Performance Optimization of a Molecular Dynamics Simulation

Catarina Felgueiras
Physics Engineering Student
Minho's University
Braga, Portugal
a100506@alunos.uminho.pt

Luís Silva
Physics Engineering Student
Minho's University
Braga, Portugal
a96534@alunos.uminho.pt

***Abstract*—The main objectives of the project are to explore shared memory parallelism (based on OpenMP) to enhance the overall runtime of the code from the previous stage and to analyse the scalability of this implementation.**

Keywords—Optimization, performance, analysis, time, execution, OpenMP, multithreading

I. Introduction

In this second phase, to improve the runtime of the code developed in the first phase by exploring shared memory parallelism with OpenMP directives, several essential steps were taken as a foundation: identifying code blocks with high computational time, analysing and presenting alternatives to exploit parallelism, selecting an approach, implementing, measuring, and discussing the performance of the proposed solution. Finally, an analysis of the stability of this implementation was conducted, justifying variations based on the number of threads.

A. Settings

All code builds in this stage were conducted using version 11.2.0 of the GNU Compiler Collection, better known as GCC. All measurements in this report were selected using the k-best 5 method, involving the selection of a set of 5 measurements with a precision of 5%. This means that, in addition to the presented measurement, there are four more measurements whose relative percentage deviation is less than 5% of the same value presented. All measurements were performed on SeARCH (Services and Advanced Research Computing with HTC/HPC clusters) on an INTEL x86-64 node, in the Ivy Bridge segment in the cpar partition, with two Intel Xeon E5-2670v2 processors (22 nm lithography). Each processor has 10 cores and 20 threads, so in this partition, 2x20 CPUs are available [1].

B. Sequential Code

In this stage, the base code will be the code from the first phase of the project, however, the number of particles has been adjusted to 5000 particles.

TABLE I. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
179 337	0,65	116 471	2 752	46,593

Fig. 1. Table with the selected measurement of various parameters related to the sequential code (the remaining measurements are attached).

As the number of atoms was increased, the parameters mentioned above increased relative to those presented in the first phase (with the exception of CPI). Based on the execution profile of the sequential code, it was observed that the *computeAccelerations* function consumes 61.63% of the execution time, followed by the *Potential* function, which occupies 32.64%. Considering this information, the initial approach is to explore parallelism in these two functions.

II. Parallelization with OpenMP

Let's analyse, in a general manner, the key points of the *Potential* and *computeAccelerations* functions: the loops in both functions have high complexity with a large number of instructions, so the pragma omp parallel for directive is applied to ensure distribution of iterations among multiple threads. The variables used within the loops are defined within them to eliminate data races, a phenomenon where different threads access and modify the same variables simultaneously, leading to unpredictable results. By defining the variables locally, each thread operates on an independent copy of the variable in question, avoiding data races and ensuring consistency in the results.

A. Potential

As there is an accumulative variable within the loop, it cannot be defined inside the loop. To eliminate data races, the clause reduction(+:Pot) is applied. This command creates a copy of the accumulative variable for each thread, and at the end of all thread iterations, it sums the accumulated value to the accumulative variable defined at the beginning of the function.

The proposed approach prevents issues when multiple threads (independent units of execution) are working with the same variable simultaneously. Without this approach, if threads read a variable simultaneously, conflicts can arise. In other words, if one thread reads the value of a variable at the same time as another, the operations in the ALU (Arithmetic Logic Unit) will be performed relative to the same value, and changes made by another thread are lost.

B. ComputeAccelerations

The approach for parallelizing the *computeAccelerations* function was identical to the one explained earlier. In this case, the reduction(+:a) is applied, where the difference lies in creating a copy of the buffer for each thread, and this copy is accumulated at the end of all iterations.

TABLE II. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
224 585	1,41	316 137	2 729	4,330

Fig. 2. Table with the selected measurement of various parameters related to the code parallelized with OpenMP (the remaining measurements are attached).

After applying parallelism to the two functions with the highest computational load, a 90.71% reduction in execution

time was observed compared to the previously measured time. Reassessing the code profile, the *computeAccelerations* function now uses 44.71% of the execution time, a *gomp_team_barrier_wait_end* function occupies 25.31%, and the *Potential* function (17.51%) is accompanied by the *gomp_barrier_wait_end* function at 10.43%. The main objective now is to reduce the computational load of the *gomp_barrier_wait_end* and *gomp_team_barrier_wait_end* functions.

III. Load Distribution

Analysing the profiling of the execution of the previous code, we conclude that there is an implicit barrier related to OpenMP's parallelism. This implies that some threads take longer to finish their work compared to other threads. Knowing that the assignment of threads to iterations is done in blocks, where the number of blocks corresponds to the number of threads, and:

```
for (int i=0; i<N-1; i++)
    for (int j=i+1; j<N; j++)
```

Fig. 3. Cycles of the *Potencia*l and *computeAccelerations* functions.

Thus, we can consider that for the first iterations compared to the last ones of the outermost loop, there is a significant difference in the number of iterations of the innermost loop. Therefore, the thread responsible for the first chunk has more overhead than the thread responsible for the last pack. The difference in the number of iterations is approximately (calculation attached),

$$\left(\frac{particle\ number}{threads\ number}\right)^2 - \frac{particle\ number}{threads\ number} \qquad (1)$$

Therefore, through the loop schedule, we can better control the load distribution among threads. The simplest implementation is a schedule(dynamic) because iterations are assigned based on thread availability. On the other hand, in this implementation, there will be time dedicated to thread assignment. This time could be reduced by using a schedule(dynamic, chunk size), where iterations are dynamically assigned in blocks. However, it would create instability in scalability as the number of threads increases, and since the improvement is not significant, only the schedule(dynamic) will be implemented.

TABLE III. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
192 132	1,33	256 807	2 320	2,576

Fig. 4. Table with the selected measurement of various parameters related to the code parallelized with load distribution (the remaining measurements are attached).

After applying load distribution to the two functions, there was a decrease in all parameters, including the execution time, which decreased by 40.49% compared to the previous time.

IV. Scalability Analysis

Considering the multithreading optimization implemented, it is necessary to analyse the scalability of our code. Thus, with multiple measurements, the following graph of the gain as a function of the number of threads is obtained:

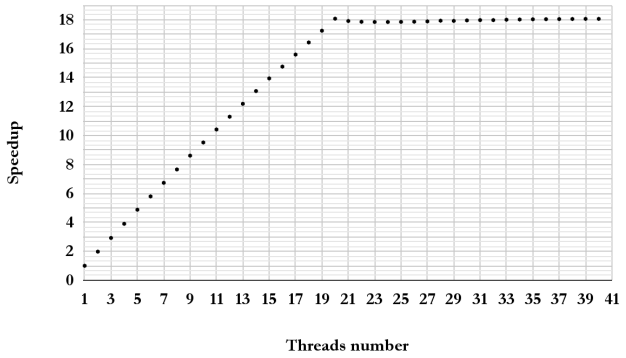

Fig. 5. "Speedup vs. Number of Threads" Graph

From the graph above and considering all the specifications of the cluster indicated in the introduction, i.e., in the used partition, we have 2 processors, and each of them has 10 physical threads, which, in turn, are composed of two logical threads. While the number of used threads is less than the number of physical threads, they will share the workload among them, and as expected, there is an approximately linear growth. This region of the graph is not perfectly linear due to the existence of sequential work in the code execution (*Amdahl's law*, strong scalability).

By increasing the number of threads beyond the number of available physical threads, logical threads are used, made possible by hyper-threading. However, the speedup remains approximately constant (Gustafson's law, weak scalability). This is because, although each thread is responsible for a smaller portion of the work, it implies a compromise of efficiency, as additional threads create synchronisation problems, competition for shared resources, and communication latencies. In this case, since we use dynamic scheduling, the use of hyper-threading is not advantageous because there will always be threads in waiting.

V. Final Remarks

In conclusion, the most recommended number of threads to compare as the initial result is the parallel version with 20 threads.

TABLE IV. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
185 513	0,69	128 339	2 801	2,574

Fig. 6. Table with the selected measurement of various parameters related to the code executed in parallel with 20 threads (the remaining measurements are attached).

Achieving a 94.47% reduction in execution time compared to the sequential code, the version with 20 threads is favourable. Comparing it to the version with 40 threads (Fig.4), it's important to note that, due to the shared cache among logical threads belonging to the same physical thread, there is an increase in L1 misses. Additionally, there is a decrease in CPI due to the reduction in cycles waiting for memory.

VI. Attachment

TABLE V. Execution Time and Other Information

Instructions ($\times 10^6$)	CPI	Clock Cycles ($\times 10^6$)	$L1_{MISSES}$ ($\times 10^6$)	Time (s)
179 337	0,65	116 471	2 752	46,593
179 344	0,67	119 262	2 753	47,710
179 337	0,65	116 478	2 752	46,596
179 343	0,67	119 815	2 753	47,931
179 337	0,65	116 618	2 752	46,652
179 343	0,67	119 228	2 753	47,696
179 340	0,66	118 366	2 753	47,352
179 337	0,65	116 471	2 752	46,593
179 342	0,66	18 466	2 753	47,392
179 337	0,65	116 481	2 752	46,597

Fig. 7. Measurements related to the sequential code.

Profiling I. Sequential Code

```
Overhead  Command    Shared Object      Symbol
........  .........  .................  ......................

  61.63%  MDseq.exe  MDseq.exe          [.] computeAccelerations
  32.64%  MDseq.exe  MDseq.exe          [.] Potential
```

Fig. 8. Profile of the sequential code.

```
double Potential() {
    double Pot = 0.;
    int i, j;
    double sigma2 = sigma * sigma;
    double rSqd, quot2, quot6,ri0,ri1,ri2;
    for (i=0; i<N-1; i++) {
        for (j=i+1; j<N; j++) {
                rSqd = ((r[0][i]-r[0][j])*(r[0][i]-r[0][j]))
                      +((r[1][i]-r[1][j])*(r[1][i]-r[1][j]))
                     +((r[2][i]-r[2][j])*(r[2][i]-r[2][j]));
                quot2 = sigma2/rSqd;
                quot6 = quot2 * quot2 * quot2;
                Pot += quot6*(quot6 - 1);
        }
    }
    return 2*4*epsilon*Pot;
}

void computeAccelerations() {
    int i, j, k;
    double f, rSqd,rSqd_inv,rSqd_inv3,
           rSqd_inv4,rij0,rij1,rij2;
    for (i = 0; i < N; i++) {
        for (k = 0; k < 3; k++) {
            a[k][i] = 0;
        }
    }
    for (i = 0; i < N-1; i++) {
        for (j = i+1; j < N; j++) {
            rij0 = r[0][i] - r[0][j];
            rij1 = r[1][i] - r[1][j];
            rij2 = r[2][i] - r[2][j];
            rSqd = (rij0 * rij0) +
                   (rij1 * rij1) +
                   (rij2 * rij2);
            rSqd_inv = 1/rSqd;
            rSqd_inv3 = rSqd_inv*rSqd_inv*rSqd_inv;
            rSqd_inv4 = rSqd_inv*rSqd_inv*rSqd_inv*rSqd_inv;
            f = 24*rSqd_inv4*(2 * rSqd_inv3 - 1);
            a[0][i] += rij0 * f;
            a[0][j] -= rij0 * f;
            a[1][i] += rij1 * f;
            a[1][j] -= rij1 * f;
            a[2][i] += rij2 * f;
            a[2][j] -= rij2 * f;
        }
    }
}
```

Fig. 9. Functions of the sequential code.

TABLE VI. Execution Time and Other Information

Instructions ($\times 10^6$)	CPI	Clock Cycles ($\times 10^6$)	$L1_{MISSES}$ ($\times 10^6$)	Time (s)
227 323	1,43	323 865	2 734	4,383
225 291	1,41	316 817	2 739	4,409
225 660	1,43	321 307	2 719	4,348
225 463	1,43	321 639	2 714	4,379
224 585	1,41	316 137	2 729	4,330
225 175	1,41	319 173	2 723	4,397
227 632	1,45	329 867	2 712	4,484
228 219	1,45	330 904	2 714	4,574
228 086	1,45	328 569	2.727	4,514
225 538	1,43	320 033	2 727	4,346

Fig. 10. Measurements related to the parallel code.

Profiling II. Parallel Code

```
Overhead  Command    Shared Object      Symbol
........  .........  .................  ..............................

  44.71%  MDpar.exe  MDpar.exe          [.] computeAccelerations
  25.31%  MDpar.exe  libgomp.so.1.0.0   [.] gomp_team_barrier_wait_end
  17.51%  MDpar.exe  MDpar.exe          [.] Potential
  10.43%  MDpar.exe  libgomp.so.1.0.0   [.] gomp_barrier_wait_end
```

Fig. 11. Profile of the parallel code.

```
#include <omp.h>
double Potential() {
    double Pot = 0.;
    double sigma2 = sigma * sigma;
    #pragma omp parallel for (+:Pot)
    for (int i=0; i<N-1; i++) {
        for (int j=i+1; j<N; j++) {
                double rSqd =
                       ((r[0][i]-r[0][j])*(r[0][i]-r[0][j]))
                      +((r[1][i]-r[1][j])*(r[1][i]-r[1][j]))
                     +((r[2][i]-r[2][j])*(r[2][i]-r[2][j]));
                double quot2 = sigma2/rSqd;
                double quot6 = quot2 * quot2 * quot2;
                Pot += quot6*(quot6 - 1);
        }
    }
    return 2*4*epsilon*Pot;
}

void computeAccelerations() {
    for (int i = 0; i < N; i++) {
        for (int k = 0; k < 3; k++) {
            a[k][i] = 0;
        }
    }
```

```
    #pragma omp parallel for (+:a)
    for (int i = 0; i < N-1; i++) {
        for (int j = i+1; j < N; j++) {
            double rij0 = r[0][i] - r[0][j];
            double rij1 = r[1][i] - r[1][j];
            double rij2 = r[2][i] - r[2][j];
            double rSqd = (rij0 * rij0) +
                   (rij1 * rij1) +
                   (rij2 * rij2);
            double rSqd_inv = 1/rSqd;
            double rSqd_inv3 = rSqd_inv*rSqd_inv*rSqd_inv;
            double rSqd_inv4 = rSqd_inv*rSqd_inv*
                               rSqd_inv*rSqd_inv;
            double f = 24*rSqd_inv4*(2 * rSqd_inv3 - 1);
            a[0][i] += rij0 * f;
            a[0][j] -= rij0 * f;
            a[1][i] += rij1 * f;
            a[1][j] -= rij1 * f;
            a[2][i] += rij2 * f;
            a[2][j] -= rij2 * f;
        }
    }
}
```

Fig. 12. Functions of the parallel code.

DEDUCTION I. DIFFERENCE IN ITERATIONS BETWEEN THE FIRST AND LAST CHUNK

Variable	Variable name
N	particule number
I	number of interaction difference
n	Thread number
If	number of first block interactions
Il	number of last block interactions

Considering the high number of particles, each block will have approximately,

$$\frac{N}{n} \tag{2}$$

elements, and therefore:

$$Il = \sum_{k=1}^{\frac{N}{n}-1} k \tag{3}$$

$$If = \sum_{k=1}^{\frac{N}{n}} \left(\frac{N}{n} - k\right) \tag{4}$$

Considering this and that,

$$\sum_{k=1}^{m} (k) = \frac{m}{2}(m-1) \tag{5}$$

thus, we have an approximation for a large number of particles:

$$I = \left(\frac{N}{n}\right)^2 - \frac{N}{n} \tag{6}$$

The larger N is in relation to n, the longer the wait time at the barrier will be.

TABLE VII. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	CPI	Clock Cycles ($\times 10^6$)	$L1_{MISSES}$ ($\times 10^6$)	Time (s)
192 295	1,33	257 307	2 314	2,582
192 292	1,33	257 283	2 314	2,601
192 132	1,33	256 807	2 320	2,576
192 174	1,33	257 062	2 315	2,580
192 169	1,33	256 901	2 310	2,577
192 204	1,33	257 050	2 315	2,578
192 910	1,33	259 393	2 318	2,605
192 296	1,33	257 216	2 311	2,581
192 307	1,33	257 430	2 325	2,587
192 387	1,33	257 674	2 313	2,585

Fig. 13. Measurements related to the parallel code with load distribution.

PROFILING III. PARALLEL CODE WITH LOAD DISTRIBUTION

```
Overhead  Command    Shared Object     Symbol
........  .........  ................  ........................................

 47.73%  MDpar.exe  MDpar.exe         [.] computeAccelerations
 23.90%  MDpar.exe  MDpar.exe         [.] Potential
  8.58%  MDpar.exe  [unknown]         [k] 0xffffffffa0f17aa0
  7.34%  MDpar.exe  [unknown]         [k] 0xffffffffa0f17aa2
  3.32%  MDpar.exe  libgomp.so.1.0.0  [.] gomp_team_barrier_wait_end
  2.82%  MDpar.exe  libgomp.so.1.0.0  [.] gomp_mutex_lock_slow
  1.90%  MDpar.exe  libgomp.so.1.0.0  [.] gomp_barrier_wait_end
```

Fig. 14. Profile of the parallel code with load distribution.

```
#include <omp.h>
double Potential() {
    double Pot = 0.;
    double sigma2 = sigma * sigma;
    #pragma omp parallel for (+:Pot) schedule(dynamic)
    for (int i=0; i<N-1; i++) {
        for (int j=i+1; j<N; j++) {
                double rSqd =
                       ((r[0][i]-r[0][j])*(r[0][i]-r[0][j]))
                      +((r[1][i]-r[1][j])*(r[1][i]-r[1][j]))
                     +((r[2][i]-r[2][j])*(r[2][i]-r[2][j]));
                double quot2 = sigma2/rSqd;
                double quot6 = quot2 * quot2 * quot2;
                Pot += quot6*(quot6 - 1);
        }
    }
    return 2*4*epsilon*Pot;
}

void computeAccelerations() {
    for (int i = 0; i < N; i++) {
        for (int k = 0; k < 3; k++) {
            a[k][i] = 0;
        }
    }
    #pragma omp parallel for (+:a) schedule(dynamic)
    for (int i = 0; i < N-1; i++) {
        for (int j = i+1; j < N; j++) {
            double rij0 = r[0][i] - r[0][j];
            double rij1 = r[1][i] - r[1][j];
            double rij2 = r[2][i] - r[2][j];
            double rSqd = (rij0 * rij0) +
                   (rij1 * rij1) +
                   (rij2 * rij2);
            double rSqd_inv = 1/rSqd;
            double rSqd_inv3 = rSqd_inv*rSqd_inv*rSqd_inv;
```

```
            double rSqd_inv4 = rSqd_inv*rSqd_inv*
                               rSqd_inv*rSqd_inv;
            double f = 24*rSqd_inv4*(2 * rSqd_inv3 - 1);
            a[0][i] += rij0 * f;
            a[0][j] -= rij0 * f;
            a[1][i] += rij1 * f;
            a[1][j] -= rij1 * f;
            a[2][i] += rij2 * f;
            a[2][j] -= rij2 * f;
        }
    }
}
```

Fig. 15. Functions of the parallel code.

TABLE VIII. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**$L1_{MISSES}$** ($\times 10^6$)	**Time (s)**
185 513	0,69	128 339	2 801	2,574
185 574	0,70	129 940	2 799	2,606
185 537	0,73	134 960	2 778	2,706
185 558	0,70	130 294	2 798	2,613
185 560	0,70	129 504	2 800	2,598
185 544	0,70	129 664	2 799	2,601
185 515	0,69	128 779	2 801	2,600
185 541	0,69	128 435	2 802	2,576
185 775	0,73	135 604	2 777	2,719
185 527	0,70	130 739	2 795	2,622

Fig. 16. Measurements related to the parallel code executed with 20 threads.

REFERENCES

[1] SeARCH | Services and Advanced Research Computing with HTC/HPC clusters. (n.d.). University of Minho. Retrieved from https://search6.di.uminho.pt/wordpress/ at 6th december of 2023