Parallel Computing - Report 1

Guilherme Sampaio
PG53851
University of Minho

Luís Pereira
PG54009
University of Minho

***Abstract*—This report explores the optimization of a C-based molecular dynamics simulation code with Lennard-Jones potential.**

***Index Terms*—optimization, c, algorithms, lennard-jones**

I. Introduction

In the first phase of this parallel computing project, we were tasked with optimizing a C code simulating molecular dynamics with Lennard-Jones potential, all while adhering to a constraint of not using multithreading or multiprocessing techniques. This report presents a comprehensive exploration of various strategies employed to enhance the code's performance, focusing on algorithmic improvements, memory management, and code organization. Our approach to optimization prioritizes readability and maintainability, ensuring that the code remains accessible and comprehensible to future developers.

II. Code Profiling and Analysis

The use of `gprof` and `perf` were instrumental in the realm of code profiling and analysis. Both `perf` and `gprof`, are performance analysis tools, they utilize data collected during program execution to identify the functions that are most CPU-intensive. This information is invaluable when seeking to optimize code, as it allows us to determine which functions are the most time-consuming. With `perf` we can record a wide range of events, including CPU cycles, executed instructions, cache missed, and more.

During each iteration of our code, we employed these tools to ascertain whether we were achieving an enhancement in the outcomes, guaranteeing better performance. Every value related to performance presented in this document was obtained via `perf` after taking an average of 10 runs.

With our initial analysis using `gprof`, we discovered that the functions `Potential` and `computeAcceleration` were the most costly, and so we focused on optimizing them. We can see the results outputed by `gprof` on the picture 1.

III. Compiler Optimization

Changing the code to be compiled with the flag *-O3* will increase its performance dramatically since it performs many optimizations including:

- **Loop unrolling**: Reduces CPI by minimizing loop overhead, which decreases the total number of cycles and instructions executed and mitigating RAW[0] dependencies;
- **Inline functions**: Lowers CPI by reducing function call overhead, resulting in fewer total cycles and instructions executed;
- **Vectorization**: Decreases instructions significantly by executing multiple data elements in a single instruction;
- **Register Allocation**: Lowers clock cycles by reducing memory accesses and increasing the utilization of registers.

[0]Read after Write

Many of the optimization presented above have the drawback of increasing code size, although this wasn't seen in practice since many of those optimizations could not be done due to the way the code was written, being a decrease of 2% in the assembly code size. This flag did indeed increase the performance of the program, as we can see in the Table I in the Attachments section.

IV. Math Optimizations

The *cmath* library from *C++* must support powering numbers to any real number and not just integers, this implies that the function *pow* must use algorithms based on the *Taylor's series* or *Logarithms and Exponentiation Series* [1]. In the scope of this project, all powers calculated use constant integers, so calculating powers using multiple multiplications of the same value is possible and far more efficient than using the *pow* function. This will decrease the instruction count since is a far simpler algorithm, and also the L1 cache misses since it's accessing the same value multiple times, which in the cases used will already be in a register, instead of having to load multiple values and constants.

V. Optimizing Lennard Jones Force

The first step to optimize the derivative of Lennard Jones force calculation seen in [2] is looking at the provided code, $f = 24 * (2 * pow(rSqd, -7) - pow(rSqd, -4))$, where `rSqd` is the distance between 2 particles squared. This is done for every pair of particles without repeating, so the force for the pair (i, j) will be calculated, but the (j, i) won't. By applying the rules set at section IV, we are left with optimizing divisions, since they are CPU intensive operations.We ended with the following code:

```
double InvrSqd = 1 / rSqd;
double InvrSqd3 = InvrSqd * InvrSqd * InvrSqd;
double InvrSqd4 = InvrSqd3 * InvrSqd;
f = InvrSqd4 * (48 * InvrSqd3 - 24)
```

VI. Optimizing Potential

The first step to optimize the potential calculation is looking at the provided code used:

```
double rnorm = sqrt(r2);
double quot = sigma/rnorm;
double term1 = pow(quot,12.);
double term2 = pow(quot,6.);
double Pot += 4 * epsilon * (term1 - term2);
```

Here, `r2` is the distance between 2 particles squared. This is done for every pair of particles with repeating, so the force for the pair (i, j) and (j, i) will be calculated. Since $distance(i, j) == distance(j, i)$ we can calculate the potential energy for one of the pairs and multiply by two the energy. We can also multiply $4 * epsilon * 2$ in the end, saving a lot of instructions, and precompute $sigma^6$ since it's a constant.

The refactored calculation will look like:

```
double InvrSqd3 = 1 / (r2 * r2 * r2)
double term2 = sigma_over_6 * InvrSqd3;
double term1 = term2 * term2;
Pot += term1 - term2;
```

VII. Joining Acceleration and Potential

Since between the start of the function `VelocityVerlet` and the start of `Potential` there are no RAW dependencies, we can join them together. Moreover, we will be joining the `computeAccelerations` function that is used within `VelocityVerlet` with the `Potential` functions. This is done because both calculate the distance between the same particles in the same order and both use the value $1/(distance(i, j)^3)$, this way, we only need to calculate it once, decreasing the instructions needed.

VIII. Code Vectorization

One of the major improvements instruction and speed wise was the manual vectorization of the code.

While enabling compiler vectorization directly with `-ftree-vectorize`, optimizes certain parts of the code, the code still needs to be vectorization compliant (i.e. no dependencies on the inner loop, aligned structures, consecutive memory accesses, etc.). This, unfortunately, was not the case for the most expensive function (`computeAcellerationAndPotential`) since the compiler, warned us about some conditional block preventing the vectorization of the code (inner loop dependeds on outer loop). Thus, the idea of manually vectorizing using AVX compiler intrinsics surged.

A. Matrix Transposition

Even when manually vectorizing code, we need to make it vectorizable, so for starters we transposed every matrix used on the `computeAcellerationAndPotential` (a, r and v) function so that each matrix had more lines than columns. Due to the nature of duality in linear algebra, we can freely transpose the matrices without altering the final result of our computation.

Transposing matrices, in this context, enhances computational efficiency by improving cache locality, reducing cache misses and optimizing memory accesses. This is only possible because the transposition ensures that sequential elements are stored contiguously in memory.

B. Matrices Alignment

We also opted to have the matrices 32 bytes aligned. Aligning matrices is not required for us to vectorize using AVX intrinsics since the `immintrin.h` API provides functions to handle unaligned values, but they are slower than their counterparts. The standard AVX register is 256 bits long, this means that we can fit 4 doubles (8 bytes each) in each register, by aligning the matrices to 32 bytes (32 * 8 = 256 bits) we ensure that on each load or store we are "pumping" in or out the full register. This leads to more efficient memory accesses, generating more performance. Although, for this to work we had to change the value of `MAXPART` to a multiple of 4, in this case, from 5001 to 5000.

C. Implementation

Finally, all left to do was to replace each operation (assignments, declarations, additions, multiplications, subtractions and divisions) with the functions provided by the AVX C API. We're using AVX instructions, which allows the processing of up to 4 doubles at each time.

As already said in the introduction, we're not only aiming for fast and optimized code, we're also striving to get maintainable and readable code which we could not achieve by using the default AVX functions. In order to address this issue, we defined a set of C macros that would replace operations and simplify our code, for example:

- Instead of adding 3 vectors with multiple calls of `_mm256_add_pd(a, b)` we would have `add3(a, b, c);`

Regarding the implementation, there is one more detail that we must keep in mind. We are working with loops and processing 4 elements each iteration, the loop might not start in a multiple of 4 so, in order, to be able to process every element, we have to handle the non-vectorizable iterations first with scalar code, and the rest with AVX instructions.

IX. Conclusion and Results

As specified by the assignment, every test took place on the SEARCH cluster using *gcc 9.3.0*. With this in mind, the Table I specifies the results obtained throughout the different stages of optimizations. The version value of each row indicates which optimization aspects were implemented at the time, this means that `(VI.A)` is the program optimized with the techniques used from section `III` to section the subsection `A` of the section `VI`. When comparing with Figure 1 to Figure 2, we can check that we were able to simplify the execution tree, now only depending on two functions which are a lot laster.

ATTACHMENTS

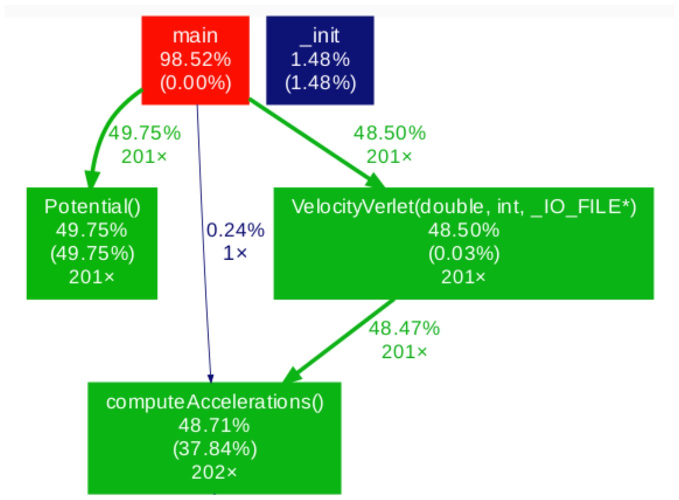

Fig. 1. Initial *gprof* analysis.

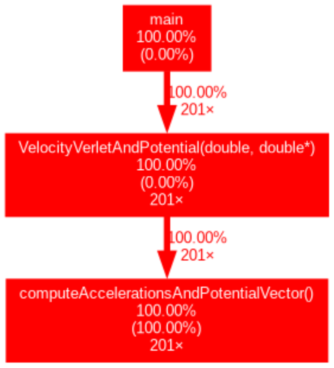
main
100.00%
(0.00%)
100.00%
201×
VelocityVerletAndPotential(double, double*)
100.00%
(0.00%)
201×
100.00%
201×
computeAccelerationsAndPotentialVector()
100.00%
(100.00%)
201×

Fig. 2. Final *gprof* analysis.

TABLE I
FINAL RESULTS

Versions	**Characteristics**				
	#I[1]	***#CC***[2]	***CPI***[3]	***#L1CM***[4]	***#BM***[5]
Default	$1,244,413,423,117$	$933,129,095,653$	0.7	$2,785,913,298$	$436,018,434$
(III)	$990,204,702,647$	$775,112,235,395$	0.8	$2,747,057,224$	$424,322,010$
(IV)	$59,308,772,361$	$48,785,187,300$	0.8	$672,509,629$	$1,751,120$
(VI)	$39,196,685,921$	$33,860,781,137$	0.9	$448,943,100$	$1,204,137$
(V)	$33,955,505,871$	$31,374,140,088$	0.9	$459,573,342$	$1,189,584$
(VII)	$23,629,235,989$	$20,718,331,431$	0.9	$333,763,827$	$697,803$
(VIII.A)	$24,101,871,484$	$16,188,107,501$	0.7	$327,521,253$	$651,865$
(VIII)	$4,763,092,920$	$6,973,788,346$	1	$333,948,377$	$547,030$

[1]Number of instructions.
[2]Number of clock cycles. [3]Number of clock cycles spent per instruction on average.
[4]Number of L1 cache misses. [5]Number of branch misses.

REFERENCES

[1] J. Stewart, *Calculus*. Cengage Learning, 2008.
[2] F. J. Jensen, *Introduction to Computational Chemistry*. Wiley, 2007.