Departamento de Engenharia Eletrónica e Informática

Licenciatura em Engenharia Informática

Desempenho com Programação Paralela e Distribuída

Multiplicação de Matrizes

Aluno:
a74800 - João Sousa

Docentes:
Dra. Maria Margarida Madeira e Moura
Rodrigo Zuolo Carvalho

2024-2025

Conteúdo

1 Introdução

1.1 Motivação

Desde o surgimento dos microprocessadores na década de 1970, o desempenho computacional tem sido impulsionado principalmente pelo aumento da frequência do relógio, da largura dos barramentos e da capacidade da *cache*. Esta evolução sustentou-se durante décadas com base na Lei de Moore [1], que previa a duplicação do número de transístores num chip a cada dois anos. No entanto, nas últimas décadas, esta tendência começou a desacelerar tornando evidente a necessidade de novas abordagens para continuar a aumentar o desempenho dos sistemas.

Desta limitação emergiram os **sistemas paralelos** e **distribuídos** como estratégias fundamentais na computação de alto desempenho. Em sistemas paralelos, múltiplas unidades de processamento partilham memória e trabalham de forma cooperativa na resolução de tarefas, como ilustrado na Figura 1. Já em sistemas distribuídos, os processadores são independentes e comunicam entre si apenas por troca de mensagens, cada um com o seu próprio espaço de memória [2], como ilustrado na Figura 2.

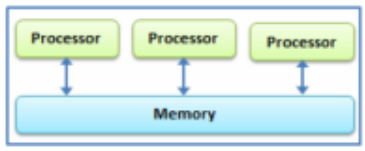

Figura 1: Representação de um sistema paralelo com memória partilhada.

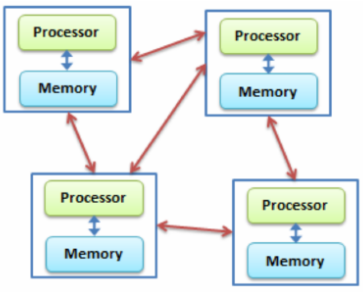

Figura 2: Representação de um sistema distribuído com memória independente.

Estas abordagens são hoje amplamente utilizadas em arquiteturas modernas, desde computadores pessoais com processadores *multicore* até supercomputadores e clusters[1]. Para explorar estas arquiteturas, utilizam-se ferramentas como *Open Multi-Processing* **OpenMP** (para paralelismo em memória partilhada) e *Message Passing Interface* **MPI** (para paralelismo em memória distribuída). A combinação de ambas, permite desenvolver aplicações paralelas híbridas que exploram tanto o paralelismo intra-nó (com *threads* via OpenMP)[2], como o paralelismo inter-nós com processos via MPI. Esta implementação é fundamental em ambientes de computação de alto desempenho, permitindo uma utilização eficiente de recursos a diferentes níveis da hierarquia de hardware [3, p. 60].

[1] Um *cluster* é um conjunto de computadores independentes que trabalham de forma coordenada como se fossem um único sistema, partilhando a carga de trabalho e interligados por rede.

[2] O paralelismo *intra-nó* ocorre dentro de um único nó (máquina física), onde os *threads* partilham memória. Já o paralelismo *inter-nó* distribui o trabalho por vários nós ligados em rede, exigindo comunicação entre processos.

1.2 Objetivos

O foco deste estudo é analisar os benefícios da computação paralela e distribuída na resolução do problema da multiplicação de matrizes, uma operação fundamental em diversas áreas da computação. Para tal, foram desenvolvidas e avaliadas quatro versões do algoritmo: uma versão sequencial (referência), uma versão paralela com recurso a **OpenMP**, uma versão distribuída com **MPI** e uma versão híbrida que combina ambas as abordagens.

O principal objetivo é comparar o desempenho destas abordagens em diferentes ambientes computacionais avaliando métricas, como o tempo de execução, a aceleração e a eficiência. Pretende-se com esta análise perceber o impacto da paralelização e identificar quais as abordagens mais adequadas em função da carga de trabalho e da infraestrutura utilizada.

Este relatório encontra-se estruturado em quatro capítulos principais. No **Capítulo 2** é apresentado o enquadramento teórico e técnico necessário para a compreensão do estudo, incluindo a descrição do caso em análise, as métricas de avaliação de desempenho, as arquiteturas utilizadas, os compiladores, os modelos de processamento paralelo e as ferramentas **OpenMP**, **MPI** e **SLURM**. O **Capítulo 3** descreve o desenvolvimento do trabalho, detalhando a metodologia adotada, os tamanhos das matrizes, a organização dos testes, o processamento dos resultados e a análise comparativa entre as abordagens implementadas. No **Capítulo 4** são apresentadas as conclusões retiradas a partir da análise dos dados e discutidas possíveis direções para trabalhos futuros.

2 Enquadramento

2.1 Apresentação do Caso

Este estudo incide sobre a implementação e análise de desempenho de um algoritmo de multiplicação de matrizes, uma operação fundamental em diversas áreas da computação, como a aprendizagem automática, a álgebra linear computacional, o processamento de imagem e a simulação científica. Foram desenvolvidas quatro versões distintas do algoritmo, todas em linguagem C: uma versão sequencial, uma versão paralela utilizando **OpenMP**, uma versão distribuída com **MPI** e uma versão híbrida com **MPI** + **OpenMP**.

As diferentes versões foram submetidas a testes em múltiplas arquiteturas computacionais, com o objetivo de comparar o desempenho entre abordagens paralelas e a versão sequencial. Os ambientes de execução utilizados foram dois servidores da Universidade do Algarve — `fct-deei-linux.ualg.pt` e `fct-deei-aval.ualg.pt` — e o supercomputador **Cirrus**, acessível através do serviço nacional de computação avançada.

2.2 Métricas de Avaliação de Desempenho

Para avaliar o desempenho das diferentes abordagens à multiplicação de matrizes, foram utilizadas as métricas clássicas da computação paralela [4]:

- **Tempo de execução** (T): corresponde ao tempo total necessário para completar a execução do algoritmo. Pode ser medido por ferramentas externas (*unix time*) ou através de funções específicas como `omp_get_wtime()` no **OpenMP**, `MPI_Wtime()` no **MPI**, ou `clock_gettime()` no **POSIX**.

- **Aceleração (*Speedup*,** S): razão entre o tempo de execução da versão sequencial (T_1) e o tempo da versão paralela (T_p) calculada como:

$$S = \frac{T_1}{T_p}$$

  Esta métrica indica o quanto mais rápido o programa se tornou com a introdução do paralelismo.

- **Eficiência** (E): mede a eficácia da utilização dos recursos paralelos obtida pela razão entre a aceleração e o número de unidades de processamento (P):

$$E = \frac{S}{P} \times 100\%$$

  Uma eficiência de 100% indica utilização ótima dos recursos disponíveis.

- **Tempo de comunicação**: representa o tempo gasto nas trocas de mensagens entre processos (no caso de **MPI**), podendo ser obtido por diferenciação entre o tempo total de execução e o tempo efetivamente dedicado ao cálculo. Este valor é relevante para avaliar o *overhead* da comunicação em ambientes distribuídos.

- **Escalabilidade**: avalia a capacidade de manter ou melhorar o desempenho à medida que se aumentam os recursos computacionais. Pode ser estudada em dois cenários:
  - ***Strong Scaling***: o tamanho do problema mantém-se fixo enquanto se aumenta o número de processadores.
  - ***Weak Scaling***: o tamanho do problema cresce proporcionalmente ao número de processadores, mantendo constante o trabalho por processador.

Estas métricas são essenciais para compreender não apenas os ganhos de tempo de execução, mas também a eficiência e viabilidade da paralelização aplicada.

2.3 Arquiteturas e Infraestruturas

Os testes de desempenho foram realizados em três ambientes distintos, com características arquitetónicas variadas, permitindo uma análise comparativa da portabilidade e eficiência das diferentes implementações.

- `fct-deei-aval.ualg.pt` — Este servidor académico possui um processador *Intel Xeon Gold 6138* com frequência base de 2.00 GHz. De acordo com a saída do comando `lscpu`, este ambiente disponibiliza 2 CPUs lógicos (núcleos), com 1 *thread* por core. A *cache* L3 tem uma capacidade de 55 MiB e o sistema suporta instruções de 64 bits. A virtualização está ativa, sendo o hipervisor identificado como Microsoft.

- `fct-deei-linux.ualg.pt` — Equipado também com um *Intel Xeon Gold 6138* a 2.00 GHz, este servidor disponibiliza 4 CPUs lógicos. A *cache* L1 e L2 apresentam 128 KiB e 4 MiB respetivamente, enquanto a *cache* L3 é significativamente superior à da máquina anterior, com 110 MiB. Tal como no servidor anterior, é um sistema x86_64 com virtualização ativa.

- **Cirrus (INCD)** — O supercomputador **Cirrus**, disponibilizado pela Infraestrutura Nacional de Computação Distribuída (INCD), é uma plataforma HPC com múltiplos nós de cálculo distribuído [5]. Cada nó de cálculo dispõe de dois processadores *Intel Xeon Gold 6238R* com 28 *cores* físicos cada (totalizando 56 cores por nó), e 192 GB de memória RAM. Este sistema utiliza o gestor de tarefas **SLURM**, que permite agendamento eficiente de jobs e alocação dinâmica de recursos.

As diferentes configurações permitem observar o impacto do número de núcleos, da arquitetura de *cache* e da distribuição de memória no desempenho das implementações. Adicionalmente, a utilização do **Cirrus** permite testar a escalabilidade em larga escala, em comparação com ambientes locais limitados.

2.4 Compiladores

Todas as versões foram compiladas com o *GNU Compiler Collection (GCC)*, versão **10.2.1** (`gcc (Debian 10.2.1-6) 20210110`), instalada em ambos os servidores utilizados para os testes.

Para a versão paralela com *OpenMP*, foi utilizada a flag `-fopenmp` de forma a ativar o suporte a diretivas de paralelização baseadas em *threads*. Já nas versões com **MPI** e híbrida, foi utilizado o compilador `mpicc`, fornecido pelas bibliotecas `Open MPI`, que funciona como um *wrapper*[3] do `gcc` , assegurando a ligação correta às funcionalidades da biblioteca MPI.

A utilização do mesmo compilador e ambiente de desenvolvimento em todas as abordagens garantiu uma base de comparação justa e controlada, sem interferências externas nos resultados de desempenho.

2.5 Processamento Paralelo

Processamento paralelo é uma técnica que consiste na decomposição de um problema em múltiplas subtarefas, que são executadas simultaneamente por vários núcleos ou processadores, com o objetivo de reduzir o tempo total de execução e melhorar a eficiência do sistema. Segundo Pacheco [3], esta abordagem permite explorar o potencial computacional de arquiteturas modernas, onde a computação sequencial já não é suficiente para lidar eficientemente com problemas de larga escala.

Existem dois modelos principais:

- **Memória partilhada**: vários *threads* acedem a um espaço de memória comum e executam tarefas concorrentes, requerendo mecanismos de sincronização;
- **Memória distribuída**: múltiplos processos independentes comunicam entre si através da troca de mensagens, sendo cada um responsável pela sua própria memória local.

A eficiência do processamento paralelo depende de diversos fatores, como a granularidade das tarefas, o balanceamento da carga de trabalho e o custo de comunicação e sincronização entre as unidades de execução.

[3]Um *wrapper* é um programa auxiliar que executa outro programa com os argumentos e bibliotecas adequadas. No caso do `mpicc`, trata-se de um empacotador que chama o `gcc` com suporte para MPI.

2.6 Linguagem C

A linguagem **C** é uma linguagem de programação de uso geral, desenvolvida originalmente para o sistema UNIX por Dennis Ritchie. A sua sintaxe concisa, eficiência e proximidade ao hardware tornaram-na uma linguagem fundamental para o desenvolvimento de sistemas operativos, compiladores e aplicações de baixo nível. A linguagem C continua a ser amplamente utilizada na atualidade, especialmente em sistemas embebidos, drivers de dispositivos e desenvolvimento de sistemas paralelos. Um exemplo da sua relevância é o núcleo do sistema operativo Linux, cujo código é maioritariamente escrito em C [**?** **?**].

2.7 *Threads*

Uma *thread* consiste numa unidade de execução leve dentro de um processo. Várias *threads* podem ser executadas de forma concorrente no mesmo processo, partilhando recursos como o espaço de memória, o que permite uma comunicação rápida entre elas. No entanto, essa partilha requer cuidados ao nível da sincronização para evitar condições de corrida e garantir a consistência dos dados [3].

O principal benefício do uso de *threads* é a possibilidade de paralelizar tarefas dentro do mesmo processo, aproveitando o potencial de sistemas multicore para melhorar o desempenho global das aplicações.

2.7.1 POSIX *Threads*

As **POSIX *Threads*** (ou *`pthreads`*[4]) são uma Application Programming Interface API padrão para programação multithread em sistemas compatíveis com POSIX, como UNIX e Linux. Permitem ao programador criar, sincronizar e controlar múltiplas *threads* dentro de um processo. Embora ofereçam grande flexibilidade, a sua utilização requer um conhecimento aprofundado de sincronização e gestão de concorrência, o que pode aumentar a complexidade do desenvolvimento [3].

2.8 OpenMP

OpenMP é uma API de alto nível para programação paralela em memória partilhada. Baseia-se no modelo de *threads* permitindo ao programador paralelizar regiões do código, como ciclos `for`, através de diretivas inseridas no código-fonte. Esta abordagem abstrai a criação e sincronização das threads, tornando o desenvolvimento mais simples e menos propenso a erros [3].

[4] `pthreads` é uma API padrão POSIX que permite o uso explícito de múltiplas *threads* em programas C/C++.

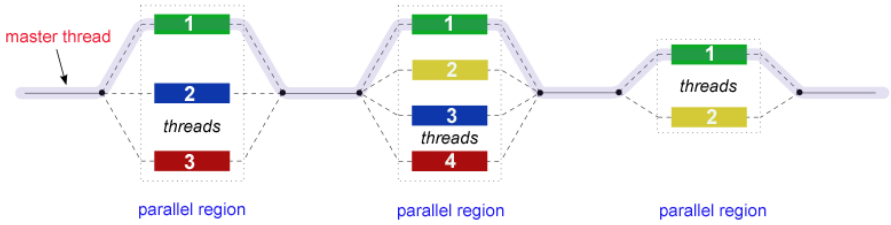

Figura 3: Modelo de execução paralelo com OpenMP (memória partilhada).

2.9 MPI

MPI é um padrão de comunicação utilizado em sistemas de memória distribuída. Cada processo possui o seu próprio espaço de memória e comunica com os outros através do envio e receção explícitos de mensagens. **MPI** suporta comunicação ponto-a-ponto (como `MPI_Send` e `MPI_Recv`) e operações coletivas (como `MPI_Reduce` e `MPI_Bcast`) [3].

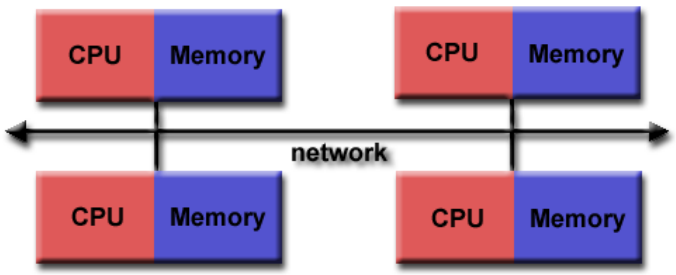

Figura 4: Modelo de comunicação entre processos com MPI (memória distribuída).

Nesta investigação foram utilizadas ambas as abordagens: **OpenMP** para paralelismo intra-nó (memória partilhada) e **MPI** para paralelismo inter-nó (memória distribuída), bem como uma versão híbrida que combina as duas, explorando o potencial completo de arquiteturas HPC.

2.10 SLURM

O **SLURM** (*Simple Linux Utility for Resource Management*) é um sistema de gestão de recursos e filas de trabalho amplamente utilizado em ambientes de (HPC). Trata-se de um *workload manager* de código aberto, desenvolvido para alocar recursos computacionais, agendar tarefas e monitorizar a execução de jobs em clusters com múltiplos nós [6].

No contexto deste estudo, o **SLURM** foi utilizado no supercomputador **Cirrus**, permitindo submeter tarefas de multiplicação de matrizes com diferentes configurações de nós e processos. A sua sintaxe baseada em `scripts bash` com diretivas `#SBATCH` possibilita uma configuração detalhada de cada tarefa submetida: número de CPUs, memória, tempo de execução máximo, entre outros parâmetros [7].

A submissão de um job ao **SLURM** é feita através de um ficheiro de submissão, onde são especificadas todas as opções necessárias para a execução da tarefa. Um exemplo simples inclui a seleção da partição, o número de nós, o número de tarefas por nó, e o comando de execução. No **Cirrus**, esta prática está documentada de forma acessível, permitindo aos utilizadores executar o seu primeiro job de forma orientada [8].

Este gestor de tarefas é fundamental para garantir a utilização eficiente dos recursos partilhados num sistema HPC, promovendo a escalabilidade dos testes e a reprodutibilidade dos resultados. Além disso, o **SLURM** fornece métricas úteis sobre o desempenho dos jobs, tais como o tempo real de execução, utilização de CPU e memória alocada.

3 Desenvolvimento do Trabalho

3.1 Metodologia

Com o objetivo de analisar o impacto das diferentes abordagens de paralelização na multiplicação de matrizes, foram desenvolvidas quatro versões do algoritmo, todas implementadas em linguagem C: uma versão sequencial, uma versão paralela com **OpenMP**, uma versão distribuída com **MPI** e uma versão híbrida que combina ambas as abordagens.

Para garantir a comparabilidade dos resultados e reduzir o impacto de variações momentâneas no desempenho do sistema, cada versão foi executada 31 vezes por conjunto de parâmetros, sendo a primeira execução descartada por forma a eliminar eventuais efeitos de aquecimento do sistema (como carregamento de bibliotecas ou *caching*[5] . As restantes 30 execuções foram utilizadas para o cálculo da média dos tempos de execução.

Os testes foram realizados em dois servidores da Universidade do Algarve, com ambientes controlados, utilizando os mesmos compiladores e configurações base. As experiências foram conduzidas de forma sistemática, variando o tamanho das matrizes e o número de unidades de execução (*threads* ou processos), de forma a observar o comportamento de cada abordagem em diferentes cenários de carga computacional.

Nos pontos seguintes descrevem-se as decisões metodológicas adotadas relativamente ao tamanho das matrizes, à organização e automação dos testes, à gestão da memória e ao processamento dos dados recolhidos.

3.1.1 Tamanhos n da Matriz

Para este estudo, foram definidos vários tamanhos quadrados de matriz $n \times n$, com o objetivo de avaliar o comportamento das diferentes abordagens em cenários com diferentes cargas computacionais. Os tamanhos escolhidos permitem observar tanto o desempenho em escalas reduzidas como a escalabilidade das soluções à medida que o volume de dados aumenta.

Os valores de n selecionados foram: **256**, **512**, **1024** e **2048**. Estes tamanhos cobrem uma gama representativa, desde matrizes pequenas (com baixo tempo de execução) até matrizes de grandes dimensões, que impõem uma carga computacional elevada, especialmente nas versões sequenciais.

A escolha destes tamanhos foi também influenciada pela capacidade de memória disponível nos servidores utilizados. Em todos os casos, procurou-se garantir que os testes fossem executados com fiabilidade e sem recorrer a mecanismos de swap, que poderiam enviesar os resultados.

[5] O *caching* refere-se ao armazenamento temporário de dados frequentemente acedidos, permitindo acesso mais rápido em execuções subsequentes. Pode afetar os resultados se não for controlado.

3.1.2 Estrutura e Automação dos Testes

A execução dos testes foi organizada de forma modular e automatizada, com o objetivo de garantir reprodutibilidade, clareza e consistência entre as várias abordagens avaliadas. O projeto está estruturado numa pasta principal denominada `project_matrix`, contendo quatro subdiretórios: `sequencial`, `openmp`, `mpi` e `hibrido`, correspondentes a cada uma das versões implementadas do algoritmo de multiplicação de matrizes.

Cada subdiretório segue a mesma estrutura e inclui:

- **Ficheiro** `Makefile` – utilizado para compilar o código-fonte de forma automática, com as opções apropriadas a cada abordagem.
- **Script** `testes.sh` – responsável pela execução dos testes com diferentes tamanhos de matrizes e variações no número de *threads* ou *processos*. Cada teste é repetido 31 vezes, sendo descartada a primeira execução. Os tempos das restantes 30 são guardados para cálculo da média e análise estatística.
- **Ficheiros de resultados** – gerados automaticamente após os testes:
  - `.csv`: contém os tempos médios por dimensão e configuração;
  - `.txt`: logs para controlo de erros.

A raiz do projeto contém ainda um *script* principal (`init.sh`), que centraliza a execução global: gera as matrizes com `gera_matrix.py`, distribui-as pelos diretórios e executa os testes de cada implementação de forma sequencial e automatizada.

Todos os ficheiros de saída seguem uma convenção de nomes que indica o tamanho da matriz e o número de unidades de execução, facilitando a organização e a análise posterior. Esta abordagem garante que todas as experiências são realizadas sob condições controladas e equivalentes, promovendo comparações justas entre as abordagens e assegurando a fiabilidade dos resultados.

3.1.3 Gestão da Memória

Devido à natureza dinâmica do problema em estudo — multiplicação de matrizes com dimensões variáveis —, a alocação de memória foi realizada de forma dinâmica em todas as implementações. A utilização de alocação estática (por exemplo, com arrays de tamanho fixo) não seria viável, uma vez que o tamanho das matrizes é definido em tempo de execução e pode atingir dimensões elevadas, tornando inviável a reserva antecipada de memória na stack.

Para garantir flexibilidade e portabilidade, todas as matrizes utilizadas (A, B e C) foram alocadas dinamicamente com recurso à função `malloc()` da biblioteca padrão do C, sendo posteriormente libertadas com `free()` após a conclusão dos cálculos. Esta abordagem assegura uma utilização eficiente dos recursos do sistema, especialmente importante em testes repetidos com matrizes de grande dimensão.

Adicionalmente, foi utilizada uma biblioteca auxiliar de deteção de fugas de memória, fornecida pelo docente Dr. João Dias no âmbito da unidade curricular de Laboratório de Programação [9]. Esta biblioteca (`leaks_checker.c` e `leaks_checker.h`)

permite verificar se toda a memória alocada foi corretamente libertada, assegurando que as implementações não apresentam fugas de memória ao longo das execuções.

Este cuidado adicional com a gestão de memória contribui para a fiabilidade e robustez das soluções desenvolvidas, garantindo um ambiente controlado e sem interferências nos resultados obtidos.

3.1.4 Processamento dos Resultados Obtidos

Após a recolha dos dados de execução em ficheiros `.csv`, o processamento dos resultados foi realizado com recurso a scripts desenvolvidos em **Python**, utilizando as bibliotecas `pandas`, `matplotlib` e `os` para análise e visualização dos dados.

Os *scripts*[6] foram concebidos para percorrer automaticamente os ficheiros de resultados de cada implementação e extrair métricas relevantes, como o tempo médio de execução, desvio padrão, tempo de computação, tempo de comunicação (no caso de MPI) e respetivas métricas de desempenho: ***speedup*** e **eficiência**.

Para cada combinação de tamanho de matriz e número de unidades de execução (threads ou processos), foram calculadas as médias a partir das 30 execuções válidas. No caso das implementações paralelas, o tempo sequencial correspondente foi utilizado como referência para o cálculo de:

- ***Speedup*** (S): $S = \frac{T_{seq}}{T_{par}}$
- **Eficiência** (E): $E = \frac{S}{p} \times 100\%$

Para o caso do **MPI**, foi ainda contabilizado o tempo de comunicação através da subtração entre o tempo total de execução e o tempo efetivo de computação, sendo esses dados utilizados para estudar o *overhead*[7] introduzido pela troca de mensagens entre processos.

Além da análise numérica, os *scripts* geraram automaticamente **gráficos comparativos** das métricas de interesse, permitindo visualizar a evolução do desempenho em função do número de unidades de execução e do tamanho das matrizes. Os gráficos produzidos incluem:

- Tempo médio de execução
- *Speedup*
- Eficiência
- Tempo de comunicação (para **MPI**)

Estes resultados foram consolidados em ficheiros `.csv` e imagens `.png`, organizados por abordagem e métrica, permitindo uma análise clara, reprodutível e extensível a novos cenários de teste.

[6] Neste contexto, *scripts* referem-se a pequenos programas automatizados (em Python ou Bash) criados para executar tarefas específicas como análise, compilação ou execução de testes.

[7] *Overhead* representa o tempo ou recurso adicional necessário para gerir uma tarefa computacional, neste caso, o custo associado à comunicação entre processos.

Com esta metodologia cuidadosamente estruturada — desde a definição dos parâmetros de teste, passando pela automatização da execução e validação da gestão de memória, até à análise estatística e visualização dos resultados — garantiu-se que os dados obtidos são fiáveis, comparáveis e representativos. A secção seguinte apresenta e discute os resultados obtidos com base neste processo, destacando os efeitos das diferentes abordagens de paralelização na multiplicação de matrizes.

3.2 Discussão dos Resultados

Nesta secção são apresentados e discutidos os resultados obtidos a partir da execução das quatro abordagens implementadas para multiplicação de matrizes: sequencial, paralela com **OpenMP**, distribuída com **MPI** e híbrida com **MPI** + **OpenMP**. Os testes foram realizados com diferentes tamanhos de matriz e variações no número de threads/processos, e os resultados analisados segundo as métricas de tempo de execução, aceleração (*speedup*), eficiência e, no caso de **MPI**, o tempo de comunicação.

A análise baseia-se em dados estatísticos extraídos a partir de 30 execuções por cenário, descartando a primeira para eliminar variações associadas à inicialização do ambiente. Os valores médios foram utilizados para a geração dos gráficos que suportam a discussão, permitindo observar tendências, limitações e potencial de escalabilidade de cada abordagem.

3.2.1 Tempo de Execução - Versão Sequencial

A análise do tempo médio de execução da versão sequencial revela uma tendência de crescimento exponencial à medida que o tamanho da matriz aumenta. Como se observa na Figura 5, os tempos crescem de forma acentuada entre as diferentes configurações, em especial a partir de $n = 1024$.

Este comportamento é esperado, uma vez que a multiplicação de matrizes apresenta uma complexidade computacional de ordem $O(n^3)$, o que significa que o tempo de execução tende a crescer rapidamente com o aumento da dimensão das matrizes. Para $n = 256$ e $n = 512$, os tempos de execução mantêm-se relativamente baixos, mas a partir de $n = 1024$ e sobretudo em $n = 2048$, verifica-se um aumento significativo do tempo necessário para completar a operação.

Esta versão serve como ponto de referência para comparação com as abordagens paralelas, permitindo avaliar o impacto da introdução de *threads* ou processos no desempenho global da aplicação.

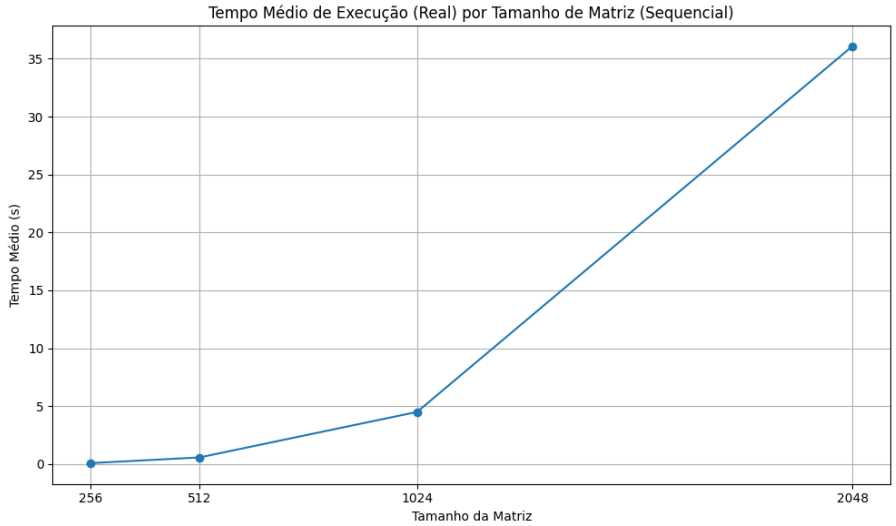

Figura 5: Tempo médio de execução por tamanho de matriz (versão sequencial).

3.2.2 Tempo de Execução - OpenMP

A Figura 6 apresenta o tempo médio de execução da versão paralela com **OpenMP** para diferentes tamanhos de matriz e números de *threads*. É possível observar uma redução significativa no tempo de execução à medida que o número de *threads* aumenta de 2 para 4, sobretudo para matrizes de maior dimensão ($n = 1024$ e $n = 2048$).

No entanto, esta melhoria não é linear para todos os casos. A partir de 4 *threads*, os ganhos de desempenho tendem a estabilizar, especialmente visível nas matrizes maiores, onde os tempos oscilam ligeiramente entre 6 e 10 *threads*. Este comportamento pode estar relacionado com o *overhead* associado à criação e sincronização das *threads*, bem como à limitação de recursos físicos disponíveis no sistema de teste.

Para matrizes pequenas ($n = 256$ e $n = 512$), os tempos de execução são baixos e os ganhos com o aumento de *threads* são menos expressivos, o que confirma que a paralelização em memória partilhada apresenta melhores resultados à medida que a carga computacional aumenta.

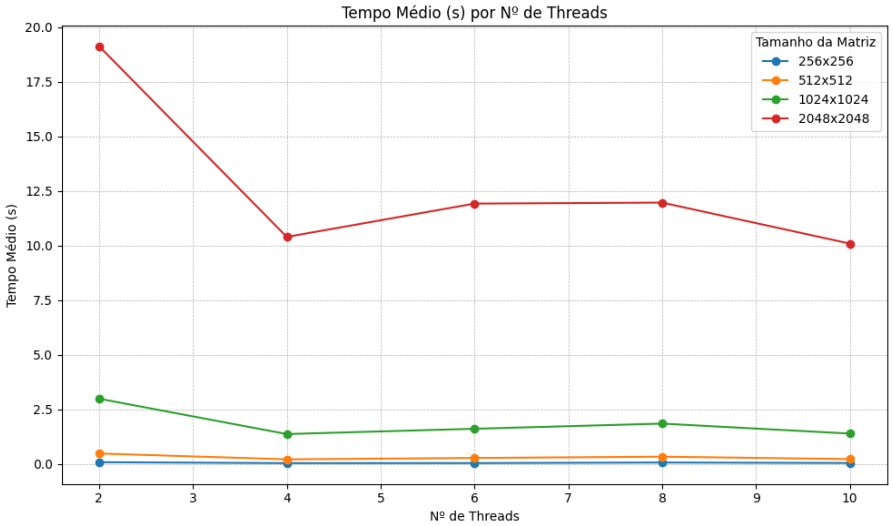

Figura 6: Tempo médio de execução com OpenMP para diferentes tamanhos de matriz.

3.2.3 Tempo de Execução - MPI

A Figura 7 apresenta o tempo médio total de execução para a abordagem distribuída com **MPI**, variando o número de processos entre 2 e 6. Ao contrário da tendência observada em OpenMP, o aumento do número de processos não traduz, de forma consistente, melhorias no tempo total de execução.

Para matrizes maiores ($n = 2048$), verifica-se uma ligeira melhoria até 4 processos, mas o tempo estabiliza ou até aumenta a partir desse ponto. Já para matrizes de dimensão mais reduzida, o aumento do número de processos não só não melhora o desempenho, como em alguns casos conduz a um tempo total superior. Este comportamento está associado ao *overhead* de comunicação entre processos distribuídos, que se torna mais penalizador quando a carga de trabalho individual é pequena.

Estes resultados sugerem que o uso de MPI é mais eficiente quando aplicado a problemas de maior escala, em que o custo de comunicação é diluído pelo volume de computação local.

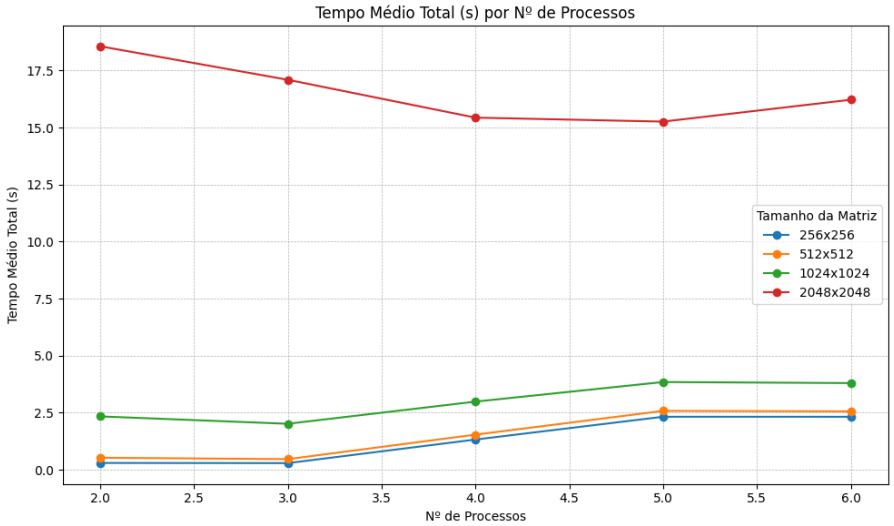

Figura 7: Tempo médio total de execução com MPI para diferentes tamanhos de matriz.

3.2.4 Tempo de Execução - Híbrido

A Figura 8 apresenta os tempos médios de execução da abordagem híbrida (**MPI + OpenMP**) na multiplicação de matrizes com dimensão 1024×1024, variando o número de processos (*np*) entre 2 e 6, e o número de *threads OpenMP* entre 2 e 8.

É possível observar que os tempos de execução tendem a diminuir com o aumento do número de *threads* e, de forma menos consistente, com o aumento do número de processos.As combinações mais eficientes encontram-se nas configurações com 3 *processos* e 4 *threads*, o que está de acordo com os limites físicos das máquinas utilizadas, que possuem entre 2 a 4 CPUs lógicos disponíveis.

Apesar da variação esperada, verifica-se que a adição de mais *threads* não garante, por si só, uma melhoria linear no desempenho. Para algumas combinações, o tempo mantém-se estável ou até aumenta ligeiramente, possivelmente devido ao *overhead* associado à gestão simultânea de múltiplas *threads* e processos, bem como à saturação dos recursos físicos da máquina.

Ainda assim, esta abordagem demonstra um bom potencial de redução do tempo de execução quando bem ajustada, tirando partido do paralelismo a dois níveis: inter-nó (**MPI**) e intra-nó (**OpenMP**).

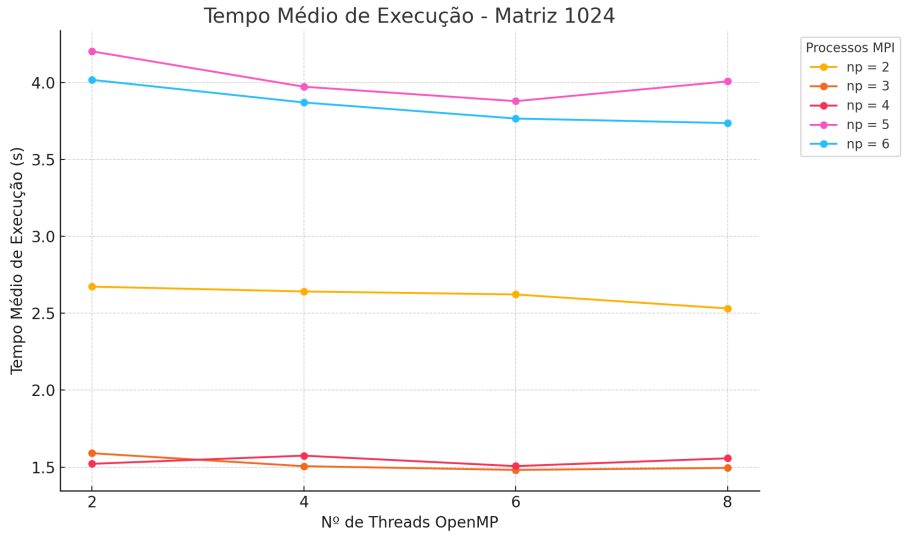

Figura 8: Tempo médio de execução da abordagem híbrida (MPI + OpenMP) com matriz 1024×1024.

3.2.5 Speedup - OpenMP

A Figura 9 mostra a evolução do *speedup* obtido com a versão paralela baseada em **OpenMP**, em função do número de *threads* utilizadas para diferentes tamanhos de matriz. Observa-se uma tendência de crescimento inicial do *speedup* com o aumento do número de *threads*, sendo mais pronunciada nas matrizes de maiores dimensões.

Para $n = 1024$ e $n = 2048$, o *speedup* alcança valores superiores a 3 com 10 *threads*, o que indica uma boa escalabilidade da abordagem paralela até esse ponto. Em particular, para $n = 2048$, o *speedup* atinge um valor próximo de 3.6, demonstrando que, para cargas computacionais mais elevadas, a paralelização permite ganhos de desempenho significativos.

Contudo, para matrizes de tamanho reduzido ($n = 256$ e $n = 512$), o ganho é limitado. O *speedup* estabiliza ou até decresce com o aumento de *threads*, refletindo o impacto do *overhead* de criação e sincronização de *threads* em cenários de menor carga.

Além disso, é visível que o *speedup* não cresce de forma linear com o número de *threads*, o que é esperado devido à existência de partes do código não paralelizáveis (lei de Amdahl), bem como às limitações físicas do hardware disponível.

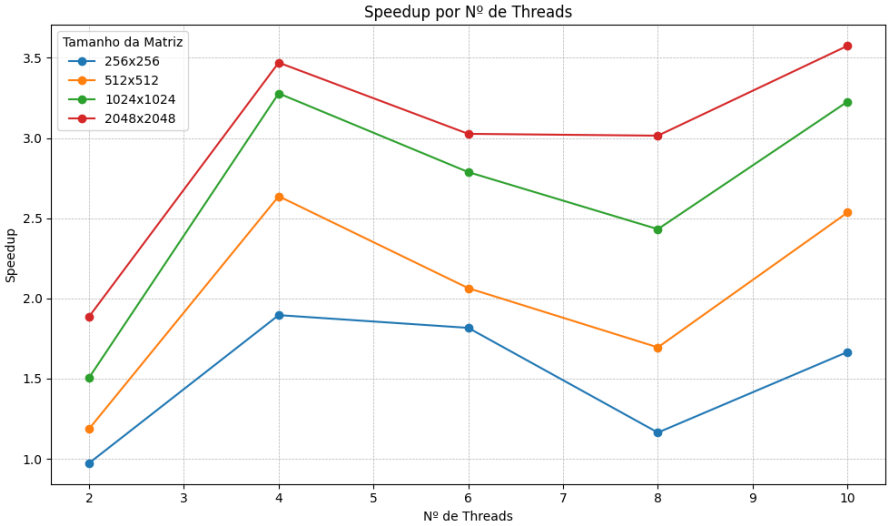

Figura 9: *Speedup* obtido com OpenMP para diferentes tamanhos de matriz.

3.2.6 Speedup - MPI

A Figura 10 apresenta os valores de *speedup* obtidos com a abordagem distribuída **MPI**, variando o número de processos entre 2 e 6, para diferentes tamanhos de matriz. Tal como esperado, os resultados mostram que o *speedup* não cresce de forma consistente com o aumento do número de processos, sobretudo para matrizes de menor dimensão.

Para $n = 2048$, observa-se um *speedup* estável entre 2 e 6 processos, com valores ligeiramente superiores a 2, indicando ganhos moderados com o paralelismo distribuído. Já para $n = 1024$, o *speedup* atinge um pico com 3 processos, mas degrada-se a partir desse ponto. Este efeito é ainda mais acentuado em matrizes pequenas ($n = 256$ e $n = 512$), onde o desempenho piora com mais processos.

Estes resultados devem-se, principalmente, ao custo de comunicação entre processos — o *overhead* introduzido pela troca de mensagens torna-se mais relevante quando a carga de trabalho atribuída a cada processo é reduzida. Assim, para tamanhos de problema pequenos, o custo da paralelização supera os benefícios da divisão do trabalho.

Este comportamento demonstra que o MPI tende a escalar melhor em problemas de maior dimensão e que o número ideal de processos depende diretamente da relação entre carga de computação e custo de comunicação.

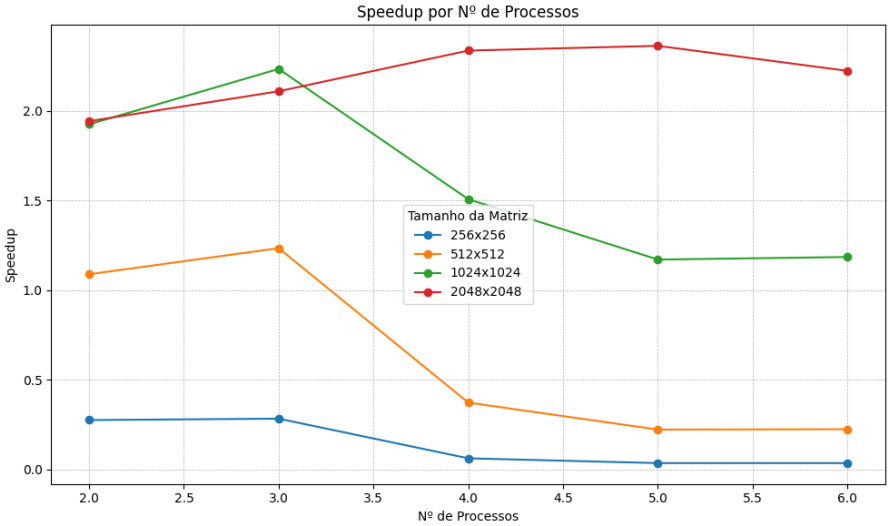

Figura 10: *Speedup* obtido com MPI para diferentes tamanhos de matriz.

3.2.7 Speedup - Híbrido

A Figura 11 mostra os valores de *speedup* obtidos com a abordagem **híbrida** (MPI + OpenMP) para a matriz 1024×1024, variando o número de *processos MPI* entre 2 e 6 e o número de *threads OpenMP* entre 2 e 8.

O *speedup* aumenta ligeiramente com o número de *threads*, mas os ganhos não são expressivos após certo ponto. As configurações com 2 e 3 *processos* revelam o melhor desempenho, com *speedup* superiores a 2.9 e próximos de 3.1, demonstrando uma utilização eficiente dos recursos até ao limite de saturação do sistema.

A partir dos 4 processos, o *speedup* estabiliza ou começa a decrescer, indicando que o paralelismo híbrido deixa de compensar devido ao aumento do *overhead* associado à gestão simultânea de múltiplos *processos* e *threads*, além dos custos de comunicação entre nós.

Estes resultados mostram que, embora o híbrido permita obter bons valores de *speedup*, os ganhos são sensíveis à configuração e não aumentam de forma linear com o número de unidades de execução. A escolha de uma boa combinação de processos e *threads* é fundamental para aproveitar o paralelismo híbrido sem introduzir penalizações de desempenho.

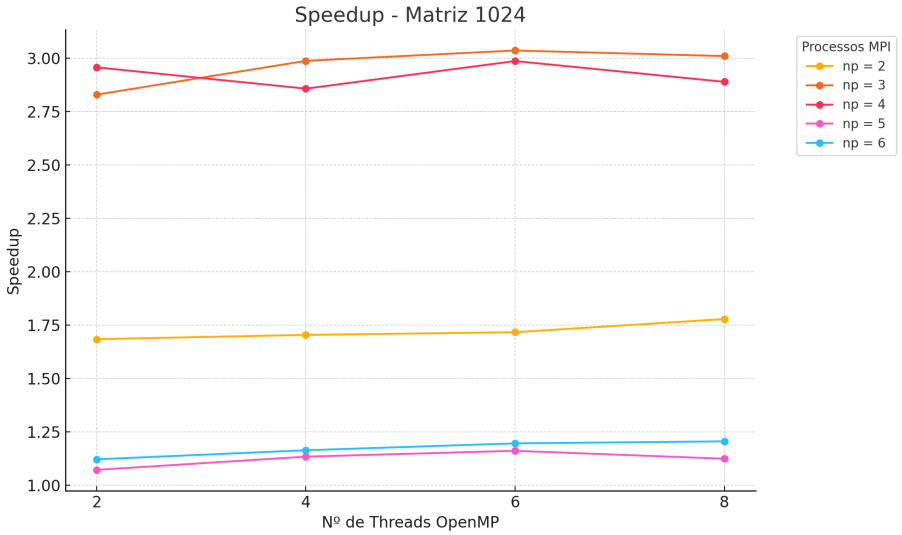

Figura 11: *Speedup* na abordagem híbrida (MPI + OpenMP) para matriz 1024 × 1024.

3.2.8 Eficiência - OpenMP

A Figura 12 apresenta os valores de eficiência obtidos na abordagem paralela com **OpenMP**, considerando diferentes tamanhos de matriz e números de *threads*. Tal como esperado, observa-se uma tendência decrescente da eficiência à medida que o número de *threads* aumenta.

Para matrizes maiores ($n = 1024$ e $n = 2048$), os valores de eficiência são mais elevados nos primeiros níveis de paralelismo (2 e 4 *threads*), chegando a ultrapassar os 80%. No entanto, à medida que o número de *threads* cresce, a eficiência decresce progressivamente, ficando abaixo dos 40% com 10 *threads*, mesmo para as matrizes mais exigentes.

Este comportamento é justificado pelo aumento do *overhead* de sincronização e pela menor relação entre trabalho computacional e custo de gestão de *threads*, sobretudo em matrizes de menor dimensão. Nestes casos, o impacto negativo do paralelismo é ainda mais evidente, com eficiências inferiores a 30% a partir de 6 *threads*.

Estes resultados confirmam que, apesar dos ganhos de desempenho com OpenMP, o paralelismo em memória partilhada apresenta limitações à medida que o número de *threads* aumenta, sendo mais eficaz quando aplicado a problemas com elevada carga computacional.

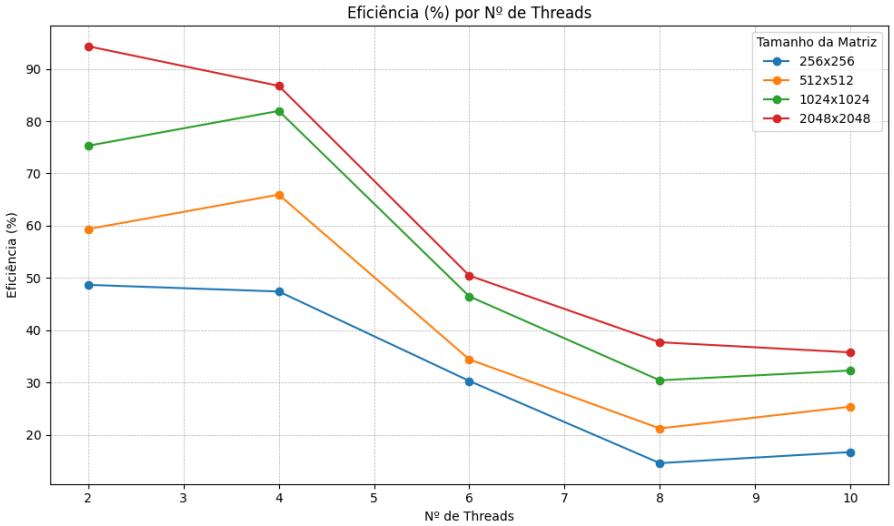

Figura 12: Eficiência da abordagem OpenMP em função do número de *threads*.

3.2.9 Eficiência - MPI

A Figura 13 apresenta a eficiência da abordagem **MPI** em função do número de processos utilizados, para diferentes tamanhos de matriz. Tal como no caso do **OpenMP**, observa-se uma tendência geral de diminuição da eficiência à medida que o número de processos aumenta.

Para matrizes grandes ($n = 2048$ e $n = 1024$), a eficiência inicial com 2 processos é muito elevada (acima dos 90%), mas decresce de forma acentuada a partir dos 4 processos. Esta queda deve-se ao aumento do *overhead* de comunicação entre processos, que se torna mais penalizador do que o benefício trazido pela divisão da carga de trabalho.

Nos casos de matrizes pequenas ($n = 256$ e $n = 512$), a eficiência é consistentemente baixa e degrada-se rapidamente, demonstrando que o uso de paralelismo distribuído é desaconselhado para este tipo de problema com pequena carga computacional por processo.

Estes resultados mostram que, ao contrário do paralelismo em memória partilhada, o paralelismo distribuído via MPI é fortemente afetado pela relação entre custo computacional e custo de comunicação. A eficiência apenas se mantém aceitável quando o volume de cálculo justifica o esforço adicional necessário para coordenar múltiplos processos.

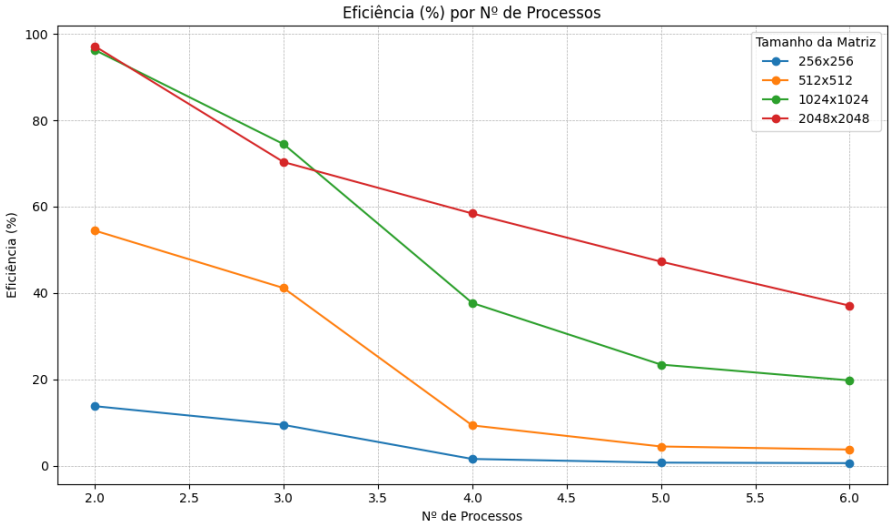

Figura 13: Eficiência da abordagem MPI em função do número de processos.

3.2.10 Eficiência - Híbrido

A Figura 14 apresenta os valores de eficiência paralela obtidos com a abordagem **híbrida** (MPI + OpenMP) para a matriz 1024×1024, considerando diferentes combinações de *processos* e *threads*.

Observa-se uma tendência clara de diminuição da eficiência com o aumento do número de *threads OpenMP*, independentemente do número de *processos MPI*. Esta degradação deve-se ao aumento do *overhead* de gestão de múltiplas *threads* em cada processo, e à saturação dos recursos físicos dos servidores, que como anteriormente referido, possuem 2 a 4 CPUs lógicos.

As melhores eficiências são alcançadas com poucas *threads* e até 3 ou 4 *processos*, atingindo valores próximos dos 40% a 47%. No entanto, com 6 ou 8 *threads*, a eficiência desce para valores inferiores a 15%, revelando que a escalabilidade da abordagem híbrida está fortemente limitada pela sobrecarga de comunicação e sincronização em cenários com elevado paralelismo.

Este comportamento reforça a importância de ajustar cuidadosamente a configuração do paralelismo híbrido em função dos recursos disponíveis, de modo a evitar penalizações no desempenho.

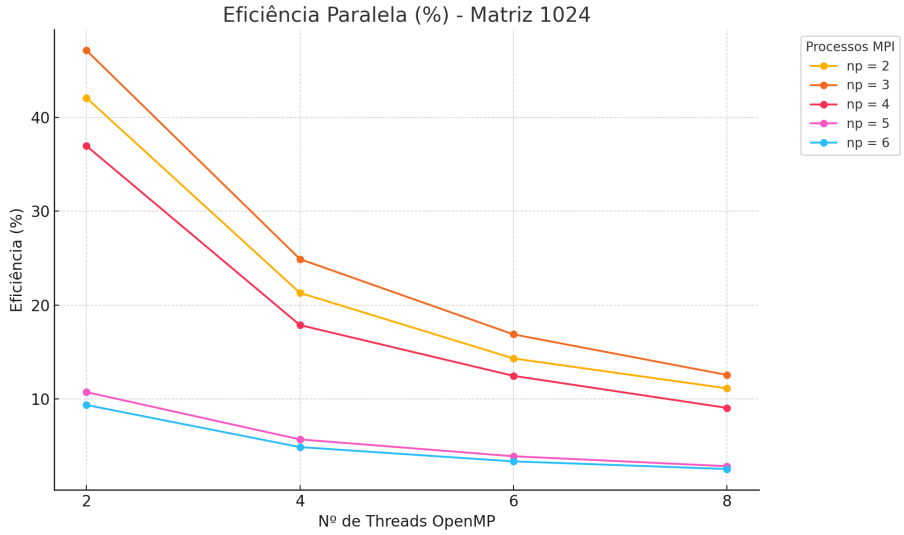

Figura 14: Eficiência paralela na abordagem híbrida (MPI + OpenMP) para matriz 1024×1024.

3.2.11 Tempo de Comunicação - MPI

A Figura 15 mostra o tempo médio de comunicação por número de processos na abordagem **MPI**, para diferentes tamanhos de matriz. Como era expectável, verifica-se que o tempo de comunicação aumenta à medida que cresce o número de processos, sendo este efeito mais pronunciado nas matrizes de maior dimensão.

Para $n = 2048$, o tempo de comunicação cresce consideravelmente até 5 processos, ultrapassando 1 segundo, e reduz ligeiramente nos 6 processos, possivelmente devido a variações momentâneas no agendamento ou balanceamento da carga. O mesmo padrão de crescimento gradual é observado para $n = 1024$, embora com valores absolutos inferiores.

Em contrapartida, para matrizes pequenas ($n = 256$ e $n = 512$), o tempo de comunicação é praticamente residual, mantendo-se abaixo de 0.1 segundos mesmo com 6 processos. Este comportamento deve-se ao menor volume de dados trocado, mas também mostra que o custo da comunicação se torna mais relevante à medida que aumenta o volume de informação e o número de processos envolvidos.

Estes resultados reforçam a ideia de que o uso de MPI deve ser cuidadosamente dimensionado em função da carga computacional disponível, já que o aumento de processos nem sempre se traduz em ganhos, devido ao impacto do *overhead* de comunicação.

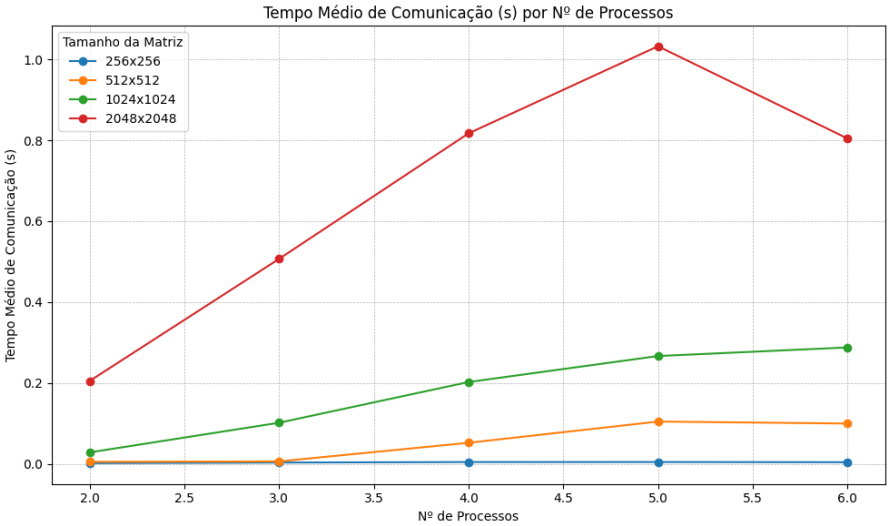

Figura 15: Tempo médio de comunicação na abordagem MPI.

3.2.12 Tempo de Comunicação - Híbrido

A Figura 16 apresenta o tempo médio de comunicação obtido na abordagem **híbrida** (MPI + OpenMP) para a matriz 1024×1024, considerando diferentes combinações de *processos* MPI e *threads* OpenMP.

Tal como nas versões puramente distribuídas, o tempo de comunicação tende a aumentar com o número de *processos*, o que se torna mais evidente a partir de 4 *processos MPI*. Com 6 processos, o tempo de comunicação ultrapassa 0.14 segundos em todas as configurações, enquanto com 2 ou 3 processos os valores permanecem bastante reduzidos, abaixo de 0.05 segundos.

É também possível verificar que, à medida que o número de *threads* aumenta, o impacto da comunicação tende a crescer ligeiramente, em especial nas configurações com mais *processos*, refletindo o maior volume de dados trocado e o esforço adicional de coordenação entre múltiplos processos paralelos que, por sua vez, utilizam múltiplas *threads*.

Este comportamento evidencia a complexidade do paralelismo híbrido, onde o aumento simultâneo de *processos* e *threads* impõe um custo crescente de sincronização e troca de mensagens, que pode anular os benefícios esperados em termos de desempenho.

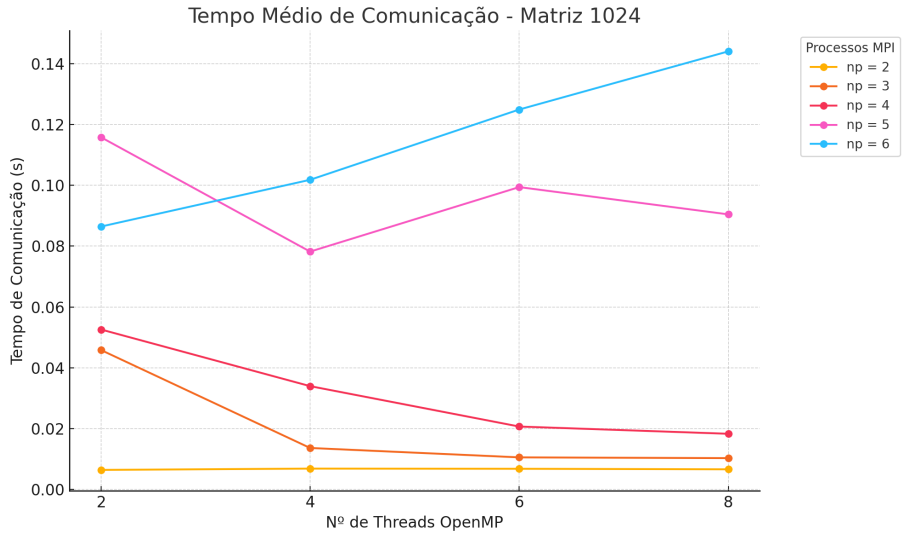

Figura 16: Tempo médio de comunicação na abordagem híbrida (MPI + OpenMP) para matriz 1024×1024.

3.2.13 Escalabilidade - OpenMP

A Figura 17 apresenta o comportamento da escalabilidade da abordagem **OpenMP** na multiplicação de matrizes com dimensão 2048×2048. Para avaliar o desempenho, foram realizados testes com 1 (sequencial), 2, 4, 6, 8 e 10 *threads*, e os respetivos valores de *speedup* foram comparados com o *speedup* ideal (linear), que representa a escalabilidade perfeita.

Os resultados demonstram que a escalabilidade é razoável até cerca de 4 *threads*, com um *speedup* próximo do ideal. A partir das 6 *threads*, observa-se uma saturação dos ganhos de desempenho, sendo o *speedup* praticamente constante, apesar do aumento do número de unidades de execução. Este comportamento reflete o impacto de fatores como o *overhead* de criação e sincronização de *threads*, e limitações impostas pela Lei de Amdahl.

A Tabela 1 resume os tempos médios de execução observados, o *speedup* obtido e a eficiência relativa em função do número de *threads*.

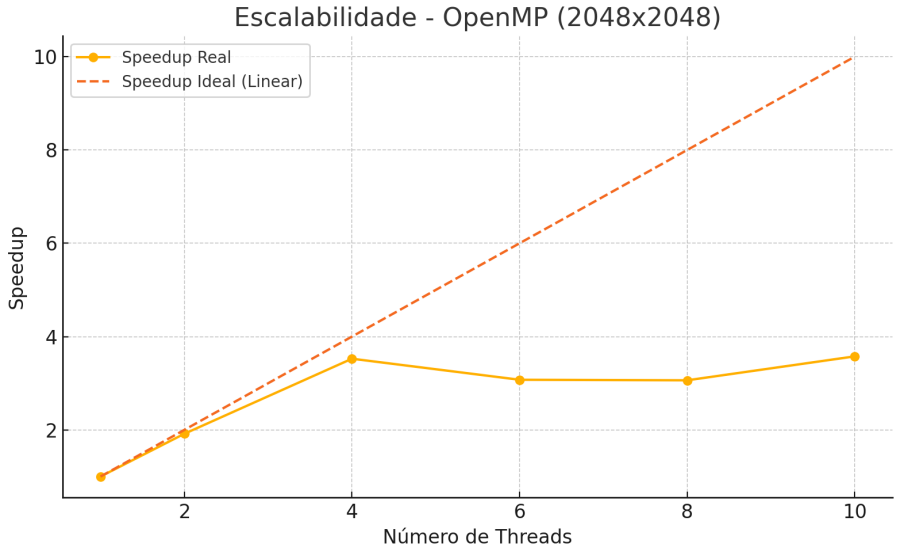

Figura 17: Speedup real vs. ideal na abordagem OpenMP (2048 × 2048).

Tabela 1: Resumo de escalabilidade com OpenMP para matriz 2048 × 2048.

Threads	**Tempo Médio (s)**	**Speedup**	**Eficiência (%)**
1	69.85	1.00	100.00
2	36.45	1.92	95.89
4	19.81	3.53	88.20
6	22.72	3.07	51.23
8	22.81	3.06	38.25
10	19.53	3.57	35.74

3.2.14 Escalabilidade - MPI

A Figura 18 apresenta o comportamento da escalabilidade da abordagem **MPI** na multiplicação de matrizes com dimensão 2048 × 2048. Para esta análise, foram realizados testes com 2 a 6 processos, e os respetivos valores de *speedup* foram comparados com o *speedup* ideal, que representa uma escalabilidade linear.

Os resultados mostram que o *speedup* cresce até aos 5 processos, mas a partir desse ponto verifica-se uma ligeira degradação. A eficiência apresenta uma tendência decrescente, iniciando com um valor elevado (97.13%) com 2 processos e descendo para 37.06% com 6 processos. Este comportamento confirma que, embora o paralelismo distribuído traga ganhos para problemas com grande volume de dados, o aumento do número de processos amplifica o *overhead* associado à comunicação, limitando a escalabilidade.

Comparando com a abordagem **OpenMP**, observa-se que o MPI mantém uma melhor eficiência inicial, mas é mais sensível ao número de processos envolvidos. A sua eficácia depende diretamente do equilíbrio entre o tempo de computação e o custo da comunicação entre nós.

A Tabela 2 resume os tempos médios de execução, o *speedup* obtido e a eficiência relativa em função do número de processos utilizados.

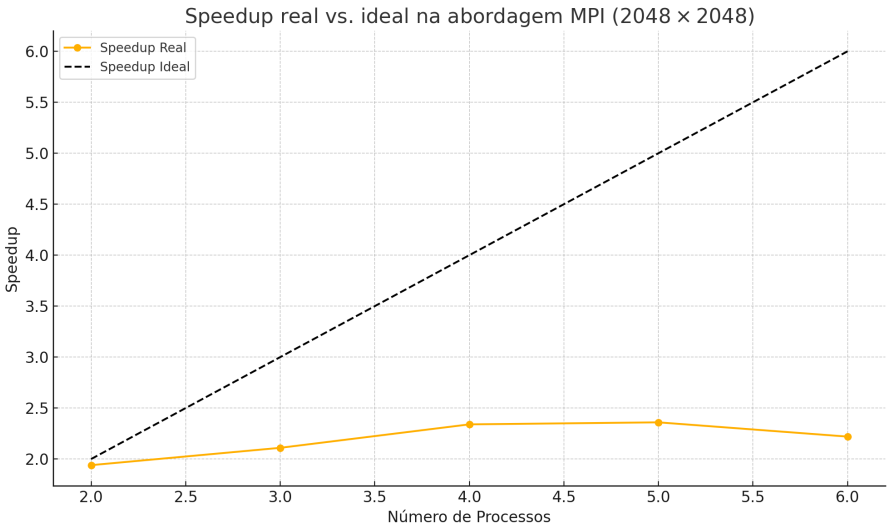

Figura 18: Speedup real vs. ideal na abordagem MPI (2048 × 2048).

Tabela 2: Resumo de escalabilidade com MPI para matriz 2048 × 2048.

Processos	**Tempo Médio (s)**	**Speedup**	**Eficiência (%)**
2	18.57	1.94	97.13
3	17.10	2.11	70.32
4	15.44	2.34	58.41
5	15.26	2.36	47.26
6	16.22	2.22	37.06

4 Conclusão

Este trabalho teve como objetivo analisar o impacto da computação paralela e distribuída na multiplicação de matrizes, comparando quatro abordagens distintas: uma implementação sequencial, uma paralela com **OpenMP**, uma distribuída com **MPI** e uma versão híbrida que combina ambas as técnicas. Todas as versões foram desenvolvidas em linguagem `C` e testadas em dois servidores da Universidade do Algarve, com recolha sistemática de dados para avaliação de desempenho.

Através de métricas como o tempo de execução, *speedup*, eficiência, tempo de comunicação e escalabilidade, foi possível identificar os pontos fortes e limitações de cada abordagem:

- A versão **sequencial** revelou um crescimento exponencial do tempo de execução com o aumento da dimensão da matriz, conforme esperado pela complexidade $O(n^3)$ da operação.
- A abordagem com **OpenMP** demonstrou boa escalabilidade até 4 *threads*, mas os ganhos reduziram-se progressivamente com mais unidades, devido ao *overhead* de sincronização e à limitação dos recursos físicos dos servidores.
- A implementação com **MPI** evidenciou vantagens em matrizes maiores e até 4 ou 5 processos, mas mostrou-se sensível ao aumento do número de processos em contextos com pouca carga por unidade, devido ao custo crescente da comunicação entre nós.
- A solução **híbrida** (MPI + OpenMP) revelou-se promissora, combinando o paralelismo inter-nó e intra-nó, alcançando *speedup* superiores a 3 em algumas configurações. No entanto, esta abordagem demonstrou ser altamente dependente da afinação entre número de processos e *threads*, e vulnerável à saturação dos recursos físicos disponíveis.

Embora tenha sido previsto o uso do supercomputador **Cirrus**, as limitações de acesso impediram a realização de testes na infraestrutura HPC, onde se antevê que a versão híbrida apresentaria resultados mais expressivos. A análise realizada baseou-se exclusivamente em ambientes locais com capacidade limitada, o que condicionou o grau de escalabilidade observável.

Durante o desenvolvimento deste trabalho, foi também analisado o artigo *"Matrix Multiplication — Optimizing the Code From 6 Hours to 1 Sec"* [10], que, embora não tenha servido de base metodológica, inspirou uma otimização concreta no código. Em particular, a alteração da ordem dos ciclos de multiplicação de `ijk` para `ikj` revelou-se eficaz na melhoria do desempenho, ao favorecer o acesso sequencial à memória — um fator crítico em arquiteturas modernas. Esta pequena alteração demonstrou, na prática, o impacto de escolhas aparentemente simples na eficiência de um algoritmo.

Conclui-se que a escolha da abordagem de paralelização mais adequada depende fortemente do problema, da dimensão das matrizes e das características da infraestrutura utilizada. O trabalho futuro poderá beneficiar de testes em arquiteturas mais robustas, da análise detalhada do consumo energético, bem como da introdução de algoritmos mais sofisticados como *Strassen* ou técnicas de blocagem (*tiling*), além do uso de ferramentas de *profiling* para identificar gargalos e otimizar o desempenho global.

Referências

[1] Nature. The chips are down for moore's law, 2016. URL https://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338.

[2] M. M. Madeira. Sistemas paralelos e distribuídos — slides da unidade curricular, 2023/2024. Material de apoio à unidade curricular, Universidade do Algarve.

[3] Peter S. Pacheco and Matthew Malensek. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2 edition, 2022. ISBN 9780128046050.

[4] M. M. Madeira. Sistemas paralelos e distribuídos — aula 6: Métricas de avaliação, 2024. Material de apoio da unidade curricular, Universidade do Algarve.

[5] INCD - Infraestrutura Nacional de Computação Distribuída. Compute node specs - cirrus (lisbon), 2024. URL https://wiki.incd.pt/books/compute-node-specs-and-information/page/incd-lisbon-(cirrusaincdpt). Acedido em 22 de março de 2025.

[6] Slurm workload manager, 2024. URL https://en.wikipedia.org/wiki/Slurm_Workload_Manager.

[7] INCD Infraestrutura Nacional de Computação Distribuída. Manage jobs - slurm, 2024. URL https://wiki.incd.pt/books/manage-jobs/page/slurm.

[8] INCD Infraestrutura Nacional de Computação Distribuída. My first slurm job, 2024. URL https://wiki.incd.pt/books/manage-jobs/page/my-first-slurm-job.

[9] João Dias. Biblioteca de deteção de fugas de memória em c, 2022/23. Disponibilizada no âmbito da unidade curricular de Laboratório de Programação, Universidade do Algarve.

[10] Vaibhaw Vipul. Matrix multiplication – optimizing the code from 6 hours to 1 sec, 2020. URL https://vaibhaw-vipul.medium.com/matrix-multiplication-optimizing-the-code-from-6-hours-to-1-sec-70889d33dcfa. Acedido em Março de 2025.