Quadrado Mágico
Desempenho com Programação Paralela e Distribuída

Ricardo Jorge Patrício Correia

2021

Faculdade de Ciências e Tecnologias
Universidade do Algarve
Faro Portugal

Abstract

Nowadays, almost every mobile service and computer that we use, rely on parallel and distributed processing in order to keep up with the ascending use of technology and the rise of new applications complexity. There are many programming languages and frameworks that allow us execute different threads in parallel. This study uses the C programming language along with some frameworks, such as POSIX Threads, OpenMP and MPI to evaluate the differences on the perfomance of the execution of an algorithm that processes square Matrixes to evaluate if they are either a Magic Square, a Semimagic Square or none of both, between the distinct approaches: Sequential execution and Parallel execution, with POSIX Threads, OpenMP, MPI and at last mixing OpenMP and MPI. The obtained results in the study show that any of the parallel approaches considerable diminish the execution time compared to the sequential approach when using an algorithm that implies summing elements of a squared matrix and it also emphazises the importance of granularity in parallel computing.

Keywords

Parallel and Distributed System,Paralell Computing, Distributed Computing, POSIX Threads, OpenMP, MPI, Magic Square

Resumo

Atualmente grande parte dos serviços móveis e computadores que usamos recorrem ao processamento paralelo e distribuído para conseguirem corresponder ao aumento da utilização da tecnologia e à complexidade de novas aplicações. Existem bastantes linguagens de programação e ferramentas que nos permitem implementar execuções de tarefas em paralelo. E estudo usa a linguagem de programação C e recorre a várias ferramentas como POSIX Threads, OpenMP e MPI para avaliar as diferenças no desempenho da execução de um algoritmo que interpreta Matrizes quadradas, para avaliar Quadrados Perfeitos, comparando os tempos de execução das diferentes abordagens: em execução sequencial e em execução paralela, com POSIX Threads, OpenMP, MPI e por último de uma forma híbrida com OpenMP e MPI. Os resultados adquiridos demonstram que qualquer uma das abordagens paralelas diminui consideravelmente o tempo de execução relativamente à abordagem sequencial quando aplicadas a um algoritmo que implique calcular somas de elementos de uma matriz quadrada e enfatiza também a importância da granularidade no processamento paralelo.

Palavras Chave

Sistemas Paralelos e Distribuídos, Computação Paralela, Computação Distribuída, POSIX Threads, OpenMP, MPI, Quadrado Mágico

Conteúdo

1 Introdução

Desde o desenvolvimento do microprocessador nos anos 70 do século XX, tem se vindo a verificar um acrescimento na melhoria de desempenho através do aumento da velocidade do relógio do microprocessador, largura dos barramentos, quantidade de memória *cache*, etc. A Lei de Moore [9], que prevê que o número de transístores dobre a cada dois anos, tem vindo a estagnar nas últimas décadas o que forçou à adoção e criação de novas técnicas.
O crescimento do mercado tecnológico e a subsequente procura por um melhor desempenho deu origem a uma nova abordagem de processamento paralelo através de Sistemas Distribuídos e Sistemas Paralelos, a última fruto da criação de uma nova tecnologia: processadores *multicore*.[10]
Hoje em dia, estas duas abordagens encontram-se na maioria dos serviços que utilizamos e todos os computadores atualmente desenvolvidos possuem, na generalidade, processadores multicore, enquanto que a maioria dos serviços para além de possuírem também eles processadores multicore, estão configurados em *cluster* que consiste num em computadores fraca ou fortemente ligados que trabalham em conjunto separando cargas da mesma tarefa entre si, podendo assim obter ganhos na eficiência de execução. É importante realçar as diferenças entre um Sistema Paralelo e um Sistema Distribuído:

Um **Sistema Paralelo** caracteriza-se como um conjunto de elementos de processamento que comunicam e cooperam na rápida resolução de problemas complexos[8], como representado na figura 1.

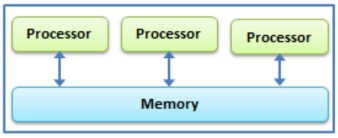


Figura 1: Representação simplificada de Sistema Paralelo, onde é realçada a partilha da memória pelas unidades de processamento.

Um **Sistema Distribuído**, por sua vez, consiste num conjunto de componentes hardware e software interligados entre si através de uma infraestrutura de comunicações, que cooperam e se coordenam entre si apenas pela troca de mensagens, para a execução de aplicações distribuídas[8], como representado na figura 2.

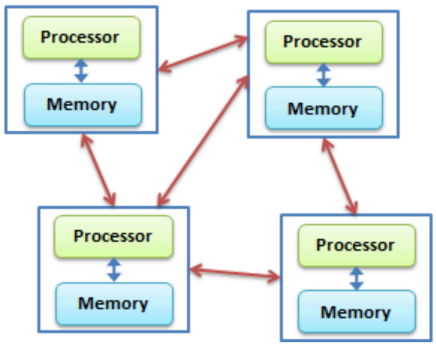


Figura 2: Representação simplificada de Sistema Distribuído, onde se denota a separação das unidades de processamento assim como a memória.

Hoje em dia para implementarmos estas técnicas de computação, são regularmente utilizadas POSIX Threads, OpenMP e ainda MPI. O foco deste estudo é verificar os benefícios da computação paralela aplicando estas técnicas e ferramentas já bastante estudadas e testadas neste tipo de computação. Para isso serão feitas 5 implementações do algoritmo para verificar **Quadrados Mágicos**, a primeira de forma sequencial, sem recurso nenhuma a paralelismo, a segunda com recurso a POSIX Threads onde é esperado um maior esforço da parte do programador na implementação, a terceira com OpenMP onde é oferecido um maior automatismo pela mesma, tornando a tarefa de implementação mais fácil, a quarta com MPI recorrendo a uma máquina alojada num domínio académico, proporcionando a possibilidade de executarmos o algoritmo num sistema distribuído, por último será feita uma implementação híbrida mantendo a execução no sistema distribuído mas correndo também ao paralelismo da ferramenta OpenMP.

No segundo capítulo é dado um enquadramento e são então apresentados alguns conceitos fundamentais, necessários para a leitura deste relatório. O terceiro capítulo descreve o problema, as implementações usadas e como é efetuada a recolha dos resultados assim como as métricas utilizadas. No quarto capítulo são apresentados os resultados das várias implementações onde são comparados, detalhadamente, entre os mesmos. É apresentado uma análise dos resultados e uma discussão. Por fim, no quinto capítulo, é exposta uma conclusão e são exibidos algumas sugestões futuras.

2 Enquadramento

2.1 Quadrado Mágico

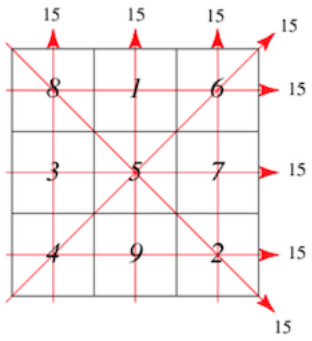


Figura 3: Representação de um Quadrado Mágico, as setas vermelhas apontam para o valor resultante da soma da dos números abrangidos pela seta.

Um Quadrado Mágico é uma matriz quadrada composta por números inteiros únicos, disposta de maneira que a soma de qualquer linha horizontal ou vertical assim como as diagonais principais tenha sempre o mesmo valor, conhecido também por constante mágica. [11] A figura 3 ilustra o quadrado mágico e a obtenção das referidas somas. No caso em que os elementos da matriz quadradas se repetem, por exemplo só 1s consecutivos, o quadrado é designado como quadrado mágico normal. Neste estudo usaremos este tipo de quadrados mágicos de forma a facilitar a obtenção de exemplares de quadrados mágicos e de forma a simplificar o algoritmo.
Caso a soma das diagonais principais falhe em ser igual à constante mágica, o quadrado é designado de Quadro Mágico Imperfeito, enquanto que a soma de alguma das linhas verticais ou horizontais falhe na condição de ser igual à constante mágica o quadrado é considerado como Não Mágico. [11]

2.2 Linguagem C

C é uma linguagem de programação de uso geral e é geralmente associada ao sistema UNIX para o qual foi desenhada e implementada por Dennies Ritchie. No entanto é um linguagem que não está amarrada a qualquer hardware ou sistema e é fácil escrever programas que corram sem alterações em qualquer máquina que suporte C. [6] C atualmente ainda é muito utilizada, particularmente no *kernel* de Linux que contém cerca de 97% do seu código em C.[1]

2.3 Threads

Uma *thread* consiste numa tarefa realizada por um determinado programa, pode ser gerida independentemente pelo escalonador do CPU podendo ser vista geralmente como uma componente de um processo.
Isto possibilita a execução múltipla *threads* no mesmo processo, concorrentemente, partilhando alguns recursos como a memória. O principal benefício das *threads* é melhorar o desempenho de um programa, estruturando-o devidamente para realizar diversas tarefas ao mesmo tempo. [7]

2.3.1 POSIX Threads

As POSIX Threads, também conhecidas por pthreads são um modelo de execução para threads geralmente encontradas em C e C++. As pthreads oferecem várias funcionalidades como a criação, sincronização e manipulação de threads num programa. Uma thread é iniciada através da função pthreads_create e pode ser posteriormente chamada a função pthreads_join para fazer esperar que as threads executem e assim sincronizá-las. [7]

2.4 OpenMP

OpenMP é uma API que usa o modelo fork-join de execução paralela representado na figura 4, é direcionada a programas que possam ser executados sequencialmente ou paralelamente com múltiplas threads. OpenMP está atualmente disponível para três tipos de linguagens de programação, são elas: C, C++ e Fortran. A implementação de paralelização com OpenMP é mais simples que com POSIX Threads dado o modelo fork-join utilizado por OpenMP que permite ao utilizador somente declarar certas blocos do programa para serem executadas paralelamente, não sendo obrigatório toda uma configuração, especificando quais são as cargas de trabalho para cada thread. [2]

Figura 4: Modelo *fork-join* do OpenMP

2.5 MPI

MPI é um padrão para a comunicação de dados que especifica uma interface de passagem de mensagens entre nós de um sistema, desenhado para sistemas distribuídos, sem memória partilhada. Esta interface é definida por um consenso pelo MPI *Forum* e prevê a ligação a C e Fortran90.[3] O seu conceito geral poder ser observado na figura 5.

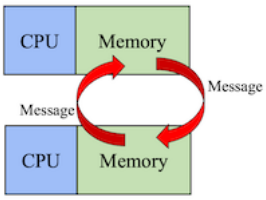


Figura 5: Representação da troca de mensagens entre nós com MPI

3 Estudo de casos

3.1 Questões

Existem algumas questões sobre o processamento paralelo, executando o algoritmo para verificar quadrados mágicos.

1. Pode a computação paralela melhorar o desempenho do algoritmo num sistema multicore?
2. Quais as diferenças no desempenho em paralelo, entre o uso de POSIX Threads, OpenMP e MPI?
3. Quais as diferenças no esforço necessário na implementação entre as seguintes abordagens?

Este estudo procura responder a estas 3 questões usando uma metodologia empírica através de testes de execução e algumas métricas, como tempo de execução e número de linhas de código.

3.2 Método

Para avaliar o desempenho das diferentes abordagens, foram criados 5 programas diferentes escritos em na linguagem de programação C. O primeiro programa consiste numa execução sequencial de código, que serve como ponto de referência para restantes programas.

Realizaram-se medições para os 5 diferentes programas, para os 3 resultados possíveis na avaliação do quadrado mágico e para vários tamanhos de quadrados. Posterior verificou-se entre o mesmo tipo de quadrado e tamanho nos diferentes programas para comparar os desempenhos dos destes.

Por último mediu-se o número de linhas de cada programa para determinar o esforço necessário para cada abordagem.

3.3 Implementações

Apresentadas quais as diferentes abordagens que foram usadas neste estudo, são neste capítulo descritas com um maior detalhe.

3.3.1 Sequencial

A primeira abordagem que se realizou, talvez a mais importante pois serviu de base às restantes implementações. A implementação seguiu-se pelo desenho de 3 grandes funcionalidades. A leitura de todo o ficheiro que seria passado como argumento ao programa, o processamento do quadrado através de funções que percorrem a matriz, somando linhas, colunas e as 2 principais diagonais, e por fim a validação do quadrado mágico, imprimindo para o utilizador o resultado da avaliação do quadrado mágico.
Para melhorar a gestão de memória, aproveitou-se uma regra da nomeação de ficheiros para os quadrados mágicos usados neste estudo que continham o número de linhas/colunas (ao qual chamaremos de N para simplificação de nome) no seu nome, para ler em primeiro lugar o nome do ficheiro e assim saber exatamente o tamanho necessário para guardar o quadrado. Após a obtenção do tamanho do número de linhas ou colunas, cria-se um *array* bi-dimensional de dimensões N por N. É de notar que este é um *array* foram alocados dinamicamente (guardados na *heap*) nas implementações, devido ao tamanho das amostras utilizadas e ao tamanho reduzido do *stack* do programa, o que pode produzir um execução menos eficiente, pois o acesso à memória no *stack* é feito numa só execução ao invés de na *heap*, onde os endereços vão ficar espalhados pela memória principal.

O array bi-dimensional é então preenchido através da leitura integral do ficheiro. Após o seu preenchimento é necessário é calculada a soma da primeira linha do quadrado,e guardada como constante mágica, valor a ser usado nas funções de avaliação do quadrado mágico. Antes de explicar os próximos passos da implementação vale a pena descrever as funções a serem usadas na avaliação do quadrado, 3 funções: avaliação de linhas, avaliação de colunas e avaliação de diagonais. Cada uma delas retorna um inteiro 0 ou 1 (útil para usarmos como resultado lógico, verdadeiro ou falso) e percorrem a matriz linha a linha, coluna a coluna ou ambas as diagonais principais, começando com o valor a retornar como 1 (verdadeiro) e no final de cada soma de linha ou coluna ou diagonal verifica se essa é igual à soma dada como argumento, valor estático calculado anteriormente. Caso esta igualdade falhe, ou por outras palavras, seja falsa o valor a retornar passa a 0 (falso), a função deixa de percorrer a matriz e o valor 0 é retornado, caso a verificação da soma seja sempre verdadeira o valor a retornar mantém-se a 1 e ao ser percorrida toda a matriz, esse é retornado.
É chamada primeiramente a função de verificação de linhas e caso seja verdadeira então é chamada a de colunas, caso alguma destas funções falhe, ou seja retorne

falso, o quadrado é imediatamente avaliado como "Não é Quadrado Mágico" e o programa termina, caso contrário é chamada a função de verificação das diagonais e caso seja verdadeira o quadrado é avaliado como "Quadrado Mágico Imperfeito," caso contrário é avaliado como "Quadrado Mágico," terminando o programa em ambos os casos.

3.3.2 POSIX Threads

Para a implementação com *POSIX Threads*, foram reaproveitadas as funções de leitura do quadrado, criação da matriz, soma da primeira linha para determinar constante mágica e a função de impressão da avaliação do quadrado mágico. Para além do legado deixado pela implementação sequencial foram necessárias várias alterações à estrutura do programa tal como à adição de novas funcionalidades e variáveis. Foram criadas estruturas para guardar a informação necessária para as *POSIX Threads*, assim com *arrays* de *Pthreads*, uma função para construir as estruturas com os respetivos índices iniciais e finais, de acordo com o tamanho do quadrado lido, separando por igual as quantidades a processar por cada *Pthreads*, funções de criação e sincronização de *Pthreads*.
Foi alterada a maneira como era avaliado o quadrado mágico, ao invés da sequencial que analisava se as funções de análise eram verdadeiras ou não, é avaliada através de uma variável global para cada função que é alterada para 0 caso seja encontrada uma soma não igual à constante mágica, nesta implementação e nas seguintes implementações paralelas. Esta técnica permite uma melhor coordenação de *threads*, pois as funções de verificação de somas só são executadas enquanto as respetivas variáveis globais forem verdadeiras.

Ao terminarem as funções as variáveis globais de cada função (linhas, colunas e diagonais) são verificadas da mesma maneira que na implementação sequencial, imprimindo subsequentemente o resultado da avaliação do quadrado.

Esta implementação permite correr as funções de avaliação do quadrado em paralelo assim como distribuir a carga de cada função por diferentes threads. A escolha do número de *pthreads* e a distribuição de carga de cada uma é analisada no capítulo 4.

3.3.3 OpenMP

A implementação paralela com OpenMP requereu menos trabalho, dado os automatismos providenciados pela ferramenta OpenMP. Partindo da implementação base (sequencial), apenas foram acrescentadas *omp sections* ,que permite que blocos de programa sejam executados paralelamente, e colocados dentro destes as funções de avaliação de linhas e avaliação de colunas, deixa a execução da avaliação de diagonais de fora, desvalorizando o seu tempo de execução (muito reduzido, como é analisado no capítulo 4) de modo a poupar uma *thread*.
Além deste blocos *omp* foram adicionados umas chamadas de OpenMP que possibilitam a paralelização de ciclos *for* dentro das funções de avaliação, de modo a distribuir a carga de processamento por diferentes *threads*, algo que teve de

ser feito manualmente com *POSIX Threads*.

3.3.4 MPI

A última ferramenta de paralelização MPI, implicou uma implementação mais exigente, porque tal como apresentada, MPI funciona através de troca de mensagens entre processos. Este pormenor pode trazer algumas problemas como *deadlocks* em que processos ficam eternamente à espera de receber de outros processos. Para evitar isto foi necessário um maior rigor na adaptação das funções já construídas na implementação sequencial. Foi usada a função *MPI_Allreduce* que consiste em na receção de um *buffer* de dados, num operação sobre o mesmo *buffer* e no envio novamente do *buffer* após a operação. Dentro deste *buffer* vai uma variável que nesta implementação, é equivalente à variável global utilizada nas outras implementações paralelas, pois com MPI os processos não vão partilhas e memória e o que invalida o uso de variáveis globais. Foi utilizada a mesma estratégia de distribuição de carga, adotada na implementação com *POSIX Threads*. Assim, foram calculados índices iniciais e finais para cada função de avaliação e pelo facto de os processos não serem sempre distribuídos equitativamente, foi guardada um variável com resto, dado às funções que ficariam com menos carga, de modo a balancear o número de comunicações para evitar *deadlocks*. Foram posteriormente chamadas estas funções, primeiro a de avaliação de linhas, sincronizando os processos após o seu fim e chamada posteriormente a função de avaliação de colunas. Após a execução destas funções são verificadas finalmente as diagonais e analisadas as variáveis de cada função, fazendo uma análise final do quadrado e impresso o resultado desta.

3.3.5 MPI e OpenMP

Por último, foram combinadas as implementações com OpenMP e MPI de modo a criar uma implementação híbrida, combinando a criação dos processos de MPI com a paralelização com *threads* de OpenMP. Esta implementação não necessitou de muitas alterações, apenas no modo como era realizada a chamada da função de inicialização de MPI. MPI está desenhado para ter compatibilidade com OpenMP.

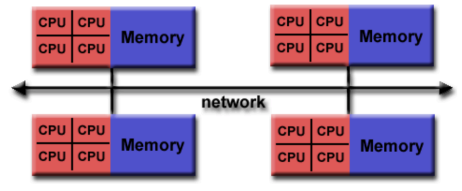


Figura 6: Representação da computação híbrida com OpenMP e MPI

3.4 Medições

Para avaliar o desempenho de cada implementação foram utilizadas várias métricas para além do **tempo de execução** como a **aceleração**, **eficiência** e **escalabilidade**. As medições do tempo de execução de cada programa foram realizadas recorrendo, primeiramente, ao comando time que apresenta estatísticas de tempo da execução do programa dado como argumento à função. [4]

Esta medição temporal provou-se pouco precisa no contexto do nosso problema, pois o tempo de execução do programa consistia em mais de 90%, dependendo do tamanho da matriz, na leitura de dados que se encontram na memória secundária.
Atendendo ao facto que a leitura era realizada da mesma maneira em qualquer um dos programas e que não beneficiava do paralelismo, foi adotada outra medição para o tempo de execução que recorre à função *gettimeofday* que guarda o tempo do relógio do sistema numa estrutura própria.[5] Foram criadas diferentes estruturas, uma para guardar os tempos de leitura, uma para cada função principal do algoritmo, uma para o tempo total destas funções (útil no paralelismo) e por fim uma para o tempo total de execução.
Imediatamente antes e após a chamada de cada função a medir foram marcados esse instantes nas respetivas estruturas e no final do programa foi feita a diferença entre os tempos para determinar o tempo de cada função do programa. Esta métrica ajudou na perceção da determinação da carga de trabalho e assim separar o trabalho pelas diferentes *threads*.

A aceleração é definida como a razão entre o tempo de execução da implementação sequencial e o tempo de execução do mesmo problema numa implementação paralela.[8] É dada pela expressão S, onde *t1* corresponde ao tempo da implementação sequencial e *tp* ao tempo de execução da implementação paralela.

$$S = \frac{t_1}{t_p}$$

Já a eficiência defini-se como a razão entre a aceleração e o número de processadores, ou seja, a fração de tempo que os processadores realizam trabalho útil [8]. Por sua vez, a escalabilidade refere-se à propriedade de um sistema manter a eficiência quando o volume de computação aumenta [8]. Por fim, o esforço para implementar os seguintes programas: sequencial, com POSIX Threads, com OpenMP, MPI e híbrido com MPI e OpenMP foram medidas o número de linhas escritas de código fonte de cada programa. Usou-se também esta métrica pois apresenta um valor com o qual podemos comparar os diferentes programas, quanto menor o valor menor o esforço.

Cada medição do tempo de execução foi realizada 31 vezes, (30) desprezando-se a primeira, em cada dimensão de quadrados analisados, para garantir uma boa amostra de dados antes de calcular a média de tempos de execução. Com

estes dados obtidos, foram então calculados as métricas: aceleração, eficiência e escalabilidade.

Os testes foram realizados em dois ambientes:
• macOS com um processador 2,2 GHz Quad-Core Intel ® Core ™ i7-4770HQ com 6 MB de memória *cache* 16 GB 1600 MHz DDR3 de memória principal.

• Debian 10 com um processador 2,0GHz Intel ® Xeon ® 6138 com 27,5 MB de memória cache e 8GB de memória principal.

A máquinas usadas para estes testes possuem ambas processadores multicore, a primeira com 4 threads com *hyper-threading*, equivalendo a 8 threads. A segunda máquina possui 20 threads, mas tem a configuração equivalente ao comportamento de 4 núcleos independentes com 4 threads em cada núcleo.

Todas os programas foram compilados com o compilador *GNU Compiler Collection*, mais conhecido por GCC, versão 10.2 no macOs e 8.3 na máquina Debian 10. Foi utilizado a opção -fopenmp para a compilação com OpenMP e utilizado o Open MPI para a implementação com MPI e Híbrida. Todos os programas foram também compilados com a otimização do GCC O3.

4 Análise de resultados e discussão

4.1 Sequencial

A avaliação do desempenho começou a ser realizada com medições do tempo de execução das implementações sequenciais, e conforme abordado no capítulo 3.3, as medições do tempo de execução foram realizadas elaborando medições funcionais do programa ao invés da medição do tempo total da execução, com o propósito de poder determinar melhor a granularidade necessária para as implementações paralelas, percebendo em que funções o programa consome mais tempo, e também para obter o tempo de execução da secção do programa que é realmente paralelizada, tornando a análise de resultados mais rigorosa. O tempo de leitura não é afetado com a paralelização neste estudo, pois as velocidades de leitura corresponderam ao máximo de velocidade possível de ser alcançada pela memória secundária do ambiente de teste. Testou-se o algortimo para avaliar Quadrados Perfeitos só com amostras de quadrados perfeitos, de modo a obter uma leitura completa da matriz e assim perceber os tempos de execução das funções, quando executados completamente. Foram utilizados vários quadrados perfeitos de dimensões NxN, começando em N = 255 até N = 20001. Como podemos analisar pela figura 7, os tempos de execução de cada função do quadrado mágico varia bastante, a leitura é a função que ocupa mais tempo, com cerca de 90% do tempo total de execução, seguido de colunas, linhas e diagonais com aproximadamente 10%, 0.2% e 0.001% , respetivamente, do tempo total.

Tendo em conta que a paralelização da leitura, e por sua vez uma redução dos

seus tempos de execução, não ser possível neste ambiente de teste, o desenho para a paralelização focou-se principalmente numa melhor paralelização das colunas. A análise da melhor configuração de granularidade e número de threads é analisada no próximo capítulo.

	Linhas (ms)	Colunas (ms)	Diagonais (ms)	Leitura (ms)	Total (ms)
255	0,03	0,04	0,00	8,00	8,13
501	0,10	0,29	0,00	32,96	33,45
1001	0,42	6,00	0,01	151,13	157,68
2003	1,67	33,27	0,04	507,55	542,69
5001	9,03	280,98	0,13	3 371,62	3 661,62
10001	38,65	1 333,54	0,33	12 893,07	14 266,56
15001	82,79	3 082,44	0,59	32 215,28	35 382,28
20001	172,95	6 374,68	1,08	53 232,38	59 783,38

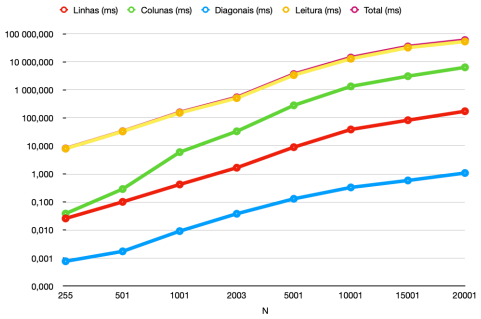


Figura 7: Análise dos tempos de execução do Quadrado Mágico, na implementação sequencial em escala logarítmica.

4.2 POSIX Threads

Com a análise dos tempos funcionais do programa da abordagem sequencial, foi a paralelização foi desenhada seguindo os seguintes objetivos.

- A execução das funções de avaliação de linhas, colunas e diagonais teria de ser feita em paralelo.
- A função de avaliação de colunas demora mais de 10x mais do que a função de avaliação das linhas, logo só as colunas necessitarão de distribuição de carga.

Assumindo isto, foram testadas várias configurações aumentando o número de *threads* na avaliação de colunas e verificando os tempos de execução. Foram usados só quadrados perfeitos para conseguir ter os tempos máximos de execução, tendo em conta que um quadrado não mágico ou um quadrado mágico imperfeito irá parar à primeira linha ou coluna que não perfaça uma soma igual à constante mágica, tornando os resultados menos coerentes. As dimensões dos quadrados testados NxN, com N mínimo de 1023 e N máximo de 20001, com o número de *POSIX Threads* a variar entre 2 e 6 (são necessárias duas *threads* para as linhas e diagonais, uma para cada), apresentaram os valores descritos na figura 8.

	2 Threads (ms)	4 Threads (ms)	5 Threads (ms)	6 Threads (ms)
1023	1,41	1,24	1,29	1,27
2047	17,15	10,90	10,80	9,69
5001	145,29	91,41	91,75	88,55
10001	690,50	442,46	452,68	442,20
15001	1 752,06	1 088,60	1 095,23	1 063,75
20001	3 821,29	2 513,84	2 393,91	2 354,72

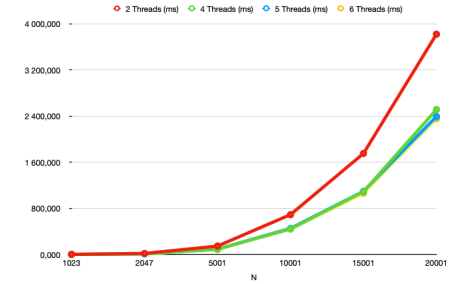


Figura 8: Análise dos tempos de execução do Quadrado Mágico, na implementação com POSIX Threads, alterando o número de threads na função de avaliação da soma das colunas em escala linear.

Em análise à implementação com 6 *Threads*, o máximo possível na máquina de teste, foram realizados vários testes para os diferentes tipos quadrados de dimensão NxN, com N com valores entre 255 e 20001 como apresentados na figura 10. Determinando as métricas apresentadas para este teste, tem-se que:
A implementação paralela com *Pthreads* tem efetivamente um tempo de execução inferior exceto no caso onde os quadrados não são quadrados mágicos, este acontecimento poderá dever-se ao tempo necessário de execução para a distribuição de carga e inicialização de *threads*.
A figura 9 apresenta as acelerações nas várias dimensões e nos três tipos de quadrados testados, na implementação com *pthreads*.

Analisando a figura 10 e 9 podemos concluir que a implementação com *POSIX Threads* apresenta uma **aceleração** média de 2,45 em quadrados mágicos perfeitos e imperfeitos, no entanto a implementação apresenta piores resultados com quadrados não mágicos, possuindo uma aceleração média de 0,15. A implementação em análise apresenta também uma **eficiência** média de 0,306. A

Quadrados Mágicos

	Aceleração	Eficiência
255	4,828E-01	6,034E-02
501	1,167E+00	1,458E-01
1023	2,449E+00	3,062E-01
2047	3,626E+00	4,533E-01
5001	3,137E+00	3,922E-01
10001	3,096E+00	3,870E-01
15001	2,877E+00	3,597E-01
20001	2,782E+00	3,478E-01

Quadrados Mágicos Imperfeitos

	Aceleração	Eficiência
255	4,516E-01	5,645E-02
501	1,455E+00	1,818E-01
1023	2,153E+00	2,691E-01
2047	3,452E+00	4,315E-01
4095	3,228E+00	4,035E-01
10001	3,108E+00	3,885E-01
15001	2,959E+00	3,698E-01
20001	2,794E+00	3,492E-01

Quadrados Não-Mágicos

	Aceleração	Eficiência
255	2,000E-01	2,500E-02
501	2,143E-01	2,679E-02
1023	2,000E-01	2,500E-02
2047	1,754E-01	2,193E-02
5001	1,044E-01	1,305E-02
10001	1,271E-01	1,589E-02
15001	7,949E-02	9,936E-03
20001	1,317E-01	1,646E-02

Figura 9: Aceleração de cada dimensão comparativamente à implementação sequencial.

escalabilidade foi estudada apenas para quadrado mágicos, a implementação revelou-se **não** ser **escalar**, apresentando eficiência com valores próximos de 1 com duas *threads* e descendo até valores de 0,3 com as 8 *threads*.

4.3 OpenMP

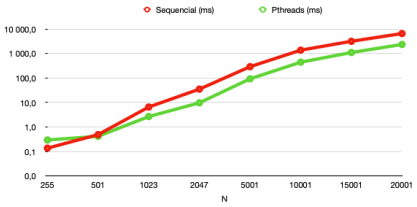
Olhando para a implementação com OpenMP, decidiu-se fazer a análise dos resultados com duas versões diferentes da implementação OpenMP. Uma com a mesma configuração da implementação com *POSIX Threads* com os melhores resultados, e outra sem definição de *threads* para cada função de avaliação, designamo-la de OpenMP Indefinido. A figura 11 apresenta os resultados das implementações com OpenMP contra os resultados da implementação sequencial, com quadrados NxN, em que N varia entre 255 e 200001.

Com estes resultados observamos que a implementação em que a configuração do número de *threads* está definida, produzem tempos de execução inferiores à implementação com OpenMP indefinido.

As acelerações das duas implementações com *OpenMP* são apresentadas na figura 12. Podemos concluir que a implementação com OpenMP definido apresenta uma **aceleração** média de 2,13 em quadrados mágicos perfeitos e de 1,83 em quadrados mágicos imperfeitos, enquanto que com quadrados não mágicos apresenta uma aceleração média de 0,09. A eficiência e escalabilidade foi estudada apenas para os quadrados mágicos. A implementação em análise apresenta também uma **eficiência** média de 0,266.

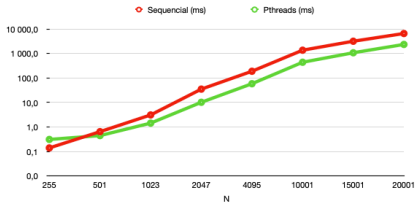
Quadrado Mágico

	Sequencial (ms)	Pthreads (ms)
255	0,14	0,29
501	0,49	0,42
1023	6,54	2,67
2047	35,14	9,69
5001	290,37	92,55
10001	1 372,94	443,49
15001	3 166,48	1 100,43
20001	6 551,35	2 354,72



Quadrado Mágico Imperfeito

	Sequencial (ms)	Pthreads (ms)
255	0,14	0,31
501	0,64	0,44
1023	3,10	1,44
2047	35,14	10,18
4095	188,79	58,48
10001	1 369,05	440,53
15001	3 180,61	1 075,02
20001	6 578,08	2 354,72



Quadrado Não-Mágico

	Sequencial (ms)	Pthreads (ms)
255	0,05	0,25
501	0,06	0,28
1023	0,07	0,35
2047	0,10	0,57
5001	0,19	1,82
10001	0,37	2,91
15001	0,62	7,80
20001	4,21	31,97

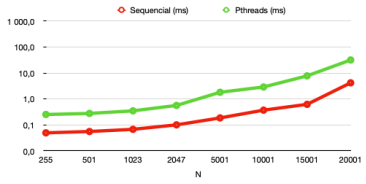
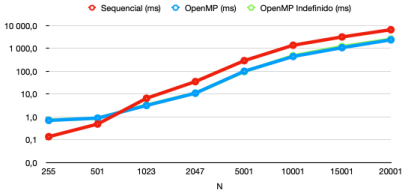


Figura 10: Sequencial vs POSIX Threads: Análise dos tempos de execução em escala logarítmica de Quadrado Mágicos, Quadrados Mágicos Imperfeitos e Quadrados Não Mágicos, na implementação com POSIX Threads.

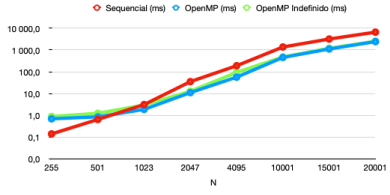
Quadrados Mágicos

	Sequencial (ms)	OpenMP (ms)	OpenMP Indefinido (ms)
255	0,14	0,70	0,72
501	0,49	0,89	0,87
1023	6,54	3,17	3,29
2047	35,14	10,90	10,94
5001	290,37	99,73	96,13
10001	1 372,94	441,65	476,08
15001	3 166,48	1 073,59	1 202,95
20001	6 551,35	2 363,45	2 538,76



Quadrados Mágicos Imperfeitos

	Sequencial (ms)	OpenMP (ms)	OpenMP Indefinido (ms)
255	0,14	0,69	0,88
501	0,64	0,85	1,23
1023	3,10	1,86	2,95
2047	35,14	11,03	12,68
4095	188,79	56,32	93,08
10001	1 369,05	454,07	478,53
15001	3 180,61	1 112,53	1 197,37
20001	6 578,08	2 418,72	2 540,22



Quadrados Não-Mágicos

	Sequencial (ms)	OpenMP (ms)	OpenMP Indefinido
255	0,05	0,73	0,79
501	0,06	0,85	0,81
1023	0,07	0,77	0,92
2047	0,10	1,01	1,06
5001	0,19	2,16	3,55
10001	0,37	1,73	6,77
15001	0,62	3,05	10,74
20001	4,21	12,86	15,32

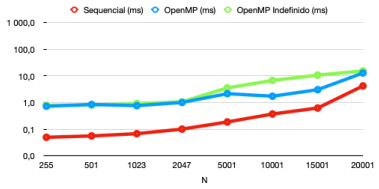


Figura 11: Sequencial vs OpenMP: Análise dos tempos de execução em escala logarítmica de Quadrado Mágicos, Quadrados Mágicos Imperfeitos e Quadrados Não Mágicos, nas implementações com OpenMP.

OpenMP Definido

Quadrados Mágicos

	Aceleração	Eficiência
255	2,000E-01	5,000E-02
501	5,506E-01	1,376E-01
1023	2,063E+00	5,158E-01
2047	3,224E+00	8,060E-01
5001	2,912E+00	7,279E-01
10001	3,109E+00	7,772E-01
15001	2,949E+00	7,374E-01
20001	2,772E+00	6,930E-01

Quadrados Mágicos Imperfeitos

	Aceleração	Eficiência
255	2,029E-01	2,536E-02
501	7,529E-01	9,412E-02
1023	1,667E+00	2,083E-01
2047	3,186E+00	3,982E-01
4095	3,352E+00	4,190E-01
10001	3,015E+00	3,769E-01
15001	2,859E+00	3,574E-01
20001	2,720E+00	3,400E-01

Quadrados Não-Mágicos

	Aceleração	Eficiência
255	6,849E-02	8,562E-03
501	7,059E-02	8,824E-03
1023	9,091E-02	1,136E-02
2047	9,901E-02	1,238E-02
5001	8,796E-02	1,100E-02
10001	2,139E-01	2,673E-02
15001	2,033E-01	2,541E-02
20001	3,274E-01	4,092E-02

OpenMP Indefinido

Quadrados Mágicos

	Aceleração	Eficiência
255	1,944E-01	2,431E-02
501	5,632E-01	7,040E-02
1023	1,988E+00	2,485E-01
2047	3,212E+00	4,015E-01
5001	3,021E+00	3,776E-01
10001	2,884E+00	3,605E-01
15001	2,632E+00	3,290E-01
20001	2,581E+00	3,226E-01

Quadrados Mágicos Imperfeitos

	Aceleração	Eficiência
255	1,591E-01	1,989E-02
501	5,203E-01	6,504E-02
1023	1,051E+00	1,314E-01
2047	2,771E+00	3,464E-01
4095	2,028E+00	2,535E-01
10001	2,861E+00	3,576E-01
15001	2,656E+00	3,320E-01
20001	2,590E+00	3,237E-01

Quadrados Não-Mágicos

	Aceleração	Eficiência
255	6,329E-02	7,911E-03
501	7,407E-02	9,259E-03
1023	7,609E-02	9,511E-03
2047	9,434E-02	1,179E-02
5001	5,352E-02	6,690E-03
10001	5,465E-02	6,832E-03
15001	5,773E-02	7,216E-03
20001	2,748E-01	3,435E-02

Figura 12: Aceleração de cada dimensão comparativamente à implementação sequencial

4.4 MPI

A análise à implementação com MPI teve infelizmente uma grande contrariedade neste estudo, devido à incapacidade de cooperação da máquina de teste com o servidor providenciado para ser integrado juntamente com a máquina de teste num sistema distribuído. Esta incapacidade de cooperação deveu-se a problemas técnicos com o uso do VPN fornecido pela Universidade do Algarve. Uma solução encontrada para conseguir simular um sistema distribuído foi testar a execução no apenas no *cluster* remoto, já que este está configurado para agir como 4 processadores independentes, podendo assim simular um sistema distribuído com 4 nós. No entanto esta máquina teve problemas em processar quadrados com tamanhos mais elevados, devido à limitação da sua memória principal (8GB), pois os testes foram corridos usando o máximo de nós possíveis (4) e os ficheiros que continham quadrados com dimensões NxN com N maior ou igual a 20001 ocupavam uma memória superior a 2GB, o que multiplicado pelos 4 processos do *cluster* ultrapassavam a capacidade de memória principal, causando custos adicionais no desempenho do programa. Atendendo a este pormenor, a dimensão máxima dos quadrados foi reduzida para 15001.

A figura 13 apresenta os tempos de execução da implementação com MPI.

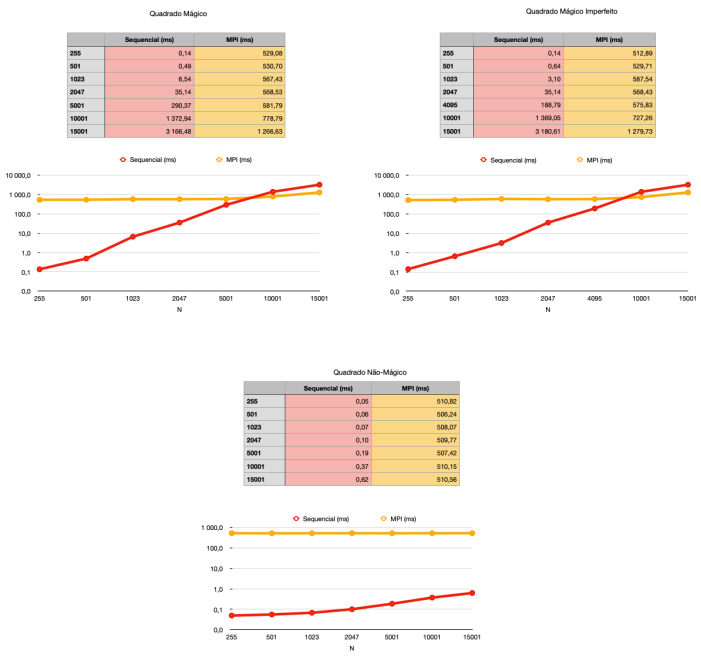


Figura 13: Sequencial vs MPI: Análise dos tempos de execução em escala logarítmica de Quadrado Mágicos, Quadrados Mágicos Imperfeitos e Quadrados Não Mágicos, nas implementações com MPI.

Os resultados observados com a implementação com MPI, apresentaram o mesmo comportamento do que as outras implementações paralelas no que aos quadrados não-mágicos diz respeito, no entanto nos outros dois tipos de quadrados, mágicos perfeitos e imperfeitos, são apresentados superiores à implementação sequencial até dimensões mais elevadas (5001 em ambos os tipos). Este facto deveu-se ao custo de processamento da inicialização e sincronização dos processos com MPI, pois os tempos de execução das funções de avaliação foram em geral inferiores às restantes implementações paralelas. Este tempo extra de execução do MPI rondou os 500ms, como se pode verificar pelos tempos de execução dos quadrados com dimensões inferiores.

Este fator teve um papel determinante nas acelerações baixando, consideravelmente, o seu valor, como é apresentado na figura 14. A **aceleração** média foi de 0,6909 0,68061 e 0,0004 para quadrados mágicos, quadrados mágicos imperfeitos e quadrados não mágicos, respetivamente. A **eficiência** média foi de 0,1725. Seriam necessários testes com dimensões superiores para perceber melhor o comportamento da eficiência, visto que esta é sempre crescente, começando com um valor de 0,00005 na dimensão mais baixa e alcançando o seu máximo na maior dimensão testada, de 0,6225.

Quadrados Mágicos

	Aceleração	Eficiência
255	2,646E-04	3,000E-05
501	9,233E-04	1,154E-04
1023	1,153E-02	1,441E-03
2047	6,181E-02	7,726E-03
5001	4,991E-01	6,239E-02
10001	1,763E+00	2,204E-01
15001	2,500E+00	3,125E-01

Quadrados Mágicos Imperfeitos

	Aceleração	Eficiência
255	2,730E-04	3,000E-05
501	1,208E-03	1,510E-04
1023	5,276E-03	6,595E-04
2047	6,182E-02	7,727E-03
4095	3,279E-01	4,098E-02
10001	1,882E+00	2,353E-01
15001	2,485E+00	3,107E-01

Quadrados Não-Mágicos

	Aceleração	Eficiência
255	9,000E-05	1,224E-05
501	1,185E-04	1,482E-05
1023	1,378E-04	1,722E-05
2047	1,962E-04	2,452E-05
5001	3,744E-04	4,681E-05
10001	7,253E-04	9,066E-05
15001	1,214E-03	1,518E-04

Figura 14: Aceleração de cada dimensão comparativamente à implementação sequencial

4.5 MPI e OpenMP

A análise à implementação híbrida com MPI e OpenMP realizou-se na mesma condição explicada na análise à implementação com MPI puro. Era esperado uma melhoria considerável dos tempos de execução com a implementação híbrida em relação à implementação com MPI, o que tal não aconteceu, apesar de efetivamente os tempos de execução terem diminuído. Analisaram-se quadrados de dimensões NxN com N a variar entre 255 e 15001. A figura 15 apresenta os tempos de execução da implementação híbrida em comparação com a implementação base, a sequencial.

A figura 16 apresenta as respetivas acelerações e eficiências para as diferentes dimensões.

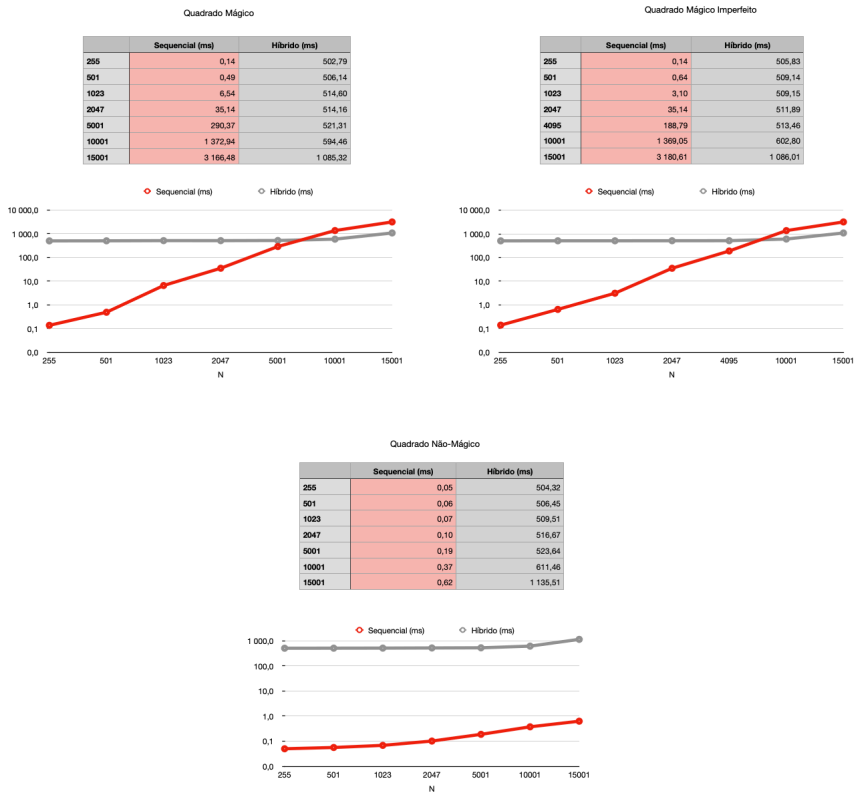


Figura 15: Sequencial vs MPI e OpenMP: Análise dos tempos de execução em escala logarítmica de Quadrado Mágicos, Quadrados Mágicos Imperfeitos e Quadrados Não Mágicos, na implementação híbrida.

A aceleração foi semelhante à observada na implementação com MPI com uma ligeira melhoria, esta teve um média de 0,838 em quadrados mágicos, 0,806 em quadrados mágicos imperfeitos e 0,00029. Quanto à eficiência esta sofreu um grande decréscimo, devido maior utilização de threads, pois nesta implementação híbrida, cada um dos processos usa 8 threads cada perfazendo um total de 32 threads.

Quadrados Mágicos

	Aceleração	Eficiência
255	2,785E-04	8,701E-06
501	9,681E-04	3,025E-05
1023	1,271E-02	3,972E-04
2047	6,834E-02	2,136E-03
5001	5,570E-01	1,741E-02
10001	2,310E+00	7,217E-02
15001	2,918E+00	9,117E-02

Quadrados Mágicos Imperfeitos

	Aceleração	Eficiência
255	2,768E-04	8,649E-06
501	1,257E-03	3,928E-05
1023	6,089E-03	1,903E-04
2047	6,865E-02	2,145E-03
4095	3,677E-01	1,149E-02
10001	2,271E+00	7,097E-02
15001	2,929E+00	9,152E-02

Quadrados Não-Mágicos

	Aceleração	Eficiência
255	9,914E-05	3,098E-06
501	1,185E-04	3,702E-06
1023	1,374E-04	4,293E-06
2047	1,935E-04	6,048E-06
5001	3,628E-04	1,134E-05
10001	6,051E-04	1,891E-05
15001	5,460E-04	1,706E-05

Figura 16: Aceleração de cada dimensão comparativamente à implementação sequencial assim como a sua eficiência

4.6 PThreads vs OpenMP vs MPI vs MPI e OpenMP

Comparando todas as implementações separadas pelos três tipos diferentes de quadrados, é de salientar que todas as implementações paralelas têm um desempenho melhor em quadrados com uma dimensão superior nos dois tipos de quadrados mágicos e por contrapartida todas as implementações paralelas têm um pior desempenho no tempo de execução em comparação com a implementação sequencial. Este agravamento de desempenho tem que ver com o acréscimo do tempo de inicialização e criação de *threads* nas implementações paralelas. A implementação com POSIX Threads tem uma ligeira vantagem sobre as outras implementações paralelas em todos os tipos de quadrado, no entanto a sua vantagem é reduzida em quadrados mágicos de dimensões superiores.

Figura 17: Sequencial vs POSIX Threads vs OpenMP vs MPI vs MPI+OpenMP : Análise dos tempos de execução em escala logarítmica de Quadrado Mágicos, Quadrados Mágicos Imperfeitos e Quadrados Não Mágicos, entre as diferentes implementações.

4.7 Esforço

Por fim foi analisado o esforço requerido por cada implementação. Tal como a figura 18 apresenta, a implementação que implicou um maior esforço por parte do programador, foi a com POSIX Threads apresentando no total 358 linhas de código, no sentido inverso a implementação Sequencial foi a apresentação que requereu menos esforço, com 271 linhas de código.

Sequencial	POSIX Threads	OpenMP	MPI	Híbrido
271,00	358,00	294,00	290,00	294,00

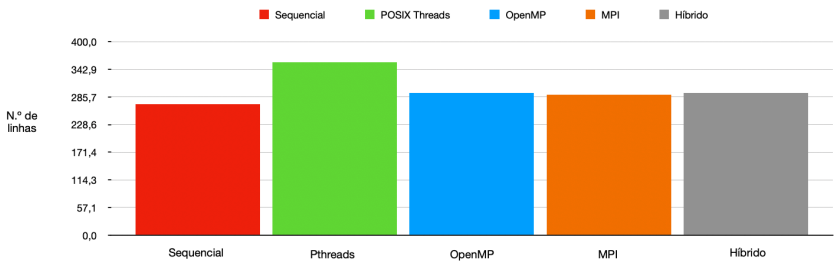


Figura 18: Esforço representado em linhas de código, para cada implementação

5 Conclusão e Comentários Finais

Este estudo debruçou-se sobre a programação paralela e distribuída, particularmente nos seus benefícios no desempenho comparativamente à programação sequencial, usando um programa de avaliação de quadrados mágicos escrito em linguagem C. Realizaram-se várias implementações com diferentes abordagens e testaram-se vários tipos de quadrados, mágicos perfeitos, mágicos imperfeitos e não-mágicos, comparando execuções paralelas contra a execução sequencial.

Os testes realizados mostraram que para conjunto de dados pequenos por vezes a paralelização não é consideravelmente benéfica nem merecedora de ser implementada, porém que conjunto de dados de maior dimensão a computação paralela exulta no que a melhor desempenho diz respeito. POSIX Threads mostraram ser a ferramenta mais eficiente para implementar a paralelização, muito por culpa do seu poder de configuração, obtendo em geral tempos de execução inferiores relativamente às restantes implementações. OpenMP provou ser uma maneira fácil e prática de paralelização, através dos bons resultados nas métricas de desempenho de execução, assim como em esforço requerido. Infelizmente,

não foi possível tirar o máximo proveito da implementação com MPI tal como a Híbrida como referido no capítulo 4.4, todavia ambas as implementações apresentaram uma característica importante: uma aceleração estritamente crescente na amostra testada, tendo inclusivé obtido o seu máximo no maior conjunto de dados. O ambiente de teste não o permitiu, mas seria interessante testar como esta implementação se comportaria com dimensões superiores. A implementação híbrida apresentou resultados mais fracos no processamento de quadrados não-mágicos devido ao facto da implementação de comunicação durante a paralelização com OpenMP, ser deveras complexa ou em alguns casos impossível, o tempo limitado para a produção deste estudo impediu aprofundar este detalhe, todavia melhorou ligeiramente o processamento de quadrados mágicos perfeitos e imperfeitos.

Em suma, a programação paralela e distribuída é recomendada para o fim de reduzir os tempos de execução de programas que necessitem de processar grandes conjuntos de dados.

R1. Pode a computação paralela melhorar o desempenho do algoritmo num sistema multicore?
Em suma, a programação paralela e distribuída é recomendada para o fim de reduzir os tempos de execução de programas que necessitem de processar grandes conjuntos de dados, porém pode não valer a pena o esforço da implementação para pequenos conjuntos de dados.

R2. Quais as diferenças no desempenho em paralelo, entre o uso de POSIX Threads, OpenMP e MPI?
As POSIX Threads em geral obtiveram melhores tempos de execução para todas as dimensões e tipos de quadrados mágicos relativamente às outras implementações. OpenMP apresentou resultados igualmente bons e semelhantes a Pthreads, MPI e até a implementação Híbrida com MPI e OpenMP apresentaram resultados promissores para conjuntos muito grandes de dados, em sentido oposto foram as implentações que obtiveram piores resultados em conjuntos de pequenas dimensões.

R3. Quais as diferenças no esforço necessário na implementação entre as seguintes abordagens? A implementação que requereu menos esforço foi a implementação sequencial, opondo-se à implementação com POSIX Threads, que foi a implementação com maior esforço exigido.

Uma área que pode ser mais explorada é na implementação híbrida com MPI e OpenMP, os resultados apresentados não foram tão conclusivos devido às dimensões testadas, seria interessante perceber como iria variar a efiência da implementação ao processar conjunto de dados de maiores dimensões. Os valores de eficiência consideravelmente baixos apresentados na amostra requerem uma maior investigação.

6 Bibliografia

[1] *Torvalds/linux*. Retrieved from https://github.com/torvalds/linux

[2] *OpenMP*. Retrieved from https://www.openmp.org

[3] *Open-MPI*. Retrieved from https://www.open-mpi.org

[4] *Linux manual page*. Retrieved from https://man7.org/linux/man-pages/man1/time.1.html

[5] *Linux manual page*. Retrieved from https://man7.org/linux/man-pages/man2/gettimeofday.2.html

[6] Brian W. Kernighan & Dennies M.Ritchie. 1988. *The c programming language*. Pearson.

[7] Dick Buttlar & Jacqueline Farrell & Bradford Nichols. 1996. *Pthreads programming*. O'Reilly Media.

[8] MM Madeira. 2021. *Sistemas paralelos e distribuídos*. Universidade do Algarve.

[9] Nature. 2016. 530. Retrieved April 2, 2021 from https://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338

[10] Balaji Venu. 2011. Multi-core processors – an overview. Retrieved April 2, 2021 from https://arxiv.org/pdf/1110.3535.pdf

[11] Eric W Weisstein. Magic square. Retrieved April 2, 2021 from https://mathworld.wolfram.com/MagicSquare.html