Work Assignment CP - Phase 1

1st Ivo Miguel Alves Ribeiro
Universidade do Minho
pg53886
V.N.Famalicão, Portugal
pg53886@alunos.uminho.pt

2nd Diogo Luís Almeida Costa
Universidade do Minho
pg53783
Trofa, Portugal
pg53783@alunos.uminho.pt

I. Introdução

Este trabalho tem como objetivo explorar várias técnicas e estratégias que podem ser implementadas para aprimorar o nosso código, assim, focando na seleção de algoritmos eficientes e na otimização de estruturas de dados, a fim de maximizar o desempenho do nosso programa em ambientes de recursos limitados.

II. Procedimentos para a otimização

A. Estaca 0

Após uma breve análise ao código fornecido, o primeiro passo do grupo foi examinar o código sem realizar quaisquer alterações, a fim de identificar suas zonas críticas. Para obter informações quantitativas corremos o código com *srun –partition=cpar perf stat ./MD.exe ¡ inputdata.txt* para termos acesso a informações como tempo de execução, número de instruções e número de ciclos, e ainda decidimos usufruir das vantagens do *Profiling* para saber quais das funções que despendiam um maior tempo de execução. Os como resultado desta analise verificamos que o código tem um tempo de execução de 457 segundos, segundo a figura 1, e que as funções mais custosas São a *Potencial()* e *computeAccelerations()*, segundo a figura 2.

Tabela I
Resultados de Performance na Estaca 0

Tempo(s)	***Instruções***	***Ins/Ciclo***	***Ciclos***
291	$1.2 * 10^12$	1.40	$0.890 * 10^12$

B. Simplificação das funções matemáticas

Antes desta fase modificamos o código modularizando-o separando a função *main()* das restantes (figura 3), para uma maior eficiência das funções retiramos grande parte dos ciclos que iteravam sobre a variável k, uma vez que eram ciclos de três iterações que podiam ser evitados (exemplo: figura 4), por fim a modificamos a Makefile introduzindo as flags *-O2 - funroll-loops* abordadas nas aulas praticas, ficando esta como ilustrado na figura 5. Após os resultados apresentados na etapa anterior o grupo decidiu focar em simplificar as funções mais custosas concentrando na simplificação das operações matemáticas nelas presentes. Na função *Potential()* (figura 6) decidimos eliminar uma operação que era constantemente chamada e apenas calcula-la uma vez por chamada de função (ponto 1), e ainda reescrevemos a função simplificando as operações matemáticas reduzindo assim a carga de operações, segundo o esboço ilustrado na figura 7 (ponto 2), obtendo como resultado a função potencial ilustrada figura 8. Quanto à função *computeAccelerations()* (figura 9) decidimos começar por retirar os ciclos que iteram sobre a variável k uma vez que assim podemos reduzir o número de operações e ainda usufruir de hierarquia de memoria, uma vez que os acessos a zonas de memoria próximas são efetuadas em sequência, e operações repetitivas são eliminadas (ponto 1), por fim assim como na função potencial decidimos reescrever, simplificando, a operação matemática segundo o rascunho ilustrado na figura 10 (ponto 2) e obtivemos uma função *computeAccelerations()* como a ilustrada na figura 11. Nesta fase fizemos ainda umas alterações na Makefile (figura 12) das quais destacamos a utilização das seguintes flags:
-funroll-loops: é uma técnica de otimização que permite substituir os ciclos de um programa por cópias repetidas no corpo do mesmo, e é capaz de reduzir a sobrecarga do ciclo.
-fno-omit-frame-pointers: que evita a otimização que remove os ”frame pointers”e mostra-se útil para debug.
-ffast-math: é uma técnica otimiza o código alterando o comportamento de operações matemáticas como reordenar operações ou associar operações entre outos.

Como resultado desta etapa obtivemos um tempo de execução de 291 segundos (figura 13), e analisando com *Profiling* podemos observar que as funções mais custosas continuam a ser as mesmas (figura 14).

Tabela II
Resultados de Performance Iniciais

Tempo(s)	***Instruções***	***Ins/Ciclo***	***Ciclos***
18,15	$0.638 * 10^12$	1.22	$0.523 * 10^12$

C. Otimização da computeAcceleration() e da Potential()

Apesar de todas as mudanças já efetuadas continuamos com uma perda de desempenho considerável nestas duas funções, então começamos por realizar analise mais pormenorizada à função *Potential()* (figura 15), onde verificamos que as operações matemáticas nela presentes podiam ainda ser simplificadas segundo o esboço ilustrado na figura 16 (ponto 1). Para além disso colocamos a variável *ep_4 = 4.*, assim como

a variável *N = 2160*, como variáveis globais constantes (ponto 2). Verificamos também acessos a posições na matriz repetidos e reduzimos esses acessos aos extremamente necessários (ponto 4). Por fim eliminamos o verificação *if* no corpo dos ciclos reformatando a função para ter um ciclo ate o j ter o mesmo valor de i e outro quando o j é maior que i até N, enviesando reduzir o número de operações no interior do ciclo que itera a variável i (ponto 3), obtendo uma função *Potential()* como a ilustrada na figura 17.

Quanto à função *computeAcceleration()* (figura 18) observamos que também poderíamos corrigir os acessos replicados a posições na memoria iguais (ponto 1), para alem disso passamos a tarefa de inicializar a *matriz a* para uma outra função que é chamada antes da mesma para obter uma melhor organização do (ponto 2), figura 19.

Nesta etapa obtivemos como resultado destas alterações um tempo de execução de 9.45 segundos (figura 20) e assim como em todas as etapas ate aqui o resultado do uso do *Profiling* indica-nos que a lacuna do nosso código continuam a ser as funções *Potential()* e *computeAcceleration()* (figura 21).

Tabela III
RESULTADOS DE PERFORMANCE INICIAIS

Tempo(s)	***Instruções***	***Ins/Ciclo***	***Ciclos***
9.45	$0.441 * 10^1 2$	1.63	$0.271 * 10^1 2$

D. Incorporação de funções semelhantes

1) Potential() e computeAcceleration() : Observando cautelosamente a função potencial da maneira que foi implementada na etapa anterior concluímos que a implementação atual utiliza dois ciclos para calcular todas as combinações possíveis de pares de partículas (i, j). Isso leva a uma duplicação dos cálculos, uma vez que a energia potencial entre as partículas i e j é calculada duas vezes (uma vez no primeiro ciclo onde i $\ll$ j e novamente no segundo ciclo onde i $\gg$ j).

Para otimizar o código, eliminamos a duplicação de cálculos e reduzimos a complexidade computacional, obtendo uma função com a ilustrada na figura 22, onde apenas iteramos uma única vez por todos os pares i j e duplicamos o valor de *ep_4 = 4.* para *ep_8 = 8.*.

Com esta alteração o tempo de execução passou para 11.8 segundos (figura 23) e agora passamos a ter como função mais custosa a computeAcceleration(), figura 24.

Tabela IV
RESULTADOS DE PERFORMANCE INICIAIS

Tempo(s)	***Instruções***	***Ins/Ciclo***	***Ciclos***
11.8	$0.335 * 10^1 2$	0.98	$0.342 * 10^1 2$

Apesar de esta otimização não resultar numa grande evolução permitiu notar que as funções *Potential()* e *computeAcceleration()* iteram agora sobre os mesmos valores nos ciclos que as compõem e ainda tem uma grande parte do código semelhante. Então deparados com este facto decidimos unificar esses cálculos comuns em uma única função capaz de devolver o valor de potencia esperado e realizar os cálculos esperados pela *computeAcceleration()*. Para tal tivemos de fazer uma reformulação do código original tornando a variável *"PE"* global para que o valor dela possa ser alterado dentro da função *VelocityVerlet()*, onde esta função criada ira ser chamada, função essa ilustrada na figura 25.

E agora sim os resultados obtidos mostram uma grande otimização do código que agora executa em 5.66 segundos (figura 26), como era de esperar a única função custosa no nosso código é a função agora criada, figura 27.

Tabela V
RESULTADOS DE PERFORMANCE INICIAIS

Tempo(s)	***Instruções***	***Ins/Ciclo***	***Ciclos***
5.66	$0.25 * 10^1 2$	1.53	$0.164 * 10^1 2$

2) MeanSquaredVelocity() e Kinetic() : Apesar da informação revelada pelo *Profiling* não identificar que as funções *MeanSquaredVelocity()* e *Kinetic()* não representam custos relevantes para a execução do código notamos que as mesmas São chamadas uma após a outra e que os acessos á memoria realizados nas duas correspondem, para tal, e enviesando baixar o número de instruções total decidimos também unificar as duas como ilustrado na figura 28.

E. Vetorização

Por fim assim como falado nas aulas praticas decidimos utilizar vetorização, ao invés do uso de matrizes de ordem 5000 por 3, decidimos utilizar arrays de 15000 posições (figura 29), que permitem um melhor uso da hierarquia de memória aproveitando de certa maneira a eficiência da cache, resultando em ganhos significativos de desempenho. Nesta fase mexemos como é lógico em todas as funções implementadas mas usamos a função *computeAccelerations_plus_potential* (figura 30) por ser mais complexa como exemplo.

Com esta fase obtivemos uma melhoria nos resultados onde atingimos os 5.08 seg de TEXEC (figura 31) e com o *Profiling* identificamos que a lacuna se mantinha (figura 32).

Tabela VI
RESULTADOS DE PERFORMANCE INICIAIS

Tempo(s)	***Instruções***	***Ins/Ciclo***	***Ciclos***
5.08	$0.203 * 10^1 2$	1.38	$0.146 * 10^1 2$

III. NOTAS FINAIS

Apesar da drástica evolução que conseguimos obter, o grupo continuou insatisfeito por isso tentamos ainda otimizar a função *computeAccelerations_plus_potential* modificando o seu ciclo interno para que ele fizesse a cada iteração o mesmo que iria fazer em três consecutivas, claro tendo em atenção todas as posições obtendo uma função como a ilustrada na figura 33, porem os resultados obtidos foram piores (figura 34) então decidimos recuar nesta nossa iniciativa, o que nos levou a apresentar a nossa ultima versão que era a mais otimizada.

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

        291718,27 msec task-clock                #    1,000 CPUs utilized
               93      context-switches          #    0,000 K/sec
                0      cpu-migrations            #    0,000 K/sec
              874      page-faults               #    0,003 K/sec
      890736088291      cycles                    #    3,053 GHz
      546888977093      stalled-cycles-frontend   #   61,40% frontend cycles idle
     1244026326805      instructions              #    1,40  insn per cycle
                                                  #    0,44  stalled cycles per insn
      129360635984      branches                  #  443,444 M/sec
         427345258      branch-misses             #    0,33% of all branches

    291,716157305 seconds time elapsed

    291,714745000 seconds user
      0,004321000 seconds sys
```

Figura 1. Resultado na estaca 0

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 63.17      35.43    35.43      201   176.29   176.29  Potential()
 36.85      56.10    20.67      202   102.33   102.33  computeAccelerations()
  0.07      56.14     0.04      201     0.20   102.53  VelocityVerlet(double, int, _IO_FILE*)
  0.00      56.14     0.00     6480     0.00     0.00  gaussdist()
  0.00      56.14     0.00      201     0.00     0.00  MeanSquaredVelocity()
  0.00      56.14     0.00      201     0.00     0.00  Kinetic()
  0.00      56.14     0.00        1     0.00     0.00  initialize()
  0.00      56.14     0.00        1     0.00     0.00  initializeVelocities()
```

Figura 2. Profiling na estaca 0

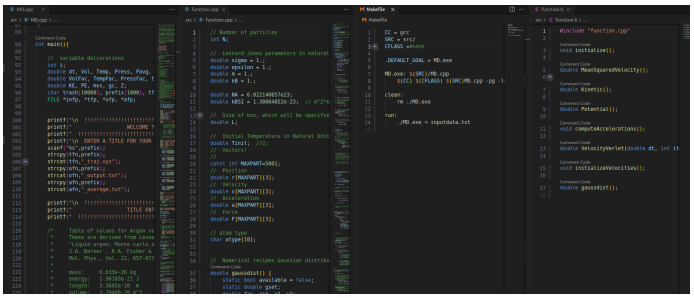

Figura 3. Código mudolarizado

```
    for (i=0; i<N; i++) {
        v[i][0] -= vCM[0];
        v[i][1] -= vCM[1];
        v[i][2] -= vCM[2];
    }

    //  Now we want to scale the average velocity of the system
    //  by a factor which is consistent with our initial temperature, Tinit
    double vSqdSum, lambda;
    vSqdSum=0.;
    for (i=0; i<N; i++) {
        vSqdSum += (v[i][0]*v[i][0] + v[i][1]*v[i][1] + v[i][2]*v[i][2]);
    }

    lambda = sqrt( 3*(N-1)*Tinit/vSqdSum);

    for (i=0; i<N; i++) {
        v[i][0] *= lambda;
        v[i][1] *= lambda;
        v[i][2] *= lambda;
    }
}
```

Figura 4. Exemplo de alteração para reduzir o número de ciclos

```
CC = gcc
SRC = src/
CFLAGS = -funroll-loops -O2 -pg

.DEFAULT_GOAL = MD.exe

MD.exe: $(SRC)/MD.cpp
    $(CC) $(SRC)MD.cpp $(CFLAGS) -lm -o MD.exe

clean:
    rm ./MD.exe

run:
    ./MD.exe < inputdata.txt
```

Figura 5. Primeira alteração da Makefile

```
// Function to calculate the potential energy of the system
double Potential() {
    double quot, r2, 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) {
                r2=0.;
                r2 += (r[i][0]-r[j][0])*(r[i][0]-r[j][0]) + (r[i][1]-r[j][1])*(r[i][1]-r[j][1])
                + (r[i][2]-r[j][2])*(r[i][2]-r[j][2]);
                rnorm=sqrt(r2);
                quot=sigma/rnorm;
                term1 = pow(quot,12.);
                term2 = pow(quot,6.);

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

    return Pot;
}
```

Figura 6. Função Potencial() antes das alterações

$$norm = \sqrt{r^2}$$

$$quot = sigma/norm$$

$$term1 = (quot)^{12}$$

$$term2 = (quot)^{6}$$

$$Pot = Pot + (ep_4 * (term1 - term2))$$

$$<=>$$

$$term1 = (quot * quot * quot)^{4}$$

$$term2 = (quot * quot * quot)^{2}$$

$$Pot = Pot + (ep_4 * (term1 - term2))$$

$$<=>$$

$$aux = quot * quot * quot$$

$$term2 = aux * aux$$

$$term1 = term2 * term2$$

$$Pot = Pot + (ep_4 * (term2 * term2 - term2))$$

$$<=>$$

$$aux = quot * quot * quot$$

$$term2 = aux * aux$$

$$Pot = Pot + (ep_4 * term2(term2 - 1))$$

Figura 7. Rascunho da simplificação das operações matemáticas

```c
// Function to calculate the potential energy of the system
double Potential() {
    double quot, r2, rnorm, term1, term2, Pot, aux;
    int i, j, k;

    Pot=0.;
    double ep_4 = 4. * epsilon;
    for (i = 0; i < N-1; i++) {   // loop over all distinct pairs i,j
        for (j = i+1; j < N; j++) {

            if (j!=i) {
                r2=0.;
                r2 += (r[i][0]-r[j][0])*(r[i][0]-r[j][0])
                + (r[i][1]-r[j][1])*(r[i][1]-r[j][1])
                + (r[i][2]-r[j][2])*(r[i][2]-r[j][2]);

                rnorm = sqrt(r2);
                quot = sigma/rnorm;
                aux = quot * quot * quot;
                term2 = aux * aux;

                Pot += ep_4 * term2 * (term2 - 1);
            }
        }
    }

    return Pot;
}
```

Figura 8. Função Potencial() após as primeiras alterações

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

    for (i = 0; i < N; i++) {  // set all accelerations to zero
        a[i][0] = 0.;
        a[i][1] = 0.;
        a[i][2] = 0.;
    }
    for (i = 0; i < N-1; i++) {   // loop over all distinct pairs i,j
        for (j = i+1; j < N; j++) {
            // initialize r^2 to zero
            rSqd = 0;

            for (k = 0; k < 3; k++) {
                //  component-by-componenent position of i relative to j
                rij[k] = r[i][k] - r[j][k];
                //  sum of squares of the components
                rSqd += rij[k] * rij[k];
            }

            //  From derivative of Lennard-Jones with sigma and epsilon set equal t
            f = 24. *(2. * pow(rSqd, -7) - pow(rSqd, -4));
            for (k = 0; k < 3; k++) {
                //  from F = ma, where m = 1 in natural units!
                double aux = rij[k] * f;
                a[i][k] += aux;
                a[j][k] -= aux;
            }
        }
    }
}
```

Figura 9. Função computeAccelerations() antes das alterações

$$f = 24 * (2 * rSqd^{-7} - rSqd^{-4})$$

<=>

$$f = 24 * \left(2 * \frac{1}{rSqd^{7}} - \frac{1}{rSqd^{4}}\right)$$

<=>

$$f = \frac{48}{rSqd^{7}} - \frac{24 * rSqd^{3}}{rSqd^{7}}$$

<=>

$$\text{aux} = \text{rSqd} * \text{rSqd} * \text{rSqd}$$

$$f = (48 - 24 * aux)/(aux * aux * rSqd)$$

Figura 10. Rascunho da simplificação das operações matemáticas

```c
//  accelleration of each atom.
void computeAccelerations() {
    int i, j, k;
    double f, rSqd, aux, aux0, aux1, aux2;
    double rij[3]; // position of i relative to j


    for (i = 0; i < N; i++) {  // set all accelerations to zero
        a[i][0] = 0.;
        a[i][1] = 0.;
        a[i][2] = 0.;
    }
    for (i = 0; i < N-1; i++) {   // loop over all distinct pairs i,j
        for (j = i+1; j < N; j++) {
            // initialize r^2 to zero
            rSqd = 0;
            rij[0] = r[i][0]-r[j][0];
            rij[1] = r[i][1]-r[j][1];
            rij[2] = r[i][2]-r[j][2];
            rSqd += rij[0] * rij[0];
            rSqd += rij[1] * rij[1];
            rSqd += rij[2] * rij[2];

            //  From derivative of Lennard-Jones with sigma and epsilon set equal to 1 in natural un
            aux = rSqd * rSqd * rSqd;
            f = (48.-24. * aux)/(aux * aux * rSqd);

            aux0 = rij[0] * f;
            aux1 = rij[1] * f;
            aux2 = rij[2] * f;
            a[i][0] += aux0;
            a[i][1] += aux1;
            a[i][2] += aux2;

            a[j][0] -= aux0;
            a[j][1] -= aux1;
            a[j][2] -= aux2;
        }
    }
}
```

Figura 11. Função computeAccelerations() após as primeiras alterações

```
CC = gcc
SRC = src/
CFLAGS = -O3 -fno-omit-frame-pointer -ffast-math -funroll-loops -pg

.DEFAULT_GOAL = MD.exe

MD.exe: $(SRC)/MD.cpp
    $(CC) $(CFLAGS) $(SRC)MD.cpp -lm -o MD.exe

clean:
    rm ./MD.exe
    rm ./gmon.out
    rm cp_average.txt
    rm cp_output.txt
    rm cp_traj.xyz

run:
    srun --partition=cpar perf stat ./MD.exe < inputdata.txt

run1:
    perf stat ./MD.exe < inputdata.txt

test:
    diff cp_output.txt Correct_res/cp_output.txt
    diff cp_average.txt Correct_res/cp_average.txt
    diff cp_traj.xyz Correct_res/cp_traj.txt
```

Figura 12. Segunda alteração na Makefile

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

         18152,06 msec task-clock                #    1,000 CPUs utilized
               21      context-switches          #    0,001 K/sec
                0      cpu-migrations            #    0,000 K/sec
              512      page-faults               #    0,028 K/sec
      52357879743      cycles                    #    2,884 GHz
      35159455438      stalled-cycles-frontend   #   67,15% frontend cycles idle
      63818028741      instructions              #    1,22  insn per cycle
                                                 #    0,55  stalled cycles per insn
       5871363541      branches                  #  323,454 M/sec
          1703223      branch-misses             #    0,03% of all branches

    18,156342146 seconds time elapsed

    18,151587000 seconds user
     0,001127000 seconds sys


pg53886@search7edu TrabalhoV3]$
```

Figura 13. Resultado na etapa B

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 72.78     12.90    12.90                             Potential()
 27.27     17.74     4.83      201    24.05    24.05  computeAccelerations()
  0.06     17.75     0.01                             VelocityVerlet(double, int, _IO_FILE*)
  0.00     17.75     0.00     3240     0.00     0.00  frame_dummy

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
```

Figura 14. Profiling na etapa B

$$norm = \sqrt{r^2}$$

$$quot = 1/norm$$

$$aux = quot * quot * quot$$

$$term2 = aux * aux$$

<=>

$$aux = \frac{1}{\sqrt{r^2}} * \frac{1}{\sqrt{r^2}} * \frac{1}{\sqrt{r^2}}$$

$$term2 = aux * aux$$

<=>

$$aux = \frac{1}{r^2 * \sqrt{r^2}}$$

$$term2 = \frac{1}{r^2 * \sqrt{r^2}} * \frac{1}{r^2 * \sqrt{r^2}}$$

<=>

$$term2 = \frac{1}{r^2 * r^2 * r^2}$$

Figura 16. Rascunho da simplificação das operações matemáticas

```
// Function to calculate the potential energy of the system
double Potential() {
    double quot, r2, rnorm, term1, term2, Pot, aux;
    int i, j, k;

    Pot=0.;
    double ep_4 = 4. * epsilon; 2
    for (i = 0; i < N-1; i++) {    // loop over all distinct pairs i,j
        for (j = i+1; j < N; j++) {

            if (j!=i) { 3
                r2=0.;
                r2 += (r[i][0]-r[j][0])*(r[i][0]-r[j][0])
                + (r[i][1]-r[j][1])*(r[i][1]-r[j][1])
                + (r[i][2]-r[j][2])*(r[i][2]-r[j][2]);

                rnorm = sqrt(r2);
                quot = sigma/rnorm;          1
                aux = quot * quot * quot;
                term2 = aux * aux;

                Pot += ep_4 * term2 * (term2 - 1);
            }
        }
    }

    return Pot;
}
```

Figura 15. Função Potencial() com as alterações destacadas

```
double ep_4 = 4.;
// Function to calculate the potential energy of the system
double Potential() {
    double r2, term2, Pot;
    double ri0, ri1, ri2;
    double Mij0, Mij1, Mij2;
    int i, j, k;

    Pot=0.;
    for (i = 0; i < N; i++) {
        ri0 = r[i][0];
        ri1 = r[i][1];
        ri2 = r[i][2];
        for (j = 0; j < i; j++) {
            Mij0 = ri0 - r[j][0];
            Mij1 = ri1 - r[j][1];
            Mij2 = ri2 - r[j][2];
            r2 = Mij0 * Mij0 + Mij1 * Mij1 + Mij2 * Mij2;

            term2 = 1/(r2 * r2 * r2);

            Pot += ep_4 * term2 * (term2 - 1);
        }
        for(j = i + 1; j < N; j++){
            Mij0 = ri0 - r[j][0];
            Mij1 = ri1 - r[j][1];
            Mij2 = ri2 - r[j][2];
            r2 = Mij0 * Mij0 + Mij1 * Mij1 + Mij2 * Mij2;

            term2 = 1/(r2 * r2 * r2);

            Pot += ep_4 * term2 * (term2 - 1);
        }
    }
    return Pot;
}
```

Figura 17. Função Potential() após as alterações

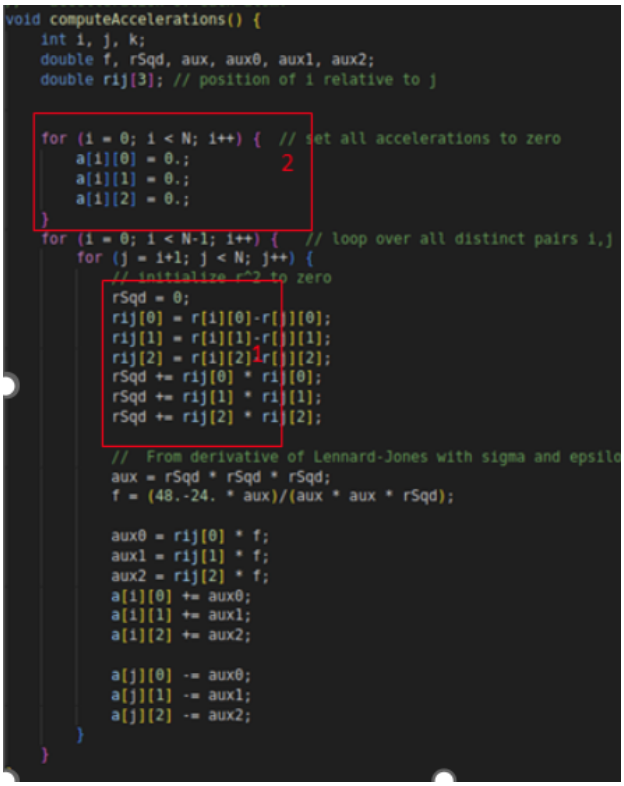

Figura 18. Função computeAcceleration() com as alterações destacadas

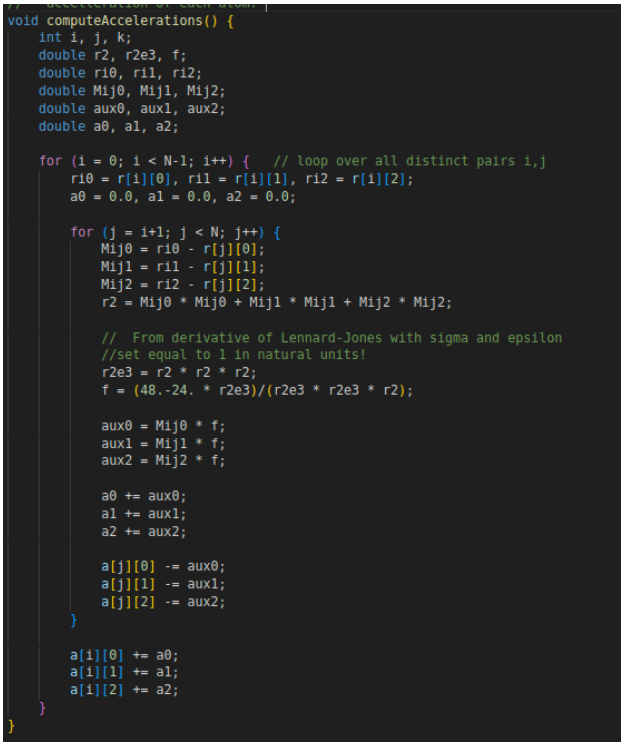

Figura 19. Função computeAcceleration() após as alterações

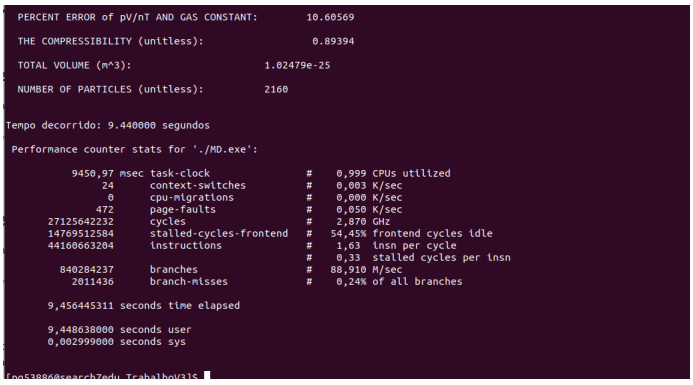

Figura 20. Resultado na etapa C

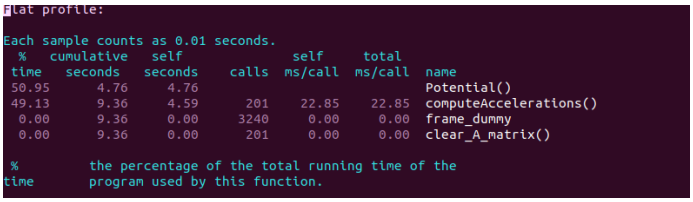

Figura 21. Profiling na etapa C

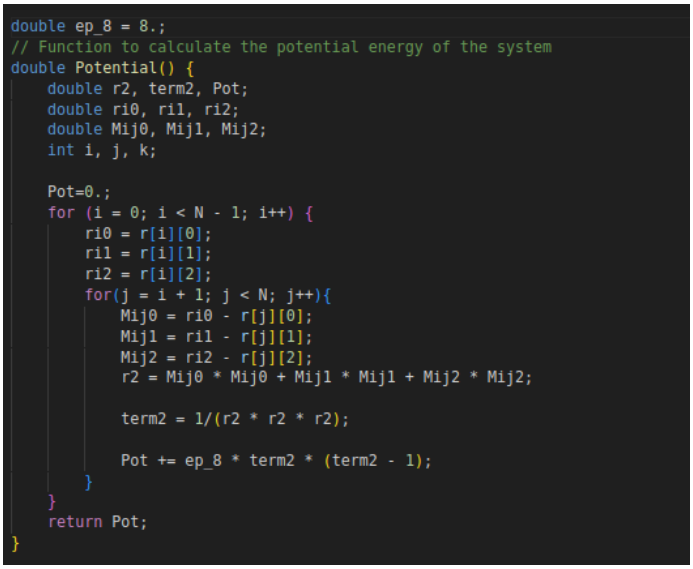

Figura 22. Função Potential() com um único ciclo no seu corpo

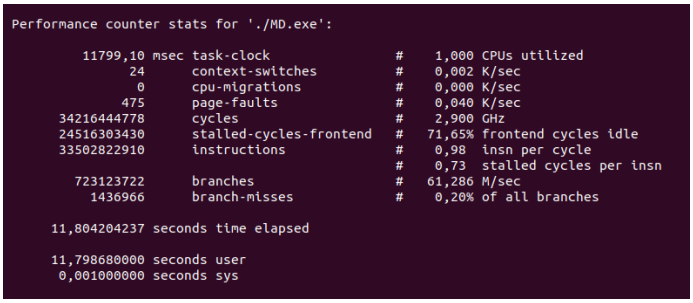

Figura 23. Resultado de unificar os ciclos da função Potential()

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 65.20      7.69     7.69      201    38.24    38.24  computeAccelerations()
 34.81     11.79     4.10                             Potential()
  0.08     11.80     0.01                             MeanSquaredVelocity()
  0.00     11.80     0.00     3240     0.00     0.00  frame_dummy
  0.00     11.80     0.00      201     0.00     0.00  clear_A_matrix()
```

Figura 24. Profiling após unificar os ciclos da função Potential()

```
void computeAccelerations_plus_potential() {
    int i, j, k;
    double r2, r2e3, f, term2;
    double ri0, ri1, ri2, Mij0, Mij1, Mij2;
    double aux0, aux1, aux2, a0, a1, a2;

    PE = 0.;
    for (i = 0; i < N-1; i++) {   // loop over all distinct pairs i,j
        ri0 = r[i][0], ri1 = r[i][1], ri2 = r[i][2];
        a0 = 0.0, a1 = 0.0, a2 = 0.0;

        for (j = i+1; j < N; j++) {
            Mij0 = ri0 - r[j][0], Mij1 = ri1 - r[j][1], Mij2 = ri2 - r[j][2];
            r2 = Mij0 * Mij0 + Mij1 * Mij1 + Mij2 * Mij2;

            //  From derivative of Lennard-Jones with sigma and epsilon
            //set equal to 1 in natural units!
            r2e3 = r2 * r2 * r2;
            term2 = 1/r2e3;

            f = (48.-24. * r2e3)/(r2e3 * r2e3 * r2);
            PE += ep_8 * term2 * (term2 - 1);

            aux0 = Mij0 * f;
            aux1 = Mij1 * f;
            aux2 = Mij2 * f;

            a0 += aux0;
            a1 += aux1;
            a2 += aux2;

            a[j][0] -= aux0;
            a[j][1] -= aux1;
            a[j][2] -= aux2;
        }

        a[i][0] += a0;
        a[i][1] += a1;
        a[i][2] += a2;
    }
}
```

Figura 25. Função computeAccelerations_plus_potential()

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

         5659,28 msec task-clock                #    0,999 CPUs utilized
              25      context-switches          #    0,004 K/sec
               0      cpu-migrations            #    0,000 K/sec
             414      page-faults               #    0,073 K/sec
    16411397276      cycles                    #    2,900 GHz
     9316935113      stalled-cycles-frontend   #   56,77% frontend cycles idle
    25031369777      instructions              #    1,53  insn per cycle
                                               #    0,37  stalled cycles per insn
      481484262      branches                  #   85,079 M/sec
         572605      branch-misses             #    0,12% of all branches

    5,664375072 seconds time elapsed

    5,657434000 seconds user
    0,002264000 seconds sys
```

Figura 26. Resultado de unificar as funções Potential() e computeAcceleration()

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.10      5.67     5.67      201    28.19    28.19  computeAccelerations_plus_potential()

 %         the percentage of the total running time of the
time       program used by this function.
```

Figura 27. Profiling após unificar as funções Potential() e computeAcceleration()

```
void MeanSquaredVelocity_and_Kinetic(){
    double velo = 0., v2, kin;

    for(int i=0; i<N; i++){
        v2 = (v[i][0]*v[i][0] + v[i][1]*v[i][1] + v[i][2]*v[i][2]);
        velo += v2;
        kin += m*v2/2.;
    }
    KE = kin;
    mvs = velo/N;
}
```

Figura 28. Função MeanSquaredVelocity_and_Kinetic()

```
//const int MAXPART=5001
#define MAXPART 15000
//  Position
double r[MAXPART];
//  Velocity
double v[MAXPART];
//  Acceleration
double a[MAXPART];
//  Force
double F[MAXPART];
```

Figura 29. Reformulação nas estruturas de dados

```
void computeAccelerations_plus_potential(){
    int i, j;
    double f, rSqd, term2, ri0, ri1, ri2, M0, M1, M2, aux, aux0, aux1, aux2, a0, a1, a2;
    PE = 0.;
    for (i = 0; i < limitM1; i += 3) { // loop over all distinct pairs i, j
        a0 = 0.0, a1 = 0.0, a2 = 0.0;

        ri0 = r[i];
        ri1 = r[i + 1];
        ri2 = r[i + 2];

        for (j = i + 3; j < limit; j += 3) {
            M0 = ri0 - r[j], M1 = ri1 - r[j + 1], M2 = ri2 - r[j + 2];

            rSqd = M0 * M0 + M1 * M1 + M2 * M2;

            aux = rSqd * rSqd * rSqd;
            term2 = 1. / aux;
            f = (48. - 24. * aux) / (aux * aux * rSqd);
            PE += ep_8 * term2 * (term2 - 1.);

            aux0 = M0 * f;
            aux1 = M1 * f;
            aux2 = M2 * f;

            a0 += aux0;
            a1 += aux1;
            a2 += aux2;

            a[j] -= aux0;
            a[j + 1] -= aux1;
            a[j + 2] -= aux2;
        }
        a[i] += a0;
        a[i + 1] += a1;
        a[i + 2] += a2;
    }
}
```

Figura 30. Função computeAccelerations_plus_potential() vetorizada

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

         5067,45 msec task-clock                #    0,997 CPUs utilized
              27      context-switches          #    0,005 K/sec
               0      cpu-migrations            #    0,000 K/sec
             406      page-faults               #    0,080 K/sec
    14695126830      cycles                    #    2,900 GHz
     8557310466      stalled-cycles-frontend   #   58,23% frontend cycles idle
    20302906580      instructions              #    1,38  insn per cycle
                                               #    0,42  stalled cycles per insn
      479261109      branches                  #   94,576 M/sec
         561988      branch-misses             #    0,12% of all branches

    5,083499463 seconds time elapsed

    5,064922000 seconds user
    0,002999000 seconds sys
```

Figura 31. Resultado da vetorização

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.71      5.05     5.05      201    25.10    25.10  computeAccelerations_plus_potential()
```

Figura 32. Profiling na vetorização

```
void computeAccelerations_plus_potential(){
    int i, j;
    double f, rSqd, term2, ri0, ri1, ri2, M0, M1, M2, r2e3, aux0, aux1, aux2, a0, a1, a2;
    double M01, M11, M21, M02, M12, M22 , r2e31, aux01, aux11, aux21, r2e32, aux02, aux12, aux22;
    double rSqd1 , rSqd2, f1, f2, term21, term22;
    PE = 0.;
    for (i = 0; i < limitM1; i += 3) { // loop over all distinct pairs i, j
        a0 = 0.0, a1 = 0.0, a2 = 0.0;

        ri0 = r[i];
        ri1 = r[i + 1];
        ri2 = r[i + 2];

        for (j = i + 3; j < limitM9; j += 9) {
            M0 = ri0 - r[j], M1 = ri1 - r[j + 1], M2 = ri2 - r[j + 2];
            M01 = ri0 - r[j+3], M11 = ri1 - r[j + 4], M21 = ri2 - r[j + 5];
            M02 = ri0 - r[j+6], M12 = ri1 - r[j + 7], M22 = ri2 - r[j + 8];

            rSqd = M0 * M0 + M1 * M1 + M2 * M2;
            rSqd1 = M01 * M01 + M11 * M11 + M21 * M21;
            rSqd2 = M02 * M02 + M12 * M12 + M22 * M22;

            r2e3 = rSqd * rSqd * rSqd;
            r2e31 = rSqd1 * rSqd1 * rSqd1;
            r2e32 = rSqd2 * rSqd2 * rSqd2;

            f = (48. - 24. * r2e3) / (r2e3 * r2e3 * rSqd);
            f1 = (48. - 24. * r2e31) / (r2e31 * r2e31 * rSqd1);
            f2 = (48. - 24. * r2e32) / (r2e32 * r2e32 * rSqd2);

            term2 = 1/r2e3;
            term21 = 1/r2e31;
            term22 = 1/r2e32;

            PE += ep_8 * term2 * (term2 - 1);
            PE += ep_8 * term21 * (term21 - 1);
            PE += ep_8 * term22 * (term22 - 1);

            aux0 = M0 * f, aux1 = M1 * f, aux2 = M2 * f;
            aux01 = M01 * f1, aux11 = M11 * f1, aux21 = M21 * f1;
            aux02 = M02 * f2, aux12 = M12 * f2, aux22 = M22 * f2;

            a0 += aux0 + aux01 + aux02;
            a1 += aux1 + aux11 + aux12;
            a2 += aux2 + aux21 + aux22;

            a[j] -= aux0, a[j + 1] -= aux1, a[j + 2] -= aux2;
            a[j+3] -= aux01, a[j + 4] -= aux11, a[j + 5] -= aux21;
            a[j+6] -= aux02, a[j + 7] -= aux12, a[j + 8] -= aux22;
        }
        for (; j < limit; j += 3) {
            M0 = ri0 - r[j], M1 = ri1 - r[j + 1], M2 = ri2 - r[j + 2];

            rSqd = M0 * M0 + M1 * M1 + M2 * M2;

            r2e3 = rSqd * rSqd * rSqd;
            f = (48. - 24. * r2e3) / (r2e3 * r2e3 * rSqd);
            term2 = 1/r2e3;
            PE += ep_8 * term2 * (term2 - 1);

            aux0 = M0 * f;
            aux1 = M1 * f;
            aux2 = M2 * f;

            a0 += aux0;
            a1 += aux1;
            a2 += aux2;

            a[j] -= aux0;
            a[j + 1] -= aux1;
            a[j + 2] -= aux2;
        }

        a[i] += a0, a[i + 1] += a1, a[i + 2] += a2;
    }
}
```

Figura 33. Tentativa de melhorar a função computeAccelerations_plus_potential()

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

          5807,44 msec task-clock                #    0,999 CPUs utilized
                20      context-switches          #    0,003 K/sec
                 0      cpu-migrations            #    0,000 K/sec
               410      page-faults               #    0,071 K/sec
     16841069411      cycles                    #    2,900 GHz
     10767412307      stalled-cycles-frontend   #   63,94% frontend cycles idle
     21264370178      instructions              #    1,26  insn per cycle
                                                #    0,51  stalled cycles per insn
       168861796      branches                  #   29,077 M/sec
          852583      branch-misses             #    0,50% of all branches

     5,810994668 seconds time elapsed

     5,806809000 seconds user
     0,001000000 seconds sys
```

Figura 34. Resultado da tentativa