Work Assignment - Phase 2

1st João Brito
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
pg53944@alunos.uminho.pt

2nd Paulo Oliveira
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
pg54133@alunos.uminho.pt

***Abstract*—This report documents the second phase of the project, focusing on the implementation of shared memory parallelism using OpenMP directives to enhance the computational perfomance of the code developed in phase 1. The main objetive in this phase is to reduce the overall execution time of the code through parallelization. The methodology involves identifying performance bottlenecks, exploring parallelization alternatives, selecting an approach supported by scalability analysis, implementing the chosen approach on the Search cluster, and finally, evaluating the performance of the parallel solution.**

***Index Terms*—Scalability, parallel, threads, OpenMP, shared memory**

I. Introduction

Shared memory parallelism with OpenMp directives is an approach to better utilize computations capabilities of modern multicore and multiprocessor systems. With this, parallele programming has become essential for achieving optimal performance in software applications.

In this report, we explore the fundamental concepts of shared memory parallelism and the utilization of OpenMP directives to facilitate parallel programming. By exploring the foundations of parallel computing and the implementation of pragma directives, which are special annotations inserted in the code to convey parallelism information to the compiler, specifying parallel regions, loops, and sections of code that can be executed concurrently, we aim to achieve a comprehensive understanding of how OpenMP can be leveraged ton enhance the performance of software applications. With this, we will follow the methodology previously outlined to develop parallel programs.

II. Application Bottlenecks

In the context of computing, bottlenecks or hot-spots refer to specific areas or portions of a program's code that consume a disproportionately large amount of computational resources, such as CPU time or memory. Since Identifying and adressing these hot-spots is essencial in execution speed and efficiency, we started out by exploring the code and figuring out which portion of the program would expense a larger ammount of resources.

Upon identifying these hot-spots by adding more detailed logging to the code for testing and coming up with the conclusion that the functon **computeAccelerations()** was taking up the most ammount of resources we decided on the best approach to handle them within the realm of parallelization. This approaches consisted in using parallelization constructs or libraries for the programming language. Some examples consisted in OpenMP directives for shared memore parallelism, MPI for distributed memory parallelism, or GPU programming for prallelism on graphics processing units.

III. OpenMP directives for shared memory parallelism

Having decided on using OpenMP directives for shared memory parallelism we started understanding the otions we had, how to implement them and the benefits of using such options such as the scalability it'd offer to the application. The scalability in the context of parallel programming refers to how well a program can efficiently utilize additional resources (such as processores or cores) as they are made available.

OpenMP provides several constructs such as **omp parallel for**, **reduction()**, **atomic**, **critical**, within others that would greatly benefit the performance of the code using multiple threads while keeping in mind the possibility of **data inconsistencies**, **race conditions**, **limited performance**, **dificulties in load balance** and offering the best possible parallelism.

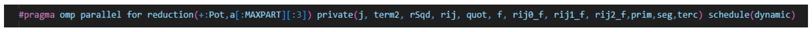
With that being said, we looked at directives such as atomic and critical to deal with some of the pre-mentioned issues but soon realised that these options and directives, while ensuring correctness, may introduce serialization and contention, potentially limiting the code's scalability. We resorted then to finding better scalable options that would deal with all possible problems but would also bring the best scalability and performance.

For our directives we used **omp parallel for** which is used to parallelize loops, instructing the compiler to creat a team of threads to execute the associated loop in parallel, making each thread work on a subset of loop iterations, allowing the program to take advantage of the available multicore architecture. This construct is generally good for scalability when the loop workload is well-balanced. It efficiently distributes work among threads, and the parallel speedup is close to the ideal case as more cores are added.

Secondly, we use the **reduction()** clause which is applied to the variables "*Pot*" and the matrix "*a*". This clause specifies that each thread mainstains a private copy of these variables, and at the end of the parallel region, the private copies are combined using the specified reduction operation ("+") in this case. The use of **reduction** is crucial for correctly aggregating results from multiple threads without data races or contention issues. It ensures that the final values of ”*Pot*” and ”*a*” reflect the contributions from all threads. This clause generally enhances scalability, but but the choice of reduction operation and the nature of the reduction variable can influence scalability.

Additionally, we employ the **private** clause which designates local, thread-private copies of the specified variables. Each thread has its private version of these variables, preventing data interference between threads. This is essential for correctness and avoiding race conditions, contributing positively to the programs scalability by allowing threads to operate on their own local data without contention, therefore reducing contention and improving parallel execution.

Lastly, we utilize the **schedule(dynamic)** directive that controls how loop iterations are dynamically assigned to threads, making them receive chuncks of work dynamically, potentially leading to improved load balancing. In terms of scalability, it is positively influenced by *dynamic scheduling* in scenarios where the workload for each iteration varies, where threads that complete their work can dynamically obtain new chunks distributing the load more evenly among threads.



```
#pragma omp parallel for reduction(+:Pot,a[:MAXPART][:3]) private(j, term2, rSqd, rij, quot, f, rij0_f, rij1_f, rij2_f,prim,seg,terc) schedule(dynamic)
```

Fig. 1. Directive used in program

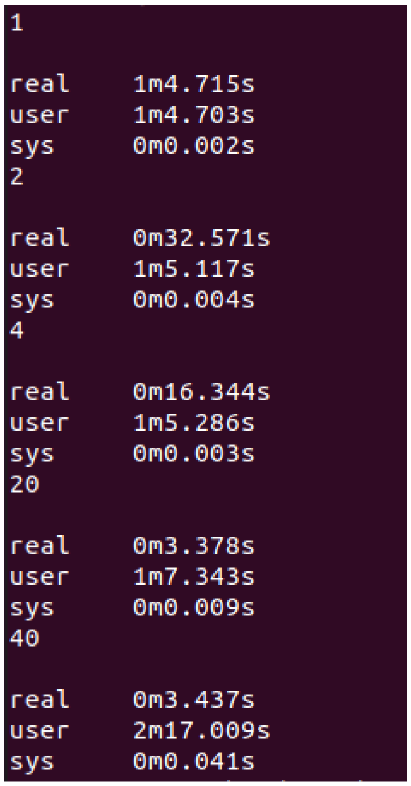
IV. Results Analysis

As shown in figure 2, when employing the **test.sh** script provided by teachers, the recorded execution times obtained for 1 thread, 2 threads, 4 threads, 20 threads and 40 threads were 65.715s, 32.571s, 16.344s, 3.378s, 3.437s, respectively. It’s apparent that the execution time improves up to 20 threads, but there is a slight increase in time beyond this point. This increase suggests a saturation point in the algorithm’s scalability, potentially due to factors such as unever distribution of work among threads. When threads complete their tasks quickly and become idle, there is a risk of load imbalance and underutilization of resources. However, we don’t consider that load imbalance is the case in our algorithm because, when we defined the schedule clause by dynamic, the execution time improved.

Another factor contributing to scalability challenges is the waiting time incurred by tasks, this may cause scalability problems due to dependencies, this case is known as synchronisation overhead.

Other reason, might stem from the management of tasks required in the parallelization process, involving additional workload. If the granularity of tasks becomes excessively large, the algorithm may encounter scalability issues.

Amdahl’s law provides another perspective on scalability problems, stating that if a significant portion of the code is not parallelizable and remains sequential, there is a limit to the performance gain. Upon analyzing our code, it becomes evident that this limitation significantly impacts the scalability of our program. The **computeAcceleration()** function, which consumes the majority of computational time, requires sequential execution of mathematical calculations for correct results, limiting potential performance gains.



```
1

real    1m4.715s
user    1m4.703s
sys     0m0.002s
2

real    0m32.571s
user    1m5.117s
sys     0m0.004s
4

real    0m16.344s
user    1m5.286s
sys     0m0.003s
20

real    0m3.378s
user    1m7.343s
sys     0m0.009s
40

real    0m3.437s
user    2m17.009s
sys     0m0.041s
```

Fig. 2. Results from test.sh

V. Conclusion

This phase enabled the group to apply the concepts taught in the subject classes by implementing an optimized parallel solution. The group is confident that the presented solution represents our best effort, validated through various tests. Ultimately, it became evident that the program’s scalability is not optimal. Despite utilizing a higher number of threads (40), the execution time remains comparable to that achieved with only 20 threads. In this report, the group endeavors to justify the reasons behind this observed phenomenon.