga, Portugal
PG54162

***Abstract*—This document is a report for the first work assignment of Parallel Computing. It discusses the optimization of a single-threaded molecular dynamics simulation code for argon gas atoms. The objective was to improve the program's execution time through code analysis and profiling, while maintaining code legibility. Key steps include identifying performance bottlenecks, exploring optimization techniques, and preserving simulation outputs. This work aims to enhance code efficiency and set the stage for subsequent assignment phases.**

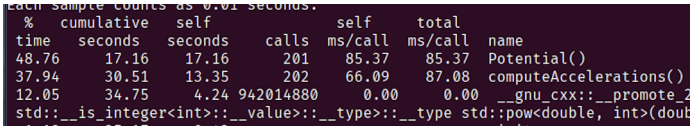
I. First Step - Analysing

A. Understanding the code

The first step was to try to understand what the code did and identify possible bottlenecks. Early on, we saw that the main function had a loop that iterates a lot. Inside that loop we saw some functions, namely: *VelocityVerlet*, *MeanSquaredVelocity*, *Kinetic* and *Potential*. Each of these functions, except for the *MeanSquaredVelocity* function, had loops within loops in them, but the ones that are note worthy are the loops in the *VelocityVerlet* and *Potential* functions (note that the VelocityVerlet function calls the *computeAccelerations*), since their loops are both iterating through the same variable, so theoretically, the time increase would be exponentially and because of that we had to analyse them further.

B. Profiling

Next up, we used the gprof profiling tool to gain insights into the functions that were the most time-consuming within our code.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 48.76     17.16    17.16      201    85.37    85.37  Potential()
 37.94     30.51    13.35      202    66.09    87.08  computeAccelerations()
 12.05     34.75     4.24 942014880     0.00     0.00  __gnu_cxx::__promote_2
std::__is_integer<int>::__value>::__type>::__type std::pow<double, int>(doub
```

Fig. 1. Code's profile

Anticipated performance bottlenecks in our code were primarily in the *Potencial* and *computeAccelerations* functions. However, we discovered a previously underestimated contributor to execution time: the pow function, which is invoked frequently and consumes a significant portion of the computation time.

II. Second step - Optimizing

A. Pow function

We started by the most basic function **pow**. Pow is a function that receives two arguments: a double **a** and an int **b** and returns a double equal to the result of **a** to the power of **b**.

$$pow(a, b) = a^b \tag{1}$$

During our code analysis, we recognized that invoking the pow function had a notable performance overhead, primarily due to the need to change the frame pointer and the fact that the C compiler does not optimize code from standard libraries. So we went through the code and removed all function calls of pow and replaced them with equivalent multiplication operations.

B. Potencial function

As we had seen earlier *Potencial* has a complexity of $\mathcal{O}(N^2)$, but when analysing further, we also noticed that the loops were impossible to vectorize, because we had a reduce in the loop.

We also realized that there were redundant calculations due to the loop iterating over all pairs of i and j. This meant that for each pair of particles, calculations were being performed twice. Specifically, the code calculated the potential energy between particles i and j, but it also did the same calculation for j and i, leading to unnecessary work. So we changed the j loop from starting from 0 to starting from i+1 and at the end of the function we doubled the result. We also highlighted the number 4 in the formula:

$$Pot+ = 4 * epsilon * (term1 - term2) \tag{2}$$

Meaning that at the end of the function we quadrupled the result. So with these two mathematical simplifications we multiplied the result by 8.

Finally, we unrolled the loop that was calculating the distance between particles (to allow more ILP) and we simplified the calculations for the potential energy between them. Before we add this to represent the potential energy between

two particles, where **r2** is the distance between the particles squared:

$$epsilon * ((\frac{sigma}{\sqrt{r2}})^{12} - (\frac{sigma}{\sqrt{r2}})^{6}) \tag{3}$$

Now we have this:

$$\frac{(1 - r2^3)}{r2^6} \tag{4}$$

Since both Epsilon and Sigma were constants set to one, we simply removed them from the calculations.

C. computeAccelerations function

As for the *computeAccelerations* function, we observed that it exhibited a similar behavior to the *potential* function. Both functions shared identical loop structures, both involved the calculation of the squared distance between two particles and the *potential* function was always invoked with the *computeAccelerations*. Therefore we joined both functions together.

We also simplified the maths, and went from this:

$$f = 24 * (2 * r2^{-7} - r2^{-4}) \tag{5}$$

To this:

$$f = \frac{48 - 24 * r2^3}{r2^7} \tag{6}$$

To minimize redundancy, we consolidated our calculations. Rather than computing the powers of 3 and 7 separately, we computed the power of 3 once and then squared the result, subsequently multiplying it by 'r2' to get the $r2^7$.

D. Flags

Initially, we introduced the optimization flag `-O2` to enhance code performance. Following that, we incorporated the `-funroll-loops` flag to unroll certain loops, but later realized that unrolling was being handled automatically by a vectorization flag we added and consequently removed the `-funroll-loops` flag.

Subsequently, we aimed to further optimize our code by enabling loop vectorization using `-ftree-vectorize` and exploiting the capabilities of the SSE4 instruction set with the `-msse4` flag. Surprisingly, we found that the `-O3` and `-Ofast` flags didn't yield performance improvements, so we retained the `-O2` flag for optimal results.

As we continued to fine-tune our compilation process, we upgraded to the AVX (Advanced Vector Extensions) instruction set with the `-mavx` flag, considering the architecture of our machine with `-march=x86-64`, and optimized floating-point arithmetic operations with the `-mfpmath=sse` flag. Our final compilation flags stood as: `-Wall -pg -O2 -ftree-vectorize -mavx -mfpmath=sse -march=x86-64`.

E. Further optimizations

We optimized the code by transitioning from matrices to arrays for the positions, accelerations, and velocities of the atoms. This change enabled us to vectorize multiple loops more effectively.

Additionally, we made improvements to two other functions within the main loop: 'MeanSquaredVelocity' and 'Kinetic.' In both cases, we eliminated loops that accumulated results in variables, which hindered code vectorization.

To overcome this limitation, we adopted a strategy that facilitated concurrent mathematical operations. Instead of accumulating results in variables, we stored them in arrays, and later, aggregated the results as needed.

III. Third step - Measuring Performance

A. Checking vectorization

To assess the compiler's ability to vectorize our code, we used the `-ftree-vectorizer-verbose=1` flag. This allowed us to gain insights into which loops were successfully vectorized or loop-unrolled.

In the process, we attempted a similar modification in the 'Potential' function, akin to what we applied to 'MeanSquaredVelocity' and 'Kinetic.' However, the verbose flag revealed that the 'Potential' loop wasn't vectorized, likely due to the complexity of the operation and the number of variables involved. Consequently, we abandoned this particular optimization approach.

In the code, we've provided comments to indicate the loops that were successfully vectorized or loop-unrolled.

B. Time of execution

Initially, we relied on gprof for execution time analysis. However, we soon discovered its lack of accuracy. To obtain more precise measurements, we transitioned to using perf stat. This involved running our program three times in succession with the following command:

```
srun --partition=cpar perf stat -r 3 \
-M cpi ./MD.exe < inputdata.txt
```

```
Performance counter stats for './MD3.exe' (3 runs):

   21,742,084,065      inst_retired.any          #      0.7 CPI
   14,882,963,451      cycles

          4.5560 +- 0.0581 seconds time elapsed  ( +-  1.27% )
```

Fig. 2. Performance

IV. Conclusion

In conclusion, our efforts in optimizing the code yielded significant improvements in terms of performance and efficiency. By streamlining the calculations, eliminating redundant operations, and adopting vectorization strategies, we managed to enhance the execution speed of our program (we were able to go from 240s to 4.5s).