Parallel Computing
Work Assignment Phase 2
Molecular Dynamics Simulation

1st Carlos Machado
Informatics Department
University of Minho
Braga, Portugal
pg52675@alunos.uminho.pt

2nd Gonçalo Sousa
Informatics Department
University of Minho
Braga, Portugal
pg52682@alunos.uminho.pt

***Abstract*—This document refers to a practical assignment within the Master's Degree program in Software Engineering, specifically in the curricular unit named Parallel Computing. The objective of this assignment is to further investigate the implementation of optimization techniques in a Molecular Dynamics Simulation program, focusing on shared memory parallelism using OpenMP.**

***Index Terms*—Parallel Computing, performance, Molecular Dynamics Simulations, Threading, OpenMP, speed-up, analysis**

I. Introduction

After exploring and applying optimization techniques to a single threaded program, we have been tasked once more to explore and apply optimization techniques, but now into a multithreaded program using *OMP* primitives. In order to achieve that solution we will discuss our approach to the following steps:

- Identifying the application hot-spots;
- Analysing and presenting alternatives to explore parallelism and selecting one approach;
- Scalability analysis;
- Conclusion.

II. Changes in the Phase 1 code

After analysing the details of the phase 1 grades, we noticed that our cache optimization was not ideal. Our idea of creating a global aux matrix was not the best approach, instead, we created a local vector of length 3 that we reduced in the inner loop and then assigned to the acceleration matrix in the outer loop. This way we reduced cache misses from 3,770 million to 1,882 million (for N=5000). (see Fig. 3 and Fig. 4)

III. Identification of the hot-spots

To identify the hot-spots, we used the perf tool. When recording, the tool wills sample the execution data at fixed time intervals and generate the profile view. After analysing the profile view, we concluded that the function *computeAccelerations()* was the hot-spot with 99.80%, with the complexity of $O(N^2)$. Since *computeAccelerations()* has a lot of instructions, we went further and analysed the hot-spots inside the function.

# Overhead	Samples	Command	Shared Object	Symbol
99.86%	60959	MD.exe	MD.exe	[.] computeAccelerations
0.07%	40	MD.exe	MD.exe	[.] VelocityVerlet
0.04%	37	MD.exe	[unknown]	[k] 0xffffffffab38c4ef
0.01%	6	MD.exe	libc-2.17.so	[.] __memset_sse2
0.00%	4	MD.exe	libm-2.17.so	[.] __ieee754_log_avx
0.00%	3	MD.exe	[unknown]	[k] 0xffffffffab396098
0.00%	2	MD.exe	libc-2.17.so	[.] __random
0.00%	1	MD.exe	MD.exe	[.] main
0.00%	1	MD.exe	libc-2.17.so	[.] __mpn_mul_1
0.00%	1	MD.exe	libc-2.17.so	[.] __GI_____strtod_l_internal
0.00%	1	MD.exe	libc-2.17.so	[.] _dl_addr
0.00%	1	MD.exe	ld-2.17.so	[.] do_lookup_x

Fig. 1. Perf report of the sequential code

We found that the zones with the higher amount of samples were inside the inner loop of the second loop. Being *vdivpd* the instruction with the most computing time, since it is a vectorized division instruction, in the inner loop, 2 times, with 323 and 539 samples. (see Fig. 2)

IV. Analysing different alternatives to explore parallelism

In order to explore parallelism, two alternatives were considered, parallelising the inner, or the outer loop. The inner loop parallelism suffers from worse granularity, because it creates a lot more tasks than the outer version, even tough that there's an implicit creation of threads in *OMP* because of the existence of the thread pool, assigning tasks to the thread has an overhead. Besides that, our inner loop has dependencies between outer loop cycles, so we can't avoid the creation of the implicit barrier that forces all threads to wait for each other before continuing to the next *i* loops.

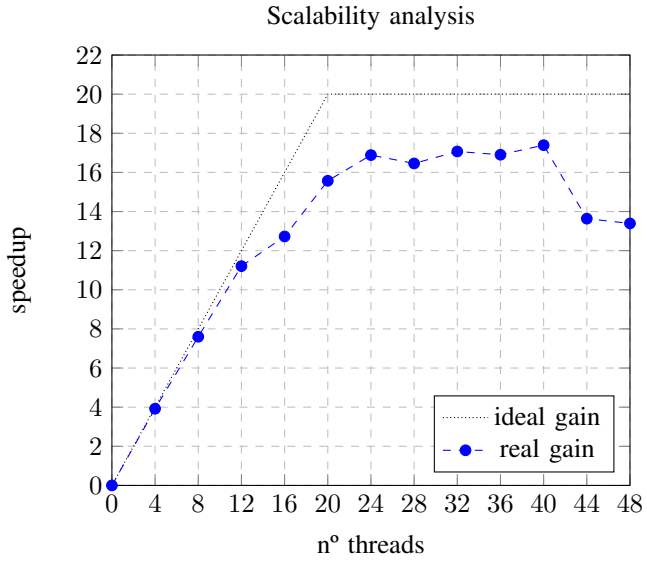
So considering that disadvantages, we opted to choose the outer loop parallelization, with also the fact that we have a reduction of the *acceleration* matrix (x, y and z). So, our solution will be the reduction of *a* and *Pot* and privating other auxiliary variables to avoid data races.

We also wanted to mention the reasoning for choosing primitive reduction instead of atomic or critical, we prefer using this synchronization primitive since it allows performing

the necessary operations locally in each thread and then the results are reduced into the original variable, that way we avoid adding the overhead of computing sequentially since as we have seen in the hot-spot analysis there are high computational time operations inside the inner loop.

V. Scalability analysis

Firstly we calculated the theoretical limit of parallelisation according to Amdahl's law, the limit is $\frac{1}{(0.20\% + \frac{(99.80\%)}{40})}$, i.e, 19.2.



As shown in the graph above, the measured speed-up plateaus at 20 threads and drops after 40, the incline of speed-up gain decreases at 8 threads, diverging from the ideal.

After 20 threads, the speed-up doesn't decay since we have Simultaneous Multi-Threading, since in our case the server microarchitecture is a *Skylake core*, we have Hyper-Threading as named by *Intel*. Therefore, since we have 40 logic cores (20 physical), we will still be able to perform multi-threading, allowing flexibility in filling the execution slots, improving the effective use of execution resources by ensuring the execution core is busy and therefore a more efficient use of the resources.

The drop after 40 threads is justified because there are 40 logic cores, so it is impossible to have more than 40 threads running in parallel (true parallelism), which implies logic scheduling (false parallelism) of the threads, decreasing the throughput and in consequence the speed-up.

Our program will never reach the speed-up 20 because of the Amdahl's law, so it's expected to have at max a speed-up of 19.2. So for 20 threads, we will not have a speed-up of 20.

In order to understand the onset of divergence at 8 threads, we must analyse the following topics about Scalability problems:

- Percentage of serial work
- Memory Wall
- Parallelism/task granularity
- Synchronization overhead
- Load imbalance

We believe that the first and third topic are not the botteneck of our solution since we have a very low percentage of serial work neither parallelism granularity since we're parallelising the outer loop as discussed in the other section.

Synchronization overhead is something we must apprise, because when we increase the number of threads, although we have improved our code with the primitive *reduction*, will have always have 2 reduces due to dependencies in the hot-spot (*Pot* and *a[]*) and possibility false sharing since we're using thread local values which implies that when altering a cache line, if we have threads working in close spaces in memory, if we alter some value that others threads are using, we end up invalidating it, forcing synchronization and cache coherence, creating unnecessary delays.

Memory wall can play a big role in our solution since as stated before, we had problems with cache optimization in our last assignment and although improving it and not having Array of pointer layout, we don't implemented loop tiling, which could be usefull in improving cache bandwidth limitation problems.

At last, Load imbalance is also something that is not well improved since one of the optimizations of our last assignment was removing the inside if of the *potential* function, that was merged into one with *computeAccelerations*, making the inside loop more irregular, as each if we maintain the default static scheduling, the first thread will have more work than the following and the following more work than its successor since the loop starts in $i + 1$, creating load Imbalances. The easiest solution was turning the scheduling dynamic, even though it has extra overhead comparing to static, we decided to choose the chunk size equal to the number of current threads, that way we try to distribute the weight among all threads instead of having a chunk size of one and making every thread competing for the next task, as increasing the chunk size make our dynamic scheduling more "static". The ideal solution in our opinion, that we couldn't implement was making a manual static as we could assign ourselves each task to each thread, but since we didn't manage to implement it, we can't share practical results.

VI. Conclusion

With the conclusion of the second phase of this assignment, we analysed the hot-spots of our code from the last assignment, identified the possible the different alternatives and explained why we chose that one. The most important part of our assignment was explaining the results of our Scalability analysis since it required further knowledge and the perpection of linking the scalability problems that we had with the architecture of the machine that we're running our parallel application.

```
119 :   401d00:       vmovupd (%rbx),%ymm4
  1 :   401d04:       add    $0x1,%esi
  6 :   401d07:       add    $0x20,%rbx
    :                       rij1 = r[1][i] - r[1][j];
  9 :   401d0b:       vmovupd (%r11),%ymm3
137 :   401d10:       add    $0x20,%r10
  1 :   401d14:       add    $0x20,%r11
    :                       rij0 = r[0][i] - r[0][j];
  2 :   401d18:       vsubpd %ymm4,%ymm12,%ymm4
    :                       rij2 = r[2][i] - r[2][j];
  9 :   401d1c:       vmovupd -0x20(%r10),%ymm1
144 :   401d22:       add    $0x20,%rax
    :                       rij1 = r[1][i] - r[1][j];
  2 :   401d26:       vsubpd %ymm3,%ymm11,%ymm3
  3 :   401d2a:       add    $0x20,%rdi
  2 :   401d2e:       add    $0x20,%rdx
    :                       rij2 = r[2][i] - r[2][j];
117 :   401d32:       vsubpd %ymm1,%ymm10,%ymm1
  2 :   401d36:       add    $0x20,%r8
  3 :   401d3a:       add    $0x20,%rcx
    :                       r_sqrd = (rij0 * rij0) + (rij1 * rij1) + (rij2 * rij2);
  1 :   401d3e:       vmulpd %ymm4,%ymm4,%ymm0
110 :   401d42:       add    $0x20,%r9
  1 :   401d46:       vmulpd %ymm3,%ymm3,%ymm2
  1 :   401d4a:       vaddpd %ymm0,%ymm2,%ymm0
  0 :   401d4e:       vmulpd %ymm1,%ymm1,%ymm2
136 :   401d52:       vaddpd %ymm2,%ymm0,%ymm0
    :                       r_sqrd_ = r_sqrd * r_sqrd * r_sqrd;
  3 :   401d56:       vmulpd %ymm0,%ymm0,%ymm9
  8 :   401d5a:       vmulpd %ymm0,%ymm9,%ymm9
    :                       f = 24 * (2 - r_sqrd_) / (r_sqrd_ * r_sqrd_ * r_sqrd);
 11 :   401d5e:       vsubpd %ymm9,%ymm15,%ymm2
129 :   401d63:       vmulpd %ymm9,%ymm9,%ymm9
  5 :   401d68:       vmulpd %ymm0,%ymm9,%ymm9
    :                       quot = s / (r_sqrd);
323 :   401d6c:       vdivpd %ymm0,%ymm13,%ymm0
    :                       f = 24 * (2 - r_sqrd_) / (r_sqrd_ * r_sqrd_ * r_sqrd);
  0 :   401d70:       vmulpd %ymm14,%ymm2,%ymm2
539 :   401d75:       vdivpd %ymm9,%ymm2,%ymm2
    :                       tempx = f * rij0;
154 :   401d7a:       vmulpd %ymm4,%ymm2,%ymm4
    :                       tempy = f * rij1;
 23 :   401d7e:       vmulpd %ymm3,%ymm2,%ymm3
    :                       tempz = f * rij2;
 26 :   401d82:       vmulpd %ymm1,%ymm2,%ymm1
    :                       a[0][j] -= tempx;
  3 :   401d86:       vmovapd -0x20(%rax),%ymm2
    :                       aux[0] += tempx;
```

Fig. 2. Perf anotate of the sequential code

```
 TO ANALYZE INSTANTANEOUS DATA ABOUT YOUR MOLECULE, OPEN THE FILE
 'cp_output.txt' WITH YOUR FAVORITE TEXT EDITOR OR IMPORT THE DATA INTO EXCEL

 THE FOLLOWING THERMODYNAMIC AVERAGES WILL BE COMPUTED AND WRITTEN TO THE FILE
 'cp_average.txt':

 AVERAGE TEMPERATURE (K):                      131.45506

 AVERAGE PRESSURE  (Pa):                   131001228.45076

 PV/nT (J * mol^-1 K^-1):                       28.47279

 PERCENT ERROR of pV/nT AND GAS CONSTANT:       242.44905

 THE COMPRESSIBILITY (unitless):                  3.42449

 TOTAL VOLUME (m^3):                       2.37220e-25

 NUMBER OF PARTICLES (unitless):           5000

Performance counter stats for './MD.exe':

    3,770,610,087      L1-dcache-load-misses

     19.213014477 seconds time elapsed

     19.206552000 seconds user
      0.001999000 seconds sys
```

Fig. 3. Perf anotate of the sequential code

```
 TO ANIMATE YOUR SIMULATION, OPEN THE FILE
 'cp_traj.xyz' WITH VMD AFTER THE SIMULATION COMPLETES

 TO ANALYZE INSTANTANEOUS DATA ABOUT YOUR MOLECULE, OPEN THE FILE
 'cp_output.txt' WITH YOUR FAVORITE TEXT EDITOR OR IMPORT THE DATA INTO EXCEL

 THE FOLLOWING THERMODYNAMIC AVERAGES WILL BE COMPUTED AND WRITTEN TO THE FILE
 'cp_average.txt':

 AVERAGE TEMPERATURE (K):                      131.45506

 AVERAGE PRESSURE  (Pa):                 131001228.45076

 PV/nT (J * mol^-1 K^-1):                     28.47279

 PERCENT ERROR of pV/nT AND GAS CONSTANT:     242.44905

 THE COMPRESSIBILITY (unitless):               3.42449

 TOTAL VOLUME (m^3):                     2.37220e-25

 NUMBER OF PARTICLES (unitless):         5000

Performance counter stats for './MD.exe':

    1,882,842,666      L1-dcache-load-misses

     15.787917748 seconds time elapsed

     15.779167000 seconds user
      0.004000000 seconds sys
```

Fig. 4. Perf anotate of the sequential code