



Universidade do Minho

Universidade do Minho
Mestrado Integrado em Engenharia Informática - 4º ano

Paradigmas de Computação Paralela

Ano Letivo de 2015/16



escola de engenharia



departamento de
informática

Open MPI

Paradigma de Memória Distribuída

Trabalho Prático 2 - Equalização do Histograma de uma Imagem

Sérgio Caldas - a57779
Filipe Oliveira - a57816

20 de Dezembro de 2015

Conteúdo

1 Caso de estudo	3
2 Caracterização das Plataformas de teste	3
2.1 Compilador, flags de compilação e ferramentas de medição utilizadas	4
2.1.1 MPICC	4
2.1.2 Bibliotecas MPIP	4
2.2 Abstração do tipo Imagem	4
2.3 Geração de dados de teste	4
2.4 Otimização Sequencial para Matriz Uni-dimensional :: Cache Misses e Loop Nesting	4
3 Solução OpenMPI	5
3.1 Solução inicial em open-MPI	5
3.1.1 Possíveis alterações ao algoritmo de memória distribuída	6
4 Encontrar o Paralelismo	6
4.0.2 Comparação do tempo de execução da versão sequencial vs tempo de execução da versão open-MPI em apenas 1 nodo e 1 core, para matrizes de diferentes dimensões	7
4.0.3 Conclusões a retirar das tabelas de tempos	7
5 Desenvolvimento da Versão Paralela	7
5.1 OverHead Paralelismo	7
5.2 Comparação do tempo total para a solução entre o algoritmo com MPI_Gather e MPI_Barrier no método calculate_histogram e o algoritmo com redução	7
5.2.1 Conclusões a retirar dos tempos de execução obtidos para algoritmo paralelo com MPI_Gather e MPI_Barrier vs algoritmo paralelo com MPI_Reduce	8
5.3 Comparação com evolução do tempo total de execução do algoritmo paralelo(MPI_Reduce) com -map-by core vs -map-by node (comunicação entre nodos r641 por ethernet)	8
5.3.1 Conclusões a retirar do tempo total de execução do algoritmo paralelo(MPI_Reduce) com -map-by core vs mapby node	9
5.4 Evolução do Tempo de Computação vs Tempo de Comunicação (comunicação entre nodos r641 por Ethernet) – através da expansão do número de processos	9
5.4.1 Análise da distribuição de tempo de comunicação pelos diferentes métodos MPI – através da expansão do número de processos MPI	10
5.4.2 Análise da distribuição de tempo de computação para os 3 métodos presentes no algoritmo MPI – através da expansão do número de processos MPI	11
5.5 Comparação com evolução do tempo total de execução do algoritmo paralelo(MPI_Reduce) vs algoritmo paralelo com Computação Extra (comunicação entre nodos r641 por ethernet)	12
5.6 Relação de ganho entre código sequencial, versão paralela OpenMP e versão paralela MPI (comunicação entre nodos r641 por ethernet)	13
5.7 Interpretação do Speedup do algoritmo final de memória distribuída	14
5.8 Evolução do tempo para a solução para (comunicação entre nodos r641 por Ethernet) – através da expansão do número de nodos	14
6 Algoritmo Híbrido	14
6.1 Interpretação do Speedup do algoritmo final de memória distribuída	15
6.2 Testes do algoritmo híbrido para o team laptop	15
6.2.1 Análise do algoritmo híbrido para o team laptop	16
A Caracterização do hardware das plataformas de computação	17
B Excerto :: Geração de Matrizes	18
C Métodos do Algoritmo Reduce	19
C.1 Método calculate_histogram ()	19
C.2 Método calculate_accum ()	19
C.3 Método transform_image ()	19
D Métodos do Algoritmo Gather	20
D.1 Método calculate_histrogram ()	20

E	Métodos do Algoritmo Extra Computação	21
E.1	Método calculate_histogram ()	21
E.2	Método calculate_accum ()	21
E.3	Método transform_image ()	21
F	Métodos do Algoritmo Híbrido	22
F.1	Método calculate_histogram ()	22
F.2	Método calculate_accum ()	22
F.3	Método transform_image ()	22
G	Comparação entre os tempos de execução por método entre as versões sequencial e MPI	23
H	Comparação do tempo total para a solução entre o algoritmo com MPI_Gather e MPI_BARRIER no método calculate_histogram e o algoritmo com redução	24
I	% de Tempo de Computação vs % de Tempo de Comunicação % para o Tempo Total de Execução (ms) para as Matrizes 2048 x 2048 e 4096 x 4096	25
J	Evolução da % do tempo de métodos para o tempo total da solução – através da expansão do número de processos	26
K	Evolução do tempo total para solução através do aumento quer do número de processos MPI, quer do número de threads openMP para o algoritmo híbrido, para matrizes de tamanho 2048*2048 e 4096*4096, para o team laptop	27

1 Caso de estudo

O presente caso de estudo, à semelhança do trabalho prático 1, paraleliza a computação e equalização de um histograma de uma imagem, e sua respetiva tradução numa imagem equalizada. A grande diferença entre ambos os trabalhos assenta no modelo de memória utilizado pelo algoritmo. No trabalho prático 1 o algoritmo recorria a diretivas Open-MP num ambiente de memória partilhada. No presente trabalho todos os processos implementam um algoritmo que recorre a memória distribuída e diretivas Open-MPI.

Como veremos adiante, é indiferente para o método de abordagem se se trata de uma imagem ou de apenas uma matriz de valores.

Foi importante basearmos a obtenção dos valores a partir de uma imagem e sua respetiva re-tradução numa imagem equalizada apenas para efeitos de validação do algoritmo – uma vez que assim conseguimos ter uma perspetiva real das alterações dos valores.

Atente no resultado obtido através do algoritmo. À esquerda encontra-se a imagem inicial e à direita a imagem devidamente equalizada:



Figura 1: Resultado da equalização

Tal como foi necessário no trabalho prático 1 estudar o grau de escalabilidade do algoritmo paralelo, no presente paradigma e algoritmo, necessitamos também de o fazer. Contudo, temos agora que analisar um fator inexistente no paradigma de memória partilhada – os processos necessitam de comunicar entre si e partilhar os dados e mensagens necessárias à correta implementação do algoritmo – há portanto um overhead adicional ao da computação paralela – o overhead da comunicação dos dados entre processos.

O grau de escalabilidade do kernel paralelo dependerá agora, também, na capacidade do algoritmo de balancear tempo de comunicação e tempo de computação.

Podemos então resumir o caso de estudo em:

- **Problema:** Contar o número de ocorrências de cada valor – no nosso caso a intensidade da imagem – numa tabela de valores (imagem). Equalizar os valores e recalcular a tabela de valores.
- **Input:** Imagem ou qualquer tipo de dados matricial não equalizado.
- **Output:** Imagem ou qualquer tipo de dados matricial equalizado.
- **Objetivo:** Minimizar o tempo total do algoritmo sequencial.

2 Caracterização das Plataformas de teste

A plataforma de testes utilizada no Search6 recorre a nodos equipados com dual-socket system, com dois processadores Intel® da micro-arquitetura Ivy Bridge. Os sistemas, doravante referenciados como nodos de

computação 641, estão equipados com dois Intel® Xeon® E5-5650 v2 e 64 GB de memória RAM DDR3 instalada, estando a memória dividida em 4 memory channels.

Para esta plataforma, dadas as duas situações distintas de comunicação entre nodos:

- Comunicação entre nodos por rede myrinet.
- Comunicação entre nodos por rede ethernet.

era nossa intenção realizar testes para os 2 tipos de comunicação e comparar os seus resultados. Contudo, dada a inviabilidade de utilizar a rede Myrinet (comprovada pelos ficheiros de profiling para um número de processos MPI superior a 32 – quando os processos eram mapeados em 2 máquinas distintas) por parte da diretiva mpiRun foi-nos impossível obter valores para comparação. Julgamos que esta é uma tremenda limitação que terá um peso extremo na análise da viabilidade dos algoritmos MPI e híbrido.

A segunda plataforma de computação, o team laptop, é um Intel® Core™ i7-3635QM (micro arquitetura Ivy Bridge). O sistema de computação tem instalados 16 GB de memória DDR3 RAM, suportada a uma frequência de 1600MHz, dividida em 2 memory channels.

As características dos CPU's das plataformas de computação enumeradas estão presentes na tabela 3 no [appendix A na page 17](#).

2.1 Compilador, flags de compilação e ferramentas de medição utilizadas

Todos os executáveis gerados foram produzidos através do compilador da gnu – gcc (GCC) versão 4.9.0, sendo as flags de compilação utilizadas as seguintes: -O3 -Wall -Wextra -std=c++11 -fopenmp.

2.1.1 MPICC

A versão do MPICC por nós utilizada foi a gcc (GCC) 4.4.6 20120305 (Red Hat 4.4.6-4).

2.1.2 Bibliotecas MPIP

A versão mpiP utilizada foi a versão 3.4.1 com Build date Jun 5 2014.

2.2 Abstração do tipo Imagem

Uma imagem não é mais do que uma matriz de pixels. Considerando o propósito deste trabalho é-nos útil abstrair-nos do tipo de dados imagem e passar a considerá-lo apenas como uma matriz de valores que oscilam entre 0 e 255.

2.3 Geração de dados de teste

Tendo prova da correção do algoritmo podemos ainda, ao invés de realizar input/output de imagens no cluster, passar a gerar matrizes aleatórias com dimensões adequadas ao problema – para o nosso caso consideramos matrizes de 2048*2048, 4096*4096 e 8192*8192.

Escolhemos estes valores pois a primeira matriz consegue ser completamente inserida em memória cache L3 dos nossos nodos de teste (os 30MB são superiores aos 17MB do tamanho aproximado da matriz), já o segundo (67MB) e terceiro (268MB) apresentam tamanhos de matriz superiores à cache L3.

Em cada uma das células da matriz encontra-se então um valor entre 0 e 255 gerado aleatoriamente pelo código anexado em [appendix B na page 18](#).

2.4 Otimização Sequencial para Matriz Uni-dimensional :: Cache Misses e Loop Nesting

Os arrays em c/c++ são guardados em memória como um bloco contíguo. Assim aceder a[x][y] é equivalente a a[x*N_COLUNAS + y]. Ao iterarmos em 2 dimensões podemos estar a não utilizar largos espaços de memória podendo causar cache misses e as respetivas penalizações.

Adicionalmente, ao reduzirmos de duas para uma dimensão estamos também a reduzir a complexidade do código e a realizar "loop nesting" para ciclos que percorram os dados.

Para o paralelismo e vetorização isto é uma mais valia uma vez que o compilador poderá tentar vetorizar código que anteriormente considerava não vetorizável.

Temos ainda o facto que muitos compiladores não suportam "nested parallelism", e mesmo que suportem o overhead adicional trará possivelmente ganhos nulos.

Por último podemos ainda referir o facto de nestes casos necessitarmos de menos sincronização entre processos/threads.

3 Solução OpenMPI

3.1 Solução inicial em open-MPI

Como pode confirmar no anexo **appendix C na page 19** existem três métodos adicionais à main sendo a função de cada um descrita de seguida – assim como a forma como o processo root e os restantes comunicam exemplificada por os respectivos esquemas.

- **void calculate_histogram ()** responsável por calcular o histograma inicial.

O processo root divide a imagem inicial em porções de tamanho idêntico para cada processo MPI. Essas porções do array inicial são seguidamente transmitidas, uma por cada processo MPI.

Após a receção da porção da imagem, cada processo MPI (incluindo o root) calcula o seu histograma local. Esse mesmo histograma local é depois enviado para o processo root e reduzido para um único histograma.

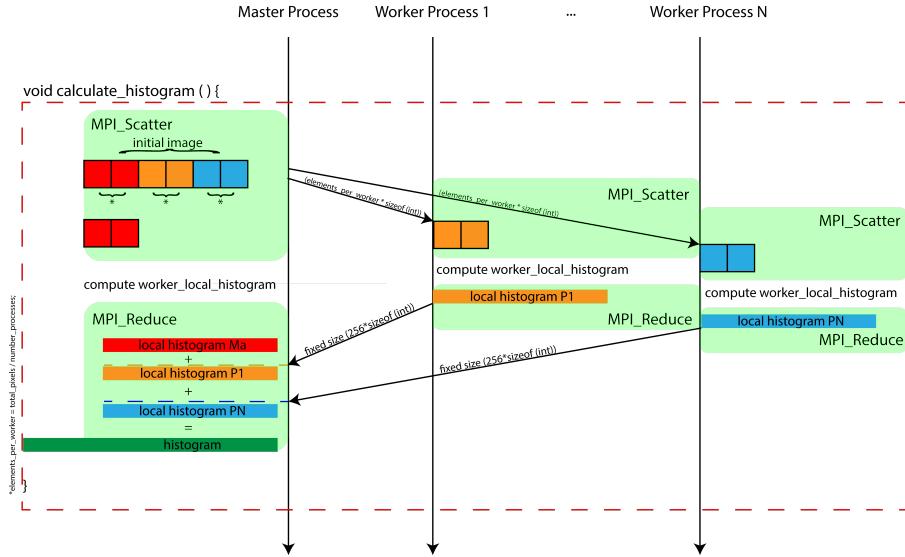


Figura 2: Esquema de comunicação entre processos do método calculate_histogram()

- **void calculate_accum (int total_pixels)** responsável por calcular o histograma acumulado.

Este método, como veremos de seguida, apresenta uma computação reduzida (apenas temos que percorrer 256 valores do histograma e calcular o acumulado). Relativamente à comunicação entre processos presente neste método, esta é feita apenas do processo root, após a computação do histograma acumulado, para os restantes processos através de MPI_Bcast.

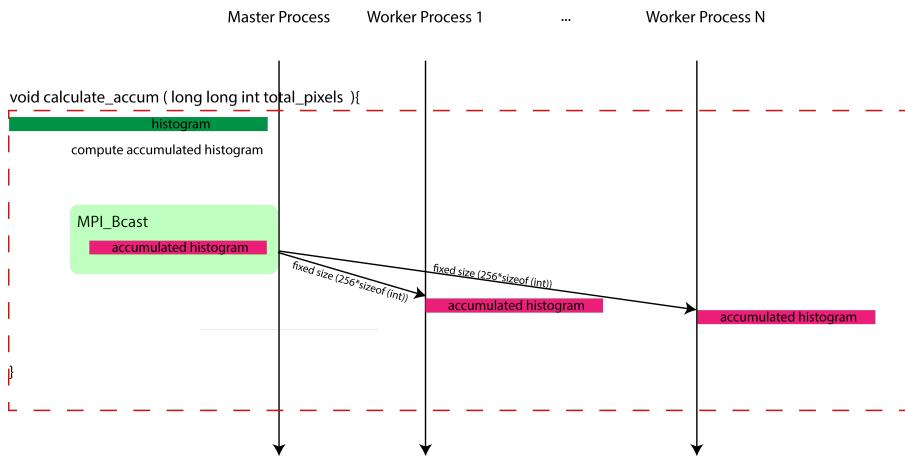


Figura 3: Esquema de comunicação entre processos do método calculate_accum ()

- **void transform_image ()** responsável por equalizar a imagem.

Cada processo, após ter recebido o histograma acumulado no método anterior, calcula a sua porção da imagem final. Essa mesma porção é depois enviada para o processo root através da diretiva MPI_Gather. Após todos os processos terem enviado a sua porção da imagem final, temos presente no processo root a imagem final completa.

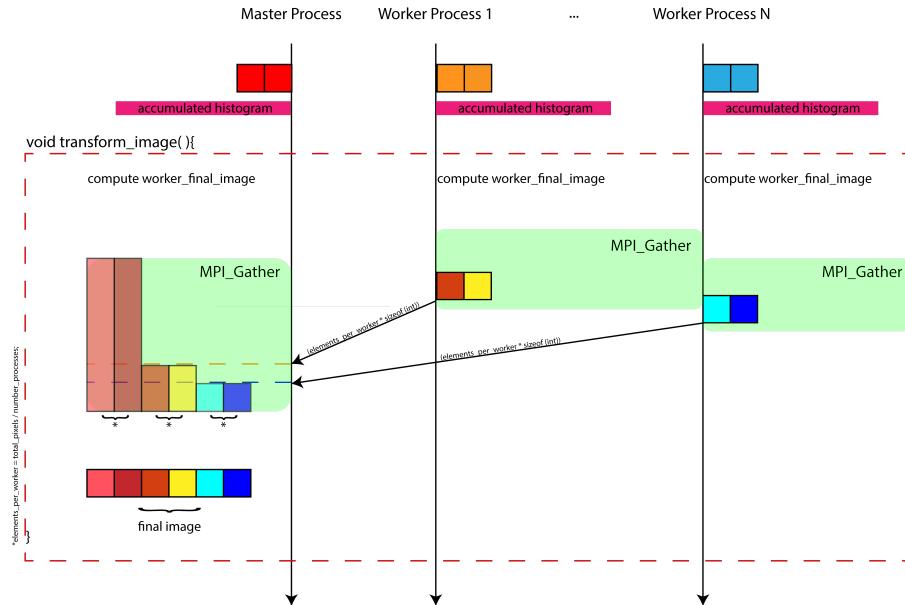


Figura 4: Esquema de comunicação entre processos do método `transform_image()`

No decorrer de todo este projeto iremos ser acompanhados pelos mesmos três métodos, sendo que alguns deles sofrerão alterações do algoritmo.

Contudo, esses 3 métodos que teremos no final deste projeto terão a mesma função que estes agora apresentados (serão apenas mais eficientes).

O algoritmo MPI final será doravante designado como MPIReduce.

3.1.1 Possíveis alterações ao algoritmo de memória distribuída

Na primeira versão do algoritmo em memória distribuída, o método `calculate_histogram` (`long long int total_pixels`), ao invés de utilizar a redução MPI_Reduce, recorria ao método MPI_Gather – recolhendo todos os histogramas locais e apenas de seguida somando-os em apenas um histograma geral. No decorrer do caso de estudo iremos apresentar essa solução como uma alternativa e iremos apresentar os resultados obtidos. Tais resultados serão discutidos nas secções seguintes do caso de estudo.

4 Encontrar o Paralelismo

Por forma a podermos analisar potenciais métodos a paralelizar necessitamos de realizar a medição do tempo de computação útil para o nosso problema.

Assim sendo, devemos registar da forma mais precisa possível os tempos de início e fim de cada método relevante.

Dado que o algoritmo sequencial em estudo é o mesmo que o analisado no trabalho prático 1, resta-nos comparar os valores obtidos para a versão sequencial do trabalho prático 1, com os valores obtidos para a execução da versão em memória distribuída (open MPI) em apenas 1 nodo e 1 core. Denote que os tempos para a versão open MPI são obtidos recorrendo ao método disponível na biblioteca MPI `MPI_Wtime()`.

Para um olhar mais atento à forma como os tempos são registados aconselha-mos a visualização do **appendix C na page 19**.

Conforme referimos anteriormente existem 3 métodos passíveis de paralelização. Recorrendo aos métodos supra referidos obtivemos os seguintes tempos (em mili-segundos) para a execução em 1 nodo com apenas 1 core para matrizes de 2048*2048, 4096*4096 e 8192*8192.

Denote que os valores apresentados correspondem ao melhor tempo obtido de uma amostra de 10 para cada tipo de matriz.

4.0.2 Comparação do tempo de execução da versão sequencial vs tempo de execução da versão open-MPI em apenas 1 nodo e 1 core, para matrizes de diferentes dimensões

Tam. Matriz	Tempo Total Seq. (ms)	Tempo Total MPI (ms)	Ganho MPI vs Seq
2048*2048	17.497	24,5459	-28,72%
4096*4096	68.954	82,4769	-16,40%
8192*8192	274.768	329,884	-16,71%

Tabela 1: Comparação entre os tempos de execução por método entre as versões sequencial e MPI e respetivo ganho pode ser consultada no [appendix G na page 23](#)

Podemos ainda extrair, dos tempos calculados anteriormente para o algoritmo MPI, a percentagem de cada um dos 3 métodos quando comparados com o tempo de execução total do kernel.

Tam. Matriz	% Tempo Kernel Calculo Histograma	% Tempo Kernel Cálculo Hist. Acumulado	%Tempo Kernel Cálculo Imagem
2048*2048	41,25%	0,05%	58,67%
4096*4096	39,87%	0,01%	60,11%
8192*8192	39,46%	0,00%	60,53%

Tabela 2: Percentagem dos tempos dos métodos quando comparados com o tempo total de execução do kernel

4.0.3 Conclusões a retirar das tabela de tempos

Como podemos constatar, os tempos do método de Cálculo do Histograma Acumulado têm uma afetação nula no tempo geral de computação –tal como constatado para a versão do trabalho prático 1.

Devemos então paralelizar o primeiro e o terceiro métodos, sendo que devemos focar mais atenção no último por representar aproximadamente 60% do tempo total de computação.

O facto de o tempo total de cálculo do histograma acumulado ter significância nula deve-se ao número de cálculos a realizar ser reduzido e à quantidade de dados a transferir entre processos ser também reduzida. Tal será comprovado nas secções seguintes do caso de estudo.

5 Desenvolvimento da Versão Paralela

5.1 OverHead Paralelismo

É tentador usar paralelismo em todos os métodos do kernel. Contudo a ideia de uma distribuição mais abrangente do esforço de computação resultante de um aumento do número de processos, poderá levar ao acréscimo de custos de valor elevado para o nosso algoritmo – os custos de sincronização e comunicação entre processos.

Infelizmente, como nos fomos apercebendo com o decorrer do trabalho, isto raramente leva a grandes ganhos de desempenho, por causa do overhead gerado pelos motivos anteriormente enunciados. Passamos agora a ter que considerar o tempo de computação, o tempo de comunicação e o tempo de ócio para cada solução.

Tivemos, portanto, que testar os ganhos potenciais do incremento unitário do número de processos para a implementação que consideramos menos penosa em termos de tempo total de execução.

5.2 Comparação do tempo total para a solução entre o algoritmo com MPI_Gather e MPI_Barrier no método calculate_histogram e o algoritmo com redução

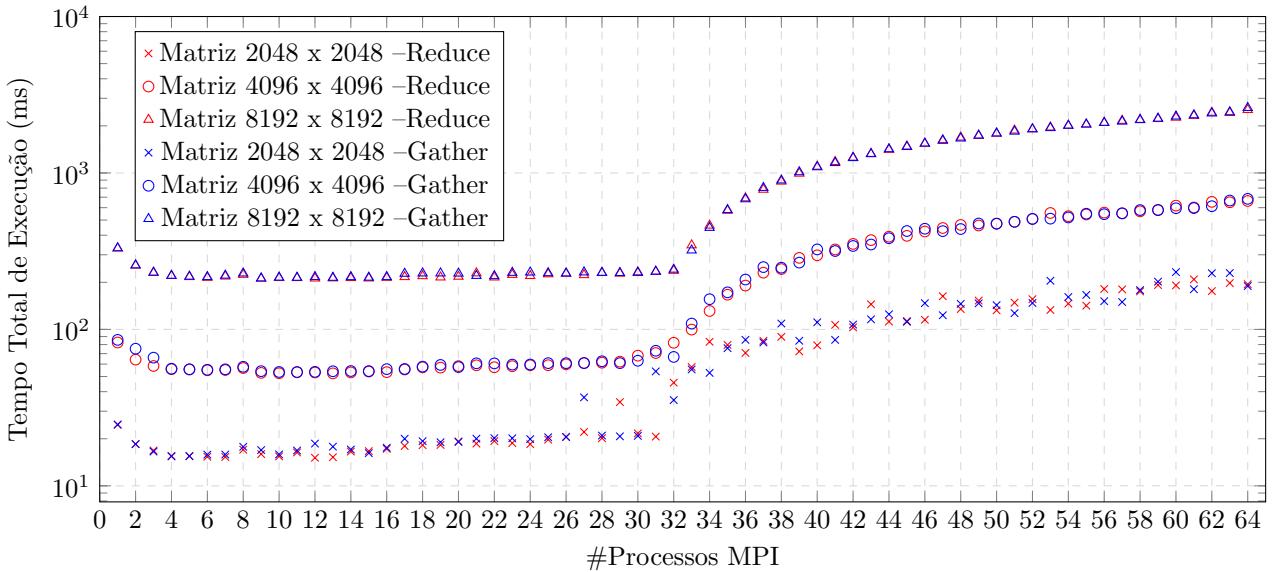
Conforme enunciado anteriormente, existe uma implementação alternativa que devemos ter em consideração. A mesma, recorre a MPI_Gather e a MPI_Barrier na sua especificação do método calculate_histogram (). Aconselha-mos a visualização da sua implementação através do [appendix D na page 20](#).

De seguida apresenta-se a relação entre as versões final e versão com MPI_Gather para um número de nodos igual a 2 e comunicação entre nodos Ethernet.

Para uma análise mais aprofundada das diferenças entre os tempos de execução aconselhamos a visualização da tabela X presente no [appendix H na page 24](#).

Recorrendo aos métodos de medição de tempo anteriormente referidos obtivemos os seguintes tempos (em mili-segundos) para execução do algoritmo com MPI_Gather e MPI_Barrier no método calculate_histogram vs o algoritmo paralelo (MPIReduce), para matrizes de 8192*8192.

Denote que os valores apresentados correspondem ao melhor valor de uma amostra de 10 com 5% de tolerância, para cada tipo de matriz,tendo sido o trabalho submetido nos nodos compute-641, com comunicação entre nodos via Ethernets.



5.2.1 Conclusões a retirar dos tempos de execução obtidos para algoritmo paralelo com MPI_Gather e MPI_Barrier vs algoritmo paralelo com MPI_Reduce

Dada a inclusão da MPI_Barrier, partimos do pressuposto que o tempo total de execução do algoritmo com esta diretiva seria superior ao algoritmo apenas com a redução. Contudo, conforme verificado graficamente, ambos os algoritmos apresentam o mesmo tempo de execução.

Para uma maior validação, podemos ainda analisar os ficheiros .mpiP gerados. Para, por exemplo, um número de processos igual a 64, foram selecionados os melhores cenários alcançados por cada algoritmo. Para o algoritmo com redução, o tempo total do kernel foi de 4.8 segundos, dos quais 2.72 foram gastos em diretivas openMPI.

Para o mesmo número de processos, o algoritmo com MPI_Gather e MPI_Barrier apresentou tempo de total de 4.93 segundos, dos quais 2.61 segundos foram gastos com diretivas openMPI.

Novamente, como se pode constatar, e ao contrário do pensado inicialmente os algoritmos apresentam aproximadamente os mesmos tempos de solução.

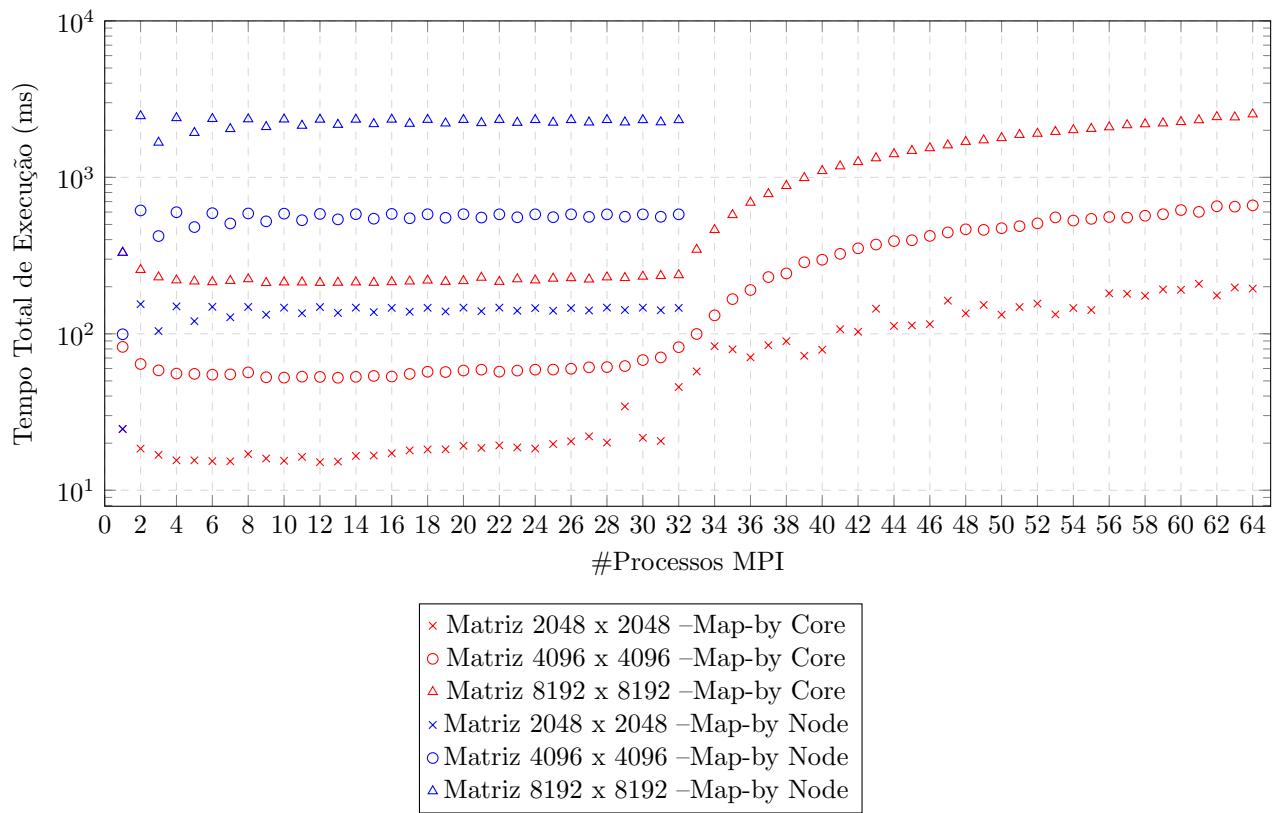
Denote que apesar da diferença de tempos ser reduzida a solução para o problema através do algoritmo com redução foi mantida.

5.3 Comparação com evolução do tempo total de execução do algoritmo paralelo(MPI_Reduce) com –map-by core vs –map-by node (comunicação entre nodos r641 por ethernet)

Uma vez que o ambiente de execução MPI, permite a especificação da forma de mapeamento dos executáveis achamos importante estudar os 2 tipos mais pertinentes de mapeamento para o nosso algoritmo – o mapeamento por core e o mapeamento por nodo.

Recorrendo aos métodos de medição de tempo anteriormente referidos obtivemos os seguintes tempos (em mili-segundos) para execução do algoritmo paralelo (MPIReduce) com -map-by core vs -map-by node , para matrizes de 8192*8192.

Denote que os valores apresentados correspondem ao melhor valor de uma amostra de 10 com 5% de tolerância, para cada tipo de matriz,tendo sido o trabalho submetido nos nodos compute-641, com comunicação entre nodos via Ethernet.



5.3.1 Conclusões a retirar do tempo total de execução do algoritmo paralelo(MPI_Reduce) com –map-by core vs mapby node

Conforme esperado, uma vez que para a solução mapeada por nodo, logo a partir do processo 1 estamos a comunicar entre nodos distintos, o overhead máximo da comunicação é rapidamente atingido.

O mapeamento por nodo deve então ser excluído para este algoritmo por ter um comportamento ainda mais penoso em termos de comunicação que o mapeamento por core.

5.4 Evolução do Tempo de Computação vs Tempo de Comunicação (comunicação entre nodos r641 por Ethernet) – através da expansão do número de processos

Como foi mencionado anteriormente, o overhead de comunicação do nosso algoritmo aumenta à medida que o número de processos MPI envolvidos na solução aumenta. De seguida apresentam-se a relação entre tempo de comunicação e tempo de computação do nosso algoritmo para matrizes de 8192 * 8192 – para comunicação entre nodos r641 por Ethernet.

Dada a semelhança da relação para tamanhos de matrizes inferiores, as mesmas podem ser consultadas no **appendix I na page 25**.

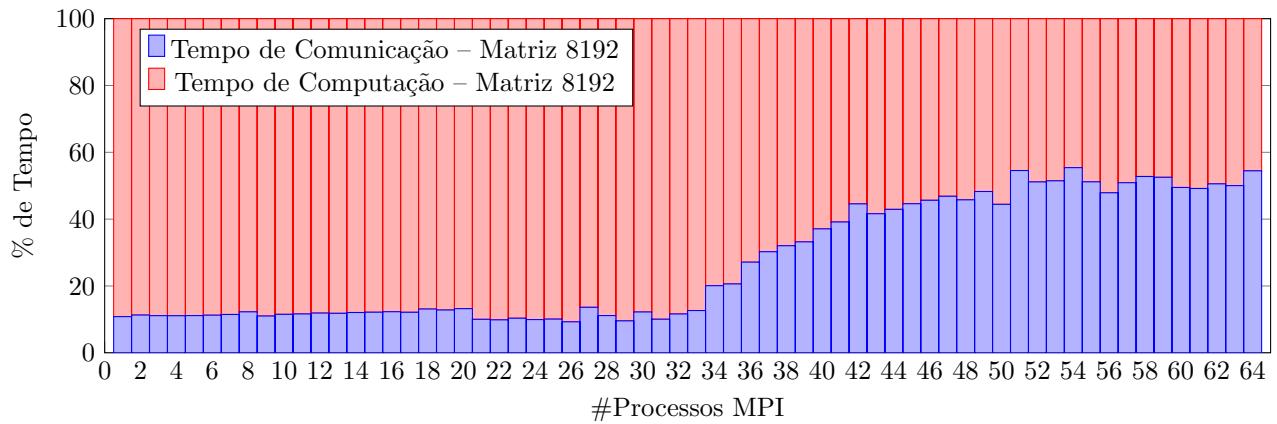


Figura 5: Percentagem de tempo de Computação vs percentagem de tempo de Comunicação % para o Tempo Total de Execução (ms) para a Matriz 8192 x 8192

Como podemos constatar pela análise do gráfico anterior, o aumento do número de processos MPI para a solução, leva ao aumento do peso do tempo de comunicação por parte do processo master em detrimento do peso do tempo de computação para o tempo total da solução.

Isso deve-se ao aumento do número de processos envolvidos comunicantes e consequentemente ao aumento da complexidade de sincronização e passagem de mensagens e dados.

Uma vez que para, por exemplo, o desenvolver do algoritmo é necessário que todos os processos enviem o respetivo histograma (calculado através do 1º método do algoritmo) basta um processo MPI atrasar o cálculo do seu histograma local, ou a rede de comunicação entre processos estar congestionada por outros dados que não os da nossa solução, o tempo total da solução estará certamente comprometido.

Devemos portanto, analisar qual dos métodos MPI consome mais tempo para a solução.

5.4.1 Análise da distribuição de tempo de comunicação pelos diferentes métodos MPI – através da expansão do número de processos MPI

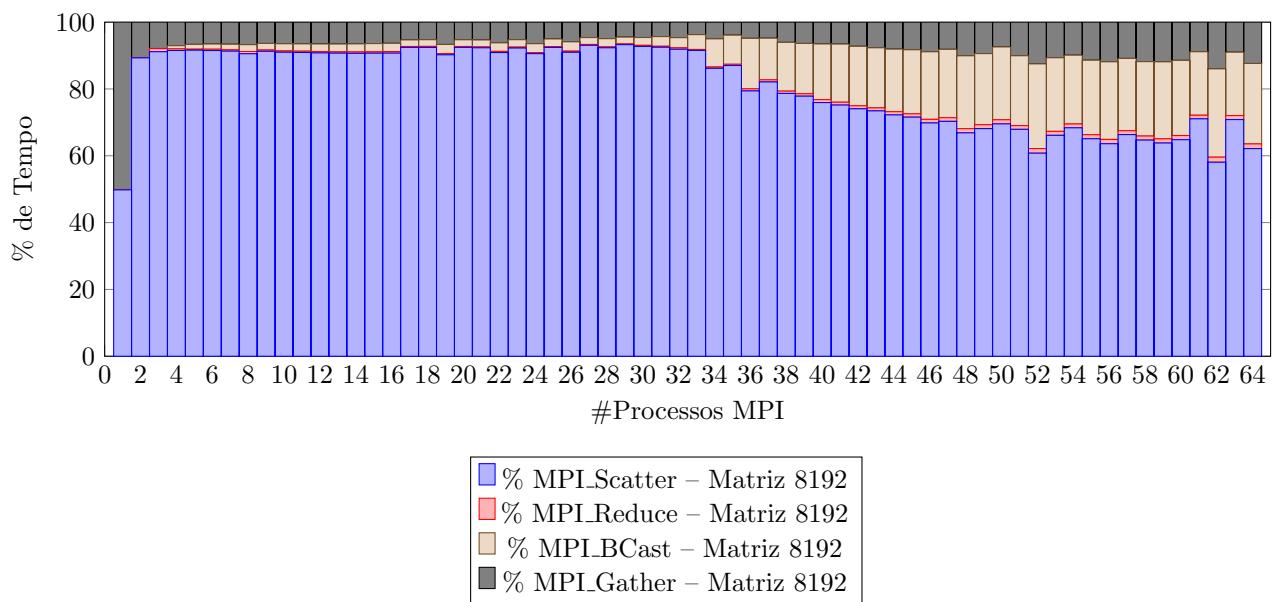


Figura 6: % de Tempo de Comunicação para os diferentes métodos MPI, para a Matriz 8192 x 8192, para comunicação entre nodos via Ethernet – através da expansão do número de processos MPI

Denote que a maioria do tempo despendido em métricas MPI está associado ao método MPI_Scatter. Esse mesmo método está presente na primeira função do algoritmo.

Para os restantes métodos MPI, podemos concluir que o tempo despendido na redução (MPI_Reduce) tem um peso quase nulo em termos de tempo MPI e consequente overhead de tempo de comunicação. Já o método

MPI_Bcast inicialmente apresenta um peso nulo em termos de overhead adicional, sendo que com o aumento de processos MPI o tempo despendido por esta métrica aumenta a ponto de o mesmo representar o 2º método com maior peso no overhead adicional de comunicação para o número máximo de processos envolvidos no caso de estudo.

O método MPI_Gather apresenta uma percentagem de tempo constante para a solução, sendo o mesmo de importância reduzida quando comparado com o tempo despendido pelo método MPI_Reduce.

5.4.2 Análise da distribuição de tempo de computação para os 3 métodos presentes no algoritmo MPI – através da expansão do número de processos MPI

Por forma a realizarmos o profilling completo do algoritmo devemos ainda analisar a percentagem do tempo MPI, despendida por cada função do algoritmo. Só assim poderemos ponderar qualquer alteração ao mesmo.

O gráfico **fig. 7** analisa a evolução da percentagem despendida pelo algoritmo para os 3 métodos do mesmo com a evolução do número de processos MPI presentes na solução, para matrizes de 8192×8192 .

Denote que os valores apresentados correspondem ao melhor valor de uma amostra de 10 com 5% de tolerância, para cada tipo de matriz, tendo sido o trabalho submetido nos nodos compute-641, com comunicação entre nodos via Ethernet.

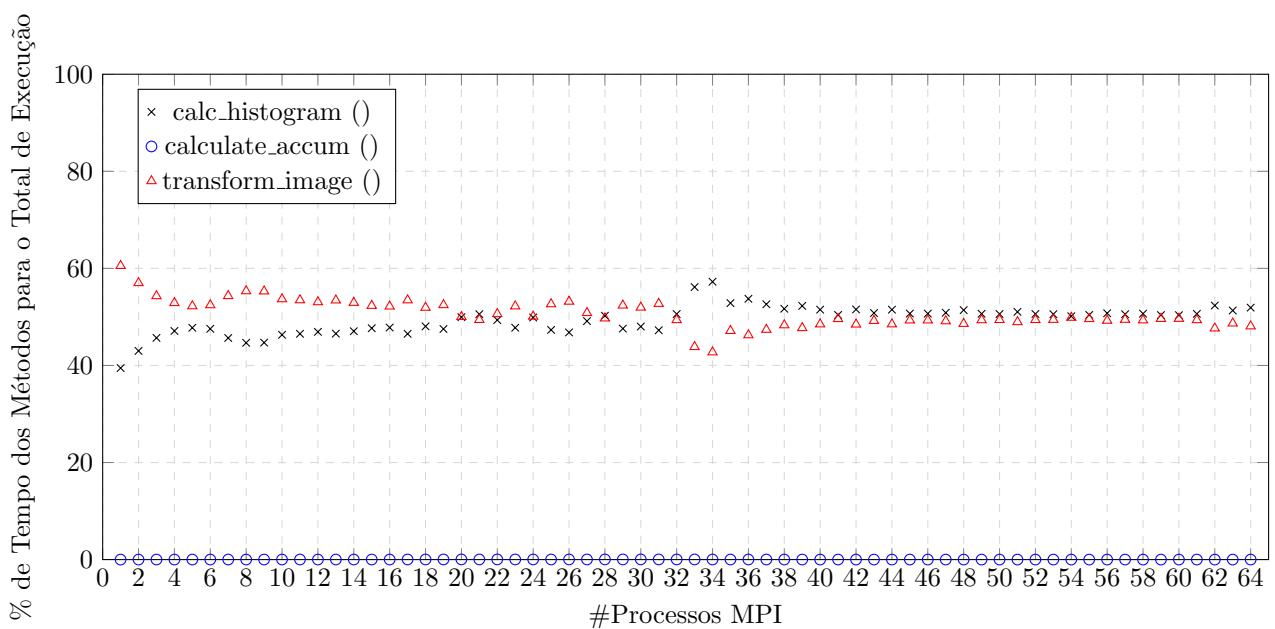


Figura 7: % de Tempo dos Métodos para o Total de Execução para a Matriz 8192×8192

Devemos ainda analisar o tempo total despendido pelos métodos MPI presentes em cada uma dessas funções do algoritmo.

De seguida apresenta-se os tempos referidos para a matriz 8192×8192 , analisando a sua evolução com o aumento de processos MPI.

Recorrendo aos métodos de medição de tempo anteriormente referidos obtivemos os seguintes tempos de Overhead em Comunicação (em mili-segundos) para execução do algoritmo paralelo (MPIReduce), para matrizes de 8192×8192 .

Denote que os valores apresentados correspondem ao melhor valor de uma amostra de 10 com 5% de tolerância, para cada tipo de matriz, tendo sido o trabalho submetido nos nodos compute-641, com comunicação entre nodos via Ethernet.

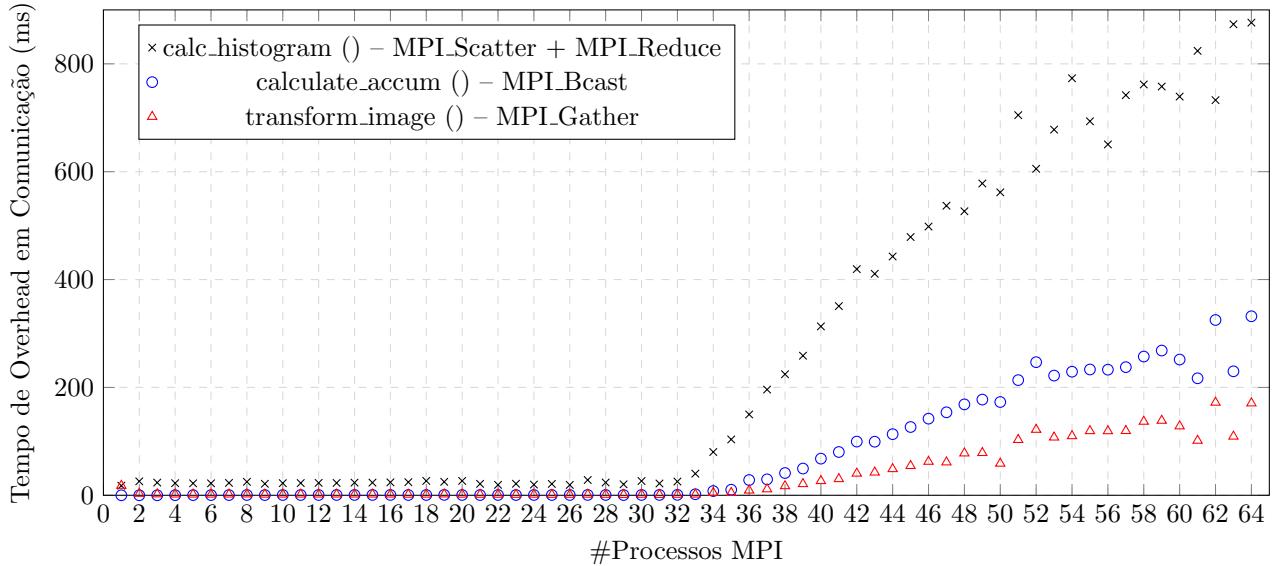


Figura 8: Tempo despendido pelos métodos do algoritmo em métodos MPI, para a execução algoritmo para a Matriz 8192 x 8192, através de Ethernet

Denote que com o acréscimo do número de processos MPI presentes na solução, o método `calc_histogram()` é o que apresenta um overhead de tempo de comunicação mais significativo. Devemos portanto analisar uma solução alternativa que reduza o overhead de comunicação geral do algoritmo através do aumento da computação em cada processo MPI envolvido na solução. Tal solução alternativa poderá ser alcançada tentando eliminar ou reduzir o método `MPI_Scatter`.

5.5 Comparação com evolução do tempo total de execução do algoritmo paralelo(MPI_Reduce) vs algoritmo paralelo com Computação Extra (comunicação entre nodos r641 por ethernet)

Tal como referido anteriormente a solução alternativa aumenta a computação em cada processo MPI, através do broadcast de toda a imagem inicial para cada processo envolvido na solução.

Desta forma retiramos a necessidade de utilizar o método `MPI_Scatter`, através da adição da computação redundante. Cada processo é agora responsável por computar o histograma completo, assim como o respectivo histograma acumulado. Após esses dois passos do algoritmo o processo MPI transforma apenas a sua porção da imagem final enviando-a para o processo master.

Toda as alterações ao código do algoritmo podem ser consultadas no [appendix E na page 21](#).

Recorrendo aos métodos de medição de tempo anteriormente referidos obtivemos os seguintes tempos (em mili-segundos) para execução do algoritmo paralelo (`MPIReduce`) vs algoritmo paralelo com Computação Extra , para matrizes de 8192*8192.

Denote que os valores apresentados correspondem ao melhor valor de uma amostra de 10 com 5% de tolerância, para cada tipo de matriz,tendo sido o trabalho submetido nos nodos compute-641, com comunicação entre nodos via Ethernet.

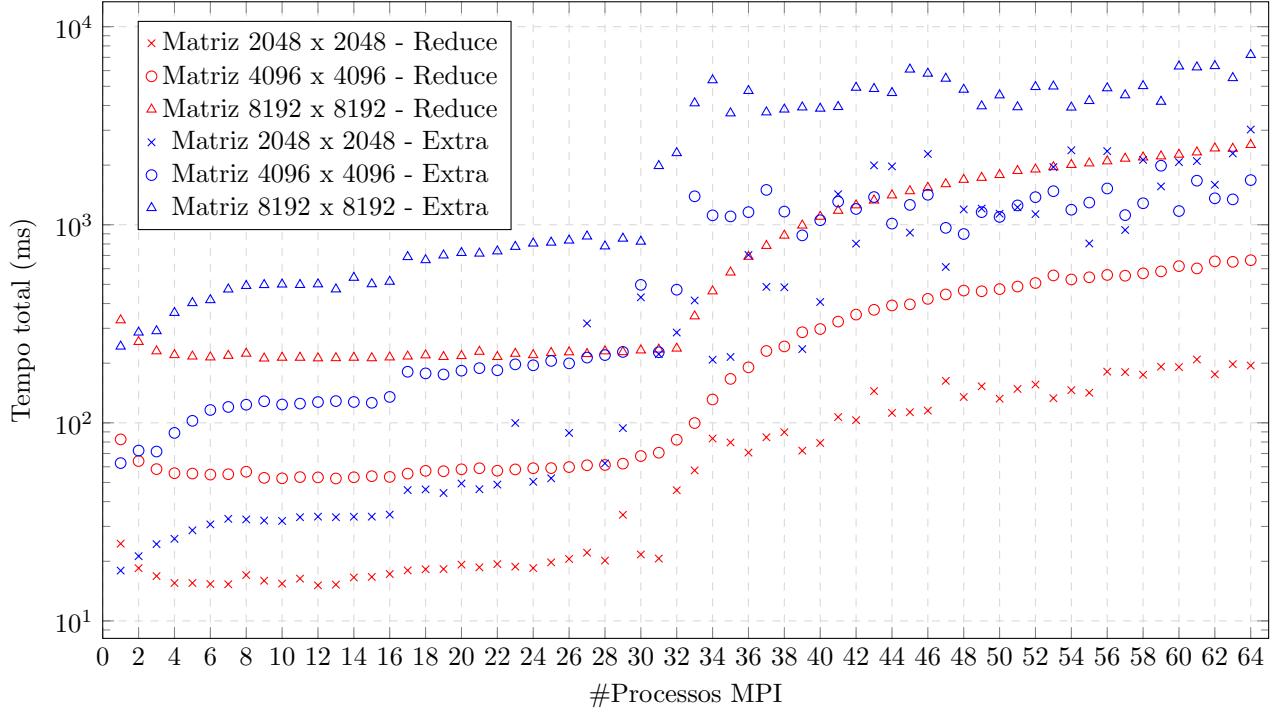


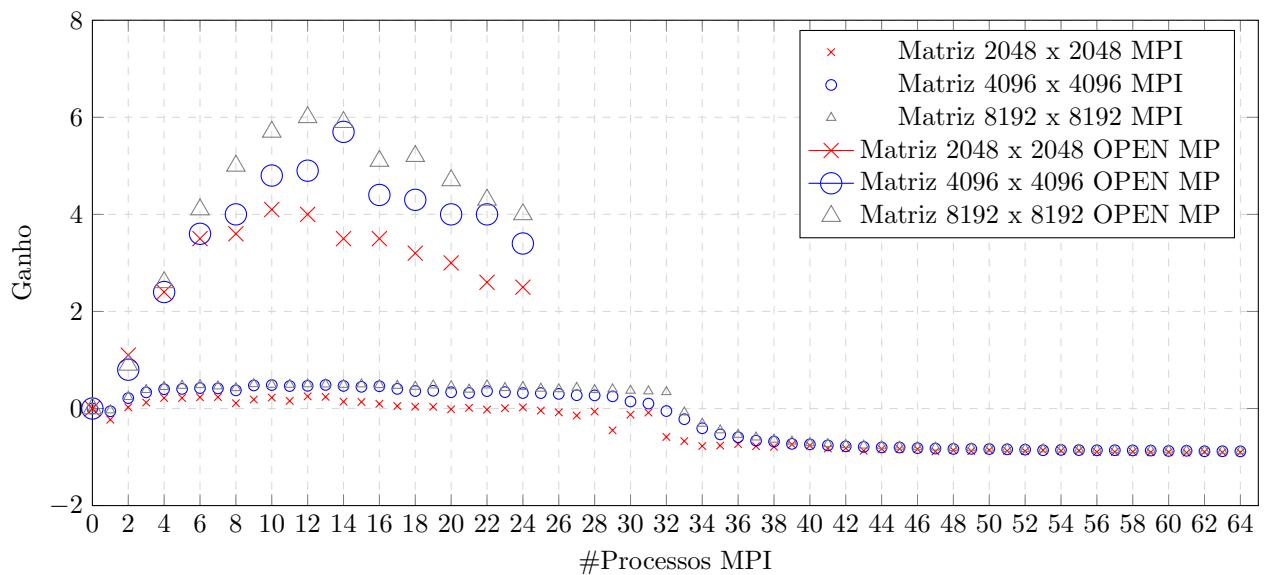
Figura 9: Evolução do tempo total para execução do algoritmo paralelo (MPIReduce) vs algoritmo paralelo com Computação Extra , para matrizes de 8192*8192.

Apesar da redução da necessidade de sincronismo entre processos MPI durante a execução do algoritmo, o esforço de computação extra aplicado não se revelou suficiente para reduzir o tempo total da solução.

Denote ainda que os tempos de execução, ao invés de sofrerem uma redução, sofreram um aumento, com a ainda penalidade de um aumento da instabilidade dos tempos do algoritmo. Esta solução, ao contrário do pensado inicialmente deverá ser portanto descartada, sendo que o algoritmo escolhido permanece o com redução e o método MPI_Scatter.

5.6 Relação de ganho entre código sequencial, versão paralela OpenMP e versão paralela MPI (comunicação entre nodos r641 por ethernet)

Uma vez que outra versão do algoritmo em memória partilhada foi já implementada e testada devemos comparar ambas através do gráfico do ganho de ambas as soluções quando comparadas com a versão sequencial.



5.7 Interpretação do Speedup do algoritmo final de memória distribuída

Como se pode confirmar pelos valores apresentados os programas paralelizados recorrendo a openMPI nem sempre escalam como esperado.

Atente na relação de ganho – denote que o tempo de computação foi, no máximo reduzido para aproximadamente metade (com 10 processos MPI), sendo que para soluções que recorram a um número de processos superior a 32 (passamos a incluir na solução 2 nodos e a respetiva penalização de comunicação) apresentam resultados em termos de tempo computação inferiores à solução sequencial.

Tal deve-se ao próprio overhead gerado pelas diretivas MPI, e pela necessidade de sincronização entre processos.

A latência de comunicação via Ethernet torna inviável qualquer tipo de solução que ultrapasse 1 nodo de computação. Dada a impossibilidade de testar a presente solução com comunicação entre processos via Myrinet nada podemos concluir relativamente ao algoritmo nessa tecnologia de comunicação.

Devemos portanto analisar o comportamento do algoritmo num ambiente híbrido – em que cada processo MPI terá um número de threads OpenMP.

De momento, a melhor solução encontrada para o problema passa pelo algoritmo de memória partilhada para 12 threads OpenMP.

5.8 Evolução do tempo para a solução para (comunicação entre nodos r641 por Ethernet) – através da expansão do número de nodos

Por forma a testarmos todas as possibilidades do nosso algoritmo decidimos ainda aumentar o número de processos MPI presentes na solução. Estendemos os testes até 5 nodos de computação.

Recorrendo aos métodos de medição de tempo anteriormente referidos obtivemos os seguintes tempos (em mili-segundos) para a solução do algoritmo paralelo (MPIReduce) com expansão do número de nodos, para matrizes de 8192*8192.

Denote que os valores apresentados correspondem ao melhor valor de uma amostra de 10 com 5% de tolerância, para cada tipo de matriz, tendo sido o trabalho submetido nos nodos compute-641, com comunicação entre nodos via Ethernet.

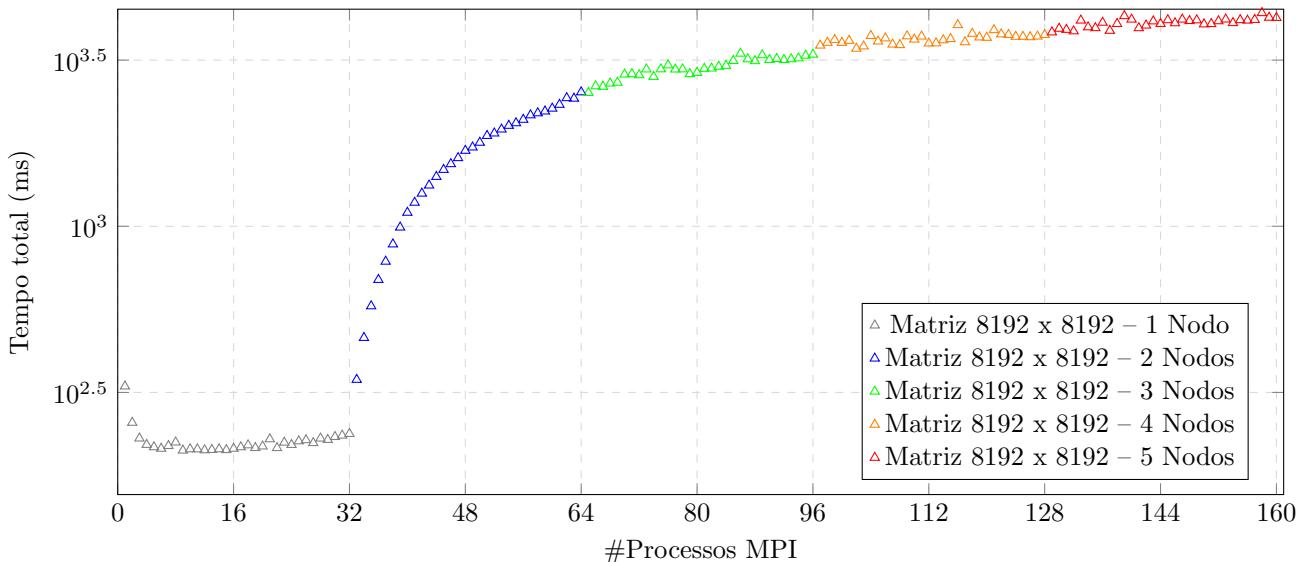


Figura 10: Evolução do tempo total para a solução através da expansão do número de nodos para a Matriz 8192 x 8192

6 Algoritmo Híbrido

Tal como foi enumerado anteriormente, resta-nos analisar as possíveis melhorias do algoritmo para uma versão híbrida.

A cada processo MPI teremos agora associado um número de threads openMP, sendo que assim alcançamos paralelismo com a adição de nodos e no próprio nodo em si por diretivas openMP.

Toda as alterações ao código do algoritmo podem ser consultadas no [appendix F na page 22](#).

Recorrendo aos métodos de medição de tempo anteriormente referidos obtivemos os seguintes tempos (em mili-segundos) para execução do algoritmo paralelo (MPIReduce) com diretivas MPI e openMP, para matrizes de 8192*8192.

Denote que os valores apresentados correspondem ao melhor valor de uma amostra de 10 com 5% de tolerância, para cada tipo de matriz,tendo sido o trabalho submetido nos nodos compute-641, com comunicação entre nodos via Ethernet.

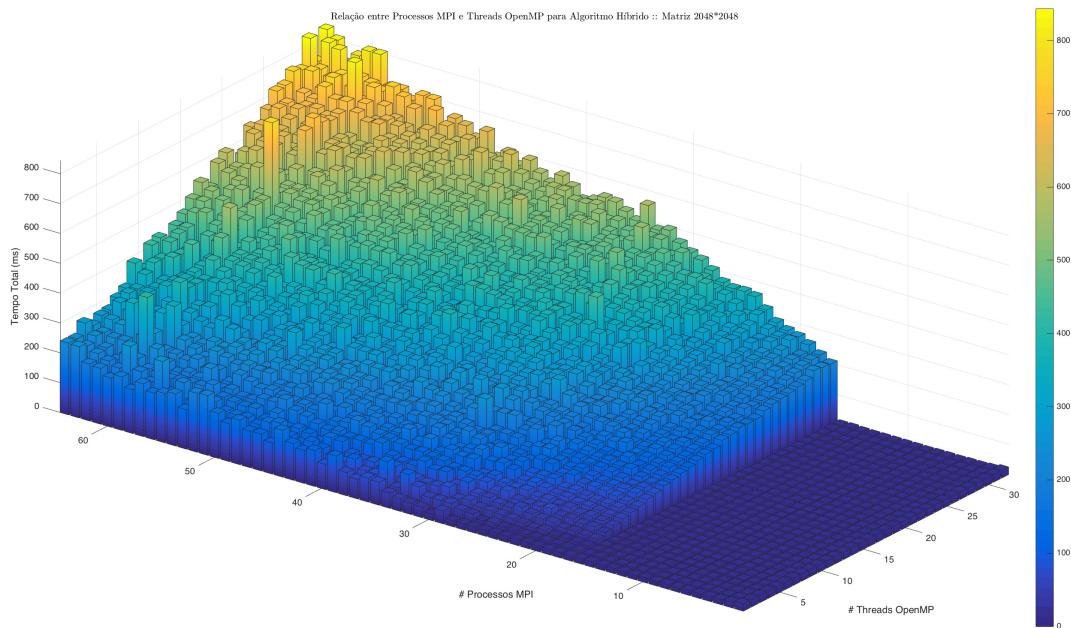


Figura 11: Análise da evolução do tempo total para solução através do aumento quer do número de processos MPI, quer do número de threads openMP para o algoritmo híbrido, para matrizes de tamanho 2048*2048, para nodos r641 com comunicação via Ethernet.

6.1 Interpretação do Speedup do algoritmo final de memória distribuída

Como se pode confirmar pelos valores apresentados os programas paralelizados recorrendo a openMPI e openMP mantêm a inviabilidade da solução. Denote que essa mesma inviabilidade se torna visível nos próprios nodos para um número de processos MPI superior a 16. É a partir desse número de processos que a solução incorpora os 2 processadores presentes no nodo. O aumento exponencial visível a partir desse ponto mesmo com a inclusão de diretivas openMP deve-se à possível inviabilização de dados para threads no mesmo processador e overhead por data races e troca de mensagens para processos e threads no mesmo nodo.

Para um número de processos superior a 32, uma vez que necessitamos de 2 nodos para solução, todo o overhead presente na solução MPI é agora exponenciado pelo overhead openMP.

Dada a impossibilidade de testar a presente solução com comunicação entre processos via Myrinet nada podemos concluir relativamente ao algoritmo nessa tecnologia de comunicação.

Podemos então concluir que o melhor algoritmo paralelo, com base nos dados e equipamentos disponíveis utilizados para os testes, é o algoritmo de memória partilhada que obtém o melhor ganho para um número de threads OpenMP igual a 12. Contudo, deverá ser realizada uma análise futura tanto aos algoritmos MPI e híbrido aquando da viabilidade de utilização da myrinet por parte do mpirun.

6.2 Testes do algoritmo híbrido para o team laptop

Recorrendo aos métodos de medição de tempo anteriormente referidos obtivemos os seguintes tempos (em mili-segundos) para a execução do algoritmo híbrido, para matrizes de 2048*2048, 4096*4096 e 8192*8192.

Denote que os valores apresentados correspondem ao melhor valor de uma amostra de 10 com 5% de tolerância, para cada tipo de matriz,tendo sido o trabalho no computador pessoal da equipa.

Dada a semelhança do comportamento do algoritmo para os diferentes tamanhos de matrizes, apresenta-se apenas neste secção o caso de matrizes de tamanho 8192*8192, sendo que para os 2 tamanhos de matrizes

restantes, estão disponíveis para análise os respectivos gráficos no [appendix K na page 27](#).

Contudo esta análise está limitada em termos de possíveis comparações com o mesmo algoritmo executados nos nodos compute-641 do Search6, uma vez que, no caso do team laptop, os tempos registados apenas possibilitam a análise de performance do algoritmo para 1 nodo com no máximo 8 processos MPI e 8 threads openMP.

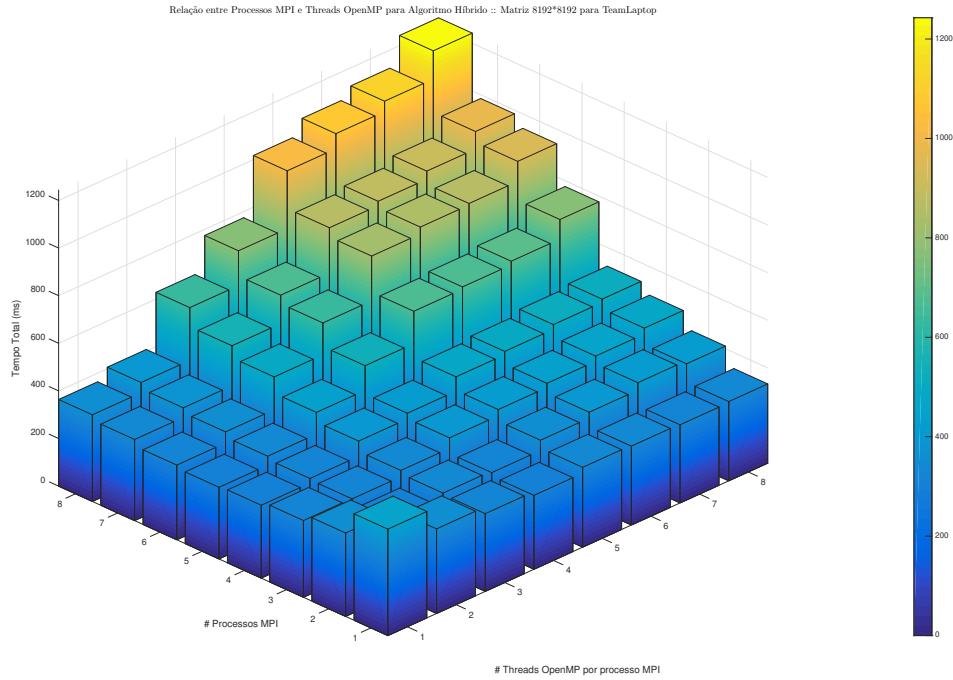


Figura 12: Análise da evolução do tempo total para solução através do aumento quer do número de processos MPI, quer do número de threads openMP para o algoritmo híbrido, para matrizes de tamanho 8192*8192, para o team laptop.

6.2.1 Análise do algoritmo híbrido para o team laptop

Tal como pode ser analisado nos gráficos anteriores, um aumento conjunto do número de processos MPI e do número de processos openMP, resulta numa diminuição de performance do algoritmo.

A melhor solução encontrada, para os 3 diferentes tamanhos de matrizes analisados, corresponde à utilização de 2 processos MPI, sendo que a cada processo MPI estarão também associados 2 threads openMP.

Denote que qualquer outra solução que inclui o aumento quer do número do número de processos MPI, quer do número de threads openMP resulta num decréscimo de performance do algoritmo relativamente ao tempo total da solução.

É expectável que o ganho da solução híbrida não seja relevante e até mesmo nulo para um grande número de threads openMP e processos MPI, uma vez que, apesar de podermos incluir na solução 8 processos MPI estes estão presentes em apenas 4 cores físicos, todos eles com memória interna do CPU partilhada (cache nível 3). Outra razão para a possível não escalabilidade do algoritmo em processadores da gama comercial do team laptop deve-se ao facto do número de registos disponíveis para cada processo ser também reduzido. Denote que, apesar do algoritmo ser implementado em memória distribuída, a nível físico a memória é partilhada. Assim sendo, e dado que para o caso do team laptop, a cache L3 não é suficiente para comportar completamente nenhuma das matrizes em estudo, para além do overhead do algoritmo comprovado também nos nodos do Search 6, temos agora também o overhead gerado por situações de cache-miss e data races – por si só suficientes para invalidar escalabilidade de algoritmos paralelos.

Em comparação com o algoritmo de memória partilhada, este é claramente inferior para o caso do team laptop, com o acréscimo de ser ainda inferior ao algoritmo sem qualquer tipo de paralelismo.

A Caracterização do hardware das plataformas de computação

Sistema	nodos compute-641	team laptop
# CPUs	2	1
CPU	Intel® Xeon® X5650 v2	Intel® Core™ i7-3635QM
Micro-Arquitetura	Ivy Bridge	Ivy Bridge
# Cores p/ CPU	8	4
# Threads p/ CPU	16	8
Freq. Clock	2.6 GHz	2.4 GHz
Cache L1	192KB 32KB por core	128KB 32KB por core
Cache L2	1536KB 256KB por core	1024KB 256KB por core
Cache L3	20MB shared	6144KB partilhada
Inst. Set Ext.	SSE4.2 & AVX	SSE4.2 & AVX
#Memory Channels	4	2

Tabela 3: Caracterização do Hardware das plataformas de teste

B Excerto :: Geração de Matrizes

```
long long int * initial_image , * final_image ;
void fillMatrices ( long long int total_pixels ) {
    initial_image = (long long int*) malloc(total_pixels * sizeof ( long long int ) );
    final_image = (long long int*) malloc(total_pixels * sizeof ( long long int ) );
    for ( long long int pixel_number = 0; pixel_number < total_pixels; ++pixel_number ) {
        initial_image [pixel_number] = ((( int ) rand ()) % (( int ) 255));
    }
}
```

1
2
3
4
5
6
7
8
9
10
11

C Métodos do Algoritmo Reduce

C.1 Método calculate_histogram ()

```
void calculate_histogram ( ) {
    hist_call_time = MPI_Wtime();
    MPI_Scatter ( initial_image , elements_per_worker , MPI_INT , worker_initial_image , elements_per_worker , MPI_INT , M
    for ( long long int pixel_number = 0; pixel_number < elements_per_worker ; ++pixel_number) {
        worker_local_histogram[ worker_initial_image[pixel_number] ]++;
    }
    // Reduce all partial histograms down to the root process
    MPI_Reduce( worker_local_histogram , histogram , HIST_SIZE , MPI_INT , MPI_SUM , MASTER , MPI_COMM_WORLD );
    hist_exit_time = MPI_Wtime();
}
```

C.2 Método calculate_accum ()

```
void calculate_accum ( long long int total_pixels ){
    accum_call_time = MPI_Wtime();
    if ( process_id == MASTER ){
        int accumulated_value = 0;
        for ( unsigned i = 0 ; i < HIST_SIZE ; i++ ){
            accumulated_value += histogram[i];
            histogram_accumulated[i] = accumulated_value * 255.0f / total_pixels ;
        }
    }
    MPI_Bcast( histogram_accumulated , HIST_SIZE , MPI_FLOAT , MASTER , MPI_COMM_WORLD );
    accum_exit_time = MPI_Wtime();
}
```

C.3 Método transform_image ()

```
void transform_image( ){
    transform_call_time = MPI_Wtime();
    for ( long long int pixel_number = 0; pixel_number < elements_per_worker; ++pixel_number ) {
        worker_final_image[pixel_number] = ( int )( histogram_accumulated [ worker_initial_image[pixel_number] ] );
    }
    // Gather all partial images down to the root process
    MPI_Gather( worker_final_image , elements_per_worker , MPI_INT , final_image , elements_per_worker , MPI_INT , ←
                MASTER , MPI_COMM_WORLD );
    transform_exit_time = MPI_Wtime();
}
```

D Métodos do Algoritmo Gather

D.1 Método calculate_histogram ()

```
void calculate_histogram () {
    hist_call_time = MPI_Wtime ();
    MPI_Scatter ( initial_image , elements_per_worker , MPI_INT , worker_initial_image , elements_per_worker , MPI_INT , ←
        MASTER , MPI_COMM_WORLD );
    for ( long long int pixel_number = 0; pixel_number < elements_per_worker ; ++pixel_number ) {
        worker_local_histogram[ worker_initial_image[pixel_number] ]++;
    }
    if ( process_id == MASTER ){
        master_histograms = (int*) malloc( number_processes * HIST_SIZE * sizeof ( int ) );
    }
    // Gather all partial histograms down to the root process
    MPI_Gather( worker_local_histogram , HIST_SIZE , MPI_INT , master_histograms , HIST_SIZE , MPI_INT , MASTER , ←
        MPI_COMM_WORLD );
    MPI_Barrier( MPI_COMM_WORLD );
    if ( process_id == MASTER ){
        for ( int pos_hist = 0; pos_hist < HIST_SIZE; ++pos_hist ){
            for ( int p_num = 0; p_num < number_processes; ++p_num ){
                histogram[pos_hist] += master_histograms[p_num*pos_hist];
            }
        }
    }
    hist_exit_time = MPI_Wtime ();
}
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

E Métodos do Algoritmo Extra Computação

E.1 Método calculate_histogram ()

```
void calculate_histogram ( long long int total_pixels ) {
    hist_call_time = MPI_Wtime();

    MPI_Bcast( initial_image , total_pixels , MPI_INT , MASTER , MPI_COMM_WORLD );
    for ( long long int pixel_number = 0; pixel_number < total_pixels ; ++pixel_number ) {
        worker_local_histogram[ initial_image[pixel_number] ]++;
    }
    hist_exit_time = MPI_Wtime();
}
```

1
2
3
4
5
6
7
8
9

E.2 Método calculate_accum ()

```
void calculate_accum ( long long int total_pixels ) {
    accum_call_time = MPI_Wtime();
    int accumulated_value = 0;
    for ( unsigned i = 0 ; i < HIST_SIZE ; i++ ){
        accumulated_value += histogram[i];
        histogram_accumulated[i] = accumulated_value * 255.0f / total_pixels ;
    }
    accum_exit_time = MPI_Wtime();
}
```

1
2
3
4
5
6
7
8
9

E.3 Método transform_image ()

```
void transform_image( ){
    transform_call_time = MPI_Wtime();
    int range_min;
    range_min = elements_per_worker*process_id;

    for ( long long int pixel_number = 0; pixel_number < elements_per_worker ; ++pixel_number ) {
        worker_final_image[pixel_number] = ( int )( histogram_accumulated [ initial_image[pixel_number+range_min] ] );
    }
    // Gather all partial images down to the root process
    MPI_Gather( worker_final_image , elements_per_worker , MPI_INT , final_image , elements_per_worker , MPI_INT , ←
                MASTER , MPI_COMM_WORLD );
    transform_exit_time = MPI_Wtime();
}
```

1
2
3
4
5
6
7
8
9
10
11
12

F Métodos do Algoritmo Híbrido

F.1 Método calculate_histogram ()

```

void calculate_histogram ( int thread_count ) {
    hist_call_time = MPI_Wtime();
    MPI_Scatter ( initial_image , elements_per_worker , MPI_INT , worker_initial_image , elements_per_worker , MPI_INT , ←
        MASTER , MPI_COMM_WORLD );
#pragma omp parallel num_threads( thread_count )
{
    int thread_id = omp_get_thread_num();
    int thread_histogram[MAX_THREADS][HIST_SIZE];
#pragma omp for nowait schedule (static)
    for ( long long int pixel_number = 0; pixel_number < elements_per_worker ; ++pixel_number ) {
        thread_histogram[thread_id][worker_initial_image[pixel_number]]++;
    }
    for ( unsigned pos_hist_local = 0; pos_hist_local < HIST_SIZE; ++pos_hist_local ) {
#pragma omp atomic
        worker_local_histogram[pos_hist_local] += thread_histogram[thread_id][pos_hist_local];
    }
}
// Reduce all partial histograms down to the root process
MPI_Reduce( worker_local_histogram , histogram , HIST_SIZE , MPI_INT , MPI_SUM , MASTER , MPI_COMM_WORLD );
hist_exit_time = MPI_Wtime();
}

```

F.2 Método calculate_accum ()

```

void calculate_accum ( long long int total_pixels ){
    accum_call_time = MPI_Wtime();
    if ( process_id == MASTER ){
        int accumulated_value = 0;
        for ( unsigned i = 0 ; i < HIST_SIZE ; i++ ){
            accumulated_value += histogram[i];
            histogram_accumulated[i] = accumulated_value * 255.0f / total_pixels ;
        }
    }
    MPI_Bcast( histogram_accumulated , HIST_SIZE , MPI_FLOAT , MASTER , MPI_COMM_WORLD );
    accum_exit_time = MPI_Wtime();
}

```

F.3 Método transform_image ()

```

void transform_image( int thread_count ){
    transform_call_time = MPI_Wtime();
#pragma omp parallel num_threads( thread_count )
{
#pragma omp for nowait schedule (static)
    for ( long long int pixel_number = 0; pixel_number < elements_per_worker ; ++pixel_number ) {
        worker_final_image[pixel_number] = ( int )( histogram_accumulated [ worker_initial_image[pixel_number] ] );
    }
}
// Gather all partial images down to the root process
MPI_Gather( worker_final_image , elements_per_worker , MPI_INT , final_image , elements_per_worker , MPI_INT , ←
    MASTER , MPI_COMM_WORLD );
transform_exit_time = MPI_Wtime();
}

```

G Comparação entre os tempos de execução por método entre as versões sequencial e MPI

Tam. Matriz	Calc. Hist. Seq.	Calc. Hist. MPI.	Calc Histo. Acum. Seq.	Calc Histo. Acum. MPI	Calc. Img. Final Seq.	Calc. Img. Final MPI.	Total Seq. (ms)	Total MPI (ms)	Ganho MPI vs Seq
2048*2048	4.4025	10,1259	0,001	0,013113	13.075	14,4019	17.497	24,5459	-28,72%
4096*4096	16.7895	32,882	0,002	0,0109673	52.1625	49,5799	68.954	82,4769	-16,40%
8192*8192	65.933	130,181	0,002	0,0100136	208.811	199,688	274.768	329,884	-16,71%

Tabela 4: Comparação entre os tempos de execução por método entre as versões sequencial e MPI para os vários tamanhos de matrizes

H Comparação do tempo total para a solução entre o algoritmo com MPI_Gather e MPI_Barrier no método calculate_histogram e o algoritmo com redução

# Processos	Tempo Total (ms) do Alg. com MPI_Gather e MPI_Reduce	Tempo Total (ms) do Alg. com MPI_Reduce	% Ganho Alg. MPI_Gather vs Alg. MPI_Reduce
1	331.035	329.884	-0.35%
2	257.304	256.506	-0.31%
3	231.305	230.065	-0.54%
4	220.627	220.091	-0.24%
5	217.911	216.555	-0.62%
6	216.487	214.334	-0.99%
7	221.153	218.328	-1.28%
8	228.372	223.812	-2.00%
9	212.169	211.556	-0.29%
10	214.88	213.525	-0.63%
11	214.331	213.442	-0.41%
12	216.538	212.024	-2.08%
13	214.258	212.485	-0.83%
14	216.328	213.513	-1.30%
15	214.681	212.274	-1.12%
16	216.396	214.28	-0.98%
17	227.361	216.517	-4.77%
18	228.448	219.458	-3.94%
19	228.481	215.363	-5.74%
20	227.974	217.697	-4.51%
21	220.323	228.835	3.86%
22	219.57	215.285	-1.95%
23	230.396	223.594	-2.95%
24	231.532	219.973	-4.99%
25	229.327	225.589	-1.63%
26	228.461	227.351	-0.49%
27	231.868	222.866	-3.88%
28	230.468	229.964	-0.22%
29	229.896	227.787	-0.92%
30	230.24	232.439	0.96%
31	233.987	234.744	0.32%
32	241.482	237.46	-1.67%
33	320.417	345.436	7.81%
34	446.836	461.783	3.35%
35	580.779	575.023	-0.99%
36	680.564	690.33	1.43%
37	804.115	783.386	-2.58%
38	897.399	882.84	-1.62%
39	1013.49	992.309	-2.09%
40	1090.69	1099.36	0.79%
41	1159.84	1178.42	1.60%
42	1247.75	1255.22	0.60%
43	1321.87	1327.58	0.43%
44	1422.18	1409.17	-0.91%
45	1468.93	1479.57	0.72%
46	1545.58	1539.56	-0.39%
47	1622.38	1605.24	-1.06%
48	1664.33	1688.84	1.47%
49	1739.39	1728.39	-0.63%
50	1793.57	1785.75	-0.44%
51	1841.76	1872.11	1.65%
52	1908.34	1904.53	-0.20%
53	1941.07	1957.33	0.84%
54	2009.97	2007.66	-0.11%
55	2045.78	2044.17	-0.08%
56	2094.81	2093.59	-0.06%
57	2134.81	2159.09	1.14%
58	2182.38	2190.05	0.35%
59	2228.98	2217.46	-0.52%
60	2299.88	2261.72	-1.66%
61	2342.33	2322.73	-0.84%
62	2406.22	2432.18	1.08%
63	2450.41	2423.28	-1.11%
64	2608.26	2532.18	-2.92%

Tabela 5: Comparação do tempo total para a solução entre o algoritmo com MPI_Gather e MPI_Barrier no método calculate_histogram e o algoritmo com redução

I % de Tempo de Computação vs % de Tempo de Comunicação % para o Tempo Total de Execução (ms) para as Matrizes 2048 x 2048 e 4096 x 4096

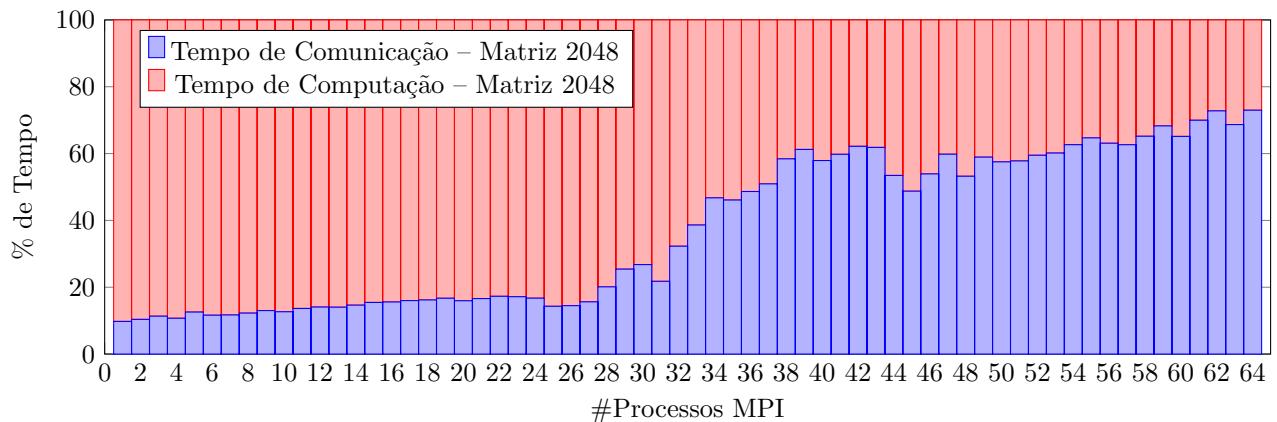


Figura 13: % de Tempo de Computação vs % de Tempo de Comunicação % para o Tempo Total de Execução (ms) para a Matriz 2048 x 2048

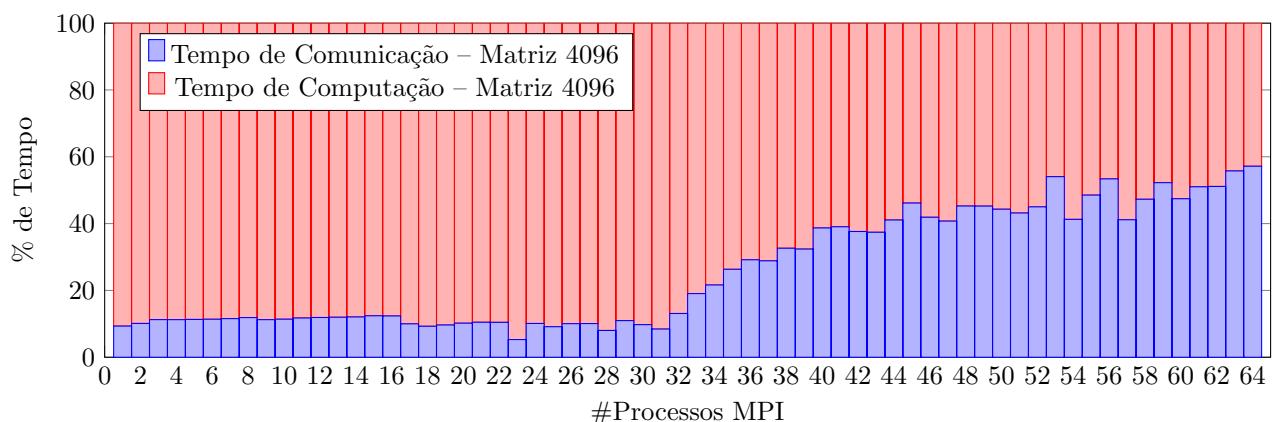


Figura 14: % de Tempo de Computação vs % de Tempo de Comunicação % para o Tempo Total de Execução (ms) para a Matriz 4096 x 4096

J Evolução da % do tempo de métodos para o tempo total da solução – através da expansão do número de processos

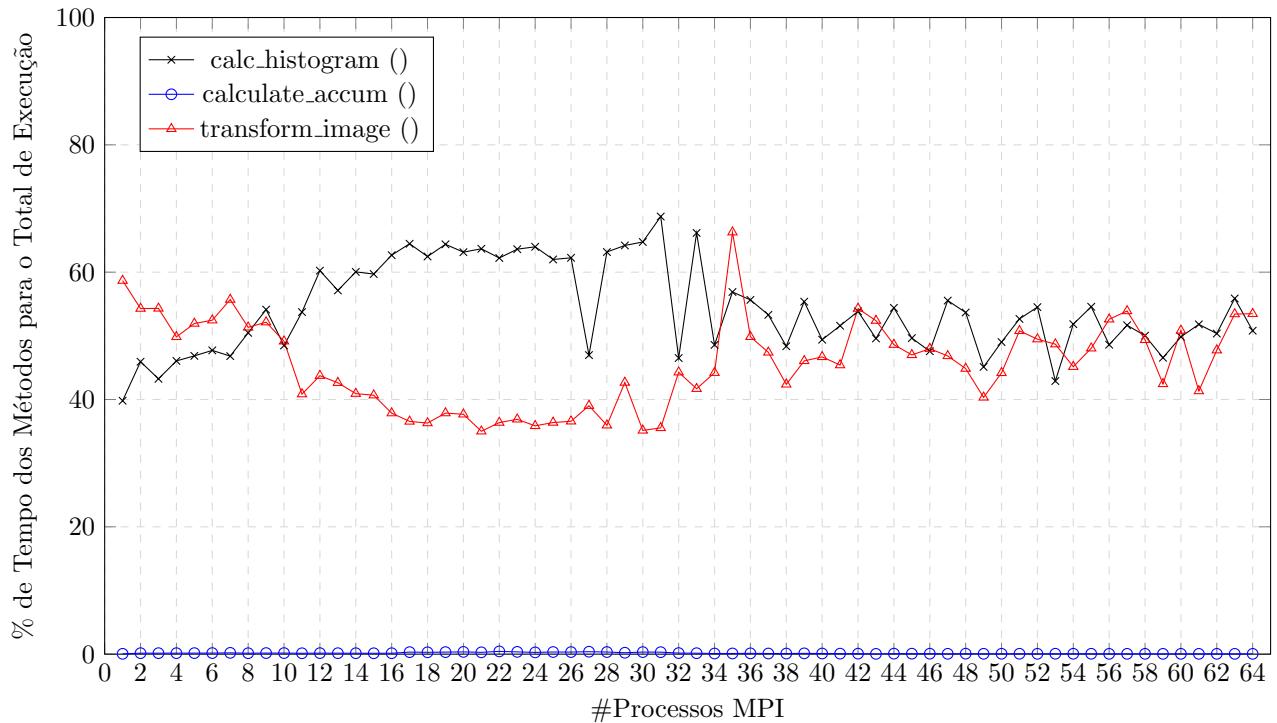


Figura 15: % de Tempo dos Métodos para o Total de Execução para a Matriz 2048 x 2048

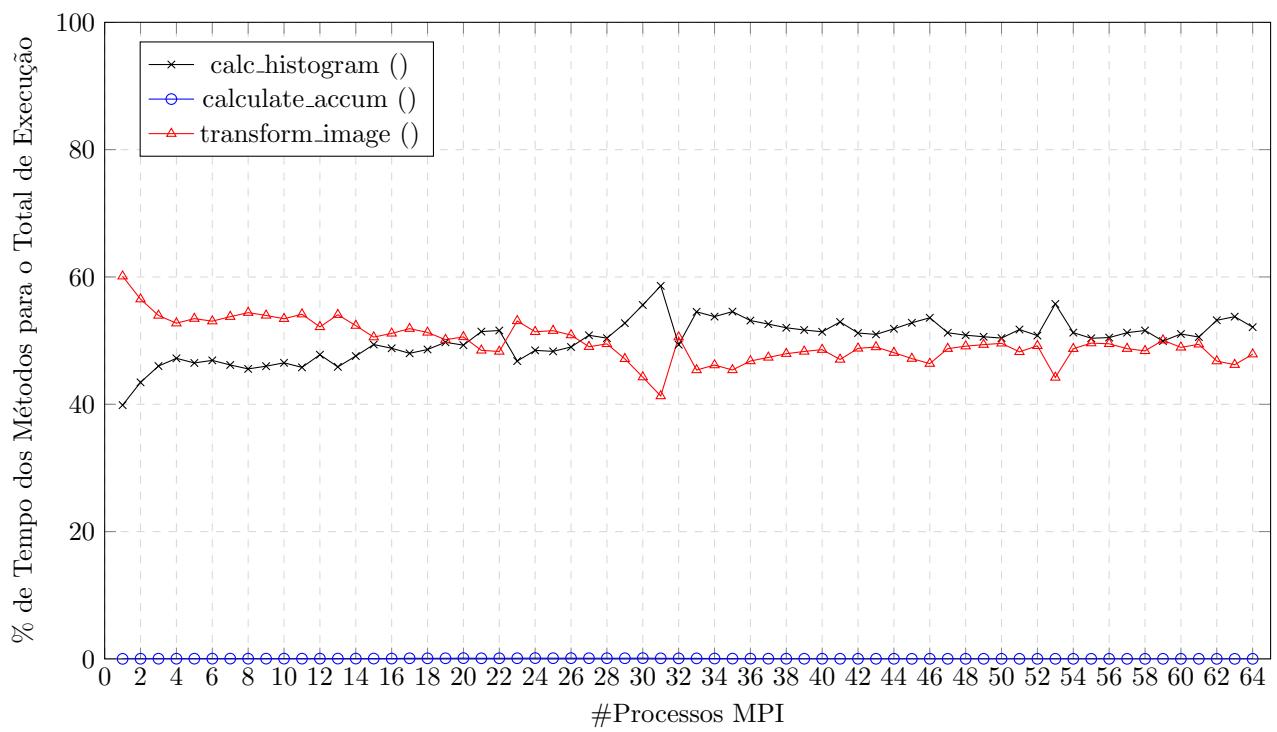


Figura 16: % de Tempo dos Métodos para o Total de Execução para a Matriz 4096 x 4096

K Evolução do tempo total para solução através do aumento quer do número de processos MPI, quer do número de threads openMP para o algoritmo híbrido, para matrizes de tamanho 2048*2048 e 4096*4096, para o team laptop

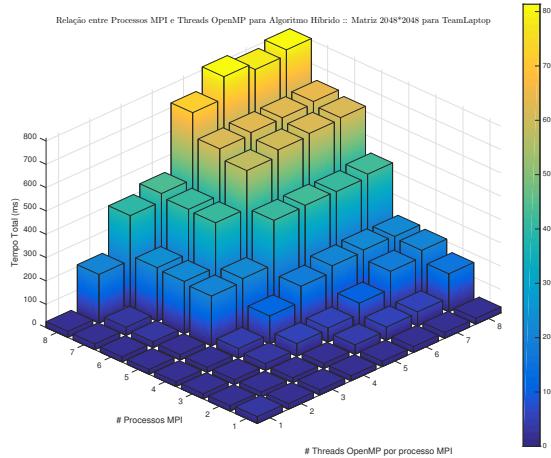


Figura 17: Análise da evolução do tempo total para solução através do aumento quer do número de processos MPI, quer do número de threads openMP para o algoritmo híbrido, para matrizes de tamanho 2048*2048, para o team laptop.

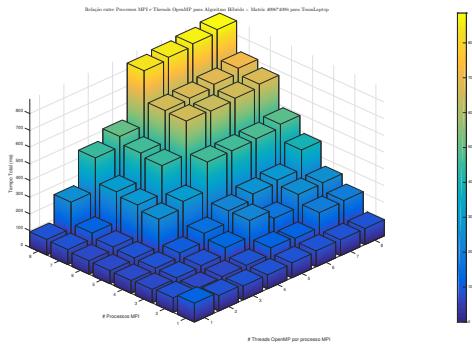


Figura 18: Análise da evolução do tempo total para solução através do aumento quer do número de processos MPI, quer do número de threads openMP para o algoritmo híbrido, para matrizes de tamanho 4096*4096, para o team laptop.