Parallel Computing - Work Assignment 2

Gabriela Santos Ferreira da Cunha
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg53829@alunos.uminho.pt

Nuno Guilherme Cruz Varela
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg54117@alunos.uminho.pt

***Abstract*—This document consists in presenting and justifying the optimisations that were made with OpenMP directives to the code that was developed by the team on the first assignment.**

***Index Terms*—optimisation, shared memory parallelism, openmp, thread**

I. Introduction

In the prior work assignment, we optimised the code of a molecular dynamics simulation using the Lennard-Jones potential with algorithmic refinements, memory management and code organisation. The goal of this second work assignment is to explore shared memory paralellism with OpenMP directives and implement threading techniques in order to improve the performance of the code developed on phase 1.

II. Sequential Code Updates

Regarding the code that was developed on phase 1, we found an improvement that could have been done so we decided to update our sequential code to include that small optimisation.

To start implementing the OpenMP directives, we also decided to remove manual vectorisation from the code due to the fact that it would be easier to visualize what we had to improve. We thought about adding this vectorisation again eventually, but since, at this stage, we gave more importance to readability instead of execution time, we decided to keep the code with automatic vectorisation as it is much more readable than the manually vectorised one.

According to Lennard-Jones potential, ϵ (field depth) and σ (distance where the field is null) are specific for each material but commonly set to 1 in simulations.

$$\phi(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}] \tag{1}$$

In relation to our code, it is important to remember that the number of iterations was halved so in each iteration we were calculating $2 \times \phi(r)$ based on two terms that were pre-computed.

$$term1 = \sigma^{12} \times 8\epsilon \tag{2}$$

$$term2 = \sigma^{6} \times 8\epsilon \tag{3}$$

$$\phi(r) = (term1 \times \frac{1}{(r^2)^6}) - (term2 \times \frac{1}{(r^2)^3}) \tag{4}$$

Since $\sigma = 1.$ and $\epsilon = 1.$, by identity property there is no need to multiply them. Also, we do not need to calculate $\frac{1}{(r^2)^6}$ previously, since we have $\frac{1}{(r^2)^3}$ in evidence. This way, potential is now calculated by the following formula:

$$\phi(r) = 8 \times \frac{1}{(r^2)^3} \times (\frac{1}{(r^2)^3} - 1) \tag{5}$$

III. Code Analysis

To reduce the execution time of the most critical sections of our code, i.e. the ones who are providing a perfomance bottleneck, we must identify them first. For that, we generated a call graph once again using *gprof*.

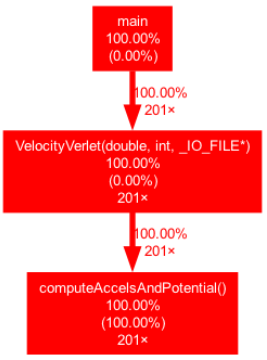

Fig. 1: Hotspots.

Examining the call graph obtained in figure 1, we see that the `computeAccelsAndPotential` function is taking 100% of the execution time. Hence, our goal is to reduce this bottleneck, trying to develop parallelism there.

IV. Parallelism Implementation

The identified hot-spot has 2 nested `for` cycles which iterate over all pairs of particles. For each iteration, both potential accumulator and accelerations array are updated. We must pay attention to these updates because they will lead to data races.

The selected approach consists in exploring parallelism in the outer cycle. In this approach we start by introducing the directive `#pragma omp parallel for`. This allows the compiler to distribute the loop iterations among multiple threads. However, as we mentioned before, we have to take a

look at the potential and accelerations structures because they will be shared by the threads and this will cause data races.

The OpenMP offers 3 directives to solve data races: `reduction`, `atomic` and `critical`. We aimed to use the `reduction` due to its better performance comparing to the other ones.

A. Atomic and Critical Directives

The first approach to deal with the data races was using `atomic` and `critical` directives because of their simplicity. Although, as we expected, it was not a good strategy due to the overhead associated with synchronisation mechanisms. The excessive use of these directives leads to performance degradation because threads spend more time waiting for access to the shared resources than actually performing useful work. In this case, the parallelised version had higher execution times than the sequential one as the referenced directives were used excessively.

B. Reduction

To solve this question in an efficient way, we decided to use a `reduction` in the potential accumulator and in the accelerations array by introducing the directive `#pragma omp parallel for reduction(+:potentialAcc, a)` which avoids the data races. The `reduction` primitive creates a private copy for each thread and in the end of the parallel section the results are reduced into a single variable.

C. Scheduling

In OpenMP, we can instruct the CPU to use scheduling strategies which determine how the iterations or tasks are allocated to threads during the execution of a parallel region. Static scheduling is commonly used as the default for parallel loops. This means the compiler determines the distribution of loop iterations at compile-time. The loop iterations are divided into chunks, and each thread is assigned a chunk of iterations to process.

On the other hand, in dynamic scheduling, the distribution of loop iterations is determined at runtime. Threads request work from a shared pool, and the runtime system assigns them a small unit of work. Though there is some overhead associated with the runtime system managing the distribution of work dynamically, this approach ensures better results in situations where the workload of iterations is uneven. Thus, `schedule(dynamic)` was added to the `reduction` primitive.

V. Results Discussion

To analyse the performance of a parallel computing system as the problem size or the number of processing elements increases, we can employ several techniques and methodologies.

Focusing on general performance, we should analyse scalability. Weak scalability focuses on keeping the problem size per PU fixed and increasing both the problem size and the number of PUs proportionally. The total amount of work per processing element remains constant, so does the execution time ideally. Conversely, strong scalability focuses on keeping the problem size fixed and increasing the number of PUs to observe how the solution time changes. In this approach, ideal speedup is proportional to the number of assigned physical PUs.

In an effort to provide a strong scalability analysis, the figure 2 represents the relationship between the number of threads that were used and the speedup ($\frac{Texec_{best\_sequential}}{Texec_{parallel}}$). Firstly, we visualize that speedup tends to grow linearly until it reaches 20 threads. The number of cores of the machine does not allow this continuous linear relationship between the variables, since the ideal speedup is proportional to the number of assigned physical PUs. In the Search machine, we therefore expect the limit to be 20, as that is the number of available physical cores. We also conclude that the speedup is not ideal and this happens thanks to different reasons: percentage of serial work, memory wall, parallelism/task granularity, synchronisation overhead and load imbalance.

Percentage of serial work (Ahmdal's Law)

According to the Ahmdal's Law the maximum speedup is limited by the percentage of code non-parallelisable. Therefore, since we are only parallelising the hot-spots and there are still some pieces that could not be parallelised, we are awared that we can not achieve the maximum speedup due to this serial work.

Synchronisation overhead

The speedup can also be limited by the synchronisation overhead of the primitives used. In the initial approach to avoid data races, we used the `critical` and `atomic` directives. However, the performance is much lower than the final version using the `reduction` primitive. This occurs because of the synchronisation overheads explained before in section IV-A. The only synchronisation overhead is the initialization of the copies and results combining in the end which is much smaller than the overhead introduced by primitives referenced before.

Load imbalance

Relatively to load balancing, it is important to understand that each iteration i of the outer loop will iterate over a new cycle starting at $i+1$. Consequently, the first iterations of the outer loop will have more work than the final ones. For this reason, the static scheduling is non-optimal on this algorithm. Taking a look at the figures 3 and 4, we confirm that we obtain smaller execution times with dynamic scheduling instead of the static one.

To summarize, with the selected parallelism approach, we achieved gains on scalability that are not too far from the ideal. The best speedup was achieved using 20 threads and corresponds to 16.916.

VI. Attachments

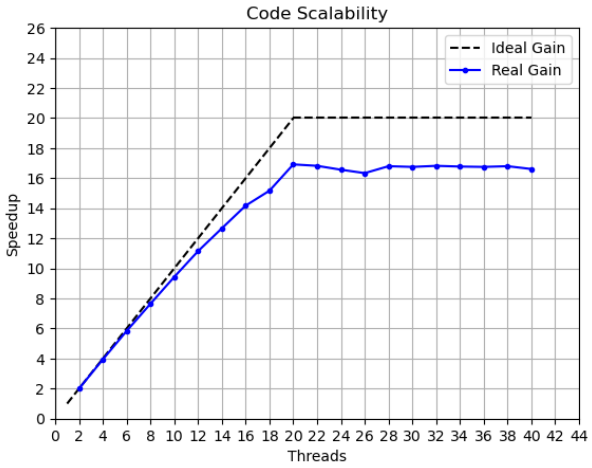

Fig. 2: Code scalability.

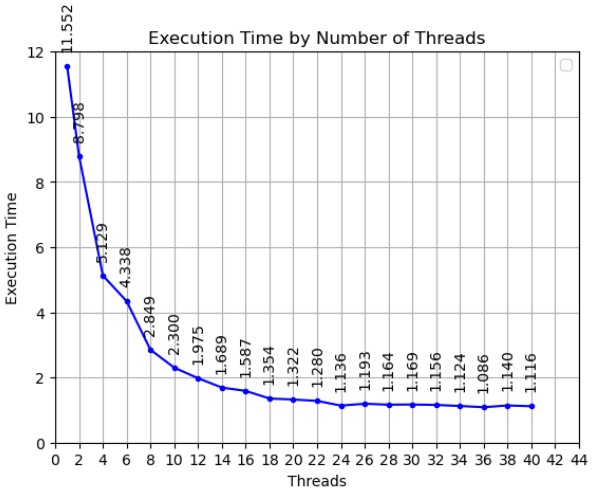

Fig. 3: Execution time by number of threads with static scheduling.

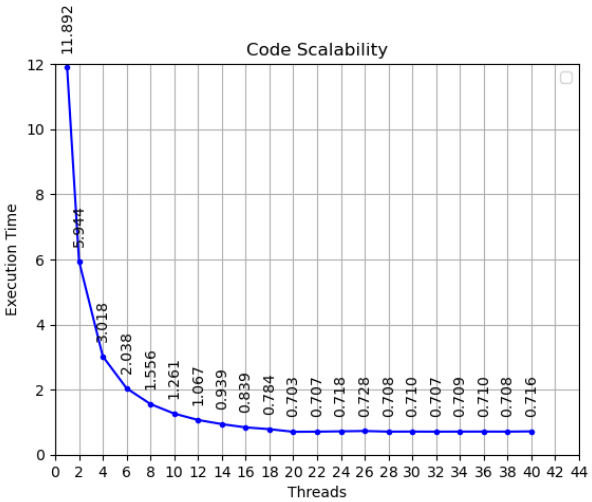

Fig. 4: Execution time by number of threads with dynamic scheduling.