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 this project stage are to explore optimization techniques applied to a single-threaded program, using code analysis and profiling tools to enhance and assess its final performance.**

Keywords—Optimization, performance, code, analysis, time, execution

I. Introduction

The program that will be analysed and optimised in this project is a code that simulates the molecular dynamics of noble gases, with a focus on the Lennard-Jones potential used to describe interactions between two particles (force and potential energy). It should be noted that natural units are used, so the values of various physical constants and parameters are defined as 1. The code includes functions for calculating the kinetic and potential energy of the system, initialising the positions and velocities of the particles, and calculating their accelerations based on intermolecular forces. The code initialises a particle system, updates their positions and velocities using the Velocity Verlet algorithm, and calculates various thermodynamic properties such as temperature and pressure.

In order to improve the code's execution time, several essential steps were taken as a basis: analysis and modification of the code, compilation of the modified code, executing it to obtain its profile and identifying the main blocks that need to undergo structural changes. Finally, checking if the code is readable and if the simulation outputs remain the same.

II. Initial Analysis

A. Configurations

All code compilations were carried out using version 7.2.0 of the GNU Compiler Collection, better known as GCC. All measurements in this report were chosen based on the k-best 5 method, which involves selecting a set of 5 measurements with an accuracy of 5%. This means that, in addition to the measurement presented, there are 4 more measurements whose relative percentage deviation is less than 5% of the same value presented.

B. Original Code

TABLE I. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{\text{MISSES}}$** ($\times 10^6$)	**Time (s)**
1 017 662	0,65	658 689	2 447	263,48

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

Based on the execution profile of the original code, it was noticed that the *Potential* function consumes 69.86% of the execution time, followed by the *computeAccelerations* function, which occupies 30.14%. With this information, the first modification aimed to optimise the *Potential* function.

III. Simplification of the Potential Function's Complexity

Based on the code of the *Potential* function, the two loops *i* and *j*, ranging from 0 to N-1, contribute to a complexity of O(N^2). However, this complexity can be reduced by half when it is noticed that r[i][k] = r[j][k] for *i* = *j*, meaning that the initial loops can be reduced by more than half (considering the *k* loop as well). Consequently, there will be a decrease in the number of instructions and memory accesses, anticipating a reduction in the execution time. *See the attachment for more details.*

TABLE II. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{\text{MISSES}}$** ($\times 10^6$)	**Time (s)**
685 642	0,64	438 128	1 724	175,26

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

Reassessing the profile of the modified code, the *Potential* function now uses 53.89% of the execution time, followed by the *computeAccelerations* function, which occupies 46.11%. As a result of the reduced complexity, a decrease in all parameters was observed, as expected, including the execution time, which decreased by 33.49% compared to the previously measured time.

IV. Optimization of Instructions within the Loops

As both functions now have roughly the same computational load, we optimise both functions by reducing the number of operations within the loops. In the first optimization, the decrease in complexity reduces the number of instructions more than expected (see attachment), which results in a higher number of instructions for calling the *pow* and *sqrt* functions. Therefore, the primary goal in this optimization is to remove these functions and, if possible, simplify the calculations.

A. Potential

Looking at the code of the *Potential* function, it is evident that there is a possibility to avoid redundant calculations and remove function calls: the 'sqrt' function is unnecessary when working with squares, and using 'pow' for small exponents is inefficient (exponentiation by squaring). Finally, we optimise the accumulation of the return value (inside the *j* loop) by multiplying the constant only once outside the loop, resulting in a reduction of instructions.

B. *ComputeAccelerations*

Since the exponents of the *pow* functions are small, calling them becomes inefficient. However, multiplying several divisions is computationally costly, as division is more computationally intensive. The alternative is to perform only one division per iteration of *j*, reducing the repetitive instructions. See the attachment for more details.

TABLE III. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
56 093	0,45	25 390	410	10,15

Fig. 3. Table with the selected measurement of various parameters related to the code after code after the second modification (the remaining measurements are attached).

Reanalyzing the profile of the modified code, the *computeAccelerations* function now occupies 72.47% of the execution time, followed by the *Potential* function, which occupies 27.53%. We observe a decrease in execution time of 94.21% compared to the previously measured time.

V. REDUCTION OF INSTRUCTIONS AND MEMORY ACCESSES

Again, the total execution time of the code is almost evenly distributed between the *Potential* and *computeAccelerations* functions. Although the code's weight is more concentrated in the *computeAccelerations* function, the optimization approach applies to both mentioned functions. Observing the assembly code, we notice that r[i][k] is unnecessarily read from memory in the *j* loop, which could be read fewer times in the *i* loop and stored in a local variable of the function. This is expected to reduce the number of instructions and clock cycles because they don't have to wait for the memory to be read as often. *See the attachment for more details.*

TABLE IV. TEMPO DE EXECUÇÃO E OUTRAS INFORMAÇÕES

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
33 984	0,65	21 926	417	8,776

Fig. 4. Table with the selected measurement of various parameters related to the code after the third modification (the remaining measurements are attached).

Reanalyzing the profile of the modified code, the *computeAccelerations* function now occupies 69.74% of the execution time, followed by the *Potencial* function, which occupies 30.03% (*see Attachment*). We observe a decrease in execution time of 13.54% compared to the previous modification.

VI. CHALLENGES WITH OPTIMIZATIONS + MEMORY HIERARCHY AND DEPENDENCY OPTIMIZATION

At this point, we couldn't optimise the code further. The fact that r[i][k] is outside the *j* loop created a loop dependency. So, we decided to take a step back, revert to the second modification, and think about another optimization.

Considering the functions with the highest computational load, to calculate the distance between two particles along an axis, we perform r[i][k] - r[j][k]. For this reason, we can take advantage of spatial locality by using r[k][i] - r[k][j], and for cases where *i* is close to *j*, there won't be L1$_{MISSES}$. However, it is not directly related to reducing L1$_{MISSES}$. While it is beneficial in terms of the number of instructions, it increases data dependency in distance calculations to take advantage of spatial locality. As it is beneficial for the functions with the highest computational load, in terms of memory hierarchy, to work with transposed matrices due to spatial locality, global variables and all operations involving the matrices are changed.

TABLE V. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
33 983	0,64	21 835	427	8,739

Fig. 5. Table with the selected measurement of various parameters related to the code after the fourth modification (the remaining measurements are attached).

Reanalyzing the profile of the modified code, the *computeAccelerations* function now occupies 69.71% of the execution time, followed by the *Potencial* function, which occupies 30.29% (see Attachment). We observe a decrease in execution time of 13.90% when compared to the execution time of optimization IV.

VII. FINAL REMARKS

To conclude this stage of the project, we would like to offer some final considerations. We have achieved a 96.68% reduction in the execution time compared to the original code. However, we were unable to apply vectorization or unrolling.

Applying the compilation option -ftree-vectorize -msse4 (vectorization) to the fourth modification does not reduce the execution time, and the same happens when we apply the compilation option -funroll-loops (unrolling). The reason for the lack of a decrease in execution time is due to data dependencies in the code loops, and unrolling cannot be applied.

In the code of the second modification, as there is no data dependency as previously mentioned (due to how particle distances are calculated), we can apply unrolling. However, applying unrolling does not reduce the execution time to less than that obtained in the fourth modification. For this reason, we consider the fourth modification to be the final version of the code for this work assignment, but the modification using unrolling is available in the attachment.

TABLE VI. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{MISSES}$** ($\times 10^6$)	**Time (s)**
33 860	0,65	22 075	416	8,837

Fig. 6. Table related to the compilation of the second modification with the unroll option.

VIII. Attachment

TABLE VII. Execution Time and Other Information

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{\text{MISSES}}$** ($\times 10^6$)	**Time (s)**
1 017 662	0,65	658 689	2 447	263,48
1 017 705	0,65	658 974	2 452	263,59
1 017 645	0,65	661 963	2 495	264,79
1 017 618	0,65	661 444	2 497	264,58
1 017 684	0,65	663 305	2 449	265,32
1 017 650	0,65	658 889	2 459	263,56
1 017 673	0,65	659 025	2 433	263,62
1 017 668	0,65	659 838	2 436	263,94
1 017 675	0,65	659 892	2 469	263,96
1 017 673	0,65	659 768	2 438	263,91

Fig. 7. Measurements related to the initial code.

DIAGRAM I. Original code

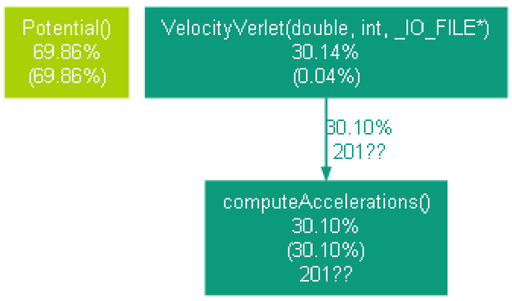

Fig. 8. Diagram with the profile of the initial code.

Prediction I. First Optimization

Potential Function in the original code:

```
double Potential() {
    double quot, rSqd, rnorm, term1, term2, Pot;
    int i, j, k;
    Pot=0.;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            if (j!=i) {
                rSqd=0.;
                for (k=0; k<3; k++) {
                    rSqd +=
(r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
                }
                rnorm=sqrt(rSqd);
                quot=sigma/rnorm;
                term1 = pow(quot,12.);
                term2 = pow(quot,6.);

                Pot += 4*epsilon*(term1 - term2);


            }
        }
    }
    return Pot;
}
```

The reduction of the total iterations of the *j* loop by half, of all the instructions inside the loop, can be observed in the assembly code of the original code:

```
.L26:
        cmpl    %r15d, %ebx
        je      .L23
        pxor    %xmm0, %xmm0
        movq    %r13, %rax
        subq    %r12, %rax
.L24:
        movsd   (%rax), %xmm1
        leaq    (%rax,%r12), %rdx
        addq    $8, %rax
        cmpq    %r14, %rax
        subsd   (%rdx,%rbp), %xmm1
        mulsd   %xmm1, %xmm1
        addsd   %xmm1, %xmm0
        jne     .L24
        ucomisd %xmm0, %xmm4
        sqrtsd  %xmm0, %xmm1
        ja      .L32
.L25:
        movsd   sigma(%rip), %xmm2
        movsd   %xmm4, 24(%rsp)
        divsd   %xmm1, %xmm2
        movsd   .LC5(%rip), %xmm1
        movapd  %xmm2, %xmm0
        movsd   %xmm2, 16(%rsp)
        call    pow
        movsd   16(%rsp), %xmm2
        movsd   %xmm0, 8(%rsp)
        movsd   .LC6(%rip), %xmm1
        movapd  %xmm2, %xmm0
        call    pow
        movsd   .LC7(%rip), %xmm1
        movsd   8(%rsp), %xmm3
        mulsd   epsilon(%rip), %xmm1
        subsd   %xmm0, %xmm3
        movsd   24(%rsp), %xmm4
        movapd  %xmm3, %xmm0
        mulsd   %xmm1, %xmm0
        addsd   (%rsp), %xmm0
        movsd   %xmm0, (%rsp)
.L23:
        movl    N(%rip), %eax
        addl    $1, %ebx
        addq    $24, %rbp
        cmpl    %ebx, %eax
        jg      .L26
```

In the loop, a conditional jump is made to call the *sqrt* function:

```
.L32:
        .cfi_restore_state
        movsd   %xmm4, 16(%rsp)
        movsd   %xmm1, 8(%rsp)
        call    sqrt
        movsd   16(%rsp), %xmm4
        movsd   8(%rsp), %xmm1
        jmp     .L25
```

Observing the assembly, we can count the number of instructions performed in each *j* loop, taking into account that there will also be an additional set of instructions in function calls. Let's consider the number of instructions as *'n'*. Thus, we can conclude that there is a reduction in the number of instructions in the *Potential* function of:

$$n\,N^2\,(Non-optimized)\;-\;(n\,N^2)/2\,(optimized)\;=\;(n\,N^2)/2$$

TABLE VIII. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	CPI	Clock Cycles ($\times 10^6$)	L1$_{MISSES}$ ($\times 10^6$)	Time (s)
685 642	0,64	438 128	1 724	175,26
685 645	0,64	438 194	1 741	175,28
685 660	0,64	438 843	1 725	175,54
685 643	0,64	438 384	1 728	175,36
685 653	0,64	438 565	1 735	175,43
685 661	0,64	438 881	1 746	175,56
685 672	0,64	438 523	1 752	175,41
685 658	0,64	438 351	1 734	175,34
685 675	0,64	438 600	1 752	175,44
685 632	0,64	438 395	1 752	175,36
685 649	0,64	438 528	1 727	175,42
685 627	0,64	438 257	1 737	175,31

Fig. 9. Measurements related to the first optimization

DIAGRAM II. FIRST OPTIMIZATION

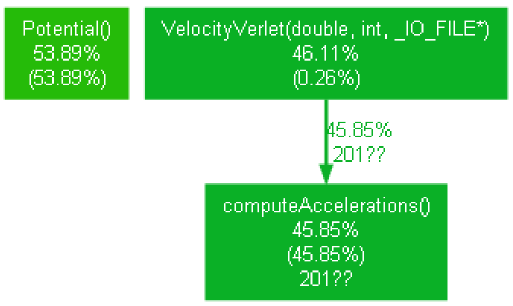

Fig. 10. Function Diagram for the First Modification

PREDICTION II. SECOND OPTIMIZATION

Potential Function from the First Optimization:

```
double Potential() {
    double quot, rSqd, rnorm, term1, term2, Pot;
    int i, j, k;
    Pot=0.;
    for (i=0; i<N-1; i++) {
        for (j=i+1; j<N; j++) {
                rSqd=0.;
                for (k=0; k<3; k++) {
                    rSqd +=
(r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
                }
                rnorm=sqrt(rSqd);
                quot=sigma/rnorm;
                 term1 = pow(quot,12.);
                 term2 = pow(quot,6.);
                 Pot += 4*epsilon*(term1 - term2);
            }
        }
    return 2*Pot;
}
```

ComputeAccelerations Function from the First Optimization:

```
void computeAccelerations() {
    int i, j, k;
    double f, rSqd;
    double rij[3];
    for (i = 0; i < N; i++) {
        for (k = 0; k < 3; k++) {
            a[i][k] = 0;
        }
    }
    for (i = 0; i < N-1; i++) {
        for (j = i+1; j < N; j++) {
            rSqd = 0;
            for (k = 0; k < 3; k++) {
                rij[k] = r[i][k] - r[j][k];
                rSqd += rij[k] * rij[k];
            }
            f = 24 * (2 * pow(rSqd, -7) - pow(rSqd,
-4));
            for (k = 0; k < 3; k++) {
                a[i][k] += rij[k] * f;
                a[j][k] -= rij[k] * f;
            }
        }
    }
}
```

In this optimization, by removing the *pow* and *sqrt* functions, considering that the *pow* function has x instructions and the *sqrt* function has y instructions, the removal of these functions, while approximating the added instruction count, is nullified by the removal of multiplying constants in the sum to the return value.

Potential Function:

$$(n\,N^2)/2\;-\;((x+x+y)\,N^2)/2\;=\;((n-x+x-y)\,N^2)/2$$

ComputeAccelerations Function (*n is not equal to that of the Potential function)*:

$$(n\,N^2)/2\;-\;((x+x)\,N^2)/2\;=\;((n-x+x)\,N^2)/2$$

Note: The value of x varies with the exponent but is calculated as a very rough approximation.

TABLE IX. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	CPI	Clock Cycles ($\times 10^6$)	L1$_{\text{MISSES}}$ ($\times 10^6$)	Time (s)
56 093	0,45	25 493	410	10,20
56 093	0,45	25 525	410	10,21
56 093	0,45	25 390	410	10,16
56 093	0,45	25 444	410	10,18
56 092	0,45	25 396	409	10,16
56 093	0,45	25 411	410	10,16
56 093	0,45	25 434	409	10,17
56 093	0,45	25 481	410	10,19
56 093	0,45	25 390	410	10,15
56 093	0,45	25 420	409	10,17

Fig. 11. Measurements related to the second modification.

DIAGRAM III. SECOND OPTIMIZATION

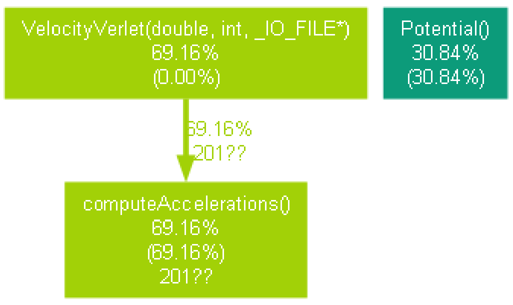

Fig. 12. Function Diagram for the Second Modification.

PREDICTION III. THIRD OPTIMIZATION

Potential Function from the SecondOptimization:

```
double Potential() {
    double Pot = 0.;
    int i, j, k;
    double sigma2 = sigma * sigma;
    double rSqd, quot2, quot6;
    for (i=0; i<N-1; i++) {
        for (j=i+1; j<N; j++) {
                rSqd=0.;
                for (k=0; k<3; k++) {
                    rSqd += (r[i][k]-r[j][k]) *
(r[i][k]-r[j][k]);
                }
                quot2 = sigma2/rSqd;
                quot6 = quot2 * quot2 * quot2;
                Pot += quot6*(quot6 - 1);
        }
    }
    return 2*4*epsilon*Pot;
}
```

Note:

$$quot^6 \times (quot^6 - 1) = (quot^{12} - quot^6)$$

ComputeAccelerations Function from the Second Optimization:

```
void computeAccelerations() {
    int i, j, k;
    double f, rSqd,rSqd_inv,rSqd_inv3,rSqd_inv4,rf;
    double rij[3]; // position of i relative to j


    for (i = 0; i < N; i++) {
        for (k = 0; k < 3; k++) {
            a[i][k] = 0;
        }
    }
    for (i = 0; i < N-1; i++) {
        for (j = i+1; j < N; j++) {
            rSqd = 0;
            for (k = 0; k < 3; k++) {
                rij[k] = r[i][k] - r[j][k];
                rSqd += rij[k] * rij[k];
            }
            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);
            for (k = 0; k < 3; k++) {
                a[i][k] += rij[k] * f;
                a[j][k] -= rij[k] * f;
            }
        }
    }
}
```

Note:

$$24 \times r^{-4} \times ((2 \times r^{-3}) - 1) = 24 \times ((2 \times r^{-7}) - r^{-4})$$

Assembly code for the *k* loop in the *Potential* function:

```
.L24:
        movsd   (%rax), %xmm1
        leaq    (%rdi,%rax), %rcx
        addq    $8, %rax
        cmpq    %rsi, %rax
        subsd   (%rcx,%rdx), %xmm1
        mulsd   %xmm1, %xmm1
        addsd   %xmm1, %xmm2
        jne     .L24
```

Assembly code for the first *k* loop in the *computeAccelerations* function:

```
.L38:
        movsd   r(%rcx,%rax), %xmm0
        subsd   r(%rdi,%rax), %xmm0
        movsd   %xmm0, -32(%rsp,%rax)
        mulsd   %xmm0, %xmm0
        addq    $8, %rax
        cmpq    $24, %rax
        addsd   %xmm0, %xmm1
        jne     .L38
```

Considering the assembly code above, reading r[i][k] from memory within the *j* loop is unnecessary, as the read can be done only within the *i* loop, reducing the number of instructions and the number of memory accesses (without affecting $L1_{MISSES}$). Based on the assembly before the modification, the following reduction in the number of instructions would be expected:

$$(N^2)/2 - (k\,N)/2$$

Where k would be greater than one, considering that the loop will be unrolled, and we will be using more registers, without the need for conditional jumps, etc.

TABLE X. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	CPI	Clock Cycles ($\times 10^6$)	$L1_{MISSES}$ ($\times 10^6$)	Time (s)
33 985	0,65	21 930	417	8,777
33 985	0,65	21 930	417	8,777
33 984	0,65	21 929	417	8,777
33 984	0,65	21 929	417	8,778
33 984	0,65	21 927	417	8,777
33 984	0,65	21 930	417	8,777
33 984	0,65	21 929	417	8,776
33 984	0,65	21 926	417	8,776
33 984	0,65	21 927	417	8,777
33 984	0,65	21 926	417	8,776

Fig. 13. Measurements related to the third modification.

DIAGRAM IV. THIRD OPTIMIZATION

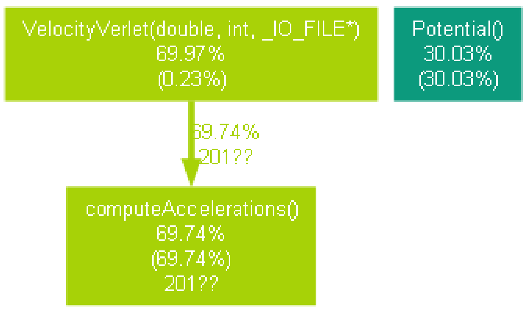

Fig. 14. Function Diagram for the Third Modification.

Potential Function from the Third Optimization:

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

ComputeAccelerations Function from the Third Optimization:

```
void computeAccelerations() {
    int i, j, k;
    double f,
rSqd,rSqd_inv,rSqd_inv3,rSqd_inv4,ri0,ri1,ri2,rij0,r
ij1,rij2;
    for (i = 0; i < N; i++) {
        for (k = 0; k < 3; k++) {
            a[i][k] = 0;
        }
    }
    for (i = 0; i < N-1; i++) {
        ri0 = r[i][0];
        ri1 = r[i][1];
        ri2 = r[i][2];
        for (j = i+1; j < N; j++) {
            rij0 = ri0 - r[j][0];
            rij1 = ri1 - r[j][1];
            rij2 = ri2 - r[j][2];
            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[i][0] += rij0 * f;
            a[j][0] -= rij0 * f;
            a[i][1] += rij1 * f;
            a[j][1] -= rij1 * f;
            a[i][2] += rij2 * f;
            a[j][2] -= rij2 * f;
        }
    }
}
```

TABLE XI. EXECUTION TIME AND OTHER INFORMATION

Instructions ($\times 10^6$)	CPI	Clock Cycles ($\times 10^6$)	$L1_{MISSES}$ ($\times 10^6$)	Time (s)
33 983	0,64	21 835	427	8,739
33 983	0,64	21 834	427	8,739
33 983	0,64	21 834	427	8,738
33 983	0,64	21 838	427	8,741
33 983	0,64	21 837	427	8,740
33 985	0,64	21 841	427	8,741
33 983	0,64	21 837	427	8,740
33 984	0,64	21 844	427	8,743
33 983	0,64	21 836	427	8,739

Instructions ($\times 10^6$)	**CPI**	**Clock Cycles** ($\times 10^6$)	**L1$_{\text{MISSES}}$** ($\times 10^6$)	**Time (s)**
33 983	0,64	21 841	427	8,741

Fig. 15. Measurements related to the fourth modification.

DIAGRAM V. FOURTH OPTIMIZATION

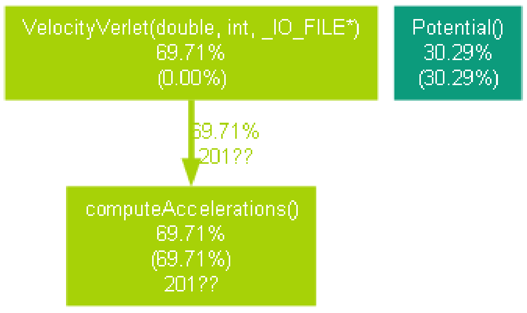

Fig. 16. Function Diagram for the Fourth Modification.