Universidade do Algarve
Faculdade de Ciências e tecnologia

Sistemas Paralelos e Distribuídos

Engenharia Informática

LAB1

Ano Letivo 2020/2021

Aluno: Rúben Cruz nº64591

Docentes: Maria Margarida da Cruz Silva Andrade Madeira e Carvalho de Moura

Table of Contents

Resumo

Este trabalho foi realizado como primeiro trabalho da cadeira de Sistemas Paralelos e Distribuídos e possui como objetivo o estudo, implementação e análise envolvidos na paralelização e distribuição de um algoritmo, nomeadamente um algoritmo capaz de determinar se uma matriz quadrada é um quadrado mágico, mágico imperfeito ou não mágico.

Ainda para mais, este trabalho possui outro objetivo, a apresentação da análise do desempenho e a utilização de métricas que possibilitam a comparação das diferentes abordagem desenvolvidas de um dado algoritmo.

Neste relatório será apresentado diferentes implementações para o mesmo algoritmo anteriormente mencionado, usando diferentes bibliotecas e tecnologias, nomeadamente PThreads, OpenMP e MPI, tendo sido desenvolvido um total de cinco implementações diferentes, uma sequencial, duas paralelas, uma com PThreads e a outra com OpenMP, uma distribuída com MPI e por fim uma implementação híbrida com MPI e OpenMP.

Por fim será apresentada cada uma das implementações em detalhe tal como será realizada uma análise e comparação do desempenho entre as diferentes implementações através das diferente métricas consideradas com uma posterior discussão dos resultados tal como sugestões para melhoria futura do desempenho das implementações desenvolvidas.

Abstract

This paper was realized as the first laboratory of the curricular unit Sistemas Paralelos e Distribuidos and has as its goal the study, implementation and analysis related with the parallelization of an algorithm, namely an algorithm capable of determining if a square matrix is a magic square, a semi magic square or is not a magic square.

Furthermore, this paper possesses yet another objective, that being, the presentation of the performance analysis as well as the use of relevant performance métrics that enable the comparison of the different developed approaches of a given algorithm.

In this report will be presented different implementations for the same algorithm that was previously mentioned, using different libraries e technologies, namely PThreads, OpenMP and MPI. In total was developed five different implementations, one sequential, two parallel solutions, one with PThreads and the other with OpenMP, one distributed solution and lastly an hybrid implementation using MPI and OpenMP.

Finally, each developed implementations will be presented in detail, as well as the analysis e comparison of the performance of the different implementations using the different performance metrics considered in this paper, followed with the discussion of said results as well as some brief suggestions on how to increase the performance of the developed implementations.

1. Introdução

Este capítulo inicial de introdução apresenta brevemente o problema, tal como descreve os objetivos e motivação deste trabalho. Responsabiliza-se também pela especificação da metodologia tomada, resume os principais resultados e conclusões e por fim estabelece a organização deste documento.

1.1. Objetivo

Este trabalho, proposto pela docente Maria Margarida da Cruz Silva Andrade Madeira e Carvalho de Moura possui os seguintes objetivos:

- A realização de 5 implementações diferentes para um dado algoritmo com diferentes bibliotecas e tecnologias. Nomeadamente as implementações a realizar consistem numa sequencial, duas paralelas, umas delas com PThreads e outra com OpenMP, uma distribuída com MPI e por fim uma implementação híbrida com MPI e OpenMP;
- O estudo e análise envolvida na paralelização e distribuição de um algoritmo;
- A apresentação da análise do desempenho através de gráficos e tabelas, tal como a utilização de métricas relevantes que possibilitam a comparação das diferentes abordagens desenvolvidas para um dado algoritmo.

Para cada uma das implementações pretende-se que sejam capazes de, para uma dada matriz quadrada, sendo esta representada por uma sequência de números inteiros positivos separados por um espaço, determinar se esta é um quadrado mágico, mágico imperfeito ou não mágico. A especificação destes conceitos estão especificados no capítulo 2.

1.2. Motivação

O desenvolvimento de hardware para computação foi previsto através da famosa lei de Moore proposta inicialmente em 1965, que diz o número de transistores num circuito integrado duplica a cada dois anos, por consequente a velocidade dos computadores em série com um só core, em termos de floating-point operations per second (FLOPS), também aumenta. Mas no início do século 21 este aumento mostrou sinais de abrandamento, devido a isto a esperança de futuros computadores em série serem capazes de resolver problemas grandes demais para os computadores em séria da época estava a acabar. Devido a isto a indústria começou a mudar o foco da inovação, de mais velocidade para relógio para o desenvolvimento de sistemas paralelos, sendo estes mais rápido e capazes de resolver problemas maiores com maior rapidez devido à divisão e consequente execução em paralelo das tarefas [3].

Mas os avanços da tecnologia não se deram apenas ao nível do CPU, mas também, ao nível de redes de computadores, isto é, nos anos 80 começou o desenvolvimento de redes de computadores de alta velocidade como LANs ou WANs que ao longo dos anos foram permitindo a transferência de dados cada vezes maiores em tempos muito

reduzidos, permitindo assim a comunicação entre diferentes sistemas e consequente criação de sistemas distribuídos [2].

Na atualidade grande parte dos computadores e outros dispositivos como smartphones possuem CPUs com múltiplos cores, tal como capacidades de rede e também a grande diversidade de áreas como a científica, industrial, militar, medicina, criptografia, entre outros, cujas possuem a necessidade de um melhor aproveitamento destes avanços tecnológicos, então é facilmente verificável que a capacidade de desenvolver aplicações capazes de tomar cada vez melhor proveito dos recursos disponíveis é uma mais valia e uma aptidão essencial para o sucesso de um produto tecnológico.

1.3. Metodologia

A metodologia seguida neste trabalho consiste na execução de cada uma das implementações desenvolvidas 31 vezes registrando num ficheiro os tempos medidos através do comando time das últimas 30 execuções, sendo estes posteriormente processados e colocados em excel onde foram criados as respectivas tabelas e gráficos. Para tal foi executado o script test_all.sh, presente dentro de todas as pastas em anexo que contém as implementações realizadas. Para correr este script basta fazer o seguinte comando:

./test_all.sh <arg>

Onde <arg> representa o sufixo da pasta onde serão guardados os tempos de execução obtidos. Esta pasta criada pelo script chama-se então R<arg> e irá conter após a execução do script, os resultados das execuções de todas as matrizes guardadas na pasta MagicSquares/ em anexo, sendo os resultados guardados com o mesmo nome que os ficheiros onde as matrizes estão gravadas. O argumento <arg> foi originalmente criado para ser possível distinguir diferentes versões da mesma implementação como versões com números de threads diferentes. É acrescentado ainda que foram apenas medidos os tempos para quadrados mágicos.
Para este script funcionar a pasta MagicSquares/ necessita de possuir o mesmo caminho relativo em relação às diretorias que contêm o código das implementações desenvolvidas.
Ainda para mais existe outro script chamado test.sh presente dentro de todas as pastas em anexo que contém as implementações realizadas, este é executado dentro do test_all.sh e faz as 31 execuções para uma dada matriz, para correr este programa usa-se o seguinte comando:

./test.sh <nome da matriz> <ficheiro onde gravar>

Onde o <nome da matriz> é o argumento que será diretamente passado como argumento aos executáveis das implementações realizadas, portanto tem que seguir a estrutura necessária para estas correrem, sendo esta mencionada no capítulo 3. E

<ficheiro onde gravar> é simplesmente o nome do ficheiro onde guardar os tempos de execução das últimas 30 execuções, obtidos a partir do comando time.
Acrescenta-se que este último script foi modificado a partir do script benchmark.sh partilhado pelo aluno Guilherme Henriques nº 61018.

1.4. Principais resultados e conclusões

Neste trabalho não foi possível demonstrar o poder dos sistemas paralelos e distribuídos em termos de diminuição dos tempos de execução, pois devido à leitura de um ficheiro, nenhum dos programas superou o tempo de execução da implementação sequencial, mas foi verificado como uma implementação distribuída pode não ser a mais indicada devido ao grande tempo perdido no envio de dados ao invés de estar a processar. Para além do mais foi verificada o poder da utilização de boas e relevantes métricas para a comparação de diferentes implementações, sendo que estas permitem diferenciar por exemplo 2 algoritmos paralelos para o mesmo problema que demoram o mesmo tempo de execução, utilizando apenas o tempo de execução seria fácil concluir que são igualmente bons, mas olhando para uma métrica como a eficiência vemos que um melhor que o outro de maneira objetiva devido a usar menos cores.

1.5. Organização do relatório

Este documento encontra-se dividido em cinco capítulos organizados da seguinte forma:

- Um primeiro capítulo chamado introdução onde é feita a apresentação dos objetivos do trabalho, a motivação, metodologia, principais resultados e conclusões e por fim a organização do relatório;
- Um segundo capítulo chamado enquadramento, onde é feito o respetivo enquadramento do trabalho, isto é, a apresentação dos conceitos teóricos como as métricas consideradas para a medição do desempenho, uma apresentação breve das tecnologias utilizadas, nomeadamente MPI, OpenMP e PThreads e por fim uma apresentação mais detalhada sobre os conceitos relacionados com a definição de um quadrado mágico, mágico imperfeito ou não mágico;
- Um terceiro capítulo onde é especificado em detalhe cada uma das implementações realizadas;
- Um quarto capítulo onde é realizada a apresentação e sequente discussão dos resultados obtidos através das medições realizadas a cada uma das implementações;
- Por fim um quinto capítulo onde são expressas as conclusões tiradas a partir deste trabalho tal como a apresentação de sugestões que poderão melhorar o desempenho de cada uma das implementações.

2. Enquadramento

Neste capítulo será feita uma breve introdução teórica aos conceitos necessários para a compreensão deste relatório, como, as métricas consideradas para a medição do desempenho, uma apresentação breve das tecnologias utilizadas, nomeadamente MPI, OpenMP e PThreads e por fim uma apresentação mais detalhada sobre os conceitos relacionados com a definição de um quadrado mágico, mágico imperfeito ou não mágico, tal como o significado das saídas do comando time.

2.1. Tecnologias e bibliotecas utilizadas

Aqui será feita uma breve introdução às tecnologias utilizadas nas implementações desenvolvidas.

2.1.1. PThreads

As PThreads, abreviatura para POSIX Threads, são uma tecnologia que permite a divisão de um processo em duas ou mais tarefas, permitindo assim o seu paralelismo, isto é, permite que estas várias tarefas executam em paralelo no sistema operativo, diminuindo assim o tempo de execução.

Estas são um API (Application Program Interface), com grande parte das implementações disponíveis serem para sistemas UNIX, embora haja para windows, para a criação e manipulação de threads, especificado pelo standard IEEE POSIX 1003.1c-1995, sendo POSIX uma abreviatura para Portable Operating System Interface for Unix.

As PThreads são definidas como um conjunto de constantes, tipos e funções da linguagem de programação C, existindo uma grande variedade de funções para manipulação de threads, mutexes, semaphores, variáveis condicionais e sincronismo [6].

2.1.2. OpenMP

OpenMP é um API (Application Program Interface), conjuntamente definido pelos mais influentes vendedores de software e hardware. Este API suporta C/C++ e Fortran para uma grande variedade de arquiteturas.

OpenMP oferece um modelo portável e escalável para o desenvolvimento de aplicações multiprocessador com memória partilhada usando o modelo fork-join [7].

2.1.3 MPI

O MPI é uma especificação de bibliotecas de Message Passing Interface que serve para realizar a comunicação entre processos a executar em máquinas diferentes com memória não partilhada. Esta é definida por consenso pelo MPI Forum e prevê a ligação a C e Fortran, sendo o paralelismo explícito, isto é, o

programador é responsável pela decomposição e/ou o escalonamento das tarefas tal como a comunicação entre as várias máquinas envolvidas no processamento.

2.2. Métricas estudadas

Neste trabalho para analisar o desempenho de cada uma das diferentes implementações não é feita apenas pela comparação das médias dos tempos, mas sim será analisado os seguintes indicadores de desempenho: tempo de execução, aceleração, eficiência, rácio R/C e por fim escalabilidade.

O tempo de execução é calculado em função da dimensão do problema (N), do numero de processadores (P) envolvidos, do número de tarefas (U) e de outras características do algoritmo e do hardware utilizado. Sendo este obtido através de duas formas, a primeira sendo a maior soma dos tempos de cálculo, comunicação e espera num determinado processador e a segunda como sendo a soma de todos estes tempos em todos os elementos de processamento dividido pelo número de processadores N. O tempo de cálculo entende-se pelos periodos de tempo consumidos em cálculo, os quais dependem da dimensão do problema.

A aceleração (S) traduz o ganho obtido no desempenho de uma determinada paralelização face à implementação sequencial. Onde temos que S é dada pela seguinte razão:

$$S = \frac{t_1}{t_p} \tag{1}$$

Onde t_1 é o tempo de execução do problema num único processador, ou seja o tempo de execução da implementação sequencial, este deve ser o tempo da solução sequencial mais rápida de entre as conhecidas ou desenvolvidas. t_p é então o tempo de execução do problema pela implementação paralela com p processadores idênticos. Idealmente S será igual a p mas na prática isto não é possível, sendo normalmente S < p. Por esta razão é também considerado a eficiência (E), que indica como um algoritmo dispõe dos recursos computacionais, sendo dada pela seguinte equação:

$$E = \frac{S}{p} \tag{2}$$

Sendo este valor definido como a fração de tempo que os processadores realizam trabalho útil, caracterizando assim a forma como um algoritmo dispõe dos recursos computacionais.

Ainda para mais temos o rácio R/C, ou por extenso o rácio do tempo de execução e tempo de comunicação, consistindo o tempo de comunicação essencialmente no período de tempo que as tarefas gastam no envio e recepção de mensagens. Existem então dois tipos de comunicação, nomeadamente inter-processador e intra-processador, neste primeiro caso as tarefas que comunicam encontram-se em unidades de processamento distintas, enquanto que a segunda as tarefas encontra-se dentro da mesma unidade de processamento.
Idealmente, o tempo de comunicação inter-processador pode ser dividido em dois parâmetros principais, o tempo de arranque (t_a), sendo este o tempo necessário para iniciar a comunicação e o segundo parâmetro é o tempo de transferência de uma palavra (t_w). Assim é possível exprimir o tempo de transferência (t_{msg}) de uma mensagem através da seguinte fórmula:

$$t_{msg} = t_a + t_w L \tag{3}$$

Com L sendo o número de palavras contidas na mensagem.

Por fim temos a escalabilidade, esta traduz a capacidade do algoritmo manter a eficiência com o aumento do número de processadores (p), assim, um sistema diz-se escalar no intervalo compreendido entre p_{min} e p_{max}, sendo p_{min} o número mínimo de processadores e p_{max} o número máximo, se nesse intervalo a eficiência do sistema se manter constante.
Considerando um sistema com um só processador assume-se uma eficiência de 100%, o que significa que a utilização deste único processador é máxima, por esta razão a zona de interesse tem início imediatamente após $p = 1$ e o que se pretende conhecer com esta métrica é saber qual o número máximo de processadores para o qual o sistema se torne ineficiente de forma inaceitável devido ao acréscimo dos custos de comunicação de acordo com o algoritmo em causa.

2.3. Quadrado Mágico

Antes de mais iremos brevemente definir o problema, isto é, o que é um quadrado mágico. Um quadrado mágico é uma matriz quadrada, n por n, onde cada entrada consiste num número inteiro positivo distinto, arrumados de maneira a que a soma de todas as entradas de uma dada linha, coluna ou diagonal pricipal seja igual ao mesmo número, sendo este número a constante mágica (M_2), sendo esta dada pela seguinte fórmula [1]:

$$M_2(n, A, D) = \frac{1}{2}n[2A + D(n^2 - 1)] \tag{4}$$

Onde n é o número de linhas da matriz, A é o número inteiro inicial e D é a diferença mínima entre duas entradas do quadrado.
Para as soluções apresentadas neste relatório foi apenas considerado matrizes onde $A = 1$ e $D = 1$, isto é, matrizes cujas entradas consistem nos números $1, 2, \ldots, n^2$, assim substituindo na expressão (4) os valores de A e D, obtém-se [1]:

$$M_2(n) = \frac{1}{2}n(n^2 + 1) \quad (5)$$

Quanto a um quadrado mágico imperfeito, ou quadrado semi mágico, este é um quadrado onde apenas as somas das linhas e das colunas correspondem à constante mágica, por outras palavras as somas de uma ou ambas as somas das diagonais principais não correspondem à constante mágica [1].

Por fim, um quadrado não mágico consiste num quadrado onde pelo menos a soma de uma linha ou coluna não corresponde à constante mágica.

2.4. Comandos utilizados

Neste trabalho para a obtenção dos tempos de execução foi utilizado o comando time do linux, sendo este estruturado da seguinte maneira:

time <opções> comando <argumentos>

Onde <opções> são as opções do comando time, cujas não foram usadas neste trabalho, comando é o executável a medir o tempo, neste caso é uma das cinco implementações compiladas, por fim <argumentos> são os argumentos de input do comando.
Este comando retorna então três tempos distintos, real time, user time e por fim sys time.
O real time é simplesmente o tempo de execução do programa mais o tempo que este processo passa bloqueado e ainda mais o tempo de outros processos que possam estar a executar ao mesmo tempo, efetivamente é o tempo percecionado por um humano enquanto espera pela resposta do programa [4].
O user time traduz o tempo que o CPU gasta a executar no espaço do utilizador o processo derivado do programa, este não inclui então o tempo em que o processo se encontra bloqueado nem o tempo de outros processos mas este inclui o total de tempo de execução em todos os CPUs, assim se duas tarefas do mesmo processo estão a executar ao mesmo tempo em CPUs diferentes ambos os tempos serão somados ao user time, devido a isto é possível que este seja maior que o real time [5].
Por fim o sys time é simplesmente o tempo que o CPU gasta a executar no espaço do kernel o processo derivado do programa.

3. Estudos de Casos

Antes de mais verifica-se que para todas as implementações, o código tal como o executável encontra-se numa pasta dedicada a cada implementação, e todas lêem a entrada de um dado ficheiro com extensão .txt que se encontra numa pasta independente das que estão dedicadas às implementações chamada MagicSquares/, presente em anexo, e o nome dos ficheiros seguem a seguinte estrutura: <t><n>.txt, onde <t> é o tipo de quadrado, isto é <t> é igual p se o ficheiro conter um quadrado mágico perfeito, i se conter quadrado mágico imperfeito e finalmente n se conter um quadrado não mágico. Quanto a <n> este é simplesmente o número de linhas da matriz guardada pelo ficheiro. Por exemplo, p5000.txt contém um quadrado mágico perfeito com dimensão 5000 x 5000.

3.1. Implementação Sequencial

Procede-se agora a explicitar cada uma das implementações realizadas começando pela implementação sequencial. Para esta, recebe-se inicialmente o nome da matriz de acordo com a formatação do nome dos ficheiros onde as entradas são gravadas, mas sem a extensão. Seguidamente extrai-se o número de linhas da matriz a partir do nome dado, isto é o valor dado em <n>, seguidamente são criados dois arrays, um com dimensão dois e outro com dimensão igual ao número de linhas da matriz, onde o primeiro irá ser usado para guardar as somas das diagonais e o segundo serve para guardar as somas das colunas. Posteriormente calcula-se a constante mágica a partir da expressão (5) e inicia-se a leitura do ficheiro presente na pasta MagicSquares/ correspondente à entrada.

Esta leitura realiza-se entrada a entrada da matriz contando qual a posição atual na matriz, sendo i o número da linha atualmente a ser lida e j a coluna. Cada vez que é lido uma nova entrada da matriz é somada ao atual valor da soma da linha atual, simultaneamente é somada ao atual valor da soma da coluna correspondente, que se encontra guardada no segundo array criado no índice igual a j, também é verificado se a entrada atual faz parte de uma das diagonais principais, caso esse for o caso soma-se a entrada ao elemento correspondente à diagonal em questão do primeiro array criado.
Ainda para mais cada vez que se chega ao final da linha verifica-se se o valor da soma da linha é igual a constante mágica, se este for o caso o valor da soma da linha é anulado e procede-se a ler a próxima linha, se isto não se verificar o programa termina e retorna que o quadrado não é mágico. Após ler todo o ficheiro é verificado se todos os elementos nos dois arrays são iguais à constante mágica, se este for o caso o programa termina e retorna que o quadrado é mágico, se um dos elementos do array das diagonais não for igual à constante mágica e todos os elementos do array das colunas forem iguais à constante mágica o programa termina e retorna que o quadrada é imperfeito e por fim se qualquer um dos elementos do array das colunas não forem iguais à constante mágica o programa termina e retorna que o quadrada não é mágico.

O código desenvolvido para esta implementação encontra-se em anexo na diretoria SequentialApproachV4/, este contém os seguintes ficheiros: main.c, MagicSquareValidation.h, test.sh e test_all.sh.
Para compilar basta correr o seguinte comando:

gcc main.c

Para correr o programa basta fazer o seguinte comando:

./a.out <nome da matriz>

Onde o <nome da matriz> segue a mesma predefinição sobre os nomes dos ficheiros das matrizes presentes em MagicSquares/ mas sem a extensão, assim por exemplo para executar esta implementação para um quadrado imperfeito com dimensão 6750x6750, basta primeiro guardar este quadrado em MagicSquares/i6750.txt e seguidamente na pasta SequentialApproachV4/ executar:

./a.out i6750

Acrescenta-se ainda que a pasta MagicSquares/ deve manter o mesmo caminho relativo em relação a SequentialApproachV4/ como o apresentado na diretoria de anexos para o programa funcionar.

Ou simplesmente pode ser executado os scripts test.sh ou test_all.sh conforme explicado no capítulo 1.3.

3.2. Implementação Paralela com PThreads

Explicitando agora como foi realizada a implementação paralela com pthreads, essencialmente consiste na mesma abordagem que a anterior mas a verificação dos dois arrays criados, o das colunas e o das diagonais, é dividida por vários threads, sendo que um thread verifica apenas o array das diagonais e os restantes verifica um segmento do array das colunas.

O código desenvolvido para esta implementação encontra-se em anexo na diretoria PThreadsApproachV2/, este contém os seguintes ficheiros: C4_main.c, C4_MagicSquareValidation.h, C5_main.c, C5_MagicSquareValidation.h, C6_main.c, C6_MagicSquareValidation.h, C7_main.c, C7_MagicSquareValidation.h, C8_main.c, C8_MagicSquareValidation.h, test.sh e test_all.sh.

O prefixo C<n> indica o número de pthreads, por exemplo C4_main.c usa 4 pthreads para realizar as suas operações.

Para compilar a implementação com o número de pthreads, n, desejados a testar basta correr o seguinte comando:

gcc C<n>_main.c -pthread

Para correr o programa basta fazer o seguinte comando:

./a.out <nome da matriz>

Onde o <nome da matriz> segue a mesma predefinição sobre os nomes dos ficheiros das matrizes presentes em MagicSquares/ mas sem a extensão, assim por exemplo para executar esta implementação para um quadrado imperfeito com dimensão 6750x6750, basta primeiro guardar este quadrado em MagicSquares/i6750.txt e seguidamente na pasta PThreadsApproachV2/ executar:

./a.out i6750

Acrescenta-se ainda que a pasta MagicSquares/ deve manter o mesmo caminho relativo em relação a PThreadsApproachV2/ como o apresentado na diretoria de anexos para o programa funcionar.

Ou simplesmente pode ser executado os scripts test.sh ou test_all.sh conforme explicado no capítulo 1.3.

Para testar uma outra implementação com um número diferente de pthreads é necessário voltar a compilar C<n>_main.c pretendido devido aos scripts test.sh e test_all.sh apenas executarem o executável a.out.

3.3. Implementação Paralela com OpenMP

Para a implementação paralela com OpenMP, esta consiste em pelo menos quatro threads, um para leitura, outro para calcular as somas das diagonais, entre um a cinco dedicados a calcular as somas das linhas e entre um até ao número de colunas do quadrado dedicados a calcular as somas das colunas. Novamente numa fase inicial é calculada a constante mágica através da expressão (5) e ambos os threads das diagonais e colunas possuem o seu próprio array onde guardam o total atual das somas da diagonal ou conjunto de colunas pelas quais são responsáveis.

Inicialmente é recebido o nome da matriz de acordo com a formatação do nome dos ficheiros onde as entradas são gravadas, mas sem a extensão. Seguidamente extrai-se o número de linhas da matriz a partir do nome dado, isto é o valor dado em <n>, seguidamente calcula-se a constante mágica a partir da expressão (5) e inicia-se a leitura do ficheiro presente na pasta MagicSquares/ correspondente à entrada da seguinte forma.

Este algoritmo divide o ficheiro que contém o quadrado em segmentos de normalmente cinco linhas e no máximo nove linhas para o último dos segmentos lidos. Existe então

um segmento global acessível por todas as tarefas e cada uma possui uma cópia local deste segmento. Posto isto, a função da tarefa de leitura é ler para o segmento local um segmento do ficheiro, depois copia o segmento local para o segmento global e desbloqueia as outras tarefas, estas começam por copiar o segmento global para o local e começam a processar o segmento local. Após todas as outras tarefas acabarem de copiar o segmento global para o local a tarefa de leitura é desbloqueado, esta enquanto espera pelas outras acabarem de copiar encontra-se a ler o próximo segmento e apenas quando se encontra desbloqueada é que procede a atualizar o global e desbloquear novamente as outras tarefas. Este ciclo repete-se até todo o ficheiro ser lido ou até alguma das somas não igualar a constante mágica.

Para o caso das tarefas dedicadas ao processamento das linhas estas dividem entre si as 5 linhas até as 9 linhas e cada uma soma os elementos das linhas a elas atribuídas, apesar de todas conterem a cópia do segmento global na sua totalidade em vez de apenas as linhas que lhes interessa, e se o resultado for igual a constante mágica, nada acontece e estas continuam a executar, senão a tarefa que detetou a diferença atualiza a variável global do resultado para o valor correspondente ao facto de uma linha ser diferente da constante mágica.

Para as tarefas dedicadas a processar colunas estas após copiarem todo o segmento global dividem entre si as colunas do segmento e adicionam o resultado da soma das cinco a nove entradas de cada coluna atribuída à entrada correspondente do array local que cada uma possui tendo este sido criado assim que cada tarefa dedicada a processar colunas são iniciadas e responsabilizado por guardar as somas das colunas atribuídas à tarefa. Quando é acabado de processar o último segmento cada uma das tarefas compara o resultado das somas das colunas atribuídas guardadas no array local anteriormente mencionado e se pelo menos uma das entradas for diferente da constante mágica então a tarefa que detetou a diferença atualiza a variável global do resultado para o valor correspondente ao facto de uma coluna ser diferente da constante mágica.

Para a tarefa responsabilizada pelo processamento das diagonais, esta apenas possui um array local de dimensão dois que guarda o resultado das somas das diagonais, para isto esta copia todo o segmento e adiciona as entradas correspondentes às diagonais à entrada correspondente do array. Após o término do processamento do último segmento esta tarefa compara o resultado das somas das diagonais guardadas no array local anteriormente mencionado e se pelo menos uma das entradas for diferente da constante mágica esta acede à variável global do resultado e verifica se o seu valor é diferente do valor correspondente ao facto de uma coluna ou linha ser diferente da constante mágica, se esse for o caso esta atualiza o valor do resultado de forma a refletir que uma diagonal é diferente da constante mágica, caso contrário não faz nada.

Ainda para mais caso a variável global para o resultado ser definida como o valor correspondente ao facto de uma coluna ou linha ser diferente da constante mágica

todas as tarefas param de executar mais cedo, devido a todas verificarem o valor desta variável em diversos momentos de execução como o caso da tarefa de leitura que verifica este valor após desbloquear as outras tarefas, ou as restantes tarefas que verificam este valor antes de copiarem o segmento global e depois de processar o segmento. E cada acesso a variável global para o resultado é realizado com acesso a uma região crítica, onde apenas uma tarefa pode aceder de cada vez.

A ideia deste algoritmo é então, tentar realizar processamento ao mesmo tempo que a leitura do ficheiro está a ser realizada, visto a leitura do ficheiro representa em maioria o tempo de execução das implementações anteriores.

O código desenvolvido para esta implementação encontra-se em anexo na diretoria OpenMPApproachV3/, este contém os seguintes ficheiros: main.c, MagicSquareValidation.h, test.sh e test_all.sh.
Para compilar basta correr o seguinte comando:

```
gcc main.c -fopenmp
```

Para correr o programa basta fazer o seguinte comando:

```
./a.out <nome da matriz>
```

Onde o <nome da matriz> segue a mesma predefinição sobre os nomes dos ficheiros das matrizes presentes em MagicSquares/ mas sem a extensão, assim por exemplo para executar esta implementação para um quadrado imperfeito com dimensão 6750x6750, basta primeiro guardar este quadrado em MagicSquares/i6750.txt e seguidamente na pasta OpenMPApproachV3/ executar:

```
./a.out i6750
```

Acrescenta-se ainda que a pasta MagicSquares/ deve manter o mesmo caminho relativo em relação a OpenMPApproachV3/ como o apresentado na diretoria de anexos para o programa funcionar.

Ou simplesmente pode ser executado os scripts test.sh ou test_all.sh conforme explicado no capítulo 1.3.

Para mudar o número de threads usados pelo programa basta fazer o seguinte comando antes de executá-lo:

```
export OMP_NUM_THREADS=n
```

Onde n é o número de threads pretendido, sendo que este valor não pode ser menor que quatro para o programa funcionar.

3.4. Implementação Distribuída com MPI

Para a implementação distribuída com MPI temos que existe 3 tipos de especialização por parte das tarefas, isto é, temos uma que está dedicado a ler o ficheiro, outro dedicado a processar os segmentos intermédios da matriz e por fim outro especializado a processar o último segmento da matriz e verificar se as somas das colunas e diagonais são iguais a constante mágica. Um processo desta implementação só possui uma tarefa especializada na leitura, uma especializada no processamento o último segmento da matriz e as restantes estão especializadas no processamento dos segmentos intermédios da matriz.

Para começar todas a tarefas começam por calcular quantas linhas tem cada segmento ($N_{segmento}$) da matriz guardada no ficheiro, equivalente ao resultado da divisão inteira entre o número total de linhas (N_{linhas}) e o número de máquinas (P) a correr este menos um mais o resto no caso do último segmento a ler, ou seja:

$$N_{segmento} = \frac{N_{linhas}}{P}$$

Ou para o último segmento:

$$N_{ultimo\ segmento} = N_{segmento} + (resto\ de\ N_{segmento})$$

Seguidamente as tarefas especializadas no processamento dos segmentos intermédios e a especializada no processamento do último segmento calculam a constante mágica através da expressão (5) e ficam a espera do seu segmento, ao mesmo tempo a tarefa de leitura começa a ler as primeiras $N_{segmento}$ linhas do ficheiro, após terminar esta envia para a primeira tarefa especializadas no processamento dos segmentos intermédios (cuja pode ser ser substituída pela tarefa especializada no processamento do último segmento), continuando a ler o ficheiro. Após ler mais $N_{segmento}$ linhas do ficheiro, envia o segmento para a segunda tarefa especializadas no processamento dos segmentos intermédios, continuando a ler mais $N_{segmento}$ linhas do ficheiro e enviando este segmento para a terceira tarefa especializadas no processamento dos segmentos intermédios e assim por diante até chegar ao último segmento cujo será enviado à tarefa especializada no processamento do último segmento. Posteriormente este fica à espera de receber o resultado vindo da tarefa especializada no processamento do último segmento.

As tarefas especializadas no processamento dos segmentos intermédios após receberem o seu segmento correspondente procedem a processar o segmento, isto é, a

calcular a soma das linhas, verificando se são iguais à constante mágica atualizando a variável de resultado para o valor correspondente ao facto de uma linha ser diferente da constante mágica, a soma das colunas com auxílio a um array semelhante à implementação sequencial, e a soma das diagonais novamente com auxílio de um array. Após o processamento caso a tarefa não for a tarefa que recebeu o primeiro segmento, fica a espera de receber os arrays das somas das colunas e das diagonais calculado pela tarefa que processou o segmento anterior tal como o valor da variável de resultado e após receber estes valores, verifica se a variável de resultado difere do valor correspondente ao facto de uma linha ou coluna ser diferente da constante mágica, se este for o caso, a tarefa procede a somar as entradas dos arrays das somas das colunas e das diagonais com as entradas de mesmo índice dos arrays das somas das colunas e das diagonais calculados durante o processamento e grava-se o resultado nos mesmos, se a variável de resultado não difere do valor correspondente ao facto de uma linha ou coluna ser diferente da constante mágica, então este passo é ignorado. Finalmente a tarefa envia os arrays das somas das colunas e diagonais, tal como o valor atual do resultado, para a tarefa que recebeu o segmento a seguir.

Para a tarefa especializada no processamento do último segmento, esta realiza as mesmas operações o outro tipo de tarefa especializada no processamento, mas em vez de enviar para a tarefa que recebeu o segmento a seguir o valor atual do resultado e os arrays das somas das colunas e diagonais, esta tarefa procede a verificar se algum valor da arrays das somas das colunas e diagonais difere da constante mágica, atualizando o valor do resultado conforme a verificação, isto é caso pelo menos uma coluna seja diferente é atribuído ao resultado o valor correspondente ao facto de uma coluna ser diferente da constante mágica, ou se todas as colunas forem iguais mas uma diagonal for diferente da constante mágica é atribuído ao resultado o valor correspondente ao facto de uma diagonal ser diferente da constante mágica. Esta verificação só é realizada caso o valor do resultado proveniente da tarefa que processou o segmento anterior, caso exista, seja diferente do valor correspondente ao facto de uma coluna ou linha ser diferente da constante mágica. Após a verificação esta tarefa envia o resultado para a tarefa de leitura e seguidamente esta retorna o resultado.

Novamente a intenção com que este algoritmo foi desenvolvido, foi para existir processamento ao mesmo tempo da leitura visto que tal como já foi mencionado anteriormente esta leitura compõe a grande maioria do tempo de execução das implementações anteriores, tal como poderemos ver no próximo capítulo.

O código desenvolvido para esta implementação encontra-se em anexo na diretoria MPIApproach/, este contém os seguintes ficheiros: main.c, test.sh e test_all.sh.
Para compilar basta correr o seguinte comando:

```
mpicc main.c
```

Para correr o programa localmente basta fazer o seguinte comando:

mpirun -np <p> ./a.out <nome da matriz>

Onde o <nome da matriz> segue a mesma predefinição sobre os nomes dos ficheiros das matrizes presentes em MagicSquares/ mas sem a extensão e <p> representa o número de workers locais, assim por exemplo para executar esta implementação para um quadrado imperfeito com dimensão 6750x6750 com 4 workers, basta primeiro guardar este quadrado em MagicSquares/i6750.txt e seguidamente na pasta MPIApproach/ executar:

mpirun -np 4 ./a.out i6750

Acrescenta-se ainda que a pasta MagicSquares/ deve manter o mesmo caminho relativo em relação a MPIApproach/ como o apresentado na diretoria de anexos para o programa funcionar.

Ou simplesmente pode ser executado os scripts test.sh ou test_all.sh conforme explicado no capítulo 1.3.

3.5. Implementação Híbrida com OpenMP e MPI

Por fim a última implementação desenvolvida, esta segue de forma geral o mesmo algoritmo da implementação anterior, exceto no processamento do segmento, na soma dos arrays das somas das colunas e das diagonais com os mesmos arrays provindos da tarefa que processou o segmento anterior e na verificação das somas das colunas através da comparação das entradas do array das somas com a constante mágica feita pela tarefa que processa o último segmento. A estes três aspetos do algoritmo anterior foi introduzido paralelismo através de OpenMP.

No processamento do segmento apenas um thread calcula a soma das diagonais e coloca os resultados no respetivo índice do array das somas das diagonais, sendo este um array global, depois cada linha do segmento a processar é dividida de forma a cada thread ficar com um certo conjunto de colunas, após esta divisão cada thread soma ao array das somas das colunas, que tal como o das diagonais também é partilhado pelos threads, a entrada do segmento a ser lida à entrada correspondente do array, tal como adiciona a mesma entrada à variável local responsável por guardar a soma do conjunto de colunas atribuídas ao thread. Depois desta ação ser realizada para cada entrada identificada pela linha em questão e pelas colunas atribuídas ao thread, cada thread soma ao verdadeiro valor da soma da linha com uma região crítica do OpenMP e fica à espera que os outros acabem. Depois quando todos acabam de processar as entradas a eles atribuídos um dos threads atualiza o valor do resultado caso o valor da soma da linha ser diferente da constante mágica e volta a atribuir o valor de zero a soma da linha, enquanto isto acontece os restantes threads voltam a esperar que este acabe de atualizar o resultado e a reiniciar o valor da soma da linha. Posteriormente todos os

threads avançam para a seguinte linha, repetindo o processo até chegar ao fim do segmento a processar.

Na soma dos arrays das somas das colunas e das diagonais com o mesmo arrays provindos da tarefa que processou o segmento anterior, é feita uma paralelização simples, isto é, cada thread é responsável por somar um subconjunto de entradas do array das somas das colunas com as entradas de mesmo índice do array das somas das colunas provindo da tarefa que processou o segmento anterior. Quanto à soma dos dois arrays de diagonais, esta continua a ser feita sequencialmente.

Por fim a verificação das somas das colunas, é atribuída um subconjunto de entradas do array das somas das colunas a cada thread, e cada um verifica se existe alguma entrada que seja diferente da constante mágica, se este for o caso, com uma região crítica é atualizado o valor do resultado para o valor correspondente, caso contrário nada acontece.

O código desenvolvido para esta implementação encontra-se em anexo na diretoria HybridApproach/, este contém os seguintes ficheiros: main.c, test.sh e test_all.sh.
Para compilar basta correr o seguinte comando:

```
mpicc main.c -fopenmp
```

Para correr o programa localmente basta fazer o seguinte comando:

```
mpirun -np <p> ./a.out <nome da matriz>
```

Onde o <nome da matriz> segue a mesma predefinição sobre os nomes dos ficheiros das matrizes presentes em MagicSquares/ mas sem a extensão e <p> representa o número de workers locais, assim por exemplo para executar esta implementação para um quadrado imperfeito com dimensão 6750x6750 usando 4 workers, basta primeiro guardar este quadrado em MagicSquares/i6750.txt e seguidamente na pasta HybridApproach/ executar:

```
mpirun -np 4 ./a.out i6750
```

Acrescenta-se ainda que a pasta MagicSquares/ deve manter o mesmo caminho relativo em relação a HybridApproach/ como o apresentado na diretoria de anexos para o programa funcionar.

Ou simplesmente pode ser executado os scripts test.sh ou test_all.sh conforme explicado no capítulo 1.3.

Para mudar o número de threads usados pelo programa basta fazer o seguinte comando antes de executá-lo:

```
export OMP_NUM_THREADS=n
```

Onde n é o número de threads pretendido, sendo que este valor não pode ser menor que quatro para o programa funcionar.

4. Análise de resultados e discussão

Neste capítulo será apresentado os valores obtidos referentes a cada uma das implementações, através de tabelas e gráficos, tal como será feita comparações entres as diversas abordagens, tal como para versões da mesma com diferentes números de threads.
Aqui todos os valores de tempo foram obtidos com recurso ao script test_all.sh tal como foi mencionado no capítulo 1.3, sendo todos corridos na Máquina 1 e sem qualquer otimização por parte do compilador, cuja descrição está especificada no próximo subcapítulo. Nas implementações com MPI os tempos foram medidos na Máquina 1, sendo esta o Master, ou aquele que está dedicado a ler o ficheiro, e sendo usado a Máquina 2 como o outro dispositivo de execução. Foram consideradas um total de 11 quadrados perfeitos para as medições dos tempos da implementação sequencial e as duas paralelas, tendo estes as seguintes medições, 1000x1000, 1501x1501, 2000x2000, 3000x3000, 4095x4095, 5000x5000, 7501x7501, 8191x8191, 10001x10001, 15001x15001 e por fim 20001x20001, sendo estes colocados na pasta MagicSquares/. Para a implementação com MPI e Híbrida foi considerada apenas quadrados perfeitos com as seguintes dimensões: 101x101, 1001x1001, 1023x1023, 2047x2047, 4095x4095, 8191x8191, 10001x10001 e por fim 15001x15001.

Os resultados obtidos encontram-se dentro das pastas onde se encontra cada uma das implementações dentro de pastas R<n>/, onde <n> é o número de threads da versão testada. Dentro destas pastas encontram-se os registos dos tempos medidos pelo script test_all.sh juntamente com o ficheiro Excel Results. Por fim junto com o source das implementações está também o ficheiro Excel chamado Results Compiled onde os resultados presentes nos ficheiros Excel Results estão compilados, sendo estes ficheiros Results Compiled os que originaram os gráficos apresentados. Algumas exceções são a pasta R2-2 no ficheiro HybridApproach em vez de R<n>, e na pasta SequencialApproachV4 esta não possui o ficheiro Results Compiled mas possui a pasta R1-MaquinaUALG, que possui os tempos medidos para o algoritmo sequencial na máquina 2.

4.1. Descrição das máquinas

Máquina 1 (Computador pessoal):

- CPU: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
- Frequência do CPU: 3.00GHz
- Número de cores: 4
- Cache 256 KB
- RAM: 8297012 KB DDR4
- Velocidade de RAM: 2400 MHz
- Versão de gcc: 8.3.0
- Sistema Operativo: Windows
- Notas: os testes realizados sobre esta máquina foram realizados em WSL

Máquina 2 (fct-deei-linux.ualg.pt)

- CPU: Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz
- Frequência do CPU: 1995.312 MHz

- Número de cores: 4
- Cache 28160 KB
- RAM: 8124784 KB
- Versão de gcc: 8.3.0
- Sistema Operativo: Debian

4.2. Implementação Sequencial

Assim conforme o mencionado na introdução do capítulo 4, e considerando como tempo de execução a soma entre os tempos user e sys retornados pelo comando time, foi obtido o seguinte gráfico:

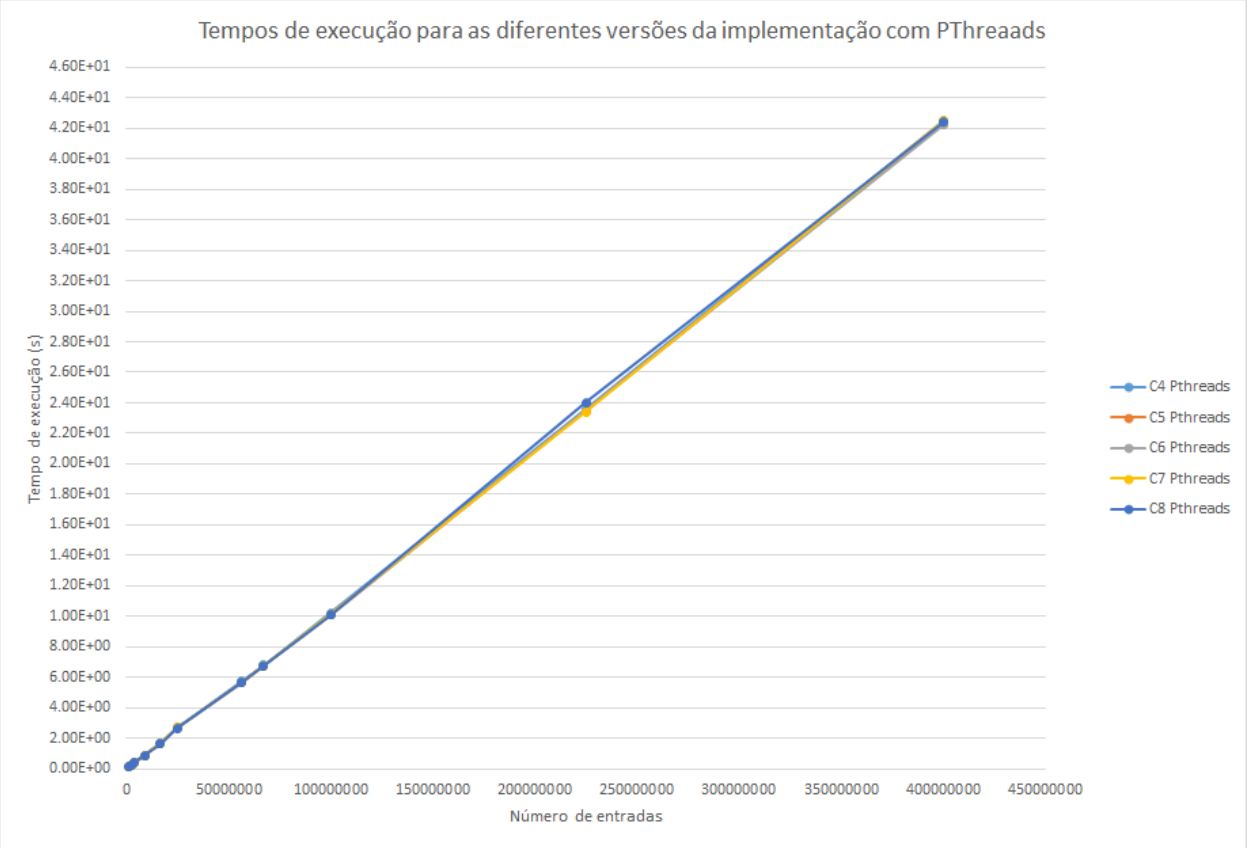
Onde o número de entradas corresponde ao quadrado do número de linhas da matriz em questão, ou seja, ao total do número de entradas da mesma. O tempo de execução é portanto a soma dos tempos user e sys do comando time representados em segundos.

Como podemos ver, esta implementação descreve um comportamento linear conforme o crescimento do número de entradas da matriz. Como podemos ver o tempo de execução para as cinco matrizes de menor dimensão é bastante reduzido, sendo este menor que quatro segundos, sendo que a menor matriz ronda os 0.09 segundos, foi devido a este tempo mínimo que foi estabelecido 1000x1000 como o menor teste onde o tempo seria considerado. Já para a matriz de maior dimensão, 20001x20001, temos que o tempo de execução desta é cerca de 41 segundos.

4.3. Implementação Paralela com PThreads

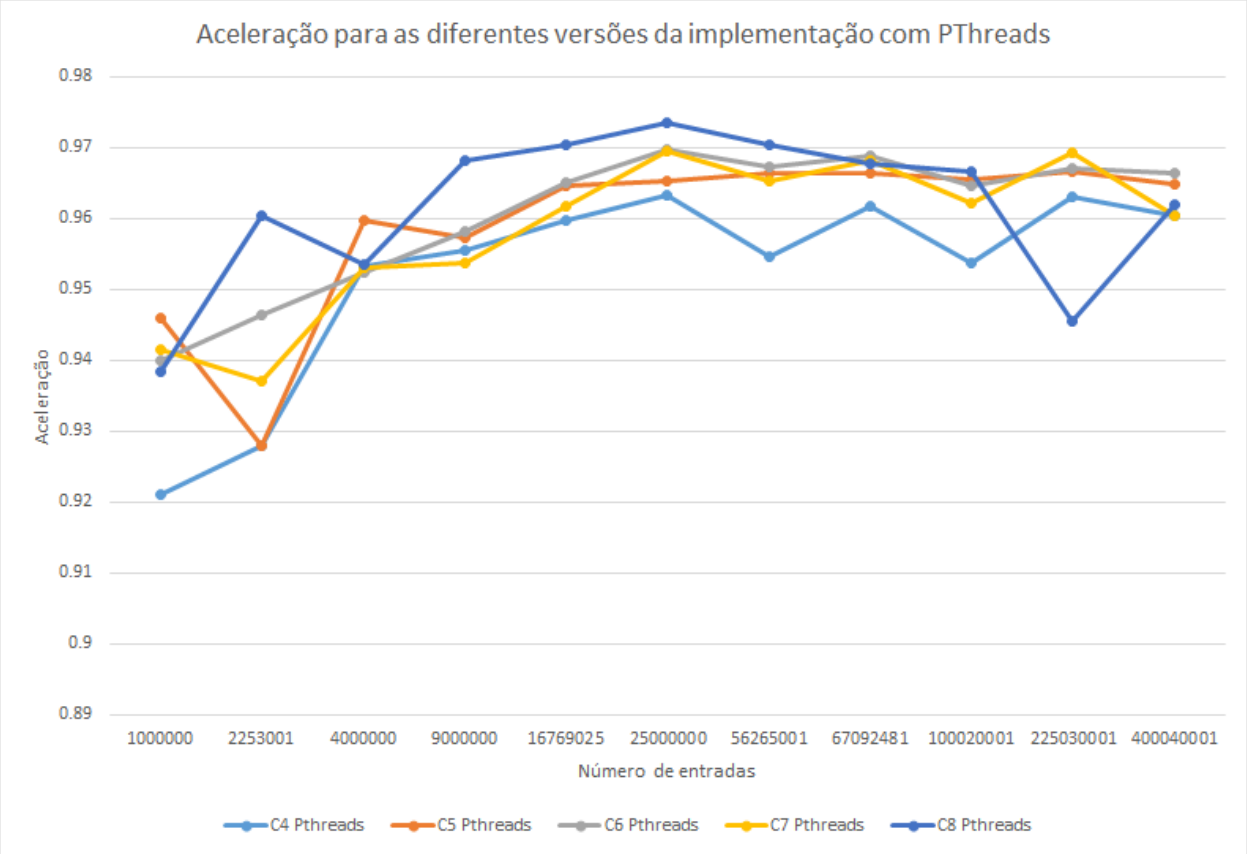
Para esta implementação já não foi considerado como tempo de execução a soma dos tempos user e sys do comando time, mas sim o tempo real, isto devido ao user e sys

representarem o tempo do processo em CPU, o que como esta implementação tem paralelismo significa que vai contar o dobro do tempo quando dois threads executam ao mesmo tempo, tal como tinha sido mencionado no capítulo 2.4.
Para além do mais foram testadas um total de cinco versões desta mesma implementação, cada uma com o seu próprio número de threads, começando com quatro e acabando com oito.
A partir disto foi construído o seguinte gráfico de tempos de execução:

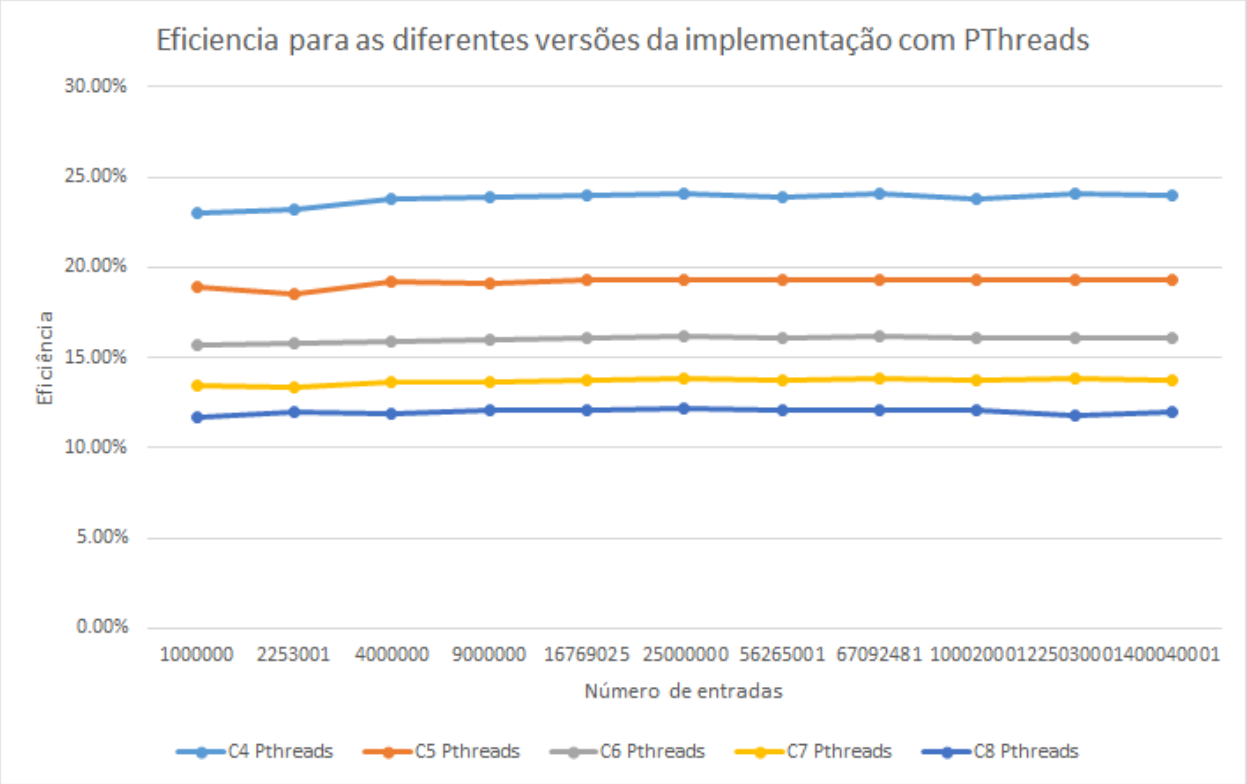


Antes de mais nota-se que as várias versões que diferem apenas no número de threads que estão a usar, estão identificados por C<n> PThreads, onde <n> é o número de threads desta versão, este sistema de nomenclatura está também presente nos próximos gráficos apresentados neste subcapítulo.
Como podemos ver os tempos são em muito semelhantes e novamente são lineares consoante o tamanho da entrada, como seria de esperar, visto a leitura do ficheiro não ser de nenhum modo dividida pelos threads, sendo esta leitura a grande maioria do tempo de execução, então seria que este tempo de leitura estaria inalterado em relação à implementação sequencial desenvolvida.
Considerando agora a aceleração calculada a partir da expressão (1), usando os melhores tempos conhecidos para a implementação sequencial, que neste caso são apenas os da implementação descrita em 3.1 e cujos resultados se encontram em 4.2. Visto os tempos de execução serem semelhantes será de esperar que a aceleração

também seja semelhante em relação às diferentes versões. Tal pode ser verificado olhando para o seguinte gráfico:



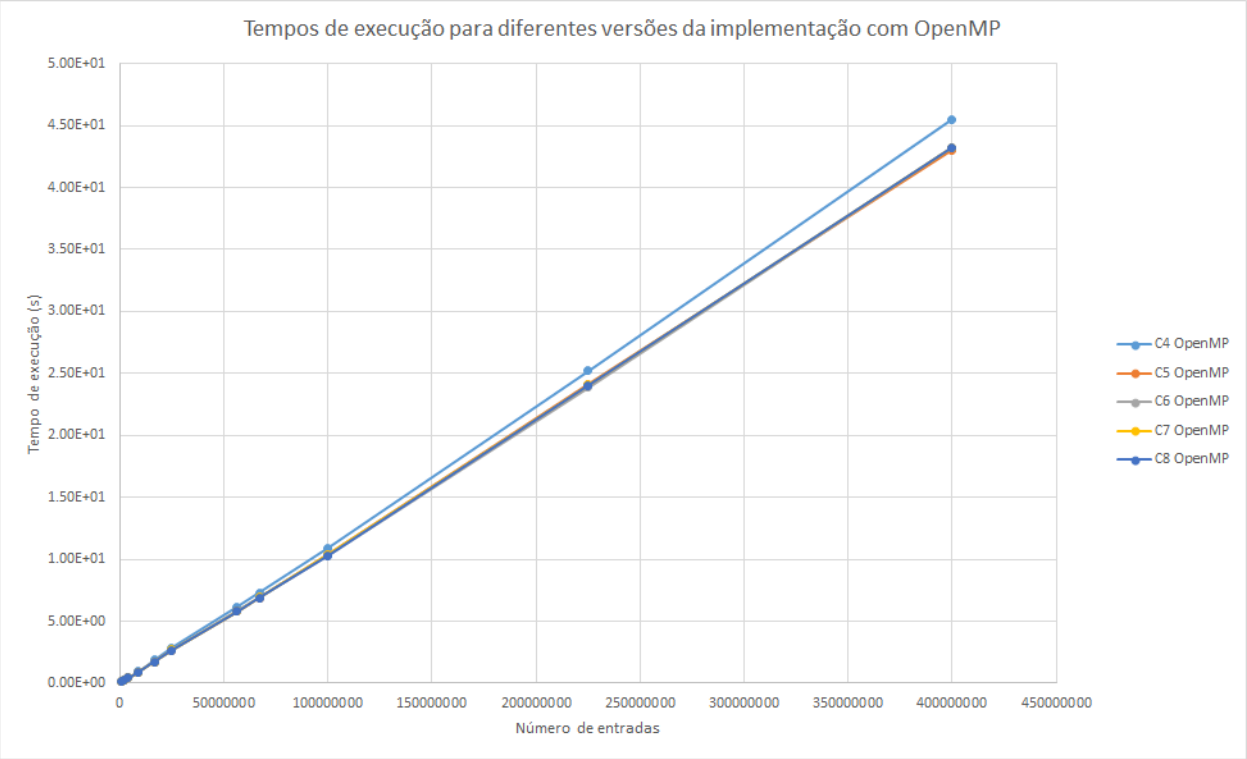
Como podemos ver todas a versões possuem acelerações semelhantes, rondando os 0.92 e os 0.985, mas, ao contrário do tempo de execução, esta métrica permite verificar qual o algoritmo que oferece uma maior diminuição do tempo de execução, assim podemos ver que em geral a versão com oito threads é mais rápida exceto para as últimas duas matrizes. Apesar desta oferecer o maior aumento de aceleração vemos que todas não só possuem os valores da aceleração muito longe da aceleração ideal, isto é igual ao número de threads, como também nenhuma das versões oferece um tempo tempo de execução menor que a implementação sequencial, isto poderá ter sido causado pelo facto de a leitura de maneira nenhuma foi modificada em relação à sequencial, o que conjuntamente com o tempo de criação dos pthreads e consequente gestão por parte do sistema operativo, levando a uma grande ineficiência por parte do algoritmo, como podemos ver no seguinte gráfico:



Como podemos ver as versões são consistentes em termos da eficiência, isto é, a eficiência é constante para qualquer um dos testes efetuados, mas é cada vez menor quantos mais threads serem utilizados, pelo que este algoritmo mostra-se não escalar. Assim verifica-se que esta implementação não toma bom proveito da tecnologia PThreads.

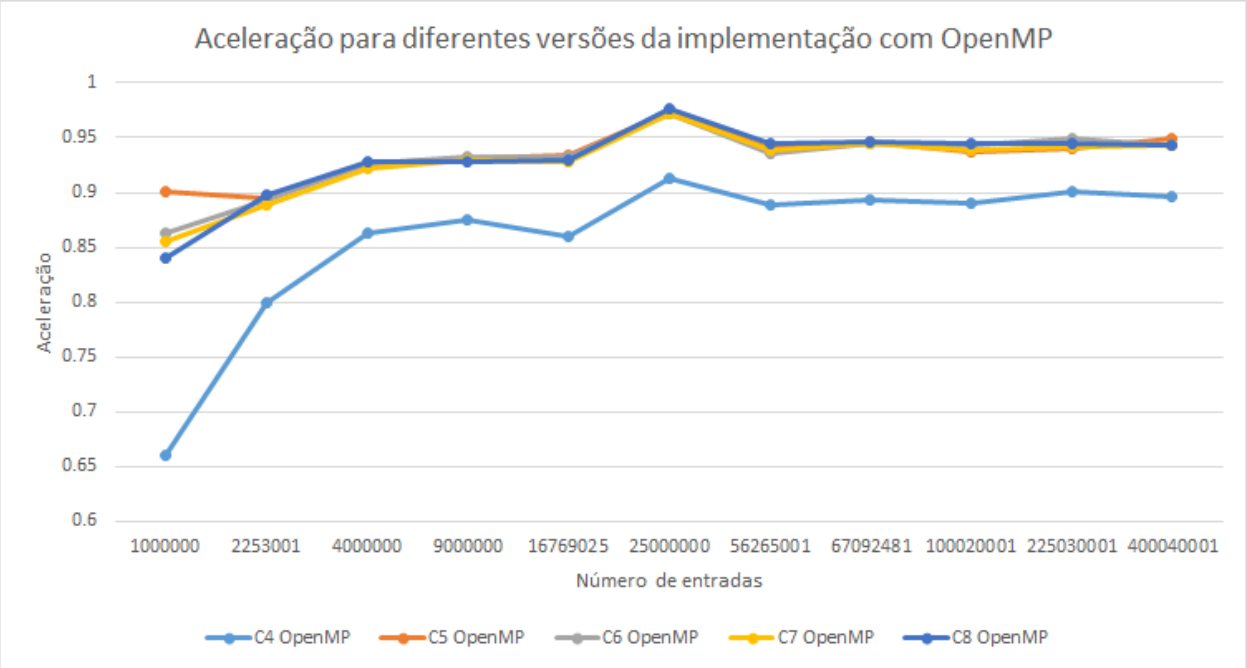
4.4. Implementação Paralela com OpenMP

Agora iremos olhar para a implementação paralela com OpenMP, tal como na implementação com pthreads o tempo de execução considerado foi o tempo real do comando time, pelas mesmas razões mencionadas em 4.3. Para além do mais também foi feitos testes com versões do mesmo algoritmo que diferem apenas no número de threads, sendo o mínimo de threads testados apenas quatro e até oito threads, posto isto é apresentado seguidamente o gráfico formado com os tempos de execução:



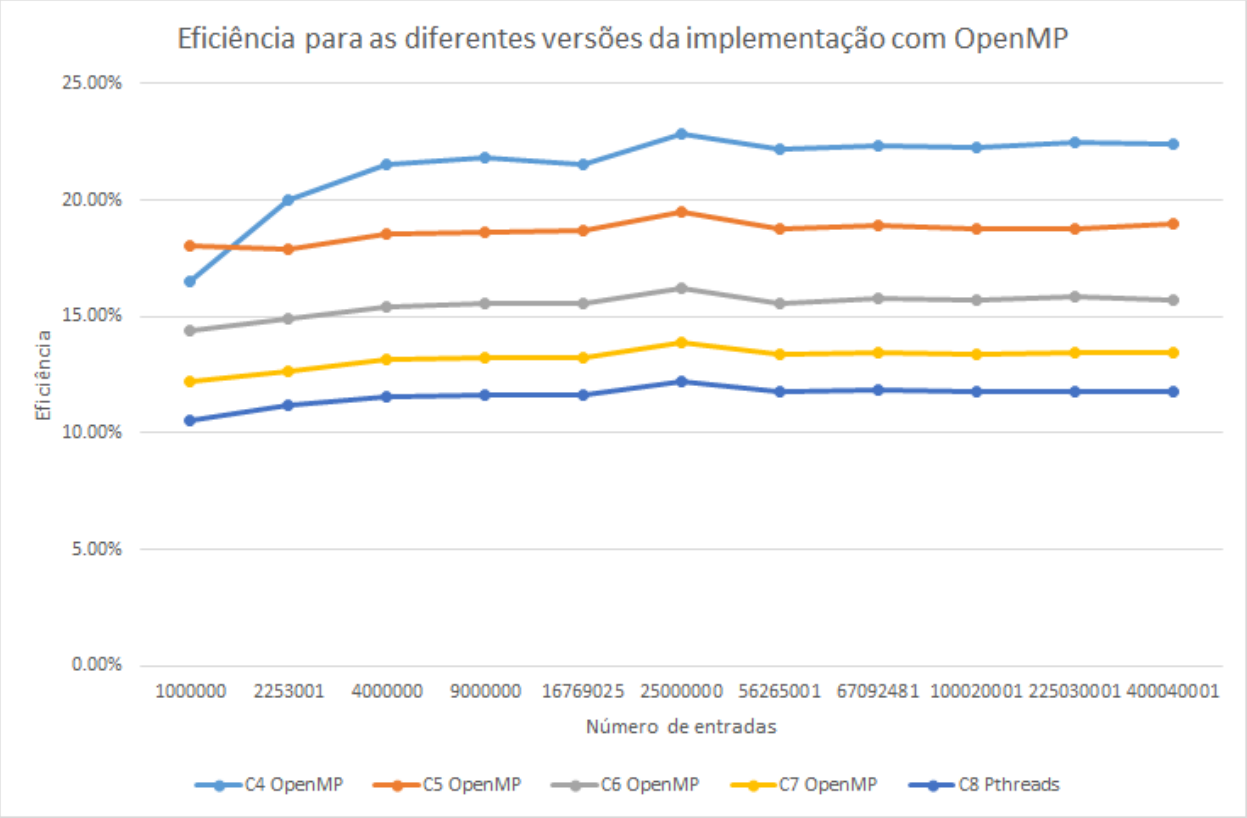
É relembrado que o sistema de nomenclatura das versões é o seguinte, C<n> OpenMP, onde <n> representa o número de threads usados pela versão da implementação de OpenMP em questão.

Novamente verifica-se o crescimento linear em termos do tempo de execução em relação ao número de entradas da matriz, sendo que a versão com quatro threads a mais demorada. Agora procede-se a comparar este algoritmo com a melhor implementação sequencial conhecida, para este efeito irá ser usada a métrica de aceleração, representada pelo seguinte gráfico:



Tal como foi verificado com a implementação com pthreads, independentemente do número de threads, esta implementação é mais lenta que a sequencial, inclusive este algoritmo com apenas 4 threads possui a aceleração mais reduzida, sendo que as outras versões possuem, aproximadamente, em geral a mesma aceleração, exceto em matrizes de tamanho reduzido.

Posto isto é verificada uma baixa eficiência por parte dos threads, como podemos ver no seguinte gráfico com as eficiências:



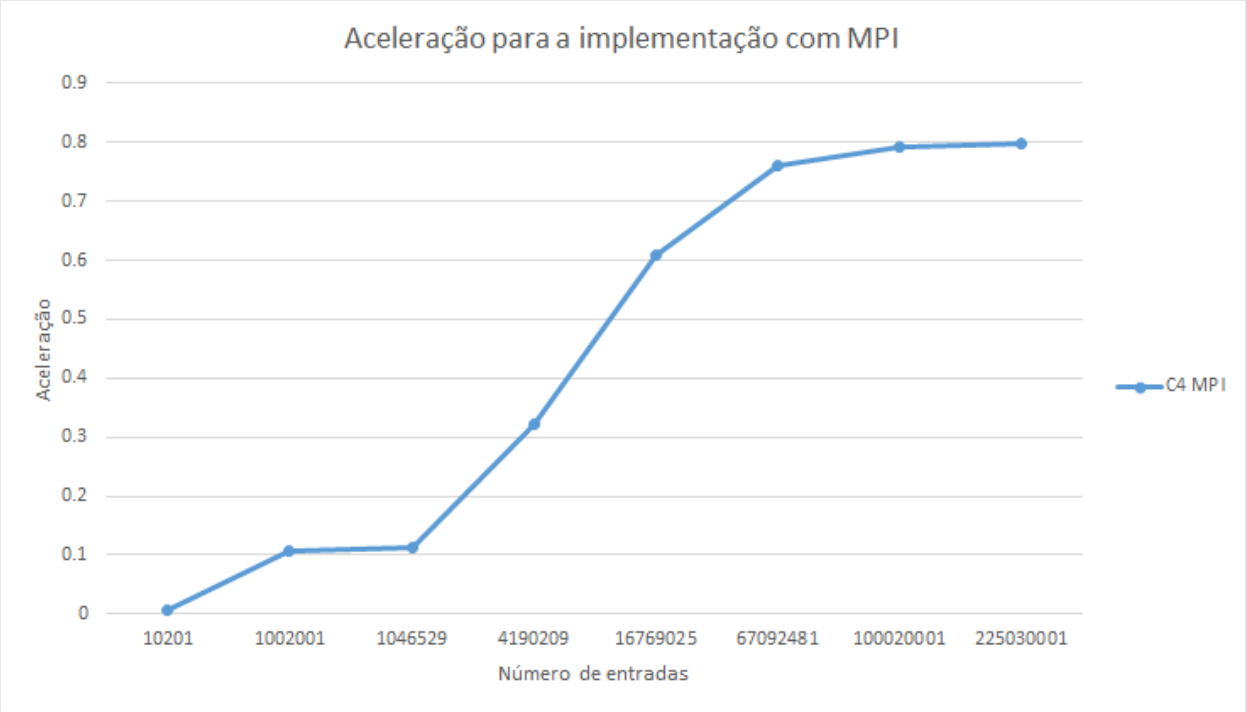
Surpreendentemente a versão com quatro threads é a mais eficiente, apesar de possuir a menor das acelerações. Olhando agora para o gráfico com uma visão mais geral verificamos que realmente este algoritmo deixa muito a desejar em termos da eficiência de como dispõe dos recursos computacionais, sendo que todas as versões possuem uma menor eficiência para matrizes com reduzido número de entradas, o que explica a reduzida aceleração, tanto no geral, como para as matrizes menores. Este resultado de certa forma era expectável devido à leitura do ficheiro pois, apesar de ser dividido em segmentos, no fundo apenas um dos threads lê o ficheiro de maneira sequencial, devido a isto seria de esperar que o tempo de leitura não diminuiria, o que faz a eficiência diminuir devido à falta da diminuição deste tempo de leitura independentemente do número de threads, outro aspeto que pode explicar a falta de aceleração pode ser os tempos de copiar os segmentos do global para o local, juntamente com a gestão dos threads por parte do sistema operativo. Adicionalmente também podemos verificar que este algoritmo não é escalar, pelo que a eficiência diminui à medida que o número de threads aumenta em vez de se manter constante.

4.5. Implementação Distribuída com MPI

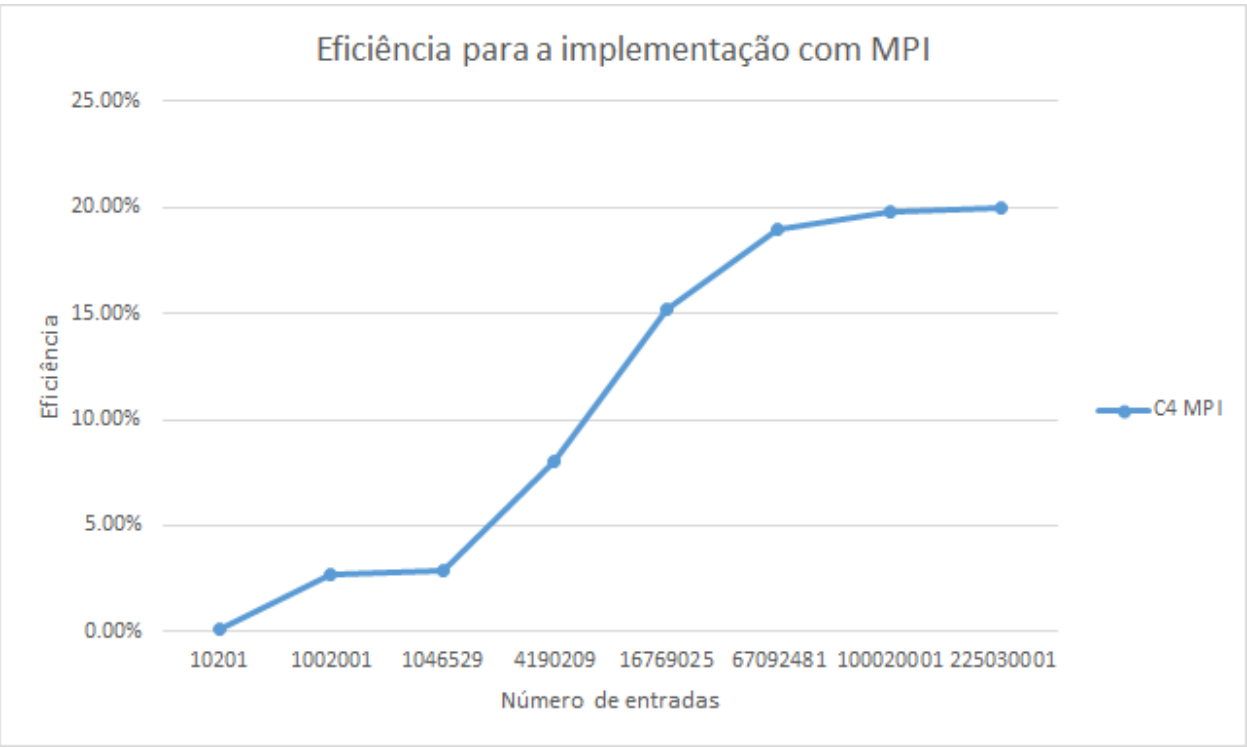
Agora será feita a análise da implementação distribuída com MPI, infelizmente devido a restrições temporais, não foi possível testar este programa com um número variável de workers, pelo que não será possível falar sobre escalabilidade, e devido a problemas encontrados ao tentar correr esta implementação em diferentes computadores, estes testes foram feitos numa só máquina, sendo esta a máquina 2, admite-se agora que

num trabalho futuro seria de interesse testar estas duas condições, de forma a enriquecer o estudo e análise destes algoritmos. Seguidamente apresenta-se os tempos de execução obtidos utilizando quatro workers e de acordo com as condições anteriores, sendo estes testes realizados quando a máquina 2 tinha apenas 2 utilizadores e uma média de carga de 0 nos últimos 15 minutos antes de iniciar os testes:

Antes de mais, estes tempos de execução foram obtidos a partir do tempo real do comando time, pelas mesmas razões que nos dois subcapítulos anteriores. A partir deste gráfico a única conclusão que é possível tirar é que o tempo cresce linearmente em relação ao aumento do número de entradas. Esta é a única comparação possível de retirar devido aos tempos terem sido obtidos por outra máquina, logo não é possível comparar diretamente com os tempos anteriormente obtidos, para este efeito temos a aceleração, cuja foi medida através da expressão (1), mas onde t_1 é o tempo de execução da implementação sequencial desenvolvida para este projeto, mas medida na máquina dois, assim apesar de não completamente perfeito, dá-nos uma boa percepção do aumento ou diminuição do desempenho em termos de tempo e como se relaciona aos outros. Posto isto, seguidamente será apresentado este gráfico:



Olhando para este podemos notar que este terá uma enorme ineficiência para os inputs com menor número de entradas, e que novamente este não oferece uma redução do tempo de execução em relação à implementação sequencial, olhando agora para o gráfico das eficiências, sendo estas calculadas pela expressão (2), temos:



Como podemos ver realmente esta implementação revela-se extremamente ineficiente em relação às matrizes de tamanho mais reduzido, isto pode ser causado pelo tempo de arranque, isto é a preparação para iniciar a comunicação ser mais pronunciada num contexto onde a quantidade de dados é bastante reduzida, devido ao tempo de envio em si e o tempo de processamento serem bastante pequenos. Apesar disto a eficiência acaba por subir à medida que o tamanho da entrada aumenta, ainda assim mostra-se bastante ineficiente, nunca ultrapassando uma eficiência de 20%. Verifica-se também que devido ao tempo de leitura que continua a ser realizado de forma sequencial pode ter também influência na grande ineficiência do algoritmo.

Por fim acrescenta-se apenas que, presente em anexo também se encontram os ficheiros usados na execução desta implementação na máquina 2, junto aos mencionados durante o capítulo 3, tanto os da implementação sequencial como a implementação com MPI, onde em tudo são iguais excepto no facto de o executável poder estar em qualquer pasta mas os inputs apenas podem estar guardados em /home/dados-SPD/inputs/.

4.6. Implementação Híbrida com OpenMP e MPI

Para a implementação híbrida temos as mesma condições que para a distribuída com MPI, isto é os testes foram realizados na máquina 2 e sem ser realizados testes a diferentes números de workers ou threads, pelas mesmas razões, sendo apenas realizado testes para 2 workers e dois threads OpenMP, foi obtido o seguinte gráfico com formado com os tempos de execução obtidos através do tempo real retornado pelo comando time:

Estes tempos foram medidos com o algoritmo a usar dois workers e dois threads OpenMP, novamente verifica-se a progressão linear do algoritmo, mas tal como para MPI devemos olhar para outras métricas para tirar mais conclusões:

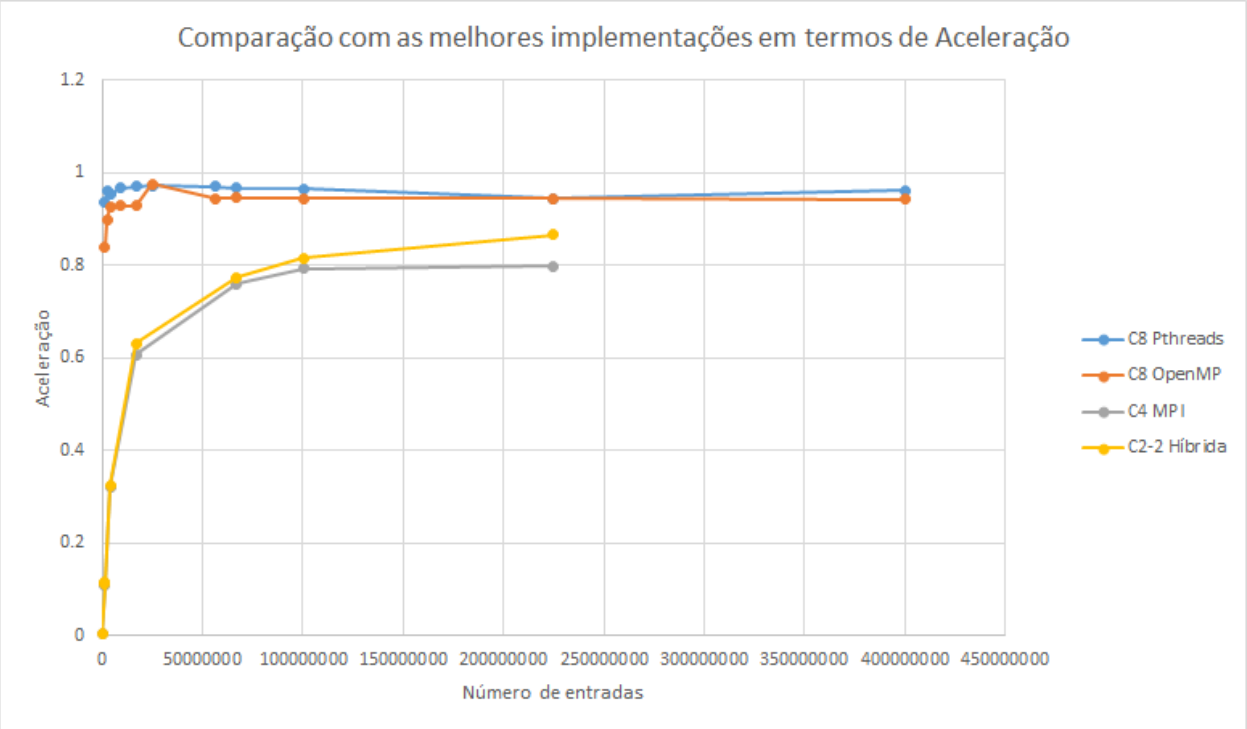
Olhando para este gráfico vemos novamente um péssimo desempenho por parte deste algoritmo quando este processa matrizes de tamanho reduzido, pensa-se que a causa é a mesma que em MPI, isto é, o tempo de preparação para transmissão é maior que a própria transmissão e processamento devido ao reduzido tamanho da matriz, e o próprio tempo de leitura que continua a ser feita de maneira sequencial, nunca permitiria a aceleração ser muito elevada, olhando agora para a eficiência observada:

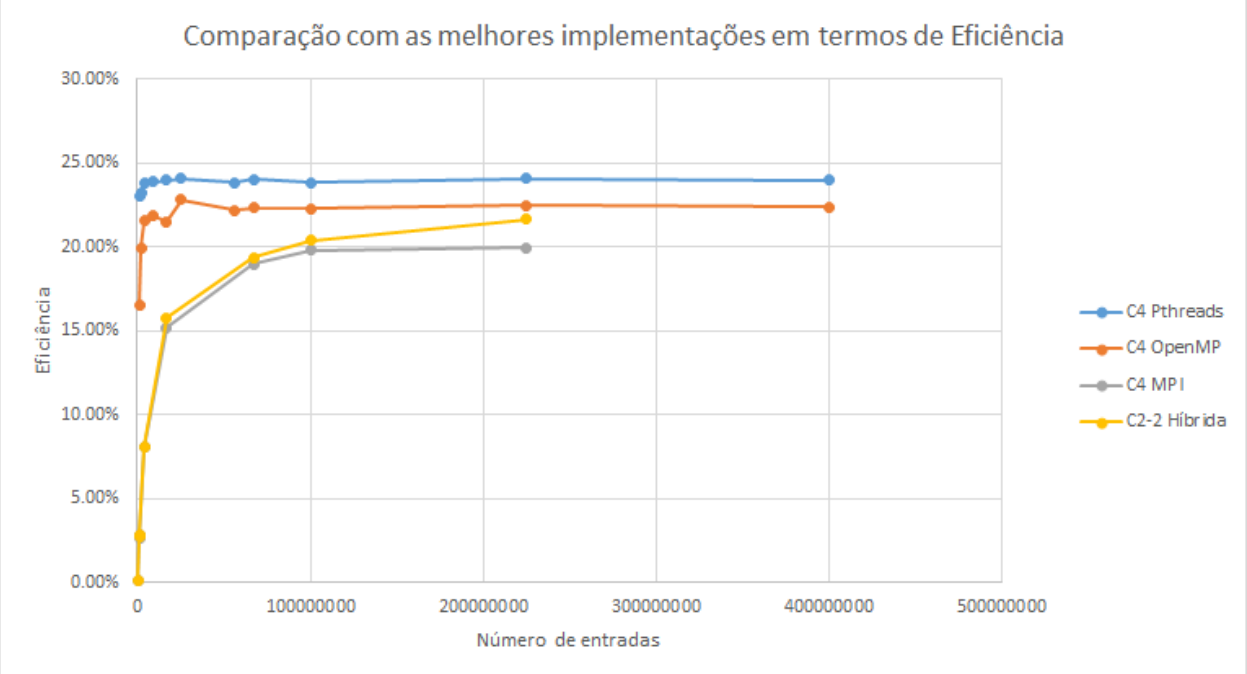
Como era de esperar ao observar a aceleração no início este algoritmo possui uma péssima eficiência para matrizes pequenas, melhorando ao longo que o número de entradas aumenta, apesar disto este algoritmo nunca atingiu uma eficiência que permitisse ser mais veloz que a implementação sequencial, isto é, uma aceleração superior a um.

Novamente os ficheiros criados especificamente para testar a implementação na máquina 2 também se encontram junto aos ficheiros mencionados em 3.5, e tal como mencionado no subcapítulo anterior, apenas diferem no aspeto em que estas versões vão buscar os inputs à diretoria /home/dados-SPD/inputs/.

4.7. Comparação entre os melhores

Apresenta-se seguidamente uma comparação através das métricas aceleração e eficiência das melhores versões de todas as implementações desenvolvidas, através de um gráfico para as acelerações seguido pelo gráfico das eficiências:





Como podemos ver a implementação com pthreads mostrou os melhores resultados em termos de aceleração e eficiência, julga-se que isto se deve ao facto que esta implementação é a que

menos difere da implementação sequencial, também poderá ter sido originado pelo facto de ser o único que lê toda a matriz sem nunca parar, isto é as outras implementações lêem apenas um segmento depois param de ler o ficheiro para realizar alguma operação, copiar para o global no caso do OpenMP e transmitir o segmento no caso do MPI e Híbrida. Outro fator que pode causar a implementação com PThreads apresentar melhores resultados é devido às interações entre tarefas ser relativamente simples, sendo que nenhum thread bloqueia outro a não ser quando é atualizado o resultado para refletir que o quadrado não é mágico perfeito, também este não perde tempo a transmitir partes da matriz como é feito pelo MPI e Híbrido.

Em relação às implementações Híbrida e MPI, considera-se que o ligeiro aumento na eficiência da Híbrida foi devido ao facto de perder menos tempo a preparar a comunicação devido a apenas transmitir a matriz completa para um só worker e seguidamente este processamento ser dividido por 2 threads. Para testar se o transmitir a matriz completa em vez de segmentos torna o algoritmo mais eficiente admite-se um caso importante de testar num estudo futuro.

Admite-se ainda que a eficiência das soluções MPI e Híbrida seria ainda mais reduzida caso os testes fossem realizados em duas máquinas diferentes, devido ao acréscimo do tempo de transmissão, mas apesar disto num estudo futuro seria de valor realmente testar esta teoria aqui proposta.

Comentários finais

Para concluir, não foi possível verificar o poder das abordagens paralelas em termos do aumento da aceleração, isto é, da diminuição do tempo de execução, isto acredita-se ter sido provocado essencialmente pela leitura do ficheiro ser sempre feita por um único thread de maneira sequencial, devido a isto considera-se importante num momento futuro testar uma implementação que permite a leitura de um ficheiro em regime paralelo, isto é, ao mesmo tempo estão dois ou mais threads a ler regiões diferentes do mesmo ficheiro.

Inversamente foi verificado que nem sempre a solução distribuída, neste caso com MPI, é a solução mais apropriada para um problema, pois para o problema de verificar se a matriz é um quadrado perfeito, imperfeito ou não mágico não requer muitos recursos computacionais, pois são apenas múltiplas somas simples, não demorando muito tempo, assim como este perde mais tempo no envio dos segmentos da matriz do que no processamento, a utilização da solução distribuída perde sentido, sendo que este efeito torna-se cada vez mais pronunciado quanto menor for o tamanho da matriz.

Por outro lado foi verificado que uma boa escolha de métricas que indicam o desempenho, tal como uma boa representação dos dados através de gráficos, em muito auxilia a análise e comparação de algoritmos desenvolvidos, permitindo assim um maior objetivismo na escolha de um melhor algoritmo, sendo assim indispensável para uma boa análise de qualquer algoritmo não sequencial.

Bibliografia

1. https://mathworld.wolfram.com/MagicSquare.html
2. Maarten van Steen and Andrew S. Tanenbaum, Distributed Systems, Third Edition, Published by Maarten van Steen, February 2017, ISBN: 978-90-815406-2-9 (digital version)
3. https://www.researchgate.net/publication/251147149_A_Glimpse_of_Parallel_Computing
4. https://thomas-cokelaer.info/blog/2018/02/meaning-of-real-user-and-sys-time-statistics/
5. https://stackoverflow.com/a/556411
6. https://hpc-tutorials.llnl.gov/posix/
7. https://hpc.llnl.gov/openmp-tutorial