Computação Paralela 23/24
Fase 3

João Castro - pg53929
Vasco Rito - a98728

***Abstract*—Através de ferramentas de análise de *profiling* é possível fazermos uma análise da performance do código. A partir dos resultados obtidos procedemos, na fase 1, a várias otimizações a nível da lógica do código, a nível do compilador e a otimizações a nível de hierarquia de memória (*cache*). De forma a explorarmos um maior paralelismo recorremos ao OpenMP de forma a otimizarmos o tempo de execução através do lançamento de várias *threads* (paralelismo grão fino) e ao mesmo tempo estudar a escalabilidade da nossa solução tendo em conta o balanceamento de carga, entre outras métricas. Para esta fase final iremos estudar e implementar uma solução em memória distribuída recorrendo ao MPI onde iremos ter vários processos a correr de forma a explorarmos paralelismo de grão grosso.**

***Key Words*—Otimização ILP, Hierarquia de Memória, Organização de estruturas de dados, vectorização, Paralelismo, Escalabilidade, OpenMP, Memória Partilhada, Threads, Secções Críticas, Data Races, Profiling, MPI(Message Passing Interface), Processes, memória distribuída**

I. Introdução

No presente relatório iremos fazer um breve resumo das fases 1 e 2 do projeto de Computação Paralela e introduzir a fase 3 onde iremos debater as estratégias de implementação do nosso programa em memória distribuída recorrendo ao **MPI**, assim como apresentar um conjunto de testes para a análise da escalabilidade e estudo de certas métricas.

II. Fase 1 - Resumo

III. Estimativa da performance de execução

Procedeu-se a uma estimativa dos tempos de complexidade de cada um dos métodos juntamente com as respetivas estimativas do número de acessos à memória de cada.

Functions	Estimated Time Complexity	
	Overall Time	***Memory Accesses***
initialize()	N^3	$3 * N^3$
MeanSquaredVelocity()	N	$6N$
Kinetic()	$3N$	$6N$
initializeVelocities()	$15N$	$24N$
VelocityVerlet()	$9N + T(CA)$	$39N + MA(CA)$
ComputeAcelerations()	$3N + ((N-1) * N)/2$	$3 * N^3$
Potential()	N^2	$N * (N-1) * 12$

* CA-ComputeAcelerations* MA - memory accesses

IV. Otimizações

A. Lógica do código

O método **Potential** consiste em dois loops aninhados, calculando **r2** quando **i** é diferente de **j**. Devido à simetria da subtração elevada ao quadrado, pode-se reduzir o número de iterações no loop de **j** começando em **i+1**. Para manter consistência, duplica-se o cálculo do **Potencial** no final. Uma análise revelou similaridade na estrutura dos métodos **Potential** e **ComputeAccelerations**, resultando na criação de um método **potAccWork**. Notou-se a possibilidade de reduzir invocações de métodos como **Kinetic()** e **ComputeAccelerations()**. **Kinetic()** pode ser **MeanSquaredVelocity()**, resultando em **Kinetic() == MeanSquaredVelocity() * N * const2m**. Finalmente, otimizou-se a invocação do método **potAccWork**, obtendo o valor do **potencial** no **VelocityVerlet** para reduzir o número de ciclos de *clock*.

B. Simplificação do cálculo de fórmulas e do modelo do potential de Lennard-Jones

O código do **potencial** foi otimizado para evitar operações desnecessárias. As simplificações incluíram a eliminação do cálculo de raiz quadrada ($\sqrt{r2}$), a substituição de *pow* por multiplicações diretas e a redução de multiplicações dentro de loops para minimizar o *overhead* de operações. O resultado final para o potencial é Pot += $\frac{1}{\text{pow}(r2,3)} - \frac{1}{\text{pow}(r2,6)}$.

C. Hierarquia de memória

De forma a explorar a hierarquia de memória é necessário garantir que no que toca a operações entre matrizes a localidade espacial é obedecida (travessias *row-order*) e que os dados utilizados no loop caberão na *cache* até que sejam reutilizados pois temos que ter em conta que as linhas da *cache* vão sendo substituídas, o que implica *cache misses* e novas leituras da memória (se for *miss* em **L3**) e que podem ter algum custo. Portanto, a estratégia utilizada foi **Loop Block Optimization**. Assim, restruturámos o código do **potAccWork** do seguinte modo:

```
1    for (int j = 0; j < N; j += blockSize) {
2        for (int i = 0; i < fun; i += blockSize) {
3            for (int jb = j; jb < j + blockSize && jb < N
     ; jb++) {
4                for (int ib = i; ib < i + blockSize && ib
      < fun && ib < jb; ib++) {
```

O **blockSize** corresponde ao tamanho da cache **L1**, que no *cluster* possui **32K** (comando **lscpu**). Existem 4 *loops* aninhados sendo que o **jb** e o **ib** são necessários para fazer a travessia de submatrizes e explorar a localidade espacial da *cache*. Para além disso, adicionámos a condição $ib < jb$ de forma a diminuir para metade o nº de iterações, já que os valores das subtrações seriam o simétrico mas como são elevados ao quadrado o *output* é o mesmo como já referímos no capítulo de Otimizações-A.

D. A nível do compilador

Flags	Estimated Time Complexity	
	Elapsed Time (seconds)	***CPI***
-O0	14,27	0,41
-O2	3,44	0,35
-O3	3,32	0,47
-O2 -funroll-all-loops	3,18	0,44
-O3 -funroll-all-loops	3,24	0,46
-O2 -funroll-all-loops -ftree-vectorize -msse4	2,88	0,38
-O3 -funroll-all-loops -ftree-vectorize -msse4	3,21	0,46
-O2 -funroll-all-loops -ftree-vectorize -mavx	2,72	0,46
-O3 -funroll-all-loops -ftree-vectorize -mavx	3,14	0,54
-Ofast	3,23	0,46
-Ofast -funroll-all-loops	3,32	0,47
-Ofast -funroll-all-loops -ftree-vectorize -msse4	3,29	0,47
-Ofast -funroll-all-loops -ftree-vectorize -mavx	3,12	0,54

Como podemos observar na flag O3 temos tempos superiores a O2 furtuito de possíveis dependências de dados que se traduzem em CPI's superiores.

V. Fase 2 - Resumo

VI. Paralelização

O passo seguinte consistiu na identificação das secções críticas que poderiam, mais tarde, originar *data races* furtuito de acessos múltiplos a variáveis partilhadas por diferentes *threads*. Após essa análise, verificámos que o incremento da variável **Pot**, assim como as operações realizadas no *array* das acelerações **a** teriam de ser protegidas devido a possíveis escritas de *threads* ao mesmo tempo, ou escritas antes de leituras atualizadas, etc.

A opção pela diretiva *reduction* foi feita visando melhorar o desempenho. Isto ocorre porque garantimos que cada *thread* na região paralela irá ter sua própria cópia privada da variável. Em seguida, as operações são realizadas independentemente, e no final, os resultados parciais são combinados. Este método mostra-se mais eficiente em termos de *performance*. Desta forma, podemos deixar para o **OpenMP** a responsabilidade das agregações serem feitas de forma correta e consistente. Em versões mais recentes do **OpenMP** é possível utilizarmos a diretiva *reduction* sob valores que não são escalares, o que será uma mais valia, já que outras diretivas como o *critical*, ou *atomic* geram *overhead* de sincronização o que acaba por afetar o tempo de execução e a escalabilidade do nosso programa. Assim, em primeiro lugar foi criada uma região paralela de forma a criar um conjunto de *threads* que irão executar o nosso bloco de código: **#pragma omp parallel reduction(+:Pot) reduction(+:a[:N*3])**. Testámos a *reduction* quer numa versão de duas dimensões, quer numa versão de um array de uma dimensão de **a** e, verificou-se que na versão de um array de uma dimensão, o CPI era ligeiramente inferior devido a um possível menor *overhead* associado ao *indexing*, ou até pela localidade já que, nessa versão, ocorreram menos *cache misses* (1 dimensão: CPI-1.0 e cerca de menos 200k *cache misses* nível L1; 2 dimensões: CPI-1.1). De seguida, introduzimos a diretiva **#pragma omp for schedule(dynamic,1) collapse(2)**, uma vez que pretendemos paralelizar o *loop* e dividir da forma mais justa possível o *workload* das iterações do *loop* dinamicamente. Ao fazermos *dynamic 1*, garantimos que quando as *threads* que acabarem a sua execução são escalonadas para executar uma nova iteração. Se o número fosse superior a 1, a diferença de trabalho seria superior podendo aumentar o *IDLE time* de outras *threads*. Por fim, o *collapse(2)* permite combinar *nested loops* num único *loop* o que ajuda a melhorar a eficiência da paralelização.
A nossa estratégia de paralelização viza em atribuir para cada *thread* um bloco de posições de forma a balancear a carga e a minimizar o *overhead* de sincronização. Logo, para esse efeito decidiu-se aumentar um pouco o tamanho do bloco para não gerar *overhead* de paralelismo *fine-grained* o que leva a maiores tempos de execução devido à elevada sincronização e coordenação entre *threads*.

VII. Análise da Escalabilidade

Após termos executado o nosso programa com diferentes números de *threads* e de ter feito um *profiling* do código, acreditamos poder estar na presença de alguns problemas de escalabilidade. Eventualmente, este código paralelizado poderia estar a ser impactado por *Memory Wall* já que existe um aumento do número de *cache misses*, assim como um aumento do número do **CPI**. Isto poder-se-ia apreender pelo facto de existir um grande número de acessos à memoria derivados de operações com *arrays* e matrizes, ou de uma redundância de cálculos, ou até de falta de localidade espacial. No entanto, o facto de termos implementado hierarquia de memória, recorrendo a *loop block optimization*, não acreditamos que este seja um problema que tenha tido um maior impacto na escalabilidade.
Quanto à *task granularity* foram testados diferentes valores para o *blockSize* e ajustámo-lo de forma a que não fosse um valor pequeno, visto que lançar uma *thread* por cada um desses blocos iria conduzir em um *overhead* de sincronização e de mudanças de contexto, já que como cada thread possui informações de registos e de estado essenciais para a retoma da exeçução da nova thread isso iria apresentar um custo acrescido, pelo que aumentámos o valor para 96. Apesar de termos obtido um melhor desempenho não podemos assegurar que esse tamanho do bloco para a execução desta tarefa seja o ideal/óptimo. Verificámos que essa melhoria é mais notória até o cenário de paralelismo de até 16 *threads* com uma diferença de tempo de execução de aproximadamente 1 segundo a menos.

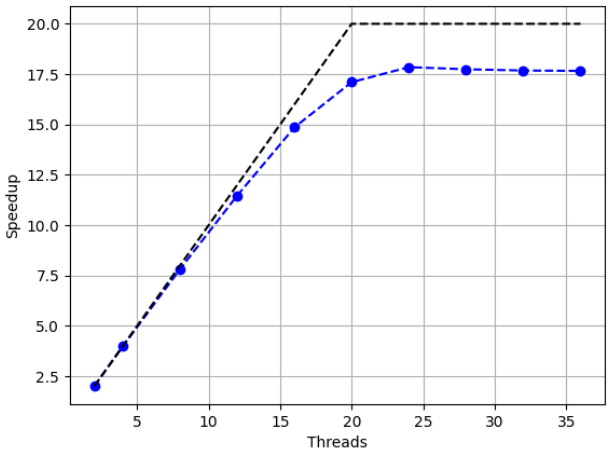

Fig. 1. Gráfico de análise de escalabilidade

No gráfico representado acima podemos averiguar que o *speedup* ideal é atingido sensívelmente entre as 2 e 12 *threads*, sendo que com 12 *threads* o desempenho já não é o ideal, uma vez que, o expectável é obtermos um *speedup* proporcinal ao número de PU's atribuídos. Como o *cluster* onde foram obtidos os *outputs* apresentam 1 *thread* por *core*, logo seria expectável obter um speedup=12. Como o presente sistema possui 20 PU's físicos o *speedup* nunca irá ultrapassar os 20. Por outro lado, podemos identificar que o nosso programa irá ter um maior *speedup* com 24 *threads*, ainda que seria necessário testar efetivamente com outro número de *threads* que não estão representados no gráfico. A partir das 24 *threads* há uma diminuição do *speedup*, ainda que os valores se mantenham constantes.

VIII. Problemas de Escalabilidade

A. Serial work (Amdahl's law)

Na função **PotAccWork**, a região paralela apresenta reduções da variável **Pot** e do *array* **a** o que introduz serialização. Segundo a lei de *Amdahl*, a existência de frações de código serializadas limitam a escalabilidade. Ora, dentro dos *loops* podemos averiguar que a maior parte do trabalho é realizado com operações efetuadas de leitura e escrita da memória (no *array* das acelerações **a**) o que poderá conduzir numa possível limitação do nosso código.

B. Memory wall

Existe um aumento do CPI com o aumento do número de threads, pelo que poderia ser indicativo de um problema de *Memory*

Wall, porém foi explorada a localidade espacial com a introdução de otimizações de *block loop*. Este aumento poderá advir da redundância de cálculos, na soma do quadrado das diferenças e Pot.

C. Parallelism/task granularity

Uma das razões que pode conduzir a este problema é a existência de tarefas pequenas que introduzem um maior *overhead*. Assim, como já foi referido optámos por aumentar o tamanho do **blockSize**, porém seria necessário testar outros valores pois podemos estar perante um paralelismo *fine-grained*.

D. Synchronisation overhead

Ao paralelizar o nosso código evitámos usar diretivas do tipo *atomic*, ou *critical*, pois estas têm uma maior carga de sincronização e posterior pior desempenho, pelo que optámos pela redução. O facto de *threads* acederem aos seus valores locais na mesma *cache line*, pode levar à invalidação da linha provocando maior número de *stalls* e consequentemente maior **CPI**. Para combater o *false sharing* teríamos de alinhar estas variáveis locais e averiguar se de facto este se tratava de um problema de escalabilidade no problema que temos em mãos.

E. Load imbalance

No que toca ao balanceamento de carga optámos por atribuir um bloco a cada *thread* e recorrendo a um escalonamento **dynamic,1**, de forma a reduzir ao máximo o *IDLE time* de outras *threads* que não estejam a efetuar trabalho. No entanto, para fazer um diagnóstico mais preciso seria necessário medir o tempo de computação de cada *thread*, de forma a verificar se existe um desfazamento grande de trabalho entre as várias *threads* ou não.

IX. Outras técnicas de análise de performance

Através da técnica de *profiling*, **perf record**, é possível fazer uma recolha de amostras para cada função ou sinais, com a respetiva estimativa de desempenho para cada uma delas. Como podemos verificar 1.08% de 1251 amostras são atribuídas à **gomp_team_barrier_wait_end**, que está relacionado com um *sinal wait* para a sincronização de *threads*, a barreira no fundo não irá permitir que as *threads* avancem sem que todas as outras tenham feito o seu trabalho. Ora 1% no contexto de desempenho não será muito significativo, porém na função **potAccWork** é onde a maior parte da execução de tempo do nosso programa é gasto.

```
# Total Lost Samples: 5989
#
# Samples: 116K of event 'cycles:uppp'
# Event count (approx.): 68770187738
#
# Overhead       Samples  Command    Shared Object     Symbol
# ........  ............  .........  ................  ........................................
#
    96.42%        111860  MDpar.exe  MDpar.exe         [.] potAccWork
     1.08%          1251  MDpar.exe  libgomp.so.1.0.0  [.] gomp_team_barrier_wait_end
     0.99%          1171  MDpar.exe  libgomp.so.1.0.0  [.] gomp_barrier_wait_end
     0.67%           780  MDpar.exe  libgomp.so.1.0.0  [.] gomp_mutex_lock_slow
     0.58%           767  MDpar.exe  [unknown]         [k] 0xffffffffab38c4ef
     0.15%           168  MDpar.exe  libgomp.so.1.0.0  [.] gomp_iter_dynamic_next
     0.06%            69  MDpar.exe  MDpar.exe         [.] VelocityVerlet
     0.01%            19  MDpar.exe  [unknown]         [k] 0xffffffffab396098
     0.01%             9  MDpar.exe  MDpar.exe         [.] MeanSquaredVelocity
     0.01%             7  MDpar.exe  MDpar.exe         [.] potAccWork
     0.01%             7  MDpar.exe  libgomp.so.1.0.0  [.] gomp_barrier_wait
```

Fig. 2. perf report

X. Fase 3

Através da implementação com **MPI** (*Message Passing Interface*) podemos ter algumas vantagens entre estas, conseguimos ter uma maior flexibilidade em correr o nosso programa em arquiteturas de memória distribuída, o que é o caso pois a arquitetura de *clusters* baseia-se numa topologia de conexões entre nodos. Desta forma podemos paralelizar o nosso programa através dessa rede de nodos que vão comunicar entre si, e transferir e computar dados entre si. Contudo, a adoção do **MPI** poderá trazer algumas limitações, principalmente, no que toca à performance, uma vez que a comunicação de rede entre os nós irá provocar algumas latências. Para além disso, teremos de fazer ajustes ao código prévio da fase 2.

No nosso programa de simulação de partículas os processos lançados vão correr simultaneamente, e de forma a aproveitarmos um melhor paralelismo a nivel de dados, cada processo irá ser responsável por executar porções de dados diferentes de modo a optimizar o tempo de execução. Apesar de cada processo ter as suas próprias variáveis locais, através das diretivas **MPI** será possível obter valores atualizados de outras variáveis de forma a executar códigos que tenham algum tipo de dependências ou não. A partir do *ID* de cada processo iremos controlar o fio de execução de cada um dos processos.

No que toca à implementação a nível de código, à semelhança do que é a estrutura típica de um programa em *MPI*, inicializámos a libraria com a diretiva *MPI_Init*, e como cada processo irá executar em concorrência cada um terá o seu próprio **rank** (*ID*), pelo que usámos a diretiva **MPI_Comm_rank** de forma a mais tarde controlarmos o código que queremos que seja computado de modo a que haja uma consistência no *output* e ao mesmo tempo um balanceamento de carga para cada um dos processos.

Em seguida, tivemos de ter em conta que na nossa **main** existe código de **stdout** para o terminal e para ficheiros, assim como também de leitura de ficheiros stdin. Ainda que fosse possível termos por exemplo vários processos a escreverem para um mesmo ficheiro teríamos de ter uma sincronização entre estes, pois a escrita concorrente poderia conduzir *data races* e dados inconsistentes. Como temos vários processos a executarem é importante assegurar que o *output* do nosso programa permaneça consistente pelo que o código correspondente a **printf's**, a leituras de *input* e a *pointers* de ficheiros deixámos encarregue ao processo de **rank=**0.

Para além disso, tivémos que ter em conta que existem excertos de código que são partilhados entre processos de forma a mais tarde obtermos o paralelismo desejado, pois existem dependências de dados. Assim, quando outros processos para além do 0 não tiverem alguma variável inicializada necessária para efetuar alguma computação, utilizámos a diretiva **MPI_Bcast** de forma a fazer *broadcast* desses dados do processo 0 para todos os outros. Desta maneira, evitámos potenciais *segmentation fault's* na execução do código.

Ainda no que toca ao código referente à main inserimos a diretiva **MPI_Barrier(MPI_COMM_WORLD)** antes da diretiva **MPI_Finalize()**, já que um dos processos pode acabar antes dos outros e fazer o *cleanup*, o que não é o comportamento que desejámos. Portanto, colocámos a barreira de forma a todos os processos esperarem depois de terem acabado o seu trabalho e de seguida fazemos o **MPI_FINALIZE()**.

De seguida, decidimos paralelizar a função **potAccWork**, uma vez que é a função que consume mais tempo de **CPU**. Portanto, optámos por explorar o paralelismo a nivel de dados, associando a cada um dos processos uma porção de dados a computar. Assim, cada um dos processos irá trabalhar com um intervalo de índices da matriz de posições **r**, de maneira a que cada processo tenha um valor local para a variável **Pot** e para o *array* de acelerações **a**.

No que diz respeito à carga de cada processo, a atribuição dos intervalos de índices foi feita da seguinte maneira:
1- Dividir o tamanho do problema **N** pelo número de processos que estão a correr (**MPI_Comm_size**).
2- Obter o resto da divisão em 1 através do cálculo do módulo.

3- Obter o índice inicial do processo (**rank * chunk**). Se o **rank** for menor que o resto, adiciona o **rank** ao índice inicial para dar a alguns processos um elemento extra.
4- Obter o índice final do processo (**start+chunk**) para garantir que cada processo tem um intervalo igual de indices. O (**rank** $<$ **resto** ? 1 : 0) é para garantir que os processos que receberam um elemento extra também considerem esse elemento no valor do índice **end**.

De modo, a que cada processo possa computar uma parte da matriz de posições **r**, em primeiro lugar, foi necessário fazer o *cast* da matriz para todos os processos através da diretiva **MPI_Bcast (MPI_Bcast(r, N * 3, MPI_DOUBLE, 0, MPI_COMM_WORLD))**. Após cada processo ter calculado o seu valor de **Pot** e ter feito as respetivas operações sobre o *array* de acelarações procedemos à redução do **Pot** e do *array* **a**. Através da utilização da diretiva **MPI_Allreduce** com os argumentos **MPI_IN_PLACE e MPI_SUM**, somámos todos os valores do **Pot** de todos os processos e colocou-se o valor final na variável **Pot**. O mesmo foi feito sobre o *array* de acelarações **a**.

```
// Sum the local Pot values from all MPI processes
MPI_Allreduce(MPI_IN_PLACE, &Pot, 1, MPI_DOUBLE,MPI_SUM,
    MPI_COMM_WORLD);

// Sum the local forces of 'a' from all MPI processes
MPI_Allreduce(MPI_IN_PLACE, a, N * 3, MPI_DOUBLE,MPI_SUM,
     MPI_COMM_WORLD);
```

Após uma análise a testes efetuados a tempos de execução numa implementação de apenas **MPI** vs numa implementação **MPI + OpenMP** podemos concluir o seguinte:

- Numa implementação integrada **MPI + OpenMP** conseguímos obter tempos de execução mais satisfatórios. Isto porque estámos a paralelizar o problema principal usando o **MPI** para decompor o trabalho do método **potAccWork**. Ao mesmo tempo estámos a explorar o paralelismo interno com o **OpenMP** para cada processo. Deste modo, temos uma aplicação com 2 níveis de paralelismo: uma que explora o paralelismo de grão grande para partilha de trabalho e passagem de informações entre processos; e outra que explora o paralelismo de grão fino com o uso de *threads*.
- Numa implementação com apenas **MPI** existe uma melhor escalabilidade em termos de avaliação da métrica *speedup*, ainda que não haja um ganho ideal com crescente aumento do número de processos lançados.

A *performance* do **MPI** obtida não foi melhor do que o cenário da implementação com apenas **OpenMP** e o tempo de execução foi maior, aproximadamente 0.5s superior, devido às possíveis seguintes razões:

- Overhead introduzido pelo **MPI**.
- Tempo gasto pelo processo mais lento (má distribuição de carga).
- Limitações de largura de banda nos processos.
- O acesso concorrente de múltiplos processos a níveis de *cache* superiores que são partilhados pelos processos poderá impactar na *performance* da aplicação.

Testámos o nosso código com 5000 partículas, variando o número de processos lançados. No primeiro gráfico iremos analisar como varia o tempo de execução numa implementação com apenas **MPI**, enquanto que no segundo gráfico iremos averiguar como variou o tempo de execução numa implementação **OpenMP + MPI**.

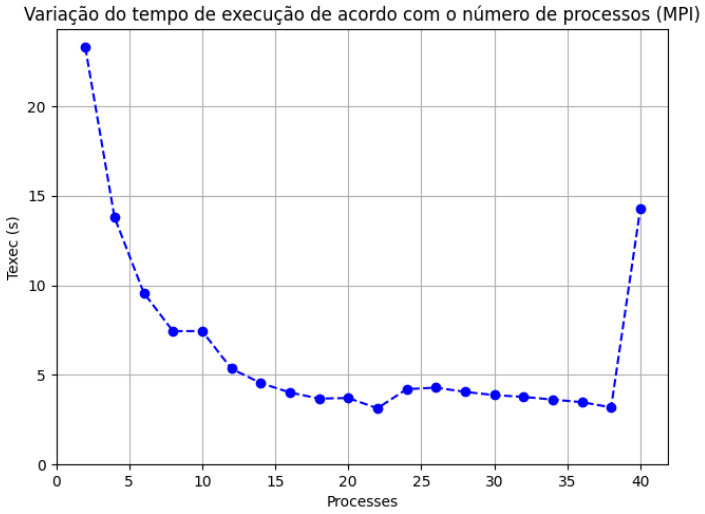

Fig. 3. Variação do tempo de execução de acordo com o número de processos (MPI)

Como podemos observar conforme vamos aumentando o número de processos lançados o tempo de execução tem tendência a diminuir ainda que não linearmente. Contudo, após lançar um número de processos superior a 38 ocorre uma queda da performance devido a um possível *overhead* de processos.
No segundo gráfico, podemos constatar tempos de execução relativamente baixos ainda que o comportamento da linha não seja o desejável já que conforme aumentámos o número de processos há um aumento grande no tempo de execução.

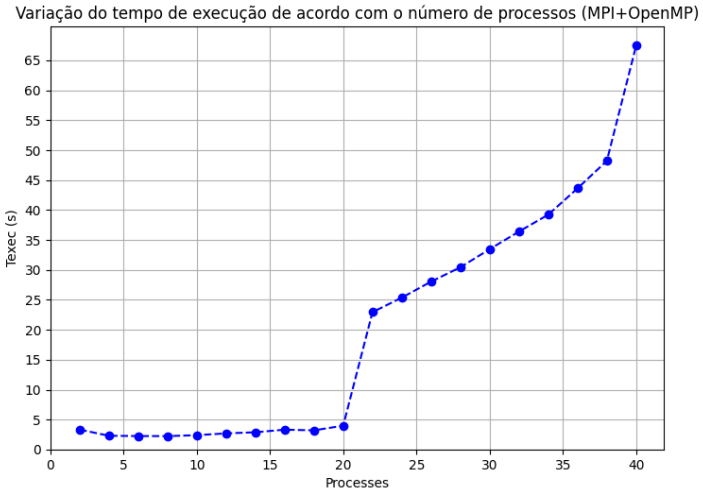

Fig. 4. Variação do tempo de execução de acordo com o número de processos (MPI+OpenMP)

Comparando o comportamento nos dois gráficos podemos concluir que numa implementação **OpenMP + MPI** conseguímos obter melhores tempos de execução (tempo mínimo de aproximadamente 2seg.) devido a paralelismo de grão fino, porém não escala com o aumento do número de processos. Já na implementação só com **MPI** obtivémos um tempo mínimo de execução superiores (tempo de execução mínimo de 3seg.), ainda que escale melhor do que a implementação anterior.
De seguida, iremos realizar alguns testes de modo a averiguar como o nosso programa de simulação se comporta se aumentarmos o número máximo de partículas e o tamanho do nosso problema **N**.

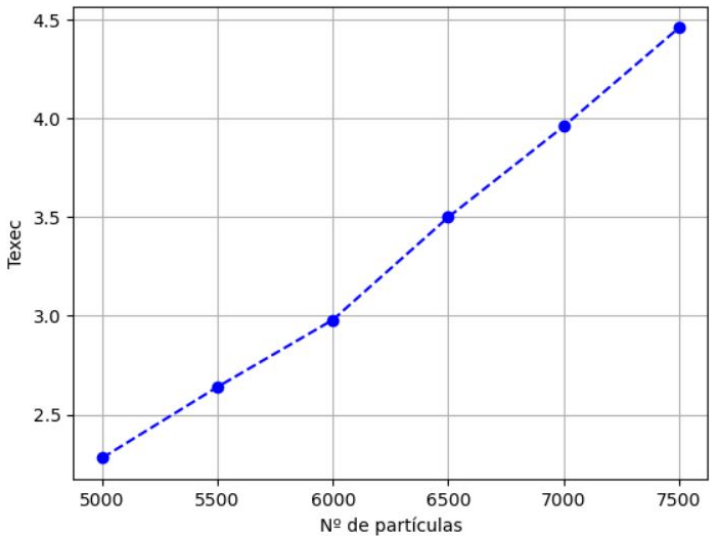

Fig. 5. Variação do tempo de execução com o aumento do número de partículas

Como podemos observar conforme aumentos o tamanho do problema **N** de 500 em 500, o tempo de execução aumenta quase linearmente, pelo que o algoritmo irá demorar mais tempo. Desta forma, podemos concluir que não conseguímos alavancar um maior paralelismo de grão fino com o aumento do **N**. Quando o **N** é pequeno isso introduz um maior *overhead* de sincronização já que o uso de diretivas do **OpenMP** introduz latências de tempo.

Adicionalmente, realizámos um teste para averiguar como variava o número de *cache misses* de acordo como o número de processos **MPI**.

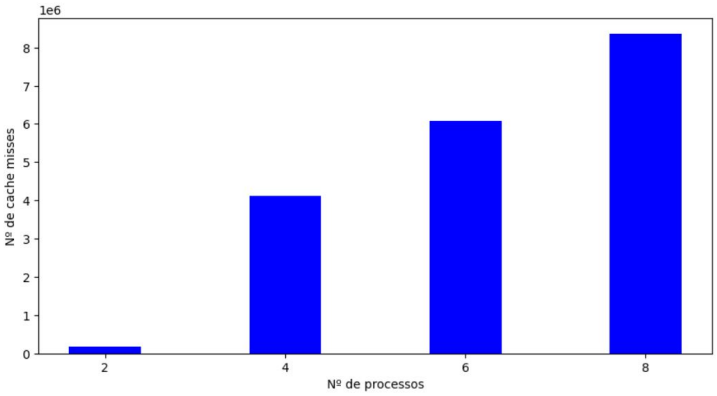

Fig. 6. Variação do número de cache misses por número de processos MPI

Este comportamento pode ser justificado devido ao crescente número de processos que são lançados e consequentemente existe uma maior troca de dados e acessos à memória o que provoca um aumento geral de *cache misses*. Apesar de termos implementado um mecanismo de balanceamento de dados equitativo para cada um dos processos lançados (atribuição de *chunks* a cada processo), esta estratégia pode não ser eficaz no que toca à minimização de *cache misses*.

XI. Conclusão

Em síntese, o desenvolvimento e otimização do código para simulação de partículas através de técnicas de computação paralela e distribuída revelou-se um desafio técnico significativo. Ao longo das fases 1 e 2, aplicamos melhorias substanciais na eficiência do código, otimizando operações críticas, explorando paralelismo com o **OpenMP** e ajustando a hierarquia de memória. A análise de desempenho, mediante a estimativa de complexidade e a seleção de *flags* de compilação, contribuiu para um código mais eficiente.

A implementação de paralelismo com o **OpenMP** permitiu uma distribuição da carga de trabalho entre as *threads*, resultando em ganhos de desempenho notáveis. Contudo, identificamos possíveis problemas de escalabilidade, destacando-se o impacto das seções críticas e da possível existência de problemas relacionados com *Memory Wall* e *Serializable Work*. O ajuste cuidadoso da granularidade das tarefas e a minimização do *overhead* de sincronização foram essenciais para otimizar o paralelismo.

Na fase 3, a introdução da biblioteca **MPI** para computação distribuída abriu novas perspetivas, proporcionando flexibilidade para executar o código em arquiteturas de memória distribuída. A paralelização da função **potAccWork** entre os diferentes processos **MPI** foi cuidadosamente realizada, atendendo à necessidade de balanceamento de carga e à minimização de conflitos de acesso a dados compartilhados.

Os resultados da análise de escalabilidade revelaram um comportamento satisfatório até um certo número de *threads*/processos, indicando um *speedup* proporcional ao aumento de recursos computacionais. No entanto, observamos uma diminuição do desempenho além desse ponto, sugerindo a necessidade de investigar possíveis causas, como serializações indesejadas, barreiras de comunicação ou desequilíbrio na distribuição de carga.