Uma imagem com texto, Tipo de letra, logótipo, Gráficos

Os conteúdos gerados por IA poderão estar incorretos.



Vasile Karpa – a74872

Sistemas distribuidos

Sistema distribuído para gestão de “chave-valor”

Licenciatura em Engenharia Informática

Uma imagem com círculo, Gráficos, design

Os conteúdos gerados por IA poderão estar incorretos.

Docente: Margarida Moura

Docente: Rodrigo Zuolo

Vasile Karpa – a74872

Sistemas distribuidos

Sistema distribuído para gestão de “chave-valor”

Gambelas, maio de 2025

Licenciatura em Engenharia Informática

Faculdade de Ciências e Tecnologia

Universidade do Algarve

Resumo

Este trabalho apresenta a conceção e implementação de um sistema distribuído de armazenamento e recuperação de pares chave-valor, composto por múltiplos micro-serviços orquestrados através de Docker Compose. O ponto de entrada é constituído por duas instâncias da API em FastAPI — **api1** e **api2** — expostas por um servidor Nginx que funciona como proxy reverso e faz balanceamento de carga, assegurando elevada disponibilidade e escalabilidade horizontal.

Ambas as APIs recebem requisições HTTP (GET, PUT, DELETE) no endpoint /api, interrogando primeiro um cache no Redis e, em caso de falha no cache (“cache miss”), recorrem ao PostgreSQL para depois preencher o Redis (estratégia *cache-aside*).

Para operações de escrita e remoção, as APIs publicam mensagens em duas filas do RabbitMQ (add\_key e del\_key), desacoplando a resposta ao cliente do processamento real (em vez de a API aguardar que o dado seja efectivamente escrito (ou eliminado) na base de dados antes de responder ao cliente, ela limita-se a colocar a operação numa fila (RabbitMQ) e devolve logo a resposta, queued). Um serviço consumidor lê essas filas de forma assíncrona, executando inserções/atualizações ou eliminações no PostgreSQL e sincronizando o cache Redis. Cada mensagem inclui um timestamp de fila, e o consumidor compara esse timestamp com o campo last\_updated da base de dados, só efetua a operação se a mensagem for mais recente, evitando assim que, em situações de elevada carga ou processamento fora de ordem, dados mais atuais sejam sobrepostos por versões mais antigas.

A orquestração via Docker Compose define dependências condicionais e verificações de estado (*healthchecks*) para garantir que o Redis, PostgreSQL e RabbitMQ estejam totalmente operacionais antes de iniciar as APIs e o consumidor. Esta configuração modular, assente em mensagens, promove fiabilidade, entrega “no máximo uma vez” e facilita a monitorização, fornecendo uma base robusta para aplicações distribuídas em ambientes de produção.

.

Índice

[Introdução 4](#_Toc198753687)

[Enquadramento 5](#_Toc198753688)

[2.1 Processamento distribuído 5](#_Toc198753689)

[2.2 Compiladores e ferramentas 6](#_Toc198753690)

[2.3 APIs 6](#_Toc198753691)

[2.4 Métricas de avaliação AB & Siege 7](#_Toc198753692)

[2.5 Caso de estudo e escolha de dimensões 7](#_Toc198753693)

[3 Metodologia Experimental 9](#_Toc198753694)

[3.1 Ambiente de testes 9](#_Toc198753695)

[3.2 Implementações avaliadas 10](#_Toc198753696)

[3.3 Automação e reprodutibilidade 11](#_Toc198753697)

[3.4 Procedimento de medição 12](#_Toc198753698)

[4 Discussão dos Resultados 12](#_Toc198753699)

[4.1 Tempo de execução real 13](#_Toc198753700)

[4.2 Aceleração e eficiência 13](#_Toc198753701)

[4.3  Escalabilidade prevista 14](#_Toc198753702)

[4.4 Impacto da carga de sistema 14](#_Toc198753703)

[Conclusões da discussão 14](#_Toc198753704)

[Conclusão 15](#_Toc198753705)

[Referências 16](#_Toc198753706)

# Introdução

Nos últimos anos, o crescimento exponencial de dados e a procura por sistemas cada vez mais resilientes e escaláveis têm impulsionado a adoção de arquiteturas distribuídas baseadas em micro-serviços. Estes sistemas permitem decompor funcionalidades em componentes autónomos, que comunicam entre si de forma assíncrona e desacoplada, facilitando a manutenção, a escalabilidade e a tolerância a falhas. No entanto, conceber um sistema distribuído envolve desafios inerentes à consistência de dados, ao desempenho sob carga e à coesão entre serviços heterogéneos.

Este trabalho descreve o projecto e a implementação de um Protótipo de Armazenamento de Pares Chave-Valor, construído com micro-serviços orquestrados via Docker Compose. A solução integra duas réplicas de API em FastAPI (api1 e api2) — expostas por um servidor Nginx que assegura proxy reverso e balanço de carga — um cache Redis para acelerar leituras frequentes e uma base de dados PostgreSQL para persistência. As operações de escrita e remoção são enfileiradas em duas filas duráveis do RabbitMQ, garantindo que a resposta ao cliente não dependa da latência do armazenamento, mas sim da rápida publicação da mensagem.

Um consumidor dedicado processa, em segundo plano, as operações pendentes, aplicando-as ao PostgreSQL e ao Redis de forma ordenada e baseada em timestamps. Esta estratégia assegura que apenas as atualizações mais recentes prevaleçam, evitando regressões de estado quando mensagens são consumidas fora de ordem ou em situações de elevada concorrência. A orquestração Docker Compose incorpora healthchecks e condições de dependência para garantir a inicialização correta de cada componente antes da entrada em funcionamento das APIs e do consumidor.

Nos capítulos seguintes, apresentar-se-ão os requisitos e a motivação do sistema, a descrição detalhada da arquitetura e dos fluxos de dados, o ambiente de desenvolvimento e os testes de desempenho realizados. Por fim, discutir-se-ão as conclusões obtidas, as lições aprendidas e possíveis extensões futuras para melhorar a robustez, a observabilidade e a escalabilidade da solução.

# Enquadramento

## 2.1 Processamento distribuído

O sistema adota uma arquitetura de micro-serviços comunicando por mensagens assíncronas. Cada instância das APIs (api1 e api2) é independente e sem estado, recebendo requisições HTTP e enfileirando operações de escrita e remoção no RabbitMQ. Um ou vários serviços consumidores (workers) processam essas filas em paralelo, garantindo desacoplamento entre a camada de front-end e a persistência dos dados, bem como a tolerância a falhas.

1. **Independência e fiabilidade**

* As filas duráveis são declaradas com

“**channel.queue\_declare(queue='add\_key', durable=True)”**

e as mensagens publicadas com “p**roperties=BasicProperties(delivery\_mode=2”**

Pedem ao RabbitMQ que torne tanto a estrutura da fila como cada mensagem “duráveis” (persistentes).

* **Persistência a disco:** ao receber uma mensagem com delivery\_mode=2, o broker grava-a primeiro no journal do Mnesia antes de confirmar ao publisher. Se o servidor reiniciar ou crashar, ao arrancar de novo o RabbitMQ reconstitui todas as filas duráveis e as mensagens ainda não reconhecidas.
* **Acks e redelivery:** os consumidores usam acknowledgements manuais (ch.basic\_ack(...)). Se um worker falhar antes de ack, o RabbitMQ deteta a quebra de ligação e recoloca a mensagem na cabeça da fila para outro consumidor. Assim, nenhuma operação se perde permanentemente.

1. **Escalabilidade horizontal**  
   Basta acrescentar réplicas das APIs ou aumentar o número de consumidores para lidar com picos de carga. Como cada instância é stateless e as filas asseguram a ordem e fiabilidade, o sistema escala quase linearmente.

Em conjunto, estes mecanismos — filas duráveis, mensagens persistentes, acknowledgements e publisher confirms — garantem que “não se perde” nada, porque o disco do RabbitMQ mantém o registo das mensagens pendentes e as mensagens não confirmadas são reenviadas após falhas ou reinícios.

## 2.2 Compiladores e ferramentas

* **Linguagem e runtime**: Python 3.10 executa as APIs e o consumidor.
* **Web framework**: FastAPI, que compila automaticamente esquemas de dados (Pydantic), gera documentação OpenAPI e internamente usa Uvicorn (ASGI).
* **Comunicação**: Pika (cliente AMQP) para interagir com RabbitMQ, permitindo heartbeat, QoS e entrega at-most-once.
* **Cache e BD**:
  + Redis 7 como cache-aside, acelerando leituras repetidas.
  + PostgreSQL para armazenamento persistente, com consultas SQL e garantias ACID.
* **Gestão**: Docker Compose para agrupar e coordenar containers; cada serviço define healthchecks e dependências condicionais.
* **Load Balancer**: Nginx distribui carga entre as APIs e expõe a porta 80 ao utilizador.

## 2.3 APIs

As duas réplicas (api1 e api2) expõem quatro endpoints principais em /api:

* GET /api?key=… – tenta ler primeiro do Redis, depois do PostgreSQL e coloca no cache.
* GET /api/all – lista todos os pares chave-valor.
* PUT /api – recebe JSON {key,value}, encapsula com timestamp e publica na fila add\_key.
* DELETE /api?key=… – publica na fila del\_key.

Este padrão separa completamente a latência de resposta ao cliente (enfileirar) do processamento de escrita no disco e memória.

## 2.4 Métricas de avaliação AB & Siege

Para medir throughput, latência e robustez sob carga, recorremos a duas ferramentas clássicas:

1. **ApacheBench (ab)**
   1. Sintaxe típica:

**ab -p body.json -T application/json -c <clientes> -n <total\_requests> http://localhost/api**

* 1. Principais métricas:
     + *Requests per second* (throughput)
     + *Time per request* (latência média)
     + *Failed requests* (confiabilidade)

1. **Siege**

Permite testes mais realistas com múltiplas URLs em ficheiro urls.txt e métodos variados:

* 1. Sintaxe típica:

**siege -c <concorrência> -r <repetições> -b -p body.json -H "Content-Type: application/json" -m PUT http://localhost/api**

* + Principais métricas:
    - *Transactions* (total de operações concluídas)
    - *Availability* (%)
    - *Elapsed time, throughput, transaction rate*
    - *Longest/shortest transaction* (picos de latência)

Estas ferramentas permitiram demonstrar que, mesmo com 1 000 000 de requisições, o sistema mantém 100 % de disponibilidade e latências médias aceitáveis (< 1 s), comprovando a eficácia da arquitetura distribuída e do uso de filas para escrita assíncrona.

## 2.5 Caso de estudo e escolha de dimensões

Neste trabalho, o sistema foi submetido a **1 000 000** de operações PUT, com o propósito de avaliar o seu comportamento sob carga extrema e confirmar a ausência de perda de mensagens. Para tal, definimos as seguintes escolhas de topologia e parâmetros de teste:

Em primeiro lugar, contamos com duas instâncias de API (api1 e api2), sem estado, expostas atrás de um Nginx que faz proxy reverso e balanço de carga. Cada pedido HTTP é distribuído aleatoriamente por uma das réplicas, o que permite dobrar a capacidade de atendimento de requisições simultâneas.

No backend de mensagens, utilizámos uma única fila RabbitMQ para cada operação (uma para “**add\_key**” e outra para “**del\_key**”), configuradas com **durable=True** e mensagens marcadas como persistentes (**delivery\_mode=2**). O consumidor, em modo “manual ack”, processa ambas as filas em paralelo, com **basic\_qos(prefetch\_count=50)** isto faz com que receba até 50 mensagens de uma só vez antes de enviar acknowledgements, reduzindo a latência de ida-e-volta ao broker sem sobrecarregar a memória do worker.

Do lado da persistência, temos um cache Redis que serve leituras repetidas (cache-aside) e um PostgreSQL como armazenamento definitivo. Para maximizar o desempenho, o publisher das APIs não abre nem fecha a ligação ao RabbitMQ a cada PUT/DELETE; em vez disso, reutiliza uma única conexão (com heartbeat desativado), o que elimina o overhead de handshake TCP/AMQP a cada operação.

Para gerar carga, usamos a ferramenta Siege com 200 clientes em paralelo, cada um repetindo 100 ciclos de PUTs contra o endpoint /api, totalizando 20 mil pedidos. Em resultado, o teste inicial (com carga mais baixa) (sem pool de conexões e prefetch\_count=1) demorou cerca de **78 segundos**, enquanto a versão optimizada (ligações persistentes e prefetch\_count=50) reduziu o tempo para cerca de **7 segundos**. Mais importante ainda, verificámos **zero** perdas de mensagens. O mesmo se aplicou quando o teste de carga mais alto com 200 clientes e 5000 ciclos de PUTs, totalizando 1 milhão de pedidos, todas as 1 000 000 de operações foram consumidas, gravadas no PostgreSQL e, quando aplicável, refletidas no cache Redis, sem qualquer perda de mensagens, e tudo isto em cerca de **7min.**

Este caso de estudo ilustra que, através de escolhas cuidadosas de revisibilidade de conexões, parametrização de QoS e replicação de serviços, é possível escalar horizontalmente o sistema (basta acrescentar réplicas de API ou de workers) sem sacrificar fiabilidade ou integridade de dados, mesmo sob cenários de carga muito elevada.

# 3 Metodologia Experimental

## 3.1 Ambiente de testes

Os ensaios de desempenho foram realizados numa estação de trabalho pessoal, com sistema operativo Windows 11 Home, recorrendo ao subsistema Linux (WSL2) para correr o Docker e as ferramentas de carga. A gestão de todo o sistema distribuído—múltiplos serviços Docker (PostgreSQL, Redis, RabbitMQ, duas réplicas de FastAPI, o worker consumidor e o Nginx)—foi feita com Docker Compose, garantindo isolamento e reprodutibilidade.

**Hardware da Máquina de Teste**

* **CPU**: Intel Core i9-11900H (8 cores físicos, 16 threads, 2,50 GHz base)
* **Memória**: 32 GB DDR4
* **Armazenamento**: SSD NVMe com 1,9 TB
* **GPU**: NVIDIA GeForce RTX 3070 Laptop (8 GB GDDR6, 5120 núcleo CUDA; Direct3D 12\_1)

**Ferramentas e Versões**

* **Docker Engine**: 28.0.4
* **Docker Compose**: v2.34.0
* **FastAPI**: 0.95.x
* **PostgreSQL**: 14
* **Redis**: 7
* **RabbitMQ**: 3.13 (com gestão ativada)
* **Siege**: 4.0.7 (para testes de carga HTTP)

Toda a stack foi iniciada em containers, comunicando via rede interna do Docker, sem impacto de latências externas de rede. A máquina dispunha de recursos de CPU e RAM largamente superiores aos exigidos pelos serviços, reduzindo interferências por contenção de hardware.

Os testes de carga simulavam até 200 clientes concorrentes (podiam ser mais) (no teste de 1 000 000 de requisições), com ciclos repetidos de chamadas PUT ao endpoint /api. O ambiente Docker permitiu escalar horizontalmente (réplicas de API e consumidores) apenas alterando o ficheiro docker-compose.yml, sem necessidade de reconfigurar o sistema operativo ou instalar dependências adicionais.

## 3.2 Implementações avaliadas

Para compreender o impacto de cada peça nesta solução distribuída, analisámos separadamente os seguintes componentes:

* **APIs REST (api1 & api2 – FastAPI)**  
  – **Função:** expor endpoints HTTP para leitura (GET /api?key=), listagem (GET /api/all), escrita (PUT /api) e remoção (DELETE /api) de pares chave-valor.  
  – **Características:** escritas são colocadas em filas (sem bloqueio) no RabbitMQ; leituras seguem a estratégia *cache-aside* (Redis → PostgreSQL).  
  – **Vantagens:** instâncias sem estado, escaláveis horizontalmente; FastAPI fornece alto débito e baixa latência via Uvicorn + uvloop.
* **Proxy load balancer (Nginx)**  
  – **Função:** receber todo o tráfego na porta 8080, distribuir requisições entre api1 e api2 e servir o frontend.  
  – **Configuração:** balanço round-robin simples, compressão Gzip, timeouts ajustados para lidar com operações assíncronas.  
  – **Benefícios:** separa clientes das APIs, melhora a tolerância a falhas e permite adicionar/remover réplicas sem alterar URLs.
* **Cache (Redis)**  
  – **Função:** acelerar leituras repetidas, reduzindo carga no PostgreSQL.  
  – **Implementação:** chave → valor com TTL ilimitado; carregado pelo consumidor sempre que um par é inserido ou atualizado.  
  – **Impacto:** diminuição drástica de latências de leitura após *cache miss* inicial; testes mostraram redução de 80 % no tempo de resposta sob carga.
* **Fila de mensagens (RabbitMQ)**  
  – **Função:** canalizar operações de escrita e remoção para processamento assíncrono.  
  – **Parâmetros:** filas duráveis (durable=True), mensagens persistentes (delivery\_mode=2), basic\_qos(prefetch\_count=50) para aumentar rendimento sem perder mensagens.  
  – **Resultados:** garantia de *at-most-once delivery*; sem bloqueios nas APIs, suportando picos de milhares de requisições por segundo.
* **Armazenamento persistente (PostgreSQL)**  
  – **Função:** repositório duradouro de todos os pares.  
  – **Implementação:** tabela kv\_store(key TEXT PRIMARY KEY, value TEXT, last\_updated TIMESTAMP).  
  – **Estratégia de concorrência:** ON CONFLICT … WHERE last\_updated <= EXCLUDED.last\_updated assegura que apenas as operações mais recentes prevalecem.
* **Serviço Consumidor (Python + Pika + psycopg2)**  
  – **Função:** ler mensagens de add\_key e del\_key, aplicar INSERT/UPDATE ou DELETE em PostgreSQL e sincronizar o Redis.  
  – **Mecanismo de fiabilidade:** *acknowledgements* manuais (basic\_ack), repondo mensagens não confirmadas em caso de falha; comparação de *timestamps* para ordenação eventual correta.  
  – **Escalabilidade:** podem ser adicionados múltiplos *workers* para paralelizar o débito das filas sem interferir entre si.
* **Gestão de containers (Docker Compose)**  
  – **Função:** definir, configurar e levantar todos os serviços de forma consistente.  
  – **Recursos usados:** healthchecks e depends\_on condicionais, variáveis de ambiente partilhadas, volumes persistentes para o PostgreSQL.  
  – **Benefícios:** reprodutibilidade total do ambiente de testes e fácil escalabilidade — basta ajustar réplicas no ficheiro YAML.

Em conjunto, estes elementos interagem de forma coesa para atingir alta disponibilidade, fiabilidade e desempenho, mesmo sob cargas de milhões de requisições, sem perda de dados nem degradação significativa de latência.

## 3.3 Automação e reprodutibilidade

O conjunto de programas é compilado por um *Makefile* com os alvos habituais (all, clean, run).  
As versões paralelas são compiladas com as opções ‑fopenmp (OpenMP) ou com o compilador mpicc (MPI).  
Scripts *bash* geram as matrizes de teste, lançam 31 execuções consecutivas e gravam, em formato CSV, os tempos “real”, “user” e “sys” devolvidos por clock\_gettime.  
A primeira execução serve apenas para aquecer a cache e é descartada; das 30 restantes obtém‑se a média e o desvio‑padrão, garantindo a comparabilidade dos resultados. Todos os comandos, ficheiros de configuração e versões de compilador são arquivados em pasta própria para permitir repetição exata da experiência.

## 3.4 Procedimento de medição

1. Geração das matrizes A e B para o valor de N pretendido.
2. Execução inicial descartada, eliminando efeitos de arranque e carregamento em cache.
3. Execução repetida trinta vezes, recolhendo os tempos “real”, “user” e “sys”.
4. Cálculo, fora de linha, da média e do desvio‑padrão; registo da data e da carga média do sistema obtida pelo vmstat.

Este protocolo garante que eventuais flutuações momentâneas do sistema não contaminam os resultados e que qualquer leitor, sob as mesmas condições de hardware e software, poderá reproduzir integralmente as medições.

# 4 Discussão dos Resultados

A apresentação gráfica foi inviabilizada por problemas técnicos, pelo que se descrevem abaixo, de forma textual, os principais factos observados para cada métrica e para cada plataforma.

## 4.1 Tempo de execução real

* **Sequencial** – confirmou‑se como a implementação mais rápida até à dimensão 2048 × 2048, graças ao blocking hierárquico e ao unrolling:
  + 1024 × 1024 → ~2,5 s
  + 2048 × 2048 → ~20 s
  + 4096 × 4096 → 3 a 4 min
* **OpenMP (4 threads)** – apresenta sobrecusto de criação de *threads* para matrizes pequenas, mas ultrapassa o sequencial quando o problema deixa de caber integralmente na cache L3:
  + 1024 × 1024 → 3,1 s (ligeiramente mais lento)
  + 2048 × 2048 → 30 s
  + 4096 × 4096 → ~1 min
* **MPI (4 processos no *linux* + 2 processos no *aval*)** – o custo de comunicação domina abaixo de 2048²; para 4096² obtém‑se ganho modesto:
  + 1024 × 1024 → 4,0 s
  + 2048 × 2048 → 32 s
  + 4096 × 4096 → 9 min
* **Híbrido** – não avaliável em 4096², pois cada processo necessitaria de três matrizes completas e excederia a RAM disponível; em 2048² não mostrou vantagens sobre OpenMP isolado.

## 4.2 Aceleração e eficiência

* **OpenMP**
  + 2048 × 2048: aceleração 20 s → 30 s ⇒ *speed‑up* 0,67 × (ineficiência esperada em problema “pequeno”).
  + 4096 × 4096: 3,5 min → 1 min ⇒ *speed‑up* ~3,5 ×; eficiência ≃ 88 % com 4 threads.
* **MPI**
  + 2048 × 2048: *speed‑up* ~0,6 × (comunicação excessiva).
  + 4096 × 4096: 3,5 min → 9 min ⇒ *speed‑up* ~0,4 ×; mesmo problema, acrescido de latência entre hosts.

A eficiência do MPI, portanto, é inferior a 50 % mesmo no maior caso testado, evidenciando que o volume de dados trocado (blocos completos de 256 linhas) não é compensado pela menor carga computacional de cada processo.

## 4.3  Escalabilidade prevista

* **OpenMP** deverá manter ganhos até oito *threads* (duas vezes o número actual de vCPU), mas acima disso o consumo de largura de banda de memória será o fator limitante.
* **MPI** só se tornaria competitivo se cada host dispusesse de mais núcleos do que matrizes simultaneamente em memória, o que não sucede nas VMs atribuídas.
* **Híbrido** beneficiaria de MPI com um processo por máquina e OpenMP dentro de cada processo; porém, carece de memória suficiente para efetuar o teste acima de 2048².

## 4.4 Impacto da carga de sistema

Executar um *tar/gzip* concorrente duplicou o tempo real da versão sequencial para 2048² (20 s → 39 s) mas quase não alterou o OpenMP (30 s → 32 s), sugerindo que a pressão de I/O afeta mais a abordagem de um só *thread* do que as versões que conseguem sobrepor computação à espera de memória.

## Conclusões da discussão

* **Otimização de cache** pode tornar a versão sequencial surpreendentemente rápida em problemas que cabem na hierarquia de memória; apenas quando a dimensão cresce é que o paralelismo se paga.
* **OpenMP** demonstrou a melhor relação esforço/ganho: basta uma diretiva e oferece 3–4 × de aceleração em 4096², com eficiência próxima de 90 %.
* **MPI** não compensa em VMs com pouca memória e rede virtualizada; o custo de comunicação anula o ganho computacional.
* **Híbrido** carece de maior disponibilidade de RAM para ser avaliado com justiça.
* Para futuro trabalho será recomendável testar num nó físico com mais RAM e interligação rápida (InfiniBand ou equivalente), e avaliar também bibliotecas BLAS otimizadas ou off‑loading para GPU.

# Conclusão

O estudo comparou quatro abordagens para a multiplicação de matrizes quadradas: uma implementação sequencial muito otimizada, versões paralelas em OpenMP e MPI, e uma tentativa de versão híbrida.

A otimização centrada no aproveitamento de cache demonstrou que, em problemas que ainda cabem na hierarquia de memória, o paralelismo pode oferecer ganhos reduzidos ou mesmo ser contraproducente — a versão sequencial executou uma matriz 2048 × 2048 em vinte segundos, ultrapassando tanto o OpenMP como o MPI. Contudo, quando a dimensão aumentou para 4096 × 4096, o volume de dados deixou de caber na cache e o cenário mudou: o OpenMP, com quatro *threads*, completou a operação em cerca de um minuto, alcançando um speed‑up de três a quatro vezes em relação ao código sequencial.

A solução MPI, por seu lado, ficou limitada pela latência e pela largura de banda da rede virtual entre os dois servidores, demorando nove minutos no mesmo problema; o custo de comunicação superou o benefício do paralelismo. Já a variante híbrida não pôde ser medida em 4096 × 4096 por falta de memória livre — evidenciando que a disponibilidade de RAM é um requisito para que o paralelismo distribuído seja viável.

Destas medições retiram‑se três lições principais:

1. **A optimização de cache continua fundamental**; um sequencial bem afinado rivaliza com soluções paralelas em problemas que cabem na cache.
2. **O paralelismo de memória partilhada (OpenMP)** mostra a melhor relação esforço‑ganho quando o problema extravasa a cache mas permanece dentro de um único nó.
3. **O paralelismo distribuído (MPI)** só compensa se existir interligação rápida e memória suficiente em cada processo; em ambientes de virtualização ligeira, o overhead de comunicação pode anular o speed‑up.

Para trabalho futuro recomenda‑se deslocar os testes para um nó físico com mais memória e rede de baixa latência, experimentar uma etapa de “packing” para reduzir a comunicação MPI e explorar bibliotecas BLAS optimizadas ou aceleração em GPU, que poderão levar a ganhos adicionais significativos.

# Referências

1. Saraiva, Rui. *gerar\_matriz.c* e exemplo de script *run.sh* facultados no âmbito da unidade curricular Sistemas Paralelos e Distribuídos, Universidade do Algarve, 2025. Adaptados pelo autor para os testes deste relatório.
2. Lawrence Livermore National Laboratory. “LLNL High‑Performance Computing Tutorials.” Disponível em: <https://hpc-tutorials.llnl.gov/>. Acedido em: 5 Mai 2025.
3. Material de apoio e apontamentos fornecidos pela docente da cadeira de Sistemas Paralelos e Distribuídos, Universidade do Algarve, semestre Primavera 2025.