# Trabalho Prático N.º 1

Computação Paralela, Mestrado Integrado em Engenharia Informática, Universidade do Minho

**Afonso Miguel Matos Bessa**
MEI
Universidade do Minho
pg53597

**Francisco Luís Rodrigues Claudino**
MEI
Universidade do Minho
pg50380

***Abstract*—Este trabalho tem como objetivo avaliar a aprendizagem das técnicas de otimização de código, através da análise do código e com recurso a ferramentas de análise de execução do mesmo.**

## Análise do Problema

A simulação de dinâmica molecular é essencial para entender como é que as partículas se comportam em sistemas complexos, permitindo-nos estudar átomos e moléculas em escalas microscópicas. Este campo admite aplicação em diversos ramos, como química, física e biologia, onde se investiga o comportamento das partículas sob diferentes condições.

Neste projeto, concentramo-nos em simulações de dinâmica molecular para vários tipos de gases, mais especificamente átomos de árgon, um tipo de gás nobre conhecido pela sua estabilidade. A base deste método segue os princípios das leis de Newton, descrevendo o movimento das partículas mediante a interação entre forças e acelerações. O código atual utiliza o potencial de Lennard-Jones para modelar as interações entre as partículas e o método de integração de Verlet, por forma a calcular as suas trajetórias.

O desafio deste projeto encontra a sua base na otimização deste código de simulação. A otimização é crucial para obter resultados mais rápidos e eficientes, especialmente em sistemas complexos com muitas partículas onde é fundamental acelerar a pesquisa científica e economizar recursos computacionais.

O nosso objetivo é analisar o código existente, identificar as áreas que consomem mais tempo de execução (com o auxilio da ferramenta *gprof*) e implementar melhorias para acelerar o desempenho da simulação, sem comprometer a precisão dos resultados. Procuramos reduzir o tempo de execução da simulação sob condições específicas, como o número de átomos e densidade, garantindo que todas as saídas da simulação permaneçam precisas.

## Exploração e Otimização do Código

### *Makefile*

Comparativamente ao ficheiro Makefile original, fornecido pela equipa docente, foram implementadas melhorias substanciais. Inicialmente, aprimorou-se o *clean* de maneira a que, cada vez que se executasse, se procedesse à limpeza de todos os ficheiros regulares, com a exceção do próprio Makefile e do ficheiro *inputdata.txt*. Posteriormente, introduziram-se as regras `srun1`, `srun2` e `gprof` no Makefile. Quer a regra um como a dois compilam o código-fonte e executam o programa *MD.exe*, realizando medições cruciais de desempenho como o número de instruções e o número de ciclos efetuados, bem como, a contagem de referências e falhas na *cache*. A terceira regra, `gprof`, replica o processo de compilação e execução, mas, desta vez, efetua uma análise minuciosa do desempenho do programa, permitindo identificar as areas onde a simulação é mais demorada. Estas regras representam um notável avanço na avaliação e otimização do código, permitindo a identificação de áreas para aperfeiçoamento e refinamento do desempenho global do programa. Finalmente, foram adicionadas *flags* relativas a cada uma das três regras supramencionadas, às quais, de seguida, iremos dedicar uma explicação pormenorizada.

1) **-ftree-vectorize**: Esta *flag* ativa otimizações de vetorização, que têm como objetivo transformar os ciclos no código em operações vetoriais, com vista a melhorar o desempenho.
2) **-O2 e -O3**: Estas *flags* ativam dois níveis de otimização diferentes. A *-O2* ativa um nível de otimização moderada, aplicando otimizações que melhoram o desempenho do código, sem prejudicar significativamente o tempo de compilação. Por outro lado, a *-O3* habilita um nível mais agressivo de otimização, aplicando uma variedade de técnicas para otimizar o código ao máximo, embora possa resultar em tempos de compilação tendencialmente mais longos.
3) **-msse4**: É uma flag que aproveita o conjunto de instruções SSE4 (Streaming SIMD Extensions 4). Essas instruções SSE4 permitem otimizar o desempenho em operações que envolvem cálculos intensivos, como, por exemplo, manipulação de matrizes.
4) **-mavx**: Esta *flag* é uma extensão da otimização que se aproveita das instruções AVX (Advanced Vector Extensions). As instruções AVX são projetadas para melhorar o desempenho de operações vetoriais, permitindo cálculos mais rápidos e eficientes em conjuntos de dados de tamanho considerável.

Em suma, todas as *flags* configuram o compilador GCC para compilar o código-fonte com otimizações direcionadas à arquitetura específica do processador e para aplicar otimizações de vetorização. Estas otimizações visam aprimorar a qualidade e o desempenho do código, aproveitando as capacidades de processamento vetorial disponíveis, como, por exemplo, o conjunto de instruções SSE4.

### *MD.cpp*

Durante este capítulo vamos enumerar todas as modificações aplicadas ao *MD.cpp*, com o objetivo de tornar o código mais eficiente e legível, mantendo a funcionalidade original.

1. **Remoção das Variáveis Sigma, Epsilon, M e kB:** Foram removidas as variáveis `sigma`, `epsilon`, `m` e `kB` para simplificar o código, uma vez que essas constantes são definidas como um nas unidades naturais usadas no código.

2. **Energia Potencial na Variável Global:** A variável `PE`, que armazena a energia potencial total, foi transformada em uma variável global para permitir o cálculo da energia potencial diretamente na função `computeAccelerations`, eliminando a necessidade da função `Potential`.

3. ***Arrays* de Posição, Velocidade, Aceleração e Tipo de Gás Dinâmicos:** Os *arrays* de posição, velocidade, aceleração e o tipo de

gás foram definidos de forma dinâmica, e a memória é libertada no final da execução do programa.

4. **Supressão de Variáveis e Apontador de Ficheiro Não Utilizados:** As variáveis `trash` e `infp` foram removidas, pois não são utilizadas no código.

5. **Cálculo do Número de Partículas (N) na Inicialização:** O cálculo do número de partículas `N` foi transferido para a função de inicialização, de acordo com a densidade fornecida, o que torna o código mais flexível.

6. **Substituição de `pow` por `cbrt` para o Comprimento da Caixa:** A função `cbrt` é usada para calcular a raiz cúbica do volume da caixa, em vez de usar `pow`, tornando o código ligeiramente mais eficiente.

7. **Supressão do `else` na Clausula `if-else`:** O bloco `else` foi removido da cláusula `if-else` porque ele aplica-se a qualquer gás, exceto para o hélio. Essa lógica simplificada torna o código mais claro.

8. **Função `initialize`:** A função `initialize` foi otimizada com a substituição de `pow` por `cbrt` para calcular o número de átomos em cada direção; Variáveis auxiliares `xPos`, `yPos` e `halfPos` foram introduzidas para reduzir o número de cálculos matemáticos em cada iteração dos ciclos aninhados que inicializam as posições das particulas; A matriz de posições foi vetorizada, resultando em apenas um terço do número de iterações do ciclo no cálculo das posições iniciais das particulas.

9. **Funções `MeanSquaredVelocity` e `Kinetic`:** As funções `MeanSquaredVelocity` e `Kinetic` foram otimizadas com a vetorização da matriz de velocidades, o que permitiu a supressão das variáveis `vx2`, `vy2` e `vz2`; Ambas as funções foram combinadas em uma única função chamada `MeanSquaredVelocityKinetic` para reduzir o número de ciclos, instruções e a necessidade de outra função.

10. **Funções `computeAccelerations` e `Potential`:** As funções `computeAccelerations` e `Potential` foram otimizadas com a vetorização da matriz de acelerações e a matriz de posições, reduzindo o número de ciclos para um terço; Variáveis auxiliares `rSqd3` e `rSqd6` foram introduzidas para otimizar o cálculo da força, eliminando as evocações da função `pow`; A função `Potential` foi incorporada na função `computeAccelerations` para calcular a energia potencial no mesmo ciclo de cálculo das acelerações.

11. **Função `VelocityVerlet`:** O argumento `iter` foi removido, já que não é fornecido ao chamar a função; A variável `dt1` foi criada para reduzir as operações matemáticas na função; As matrizes de posição e velocidade foram vetorizadas, o que permitiu executar apenas um terço das iterações dos ciclos ao calcular as posições, velocidades, atualizar as acelerações e calcular as paredes elásticas; Uma cláusula `if` foi suprimida, pois as operações eram semelhantes, permitindo testar as duas condições na mesma cláusula; O valor a ser retornado foi calculado apenas na operação `return` para reduzir as operações matemáticas no ciclo.

12. **Função `initializeVelocities`:** A matriz de velocidades foi vetorizada, o que permitiu executar apenas um terço das iterações do ciclo ao atribuir um número da distribuição gaussiana, subtrair o centro de massa da velocidade e calcular a velocidade final. A matriz `vSqdSum` foi vetorizada ao calcular o centro de massa de velocidade e ao calcular a escala da velocidade do sistema.

13. **Função `gaussdist`:** Foi criada uma variável `returnValue` para permitir que a função tenha apenas uma operação `return`.

Como já mencionado anteriormente, foram aplicadas várias otimizações de vetorização e reestruturação por forma a melhorar a eficiência da simulação molecular. Isso incluiu a vetorização das matrizes de posições, velocidades e acelerações, transformando-as em arrays unidimensionais. Essa vetorização reduziu o número de ciclos e operações matemáticas necessárias, tornando a simulação mais rápida. Além disso, a otimização do código envolveu a introdução de variáveis auxiliares para minimizar cálculos repetitivos em ciclos, contribuindo para um código mais eficiente e organizado. Esse processo pode ser resumido com a figura 1 do capítulo Anexos.

## Análise de Resultados das Métricas

A execução do código original apresentou uma ineficiência notável, com um tempo de execução que chegou a quase quatro minutos, além de um número extraordinariamente alto de instruções e ciclos, na ordem dos triliões, conforme ilustrado na figura 2.

Depois de aplicarmos todas as otimizações discutidas durante a exploração e melhoria do código, obtivemos duas soluções bastante semelhantes:

- Na primeira abordagem, ao utilizar as *flags* de compilação `-ftree-vectorize -O3 -msse4 -mavx`, conseguimos reduzir drasticamente o número de instruções em cerca de 98.37%, totalizando aproximadamente 20 biliões. Da mesma forma, observamos uma redução significativa de 97.94% no número de ciclos, totalizando cerca de 16 biliões. Essas melhorias resultaram num CPI de 0.8. O Wall Time obtido foi de apenas 5.09 segundos.
- Por outro lado, na segunda abordagem, com a utilização das *flags* de compilação `-ftree-vectorize -O2 -msse4 -mavx`, conseguimos reduzir consideravelmente o número de instruções em aproximadamente 98,25%, totalizando cerca de 22 biliões. Da mesma forma, observamos uma significativa diminuição de 98,10% no número de ciclos, totalizando cerca de 15 biliões. Estas melhorias resultaram num CPI de 0.7. Comparativamente à solução anterior, esta apresenta um Wall Time inferior, de cerca de 4.8 segundos.

Tambem é importante referir que, com o auxílio do *gprof*, uma ferramenta que permite avaliar o desempenho de cada função, bem como a percentagem de tempo que cada uma consome na execução do programa, ao observarmos a Figura 5, que representa a análise do *gprof* no código original, podemos concluir que o programa dedica mais tempo ao cálculo da energia potencial (aproximadamente 64.4%) e ao cálculo das acelerações (cerca de 35.5%). Dada esta análise, o foco desta primeira fase do trabalho prendeu-se na otimização do código da funções *Potential* e *computeAccelerations*, que é onde ocorrem estes cálculos. Por sua vez, no código otimizado (figura 6), como o cálculo da energia potencial foi transportado para a função que cálcula as acelerações, cerca de 99.8% do tempo de execução do programa é gasto nestes dois cálculos, na função *computeAccelerationsPotential*.

## Conclusão

Neste projeto, realizamos uma análise detalhada do código de simulação de dinâmica molecular para átomos de árgon, com o objetivo de otimizá-lo e melhorar o seu desempenho. Implementamos várias otimizações, incluindo a vetorização de matrizes, a simplificação do código e a utilização de *flags* de compilação específicas. As métricas de desempenho mostraram que as otimizações tiveram um impacto significativo, resultando em reduções substanciais no número de instruções e ciclos, tornando o código mais eficiente em termos de recursos computacionais.

Por fim, para otimizar ainda mais o código, podemos recorrer à programação paralela com o uso de *threads*, permitindo a execução de múltiplas instruções em simultâneo, o que elevará o código a um nível superior de otimização.

## ANEXOS

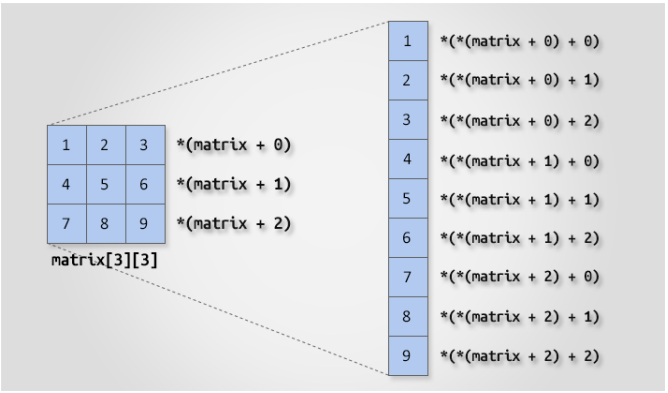


Fig. 1. Vetorização

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

      12,784,002      cache-references
       1,851,049      cache-misses              #   14.479 % of all cache refs
1,243,581,352,681      inst_retired.any          #      0.6 CPI
  779,927,446,905      cycles

    236.663752963 seconds time elapsed

    236.666444000 seconds user
      0.003000000 seconds sys
```

Fig. 2. Código Original

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

         190,209      cache-references
          44,244      cache-misses              #   23.261 % of all cache refs
  20,325,438,535      inst_retired.any          #      0.8 CPI
  16,033,457,413      cycles

     5.044312339 seconds time elapsed

     5.036717000 seconds user
     0.002999000 seconds sys
```

Fig. 3. Versão -03

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

         204,196      cache-references
          42,881      cache-misses              #   21.000 % of all cache refs
  21,739,851,044      inst_retired.any          #      0.7 CPI
  14,871,021,513      cycles

     4.704399469 seconds time elapsed

     4.696584000 seconds user
     0.002999000 seconds sys
```

Fig. 4. Versão -02

| time | seconds | seconds | calls | ms/call | ms/call | name |
|---|---|---|---|---|---|---|
| 64.41 | 56.15 | 56.15 | 201 | 279.33 | 279.33 | Potential() |
| 35.48 | 87.08 | 30.93 | 202 | 153.12 | 153.12 | computeAccelerations() |
| 0.20 | 87.25 | 0.17 | 201 | 0.85 | 153.97 | VelocityVerlet(double, int, _IO_FILE*) |
| 0.01 | 87.26 | 0.01 | 201 | 0.05 | 0.05 | MeanSquaredVelocity() |
| 0.00 | 87.26 | 0.00 | 6480 | 0.00 | 0.00 | gaussdist() |
| 0.00 | 87.26 | 0.00 | 201 | 0.00 | 0.00 | Kinetic() |
| 0.00 | 87.26 | 0.00 | 1 | 0.00 | 0.00 | initialize() |
| 0.00 | 87.26 | 0.00 | 1 | 0.00 | 0.00 | initializeVelocities() |

Fig. 5. GProf Ficheiro Original

| time | seconds | seconds | calls | ms/call | ms/call | name |
|---|---|---|---|---|---|---|
| 99.57 | 11.65 | 11.65 | 201 | 57.96 | 57.96 | computeAccelerationsPotential() |
| 0.69 | 11.73 | 0.08 | | | | VelocityVerlet(double, _IO_FILE*) |
| 0.00 | 11.73 | 0.00 | 6480 | 0.00 | 0.00 | gaussdist() |
| 0.00 | 11.73 | 0.00 | 1 | 0.00 | 0.00 | _GLOBAL__sub_I_N |

Fig. 6. GProf Ficheiro Otimizado