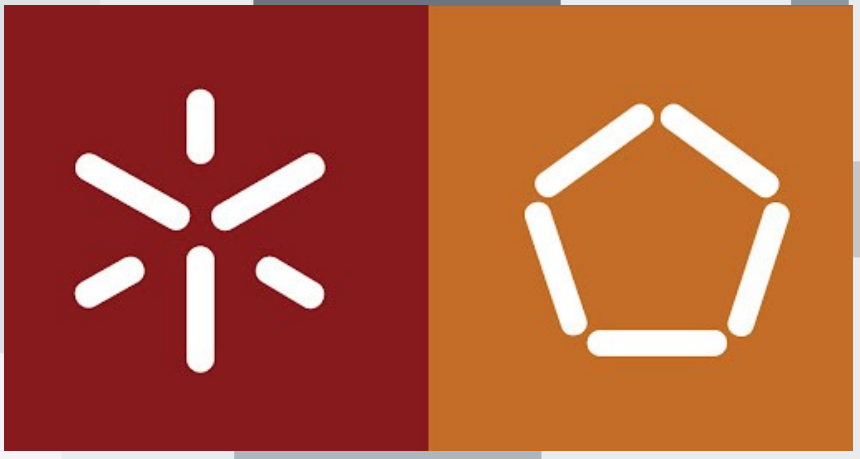


Computação Paralela

MEI - 2023/2024

Afonso Xavier Cardoso Marques - PG53601
Renato André Machado Gomes - PG54174

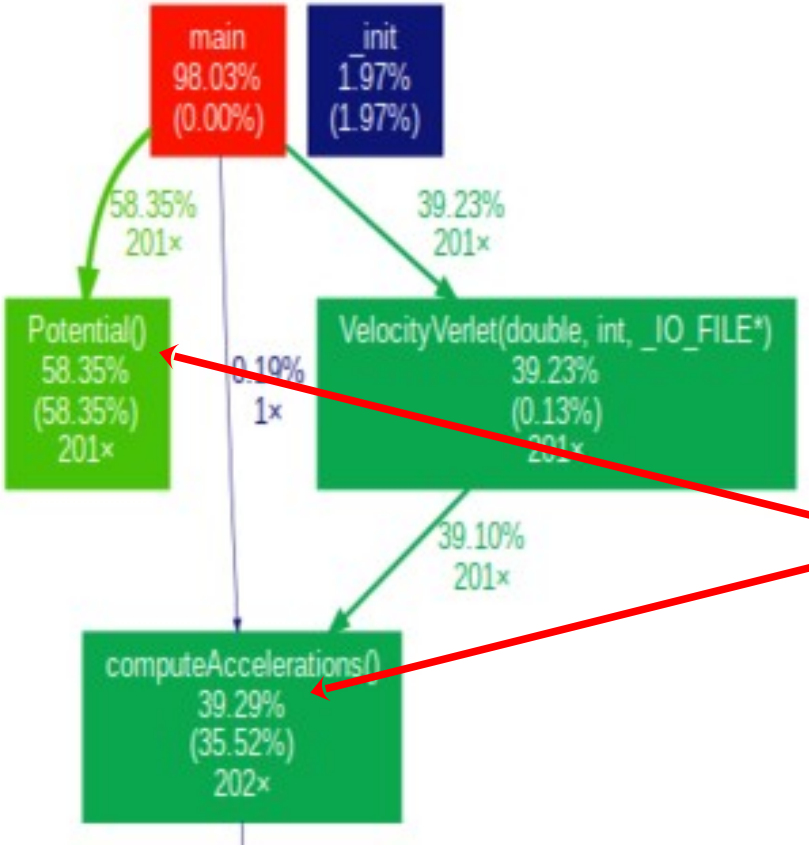
O Programa MD

- O programa a analisar e otimizar faz parte de um código simples de simulação de dinâmica molecular aplicado a átomos de gás argônio. O código segue uma abordagem genérica para simular movimentos de partículas ao longo de passos temporais, utilizando as leis do movimento de Newton.
- O código utiliza o potencial de Lennard Jones para descrever as interações entre duas partículas (força/energia potencial). O método de integração de Verlet é utilizado para calcular as trajetórias das partículas ao longo do tempo. As condições de fronteira (por exemplo, partículas que se movem para fora do espaço de simulação) são geridas por paredes elásticas.

Trabalho Prático 1

- O principal objetivo foi avaliar os benefícios da otimização do código em uma única thread, por meio de alterações em certos algoritmos presentes no código original.
- Em resumo, foram removidos cálculos complexos de dentro de *loops* sendo substituídos por alternativas de melhor desempenho que não interferem nos valores finais de *output*.
- Foram também desconstruídos *loops* com tamanho conhecido, o que permitiu remover instruções adicionais e melhorar o desempenho, mesmo que ligeiramente.
- Realizamos testes usando a ferramenta *perf* para avaliar o desempenho da nossa versão do código.
- Por fim, os resultados obtidos foram analisados e comparados com os da versão original.

Utilizamos a ferramenta *gprof* para avaliar quais os pontos do código que consumiam mais tempo e poder computacional.



- Ao analisar o gráfico conseguimos perceber que as funções que consomem a maior parte do tempo de execução são *Potencial()* e *computeAccelerations().*
- Isto significa que são as melhores candidatas para otimização.

original.txt - Bloco de notas

Ficheiro Editar Formatar Ver Ajuda

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
58.35	17.79	17.79	201	88.51	88.51	Potential()
35.52	28.62	10.83	202	53.61	59.31	computeAccelerations()
3.77	29.77	1.15	942014880	0.00	0.00	__gnu_cxx::__promote_2<decltype (((__gnu_cxx::__promote_2<double, std::__is_integer<double>::__va
1.97	30.37	0.60				_init
0.26	30.45	0.08	1	80.00	80.00	__gnu_cxx::__promote_2<decltype (((__gnu_cxx::__promote_2<int, std::__is_integer<int>::__value>::_
0.13	30.49	0.04	201	0.20	59.51	VelocityVerlet(double, int, _IO_FILE*)
0.00	30.49	0.00	6480	0.00	0.00	gaussdist()
0.00	30.49	0.00	201	0.00	0.00	MeanSquaredVelocity()
0.00	30.49	0.00	201	0.00	0.00	Kinetic()
0.00	30.49	0.00	1	0.00	80.00	initialize()
0.00	30.49	0.00	1	0.00	0.00	initializeVelocities()
0.00	30.49	0.00	1	0.00	0.00	__gnu_cxx::__enable_if<std::__is_integer<int>::__value, double>::__type std::floor<int>(int)

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

cumulative seconds — a running sum of the number of seconds accounted for by this function and those listed above it.

self seconds — the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls — the number of times this function was invoked, if this function is profiled, else blank.

self ms/call — the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total ms/call — the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name — the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Ln 7, Col 77 | 100% | Windows (CRLF) | UTF-8

Na função *Potential()*:

- Removeu-se a função *sqrt()*, passando a ter uma complexidade de cálculos muito mais baixa:

$$\text{rnorm} = \sqrt{r2} \qquad \text{quot} = \frac{sigma}{(\sqrt{2})^2}$$

$$\text{quot} = \frac{sigma}{rnorm} \quad \equiv \quad \text{term1} = quot^6$$

$$\text{term1} = quot^{12} \qquad \text{term2} = quot^3$$

$$\text{term2} = quot^6$$

- Eliminamos a utilização da função *pow()*, substituindo-a por multiplicações.
- Remoção da condição *if (j != i)* que garantia que uma partícula não interage consigo mesma. Removendo essa condição e começando os *loops* a partir de i+1, garantimos o mesmo, mas com melhor desempenho.

Na função *computeAccelerations()*:

- Novamente, eliminamos a utilização da função *pow()*, substituindo-a por multiplicações diretas.
- Os *loops* sobre k foram descontruidos, ou seja, cada elemento é calculado separadamente, sem um loop explícito, o que leva a um desempenho ligeiramente melhor.

Resultados do TP1:

TABLE I
TABLE WITH EXECUTION TIME FOR 3 ITERATIONS OF ORIGINAL VERSION OF MD

Iteration	Texe
1	236,95 sec
2	236.79 sec
3	236,92 sec

TABLE II
TABLE WITH EXECUTION TIME FOR 3 ITERATIONS OF OPTIMIZED VERSION OF MD

Iteration	Texe
1	6.82 sec
2	6.43 sec
3	6.65 sec

TABLE III
TABLE WITH METRICS FOR EACH VERSION OF MD

Version MD.cpp	#I	Average #CC	Average CPI	Average Texe
Original	1.243.580.424.482	780.836.157.356	0,6	236,8770 sec
Optimized	35.346.319.145	21.426.058.674	0,6	6,654 sec

Trabalho Prático 2

- O principal objetivo foi avaliar os benefícios da otimização do código com múltiplas *threads*, utilizando a ferramenta OpenMP para paralelizar a execução do código MD.
- O número estático de partículas, N, aumentou de 2160 para 5000.
- Foi feita uma análise de quais instruções potenciais do OpenMP poderíam ser usadas para melhorar a paralelização com base em experiências anteriores realizadas nas aulas práticas do curso.
- Testamos diferentes diretivas *pragma* nos vários *loops* contidos nas funções *Potential* e *computeAccelerations* analisando os resultados de desempenho com o *perf*.
- Alternamos o número de *threads* em execução para determinar qual o melhor valor de *speedup*.
- Os resultados obtidos foram analisados e comparados com os resultados da versão sequencial do TP1.

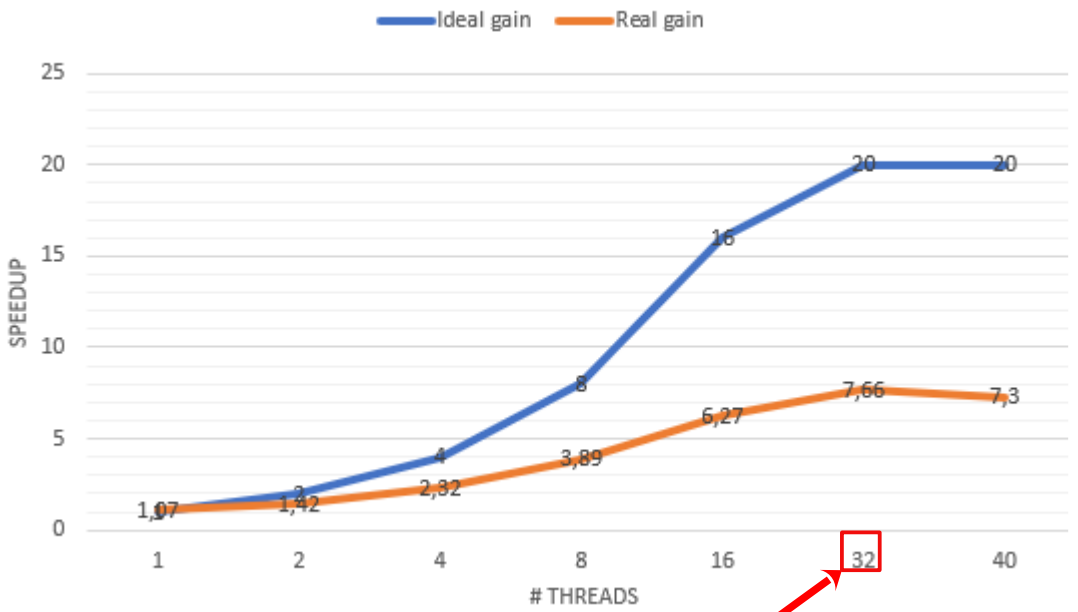
Na função *Potential()*:

- Diretivas OpenMP usadas aqui ...

Na função *computeAccelerations()*:

- Diretivas OpenMP usadas aqui ...

Resultados do TP2:



$$speedup = \frac{bestSeqTime}{parExecTime}$$

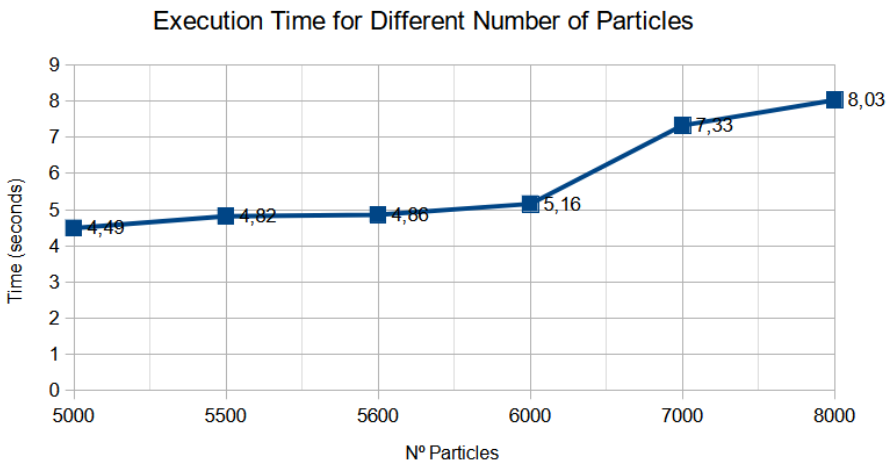
- Apesar de conseguirmos um *speedup* bastante bom com 32 *threads*, houve ocorrências de *data races* que não detetamos no momento da entrega do trabalho.
- Estas foram corrigidas na entrega final mas com valores de performance piores.

Trabalho Prático 3

- (...)

- (...)

Resultados do TP3:



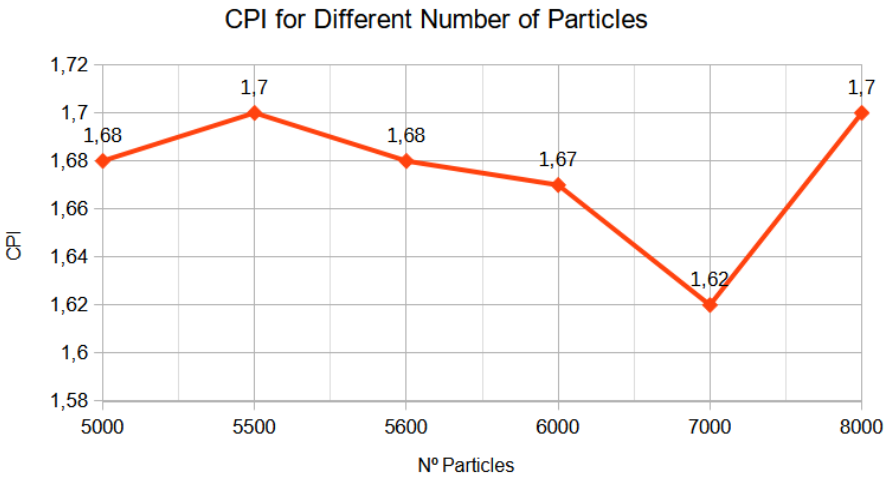
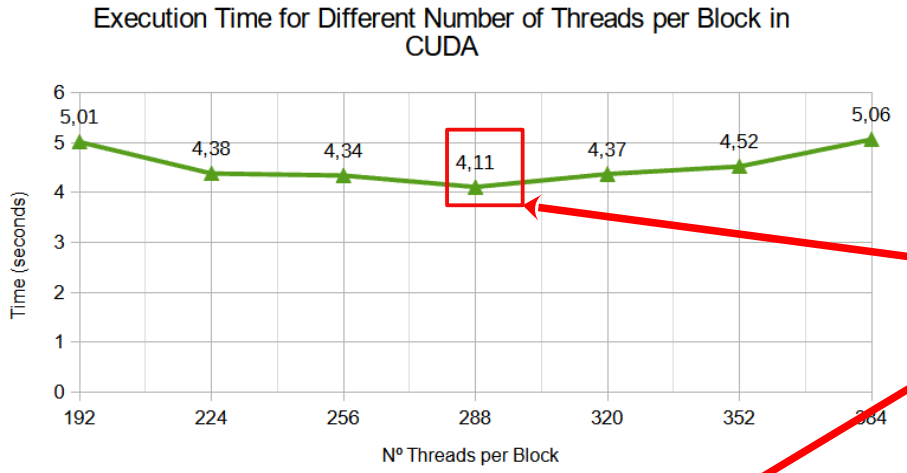


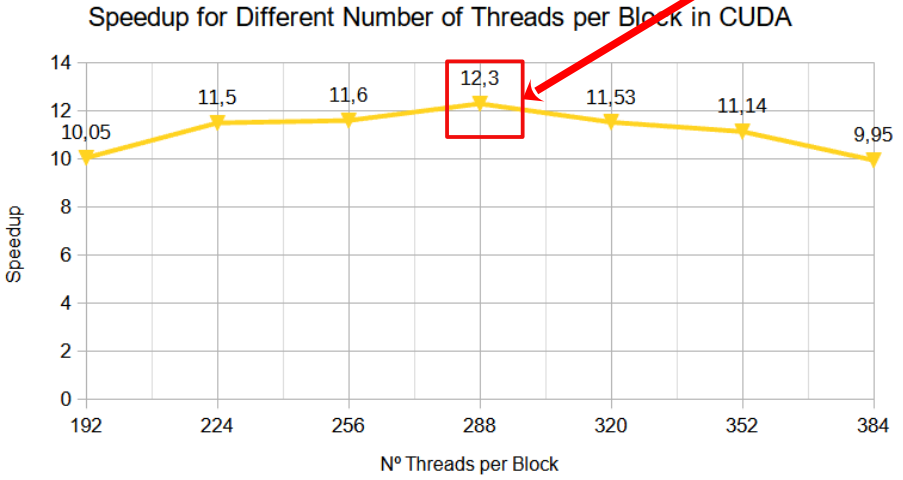
TABLE I

TABLE WITH EXECUTION TIME OF *MDpar_CUDA* IN MULTIPLE RUNS

#Run	Execution Time
1	0 m 4.11 s
2	0 m 4.16 s
3	0 m 4.18 s
4	0 m 4.06 s
5	0 m 4.17 s

Resultados do TP3:



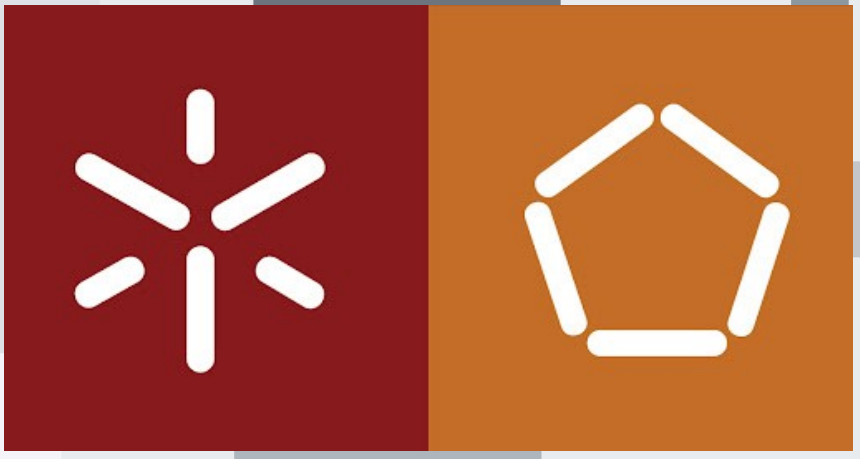


- Ao analisar os gráficos conseguimos perceber que o melhor tempo e *speedup* são obtidos se usarmos 288 *threads* por bloco dentro do *kernel* em CUDA.

$$speedup = \frac{bestSeqTime}{CUDAExecTime}$$

Conclusão

- O TP1 permitiu diminuir a complexidade dos cálculos e eliminar redundâncias nos mesmos, o que se traduziu numa boa performance para um número de partículas pequeno.
- O TP2 introduziu paralelização com OpenMP. Os resultados, inicialmente, continham *data races; foram* feitas melhorias à solução proposta, no entanto os valores de performance ficaram áquem das expectativas.
- No TP3 explorou-se a ferramenta CUDA que permitiu utilizar um processador gráfico para executar cálculos. Foi o mais satisfatório em termos de resultados, conseguindo manter a integridade dos valores de saida e ter uma tempo de execução e *speedup* muito bons para grandes números de partículas.



Computação Paralela

MEI - 2023/2024

Afonso Xavier Cardoso Marques - PG53601
Renato André Machado Gomes - PG54174