Computação Paralela

Phase 3

Afonso Laureano Barros Amorim
Departamento de Informática
Universidade do Minho
Braga, Portugal

João Carlos Fernandes Novais
Departamento de Informática
Universidade do Minho
Braga, Portugal

***Abstract*—This document gives a summary of the tasks completed in the Parallel Computing course acrosse the three assignments.**

I. Introduction

The main goal of these three tasks was to reduce the total execution time of the given program by applying diverse optimization techniques. The optimization process encompassed several steps, including the manual refinement of the code, the integration of shared memory parallelism through OpenMP, and the creation and implementation of a parallel version of the program using accelerators.

II. Previous Phases

A. Phase 1

While analysing the code, we had in consideration 2 factors: readability and performance.

1) Readability: The original code stored the positions, velocities, etc. as 2 dimensional $MAXPART * 3$ arrays, which was one of the things we decided to approach in terms of readability. To solve that issue, we created a struct called Vect3d, that holds 3 doubles (x, y and z), and changed it so that the 2 dimensional arrays were arrays of Vect3d. In practical terms it makes no difference, given that te memory used will be the same and will be aligned in the same way as well, but it is easier to understand the code this way.

2) Performance: To measure the initial performance, we decided to use gprof to profile the execution of the original code and see where the main performance bottlenecks were. Analysing the output of gprof, we were able to single out 2 functions which ammount to almost 90% of the running time: ***Potential()*** and ***computeAccelerations()***.

Knowing that we decided to look at te code inside those functions and understand what the main problems were, and by doing that we discovered that the reason those functions took such a long time was that they did unnecessarily heavy double computation, and that they didn't take advantage of the temporal locality.

The optimizations we implemented will be explored in the next section.

3) Implemented Optimizations: The optimizations we implemented were the following:

- Loop Unroling
- Loop Optimization
- Change the *pow* function to one created by us
- Vectorization
- Optimization of the Mathematical Equations
- Merging Potential and computeAccelerations

4) Loop Unroling: We decided to use Loop Unroling because it helps take advantage of the L1 Cache and also helps with the Instruction Level Paralelism. Given the Cluster's specifications we are able to store 8 doubles in every one of the L1 cache's rows (64 bytes of size in each row).

We implemented this manually in some sections of our code, in the places where the different components of the particles were accessed in a loop:

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

We also used the compiler flags *-funroll-loops* and *-faggressive-loop-optimizations* to unroll automatically other loops.

5) Loop Optimization: While looking at the code and the context of the problem, we noticed that, in order to calculate the Potential energy, we didn't need to loop through the arrays the way that was being done, because, while comparing positions, the difference between A and B is the same as the difference between B and A, so we could change the loops from:

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

to:

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

, just by doubling the result of each iteration on the latter loop, reducing the number of iterations needed to get the same result, wich would result in less instructions, helping the reducing of the execution time.

Other optimization of the loops was done by doing less data storage inside the loops themselves, for example, storing the acceleration values in local variables while computing the accelerations, and updating the accelaration realtive to the outer loop only in the end of the outer loop iteration, and by minimizing the intermediate double operations, for example, multiplying the potential energy by $4 * \epsilon$ in the end of the function, instead of every iteration.

6) Change the pow function to one created by us: As we said earlier, the code used some heavy double calculation where it wasn't needed, and one example of that is the use of the *pow* function, from math.h, which is a unefficient function considering the context of the program, where the exponentiations used are rather simple most of the time. In order to get rid of the most possible uses of that function, we decided to create our own function, called *myPow*, which is written in the most efficient way possible considering the context of the project (it is recursive and the most common recursion case is defined earlier in the function, minizing the number of conditional jumps).

```
double myPow(double base, int exp)
{
    if (exp > 0)
        return base * myPow(base, exp-1);
    if (exp == 0)
        return 1;
    return 1 / myPow(base, -exp);
}
```

7) Vectorization: Vectorization allows operations that operate with multiple data at once, wich can be helpful in this project when the same operations are done to the different components of a particle.

We achieved vectorization by using the compiler flags *-Ofast*, *-mavx* and *-ftree-vectorizer-verbose=2*.

8) Optimization of the Mathematical Equations: While computing, one should note that floating point operations are really costly (mainly multiplying and dividing), even more while dealing with double-precision floating-point operations, so, optimizing the mathematical equations with ones that require less of those operations is an optimization that can help a lot with performance.

In order to optimize our program, we changed the following equations:

```
// before
rnorm=sqrt(r2);
quot=sigma/rnorm;
term1 = pow(quot,12.);
term2 = pow(quot,6.);
Pot += 4*epsilon*(term1 - term2);
f = 24 * (2 * pow(r2, -7) - pow(r2, -4));

// after
r8 = myPow(r2, 4);
r6 = myPow(r2, 3);
// the multiplication by 4 * epsilon
// is done outside the loop
Pot += (sigma-r6) / (r6*r6);
// the multiplication by 24
// is done outside the loop
f = (2 - r6) / (r8*r6);
```

9) Merging Potential and computeAccelerations: After the changes talked about in section II-A5, we noticed that the loops of the *Potential* and *computeAccelerations* functions did the exact same iterations, so we decided to merge them together into the function *computeAccelerationsAndPotential*, allowing les function calls and reducing the number of iterations in half.

B. Phase 2

The objective of this phase was to optimize our code using some parallelization techniques, learned in the practical classes.

1) Overall Analysis: We changed the N variable to 5000 and it took 10.5439 seconds to finish, and we noticed that, similar to the other phase, most of the time is spent in the ComputeAccelerationsAndPotential function, so it would be in this function that it would be crucial to explore parallelism.

2) Implementation: On an initial phase of the parallelization, we had to start by analysing the code of the function we were changing. The main portion of the code where the program spent the most time where the 2 nested loops that where calculating the accelerations and the potential, so that's where we had to implement our optimizations.

On the outer loop, used the *pragma omp parallel for* directive, so the different threads that were being used on the program would run the for in parallel, using the optimizations we will mention ahead. We also used the *reduction(+:Pot)* and *reduction(addVect3d:a[:N])* clauses.

On the first phase we decided to used the structure *Vect3d* to have a better legibility, but that made it so that we couldn't use any of the default OpenMP reduction operations, so we had to use the following clause to define a custom operation that would be able to make the reduction of the Vect3d array possible:

```
#pragma omp declare reduction \\
        (addVect3d : Vect3d :  \
        omp_out.x += omp_in.x, \
        omp_out.y += omp_in.y, \
        omp_out.z += omp_in.z)
```

We used the *reduction* clause in order to prevent data races on the *Pot* and *a* variables, that would compromise the output of the parallelized program. The way this clause works is by creating different copies of the variables where the reduction will occur, for each different thread, and when all of the work is done, the content of the copies is joined using the reduction operation defined inside the clause, before the colon.

As an alternative to *reduction*, we could have used the atomic and critical clauses, but we didn't because:

- When there are several threads trying to enter a block of those clauses, they wait for the first one that enters to exit, which can lead to a starvation occasion, where a thread waits and ends up being blocked for a long time without be able to perform the task.
- Overhead, since both alternatives have a large overhead (higher in critical than in atomic) when containing the code block. Reduction has it too, but it is just the overhead of joining copies of the variables used in the reduction process by each thread.

While applying multi-threading on a loop, an important task is to decide how the scheduling of the iterations will be made between the different threads. We decided to use the following OpenMP directive for that end:

```
schedule(dynamic, 50)
```

Using this directive, the iterations are divided in chunks of 50 iterations each, and each one of those chunks is attributed to a thread, one at a time, and, when that thread finishes that work, it is given another chunk, which makes the distribution of the workload more balanced, because there will be no threads that aren't working until the loop is fully processed. We chose 50 for the chunk size after some experimenting between different numbers, because that's what we thought was the number that led to a more balanced workload while not having a big overhead.

We opted for the dynamic clause because:

- Using the static clause would lead to the workload being distributed equally between threads, which is something that might not be a good thing because different threads might take different times processing their workload, and that could lead to having threads that would be completely stopped while others would still have half of their work due.
- Using the guided clause could possibly give better results, but it has a really big overhead because of the way it works, because not only the attribution of chunks to threads is dynamic, but also the size of the chunks, and that's what makes it not a viable option, in our opinion.

3) Scalability: We decided to test locally on our machines how the number of threads impacted the program's scalability. The machine we tested on has 4 cpu cores.

We first decided to run the sequential code, as we said in the overall analysis section of this report, in order to be able to rate the optimizations made to the code.

We ran the code with 1, 2, 4, 8, 16, 20, 32 and 40 threads, and got the following results, according to the metrics time, speedup and efficiency:

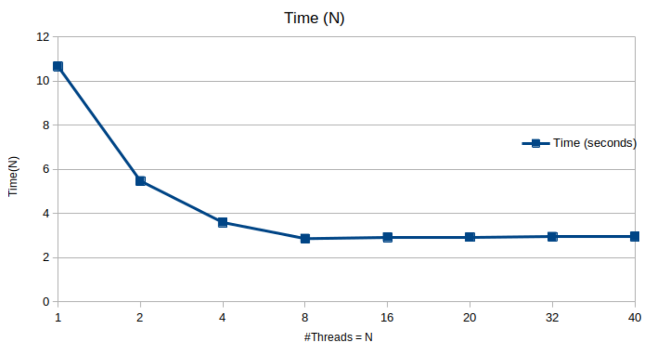

Fig. 1. Time in seconds of each run

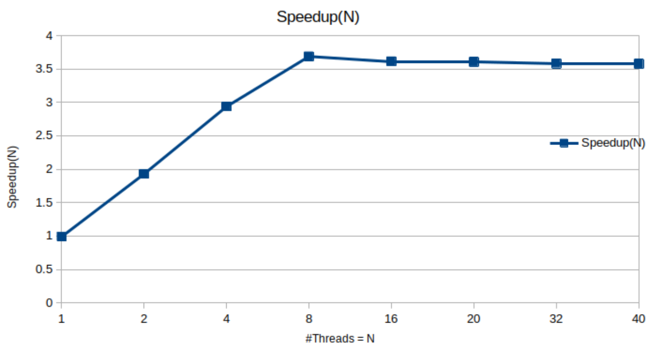

Fig. 2. Speedup(N) = Tseq / Tpar(N)

By analysing the line plots, we can notice that, as the number of threads increases:

- the time decreases, until it reaches a point where it stays the same (in average). This goes in line with what the Amdahl's Law says, that is that only the parallel part of a program can be optimized, and the sequential part of the code stays the same, and that's what thinks happens from the 8 threads onward, where we verify that the time stays the same, because the parallel part of the code can't be more optimized than it already is.
- the speedup increases, also reaching a point where it stagnates, that aligns with the same point where the time also stagnates. Also one thing to point out is that the maximum speedup never reaches the theoretical value, it stays around the 3.6 mark, and the maximum would be 4, as we are in a 4 cpu core machine. That happens because the machine can only support correctly the 4 physical threads it has and 4 virtual ones, and anymore beyond that doesn't change much.
- the efficiency decreases, and that happens because the efficiency calculates how much the threads are being used, and, when the number of threads increases, the total workload stays the same, so the threads might not be working at 100% all the time, and that's why it decreases.

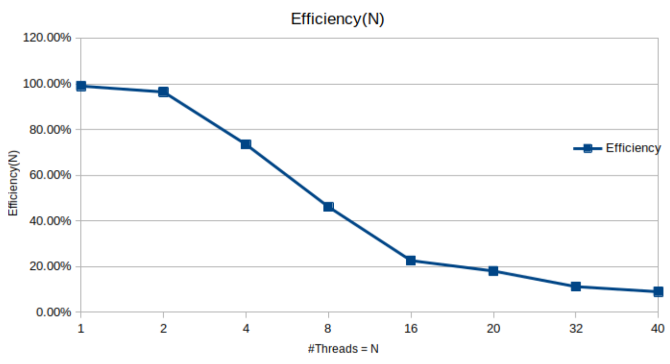

Fig. 3. Efficiency(N) = (Tseq / N) / Tpar(N)

III. Phase 3

A. Proposed work

In this phase we had to choose between correcting/improving the OpenMP implementation from phase 2, designing and implementing a new version with accelerators (GPU with CUDA) or designing and implementing a new version for distributed memory (C with MPI). We chose the second option, working with accelerators, and we chose it because it was the one that incited more curiosity on us.

Analysing the code we decided that we should make a version of the *computeAccelerationsAndPotential* function that runs on the device (GPU), because, similar to the other phases, it is the function on our code that the program spent the most time on, so it should be the one where we focused the optimizations.

B. Implementation

1) Launching the Kernel: Because the *computeAccelerationsAndPotential* function will run on the device, we should have a function that takes care of sending the information needed to the GPU, launching the kernel and retrieving the modified values from the GPU, and this sub section will be dedicated to that.

We decided that we only needed to send the information from 3 buffers to the GPU, with them being the information on the position and acceleration arrays (3-Dimensional points implemented via structs) and the variable where the potential should be written, and we decided to begin with that, defining those variables, alocating memory for them (with cudaMallocAsync), setting their initial values, with them being the current positions, for the copy of the positions (with cudaMemcpyAsync), and setting the initial values of the accelerations and potential as zero (with cudaMemsetAsync).

Before calling the kernel we have to calculate the number of blocks necessary (so that $numBlocks * numThreadsPerBlock >= N$), because we made it so that the program can receive the number of threads per block and the value of N via command line arguments while running the program.

After the kernel finishes we retrieve the final acceleration values and the final value of the Potential energy (with cudaMemcpyAsync), and then finally we free the memory allocated (with cudaFree).

As it can be seen, we used the async versions of the various data copying/allocation functions, always referencing a CUDA stream creating with the call cudaStreamCreate. This allows the asynchronous execution of the kernel code. This isn't that usefull with only 1 stream, and we could have possibly made it so that we could use more streams running concurrently to have better results, but that would make it so that almost double the ammount of data had to be transferred to the GPU, and that would obviously not be good.

2) Kernel: The code that runs on the device is in the kernel, and this section will cover the optimizations inside this function.

Every thread that executes this function will basically make 1 iteration of the outside loop of the *computeAccelerationsAndPotential* function of the other phases, so we had to think about every possible optimization to be done.

Firstly we realized that, by changing the loop iterations from i¡j¡N to 0¡j¡N (if j!=i), we could simply replace the final portion of the inside loop code from:

```
ax += x;
ay += y;
az += z;

a[j].x -= x;
a[j].y -= y;
a[j].z -= z
```

to:

```
ax += x;
ay += y;
az += z
```

where ax, ay and az are the accumulated additions of the distante from every point j to the point i. This makes it so that we don't need to have any atomicAdds calls, which helps a lot with the performance, because no thread will be waiting for resources while running the loop.

The other major implementation inside the function is the used of a shared variable (using __shared__) to hold the Potential calculated in every thread of a thread block. Before exiting the function the thread with id 0 of every thread block will sum the Potential values of the threads in their group and then use atomicAdd to sum that value to the potential value that will be read by the host.

C. Result Analysis

Before talking about the tests and results themselves it should be noted that the tests were made on one of our personal computers. It has a AMD Ryzen 7 3750H CPU with 2.3 GHz of base clock speed and a NVIDIA GeForce GTX 1050 GPU with 3GB of v-Ram. We didn't do the tests on the SEARCH cluster because when we tried to the cluster was down.

As requested for this third phase we also had to make a testing script for the final program, to test its scalability. In this script we made the value of N range between the values 2500, 5000 and 10000, and the number of threads per block range between 16, 32, 64, 128, 256, 512 and 1024, with the number of blocks being automatically calculated. The following graph and tables will show the results of the tests.

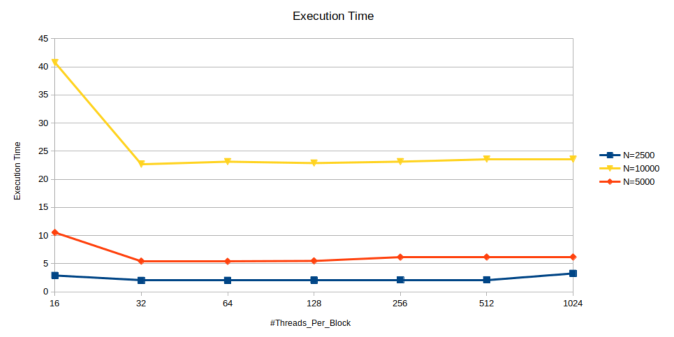

Fig. 4. Time in seconds of the executions of the program for the different values of N and THREADS_PER_BLOCK

Threads per Block	N	Time(s)
16	2500	2.892
	5000	10.529
	10000	40.751
32	2500	2.009
	5000	5.439
	10000	22.673
64	2500	2.021
	5000	5.403
	10000	23.13
128	2500	2.036
	5000	5.485
	10000	22.88
256	2500	2.096
	5000	6.147
	10000	23.134
512	2500	2.11
	5000	6.164
	10000	23.587
1024	2500	3.262
	5000	6.168
	10000	23.547

TABLE I
EXECUTION TIMES FOR THE PROGRAM WITH DIFFERENT N AND THREADS PER BLOCK VALUES

Analyzing the values we can see that with a high N value and a low number of threads per block the application is really slow, compared to the other times for the same N. That happens because having too many thread blocks with not that much threads per block doesn't take full advantage of the GPU, and that happens when the number of blocks is near the same as the number of multiprocessors, and the threads per block are a multiple of 32.

When the number of threads per block is 32, 64 and 128, we verify that the times are the best, because now, with this numbers, the threads per block are a multiple of 32, and it makes it so that the number of blocks is the ideal value for the number of multiprocessors on the GPU.

When the number goes above those, we noticed that the time starts going up. We think that happens because now the shared memory buffers start to have a lot of information, so the kernels take longer to allocate that memory and also fill it.

IV. Conclusion

In conclusion, this project provided a practical application of the knowledge gained from theoretical classes, allowing us to optimize a molecular dynamics simulation code effectively. The implemented techniques, ranging from loop unrolling to OpenMP parallelization, demonstrated the importance of algorithmic improvements in achieving better performance.

The introduction of CUDA in the third phase marked a significant leap, enabling us to harness the power of GPU acceleration. This step not only showcased the relevance of hardware-specific optimizations but also emphasized the growing importance of utilizing accelerators for scientific computing.

In retrospect, now in a more introspective tone, we think that our work, in general, was really satisfactory. We gained knowledge in regards to multithreading, code analysis and even the usage of GPU accelerators, but we think we could've maybe explored a little bit more about the latter one. We think that the work done in this third phase could've been a little bit better if we had pushed ourselves a little bit more, but we are still happy with the final results.