Paralell Computing
Work Assignment Phase 1
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 from a Master's Degree in Software Engineering curricular unit named Parallel Computing. The goal of this assignment is to explore optimization techniques applied to a single threaded program which is part of a simple molecular dynamics simulation code applied to atoms of argon gas.**

***Index Terms*—Parallel Computing, performance, Molecular Dynamics Simulations, optimizations, analysis, speedup**

I. Introduction

This document is a report on the optimization techniques used to optimize part of a simple molecular dynamics simulation, these included ILP (Instruction Level Parallelism), Memory Hierarchy optimizations, data structure organization and Vectorization.

II. Analysis of the Code

A. What the Simulator Does

A molecular dynamics simulator tries to calculate the behavior and interaction of particles through time, computing thermodynamic properties of a material. It does this by initializing the velocities of the particles according to a Gaussian distribution, computing their initial acceleration and cycling though time computing the new position, velocity, and acceleration according to the laws of Newton's mechanics, each cycle the mean squared velocity, kinetic energy and potential energy are calculated and recorded.

B. Data structures

In the original version of the given code, the main data structures are the *a[MAXPART][3]*, *v[MAXPART][3]* and *r[MAXPART][3]*, which store the x, y, z coordinates of the acceleration, velocity and position of each particle. The *MAXPART* variable stores the maximum line length that each matrix can have.

C. Functions

There are seven main functions called during the execution of the simulator: Here *N* refers to the number of particles and *NumTimes* to the ammount of states computed.

1) **Initialize**
   - Sets the initial positions of the particles and calls InitializeVelocities.
   - Called once.
   - $\mathcal{O}(N)$.
2) **InitializeVelocities**
   - Sets the initial velocities randomly according to a Gaussian distribution.
   - Called once.
   - $\mathcal{O}(N)$.
3) **gaussdist**
   - Numerical recipes Gaussian distribution number generator.
   - Called once.
   - $\mathcal{O}(1)$.
4) **VelocityVerlet**
   - Updates the position, velocity and position of each particle and returns the sum of dv/dt*m/A (aka Pressure) from elastic collisions with walls.
   - Called NumTime times.
   - $\mathcal{O}(N)$.
5) **MeanSquaredVelocity**
   - Calculates the averaged Velocity Squared.
   - Called NumTime+1 times.
   - $\mathcal{O}(N)$.
6) **Kinetic**
   - Calculates the kinetic energy of the system.
   - Called NumTime+1 times.
   - $\mathcal{O}(N)$.
7) **Potential**
   - Calculates the potential energy of the system.
   - Called NumTime+1 times.
   - $\mathcal{O}(N^2)$.
8) **ComputeAccelerations**
   - Calculates the change in acceleration for each particle.
   - Called NumTime+2 times.

- $\mathcal{O}(\frac{N^2}{2})$.

(For more information see Fig. 1)

D. Performance Analysis

Before any optimizations a performance analysis must be performed, to that effect *perf* and *gprof* were used. Initial tests revealed roughly 2.4 billion L1 cache misses and around 1.2 trillion instructions. These Could be reduced immediately to around 670 million L1 cache misses and roughly 51 billion instructions by compiling with the *-Ofast* flag. Running *gprof* showed that, as expected, the two most time-consuming functions were *computeAccelerations* and *Potencial*.

III. Optimized code

A. Data structures

In order to improve the vectorizability of the code the arrays were changed to a[3][MAXPART], v[3][MAXPART] and r[3][MAXPART], so that operations for multiple particles can be done in a single instruction.

B. Optimized Functions

The main focus of our work was to improve Potencial() and computeAccelerations() mainly because they hold most of the execution time, providing a better overall speedup. Initially Potential was altered to have a complexity of $\mathcal{O}(\frac{N^2}{2})$ by skipping repeated pairs and doubling the *Pot* variable on return, furthermore all operations on the variable Pot were moved to the outside of the loop. computeAccelerations() was changed to make use of an auxiliary array in order to enable the vectorization of all the operations which could be vectorized. Minor changes were made to both the algorithms to improve ILP and reduce operation count. Potential() and computeAccelerations() were merged to take advantage of the fact that both these functions compute the distance between particles.

C. Other Optimizations

Other than changes to the two most time-consuming functions, one of the largest time saves came from the removal of the *math* functions used throughout the code.

Furthermore, the functions MeanSquaredVelocity() and Kinetic() were merged since they initially performed the same computation (the Squared Velocity). Although, given their relative time complexity, little performance gain was observed.

D. Optimization Analysis

In spite of the use of an auxiliary matrix, the amount of L1 cache misses stayed mostly similar to the original version (Fig. 5, 4) compiled with the *-Ofast* flag. When both are compiled with the *-O2* flag, the optimized version is a 73% improvement compared to the original (see Fig. 7, 8), having around 657 million cache misses. When it comes to the number of instructions, the optimized version with no vectorization and compiled with *-Ofast* executed around 20.3 billion instructions, a 61% improvement over the original one compiled with the same flags which executed 52.2 billion instructions (see Fig. 6, 2), after the vectorization flags are applied the optimized version's instructions are 37% of the previous count(Fig. 4, 6), slightly more than the expected 25% for perfectly vectorized code. The original version with the vectorization flags applied has about 83% of the instructions (see Fig. 5, 2), we can see, therefore, that the optimized version is a lot more vectorizable.

E. Results

The optimized code was repeatedly tested with perf stat in the **Search Cluster**, with an average of around 3 seconds. The best recorded time was 2.7 (see Fig. 4) seconds.

Both of the following results were compiled with the flag *Ofast* and the respective vectorization flags. The first one is the original code (see Fig. 2) and the second one is our recorded best time (see Fig. 4).

TABLE I
Performance Metrics

	size	Texe		CC	I	I Estimated
Average CPI						
						(calculated)

F. Comparing Call-graphs

Comparing the first version Call-graph (see Fig. 1) and our best version Call-graph.

We can see that in the first one that Potential takes 74.4% of the execution time and computeAccelerations() 25.6% of the execution time.

The last one, because of the merging, the Potential() function is no longer executed and computeAccelerations() accounts for almost 100% of the execution time.

IV. Conclusion

In conclusion, we consider that the goals of this assignment were achieved and that the result is a lot more optimized than the original version. Overall progress was uneven as the behaviour of the compiler and of the program was not fully understood initially, there were also numerous ideas for optimizations that, while theoretically seemed to be improvements, ended up yielding inferior results. We believe that further improvements could be achieved with the use of manual vectorization, our version of the code is built to maximize the vectorization done by the compiler, but better results arise from the use of vectorization primitives within the code itself.

```
granularity: each sample hit covers 2 byte(s) for 0.05% of 18.24 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     74.4   13.57    0.00                 Potential() [1]
-----------------------------------------------
                4.67    0.00     201/201         VelocityVerlet(double, int, _IO_FILE*) [3]
[2]     25.6    4.67    0.00     201         computeAccelerations() [2]
-----------------------------------------------
                                                 <spontaneous>
[3]     25.6    0.00    4.67                 VelocityVerlet(double, int, _IO_FILE*) [3]
                4.67    0.00     201/201         computeAccelerations() [2]
-----------------------------------------------
                0.00    0.00    6480/6480        initializeVelocities() [12]
[11]     0.0    0.00    0.00    6480         gaussdist() [11]
-----------------------------------------------
                0.00    0.00       1/1           initialize() [13]
[12]     0.0    0.00    0.00       1         initializeVelocities() [12]
                0.00    0.00    6480/6480        gaussdist() [11]
-----------------------------------------------
```

Fig. 1. Call-graph of the Original Code

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

        670324924      L1-dcache-load-misses
      43703166473      cycles
      52254465442      instructions              #    1,20  insn per cycle

      52254465442      inst_retired.any          #      0,8 CPI
      43703166407      cycles

     15,075569643 seconds time elapsed

     15,067926000 seconds user
      0,002999000 seconds sys
```

Fig. 2. Perf Status of the Original Code without vectorization and with the Ofast flag

```
granularity: each sample hit covers 2 byte(s) for 0.21% of 4.84 seconds

index % time    self  children    called     name
                4.84    0.00     201/201         VelocityVerlet(double, int, _IO_FILE*, double*) [2]
[1]    100.0    4.84    0.00     201         computeAccelerations() [1]
-----------------------------------------------
                                                <spontaneous>
[2]    100.0    0.00    4.84                 VelocityVerlet(double, int, _IO_FILE*, double*) [2]
                4.84    0.00     201/201         computeAccelerations() [1]
-----------------------------------------------
                0.00    0.00    6480/6480        initializeVelocities() [11]
[10]     0.0    0.00    0.00    6480         gaussdist() [10]
-----------------------------------------------
                0.00    0.00       1/1           initialize() [12]
[11]     0.0    0.00    0.00       1         initializeVelocities() [11]
                0.00    0.00    6480/6480        gaussdist() [10]
-----------------------------------------------
```

Fig. 3. Call-graph of the Optimized Code

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

       647849714      L1-dcache-load-misses
      8809045219      cycles
      7540961600      instructions              #    0,86  insn per cycle

      7540961600      inst_retired.any          #      1,2 CPI
      8809045174      cycles

     2,676419899 seconds time elapsed

     2,671621000 seconds user
     0,001000000 seconds sys
```

Fig. 4. Perf Status of the Optimized Code with vectorization and the compiler flag Ofast

```
Performance counter stats for './a.out':

     685,724,303      L1-dcache-load-misses
  55,555,446,959      cycles
  43,319,436,257      instructions              #    0.78  insn per cycle
  43,319,436,257      inst_retired.any          #      1.3 CPI
  55,555,446,887      cycles

    17.914962233 seconds time elapsed

    17.908327000 seconds user
     0.002000000 seconds sys
```

Fig. 5. Perf Status of the Original Code with vectorization and the compiler flag Ofast

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

       690051367      L1-dcache-load-misses
     18042032089      cycles
     20375035803      instructions              #    1,13  insn per cycle

     20375035803      inst_retired.any          #      0,9 CPI
     18042032032      cycles

     6,224243348 seconds time elapsed

     6,216950000 seconds user
     0,002000000 seconds sys
```

Fig. 6. Perf Status of the Optimized Code not vectorized and with Ofast flag

```
Performance counter stats for './a.out':

    2,467,210,622      L1-dcache-load-misses:u
  788,546,120,111      cycles:u
1,242,636,910,047      instructions:u            #    1.58  insn per cycle
1,242,637,139,310      inst_retired.any:u
  788,545,242,712      cycles:u

    240.579735244 seconds time elapsed

    240.582801000 seconds user
      0.001000000 seconds sys
```

Fig. 7. Perf Status of the Original code with the O2 flag

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

       666617226      L1-dcache-load-misses
     20962375356      cycles
     23173142099      instructions              #    1,11  insn per cycle
     23173142099      inst_retired.any          #      0,9 CPI
     20962375290      cycles

     6,597186353 seconds time elapsed

     6,591801000 seconds user
     0,001000000 seconds sys
```

Fig. 8. Perf Status of the optimized code with the O2 flag