Parallel Computing-Work Assignment Phase 2

1stHugo dos Santos Martins
Universidade do Minho
a95125
Barcelos, Portugal
a95125@alunos.uminho.pt

2nd João Bernardo Teixeira Escudeiro
Universidade do Minho
a96075
Valpaços, Portugal
a96075@alunos.uminho.pt

***Abstract*—This document is a brief report that explains all stages covered since the initial until the final code. It has a discussion about the optimization of a program that initially was single-threaded and all the changes the group has done in order to improve the code performance, maintaining the code legibility.**

I. Introduction

The code we were supposed to analyse and improve was the code changed in the previous phase. The content given in classes helped us understanding that we would have to include threads,and we distrust that without changes,the heavier function and the spot where we could possibly make the bigger changes was the function **computeAccelarations()**.

II. Initial Analysis

After taking a quick look through the objectives of the second phase we realised that we had to use the code that was previously changed for us in the previous phase. As there were a few improvements to be done, we decided to start changing some redundancies we had in the code. The first thing to be done was changing the number of atoms **N**, that was 2160, and now is 5000. With this changing, it's expected that the program is going to become computationally heavier and slower. Given this, we decided to test the result code from the previous phase with this values.

A. Profiling

Taking advantage of some tools that are within our reach to test the program's performance, we used some of them in order to see check inconsistencies and to test wether function was taking more time to execute. Using **GProf** we got this initial results.

Given the results of **GProf**, we realised that the function **computeAccelarations()** hampers the program's performance.We knew that we had to work on the function using metrics as well as some new libraries in order to try to refine the function, as we know that is the key to improve the program's performance.

III. Optimizations

The first step we decided to take was improving some functions from the previous phase. We noticed that both functions **MeanSquaredVelocity()** and **Kinetic()** had similar type of iterations, and we could reduce both into one singular function, only by changing the way that both variables **mvs** (mean squared velocity) and **KE** (Kinetic energy) were calculated.Both were reduced into a single function named **MeanSquaredVelocityKin()** which reduced the number of loops, instructions, and the complexity of calculations. Similarly to the contents given in classes we knew that one change that could improve a lot the code performance was the inclusion of multithreading and parellelism through the use of the **OpenMP** library.

We decided to focus most of our attention on the **computeAccelerations()** function, which was the function that would be taking up most of the time (as we can see in Figure 2) and that could be improved through parallelism.

In these function there is a loop that initialises the value of all the **a** array positions with 0. This array will be changed during all the process of calculating the Potential Energy. We decided to add the primitive **omp parallel for** in order to parellize the initiallization loop of array a. Then, in the loop where all the values are calculated we added also the same primitive with the aim of minimizing the global execution time of the function, adding multiple threads and trying to break down the iterations both loops into independent tasks that can be executed simultaneously, typically across multiple processor cores. We didn't add any other primitive, and so it was expected results would be wrong due to the occurence of data races(concurrency error that occurs in a program when two or more threads or processes access shared data concurrently, and at least one of them modifies the data), when doing operations that had dependencies of Read After Write and Write After Read while accessing the variables without access control. Despite the fact that the time reduced significantly, the results,as expected, were wrong. Realising this, the group tried to find possible places where dataraces could happen. Easily one of them was found : changing the **a** array values.

Then we decided to add some primitives seen in classes in order to avoid data races derivatives from this plus and minus operations. The first we used was **atomic** before each calculation that accesses the array. The results were still not correct.After this we also noticed that one variable **Pot** was could also be a spot where threads can access without competition control. The clause **reduction(:+Pot)** specifies that variable **Pot** should be private to each thread and its values are accumulated across all threads using the + operator at the end of the parallel region.

Given this we saw that all results were now correct although the time was still a bit long(up to 17 seconds in cluster).As we learnt in practical classes, analysing the three primitives (private,reduction and atomic) the one with better performance is definitely the reduction one .We decided to take a look through the lecture slides and find how to reduce an array. We added the clause and obsiously removed the atomic operations added before. This change had a lot of impact in our code performance, since the code execution time had a decrese from 17 seconds to 7 seconds using only 8 threads. We could see improvements and kept on trying to optimize, with the desire of having a program as fast as possible, with the maximum amount of parallelism. In this phase we thought it was a good idea to test our code through the cluster with the **script** given and see the multiple results we achieved with our code optimizations. The results we got are in the Table below.

Number of threads	8	12	16	18
Time	10.655	7.314	6.137	4.988

TABLE I
EXECUTION TIME AFTER REDUCING THE ARRAY.

We could realise a pattern where as the number of threads increased, the performance of the program increases as well.It is expected that , as we learnt in theoretical classes, at some point, the program reaches the best performance and adding more threads will not adding any benefits to the code performance.

Although we had some good improvements, we knew we could reach a little bit less time.We kept on searching and we found another **OpenMP** clause named **schedule** that allows specify how loop iterations are divided among threads. We used all the different types of scheduling, as auto,dynamic,static or even guided, but the one we saw the best results with was **schedule(runtime)** that allows the scheduling type and chunk size to be determined at runtime rather than at compile time. This provides flexibility as the scheduling decision can be made dynamically based on the execution environment and input parameters.The other primitives didn't have the best load balancing, as we noticed , for example when using **Dynamic** with the chunk size of 40.

A. *Flags*

In this phase we included **-fopenmp** which enables support for the OPENMP , **-march= native** march=native that instructs the compiler to generate code that is optimized for the host machine's architecture, **-funroll-loops** which instructs the compiler to unroll loops, **-finline-functions** that suggests to the compiler that it should attempt to inline functions marked as inline and **-fomit-frame-pointer** that instructs the compiler to omit the frame pointer for functions that don't require it.

At this point and as we were quite satisfied with the results we decided to improve code legibility and to try and evaluate code performance.

IV. MEASURING PERFORMANCE

When testing our code performance in the **Search Cluster** the best result we got was close to 1.9 seconds, that is pretty decent when comparing to the sequential code, as we can see in the table II.

Number of threads	32	35	37	40
Tempo	1.841	1.821	1.817	1.836

TABLE II
FINAL EXECUTION TIMES IN CLUSTER

An important point to be mentioned is the improvements of time since the sequential code until the final parallel code. We created a plot in python that was based in our time results in cluster. In Figure 3 we can see the graph and how the number of threads influences the execution time . We used the **Amdahl Law** to calculate the speed up we got using parallelization. Analysing this graph in detail, we can observe a pattern. From 0 to 20 threads the speed up grows in a linear way. We have a total speedup from 0 to 17 percent.Comparing to the expected speed up it is a little bit below. At some stage close the 20 threads until 40 we realise that time stops reducing, probably because some overhead that is introduced with parallelization and if the computational workload is not large enough to justify the overhead, it may offset the benefits of parallelization(Resource Constraints). We also tried to use another primitives on other functions for example **velocityVerlet()** but we thought this could be worse when we think of load balancing between threads and consequently.

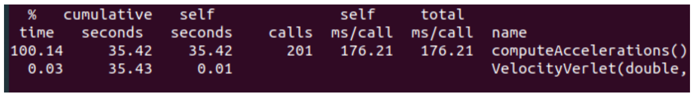
V. CONCLUSION

In conclusion, the selection of omp parallel for, reduction, and scheduling(runtime) in our parallel programming approach reflects a strategic balance between simplicity, data consistency, and adaptability. The omp parallel for directive efficiently parallelizes loops, providing a straightforward means of exploiting parallelism. The use of reduction ensures thread safety when updating shared variables within parallel regions, enhancing data consistency. Additionally, the dynamic workload balancing achieved through scheduling(runtime) allows the system to adapt to varying computational demands, optimizing resource utilization. This combination of OpenMP primitives offers a comprehensive solution for achieving efficient parallel execution without compromising on key aspects of parallel programming. These choices not only streamline development but also pave the way for scalable and responsive parallel applications in dynamic computing environments.

VI. NOTE

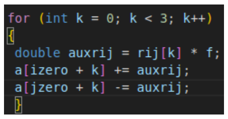
We added a **make runparscript** that helps to run the code with the script provided. We kept **make runpar** as it was provided in the original Makefile. The runparscript uses Sbatch and provides a "slurm.out" file where execution times for each number of threads in the script is specified.

VII. Attachments



```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.14     35.42    35.42      201   176.21   176.21  computeAccelerations()
  0.03     35.43     0.01                             VelocityVerlet(double,
```

Fig. 1. Initial code profile.



```
for (int k = 0; k < 3; k++)
{
 double auxrij = rij[k] * f;
 a[izero + k] += auxrij;
 a[jzero + k] -= auxrij;
}
```

Fig. 2. Local with possible data races when acessing array a.

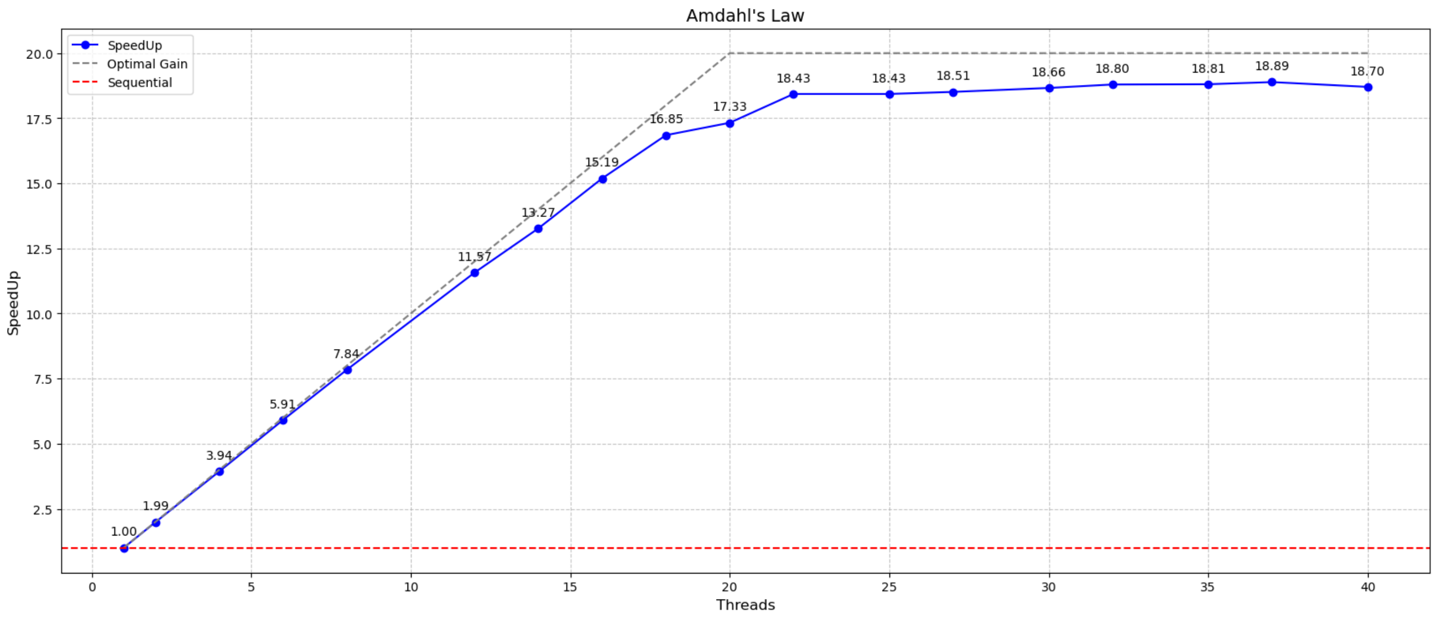


Fig. 3. Threads vs SpeedUp