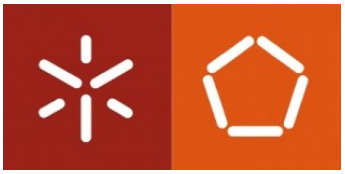
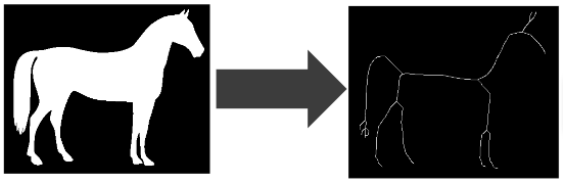


Universidade do Minho

Mestrado Integrado em Engenharia Informática

Algoritmo Esqueletização em memória distribuída

Implementação em openMPI



Paradigmas de Computação Paralela - 18/19

Luís Pedro Pereira Fonseca a60993
Filipe de Sousa Marques a57812

Conteúdo

1 Caso de Estudo

1.1 Esqueletização de uma imagem binária

O algoritmo em estudo será a esqueletização de uma imagem binária, que tal como o nome indica, permite o fabrico de um esqueleto representativo de uma imagem original a preto e branco. Para alcançar esse efeito, a imagem será iterada para se proceder à remoção dos pixeis em excesso. O resultado final será uma linha com 1 pixel de espessura. Para proceder à paralelização do algoritmo, utilizaremos a linguagem C++, e o protocolo de comunicação OpenMPI, que permitirá distribuir os dados por diversos nós de computação e assim dividir o esforço computacional por diversas máquinas, resultando num menor tempo de execução do programa e/ou permitindo a execução do programa com inputs de maior dimensão mantendo o tempo de execução em relação à versão original do algoritmo não-paralelizada.

1.2 Descrição do algoritmo sequencial

Para a realização da versão sequencial optamos por seguir o mais fielmente possível os cálculos e verificações fornecidos no enunciado do trabalho.

Ao iniciar o cálculo, este prepara algumas variáveis para auxiliar no processamento da imagem, sendo que, as importantes de referir são, a variável a que chamaremos ch image (changing image), representando o espaço alocado com o mesmo tamanho da imagem original, ao qual se trata as bordas de padding internamente e, dois arrays de tamanho fixo a representar as localizações relativas dos vizinhos ao pixel pela ordem dada pela figura 1.

P_9	P_2	P_3
P_8	P_1	P_4
P_7	P_6	P_5

Figura 1: Ordem pré-definida dos vizinhos do pixel a calcular

Chegando ao ciclo principal, é contabilizado o número de iterações, representantes do número de travessias totais da imagem, sendo que será sempre um número par de travessias para poder ser atingida pelas condições de ambos os sentidos. Neste ciclo, os pixeis da imagem são seguidos sequencialmente, e por cada um, as seguintes verificações são feitas:

- caso o valor do pixel seja 0 (representando um pixel preto), na mesma posição da ch image, é colocado o valor 0 na mesma sem mais algum cálculo realizado para este pixel;
- caso o valor do pixel seja 1 (representando um pixel a branco), é verificado se se trata de um pixel da borda da imagem e, no caso afirmativo, é retirado e marcado como pixel preto no ch image, caso contrário, continua como pixel branco.

A verificação do pixel como borda passa por:

- verificar se tem no máximo 6 vizinhos e no mínimo 2 que tenham o valor 1;
- verificar se, no sentido dos ponteiros do relógio, o número de transições dos vizinhos de 0 para 1 apenas acontece uma vez, isto examina que o pixel não represente um pixel de ligação entre partes da imagem;

- e, finalmente, examinando os valores de vizinhos específicos para testar, quer num sentido nas iterações ímpares, quer noutro, nas iterações pares, se o pixel é uma borda lateral, de uma secção da imagem não linear, e que retirar-lo não cortará a imagem em pedaços diferentes. As funções lógicas representativas deste teste podem ser vistos na imagem a seguir, retirada do enunciado do trabalho.

$$\overline{P_4} + \overline{P_6} + \overline{P_2}\,\overline{P_8} = 1 \qquad \overline{P_2} + \overline{P_8} + \overline{P_4}\,\overline{P_6} = 1$$

No fim de cada iteração, a imagem de onde se lê e a imagem para onde se escreve trocam de posição (faz-se swap dos apontadores), para continuar o ciclo, sendo que este só termina quando nenhum valor for removido durante uma iteração completa pela imagem, e o número de iterações for par. Este passo, é importante pois cada iteração é a base para a iteração seguinte.

1.3 Optimizações

1.3.1 Matriz uni-dimensional

Os arrays em c/c++ são guardados em memória como um bloco contíguo. Assim aceder a[x][y] é equivalente a aceder 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 facilitar o compilador no processo de optimização. Para o paralelismo e vetorização isto é uma mais valia uma vez que o compilador poderá tentar vetorizar código que anteriormente considerava não vetorizável. Por último podemos ainda referir o facto de nestes casos necessitarmos de menos sincronização entre processos/threads.

1.3.2 Alinhamento dos recursos de memória

De forma a não desperdiçar linhas de cache alinhamos as alocações de memória, para a imagem original e para a matriz resultante com a função memalign fazendo com que o apontador para esses chuncks de memória sejam múltiplos de 32. Com isto reduzimos as cache misses, e potencializamos as oportunidades de vectorização do código.

1.3.3 Redução nos acessos à memória e aproveitamento da L1

Tal como explicitado anteriormente, para cada pixel teremos que conhecer os seus vizinhos. Estando os mesmos em diferentes linhas da matriz, podemos ser tentados em extrair os mesmos para uma sub-matriz 3x3, num outro espaço de memória, para depois ser testada pelas funcões de teste e remoção do pixel. No entanto tal operação revela-se extremamente cara do ponto de vista computacional. Não só não tiramos partido da L1, como estamos a acrescentar um overhead desnecessário de leitura/escrita na memória para um array temporário. A solução adoptada foi a criação de dois arrays estáticos (XX-index e YY-index) com as coordenadas relativas a um dado pixel. Para conhecer os vizinhos de um pixel apenas precisaremos de saber as suas coordenadas, ao qual depois somamos os valores de XX-index e YY-index.

1.3.4 Eliminação de chamadas de funcões dentro do ciclo

Dentro do ciclo while principal, optou-se por desdobrar todas as funções de teste de remoção de pixel, eliminando qualquer overhead provocado pela inicialização de uma nova stack frame e mudança de contexto.

2 Paralelização do algoritmo

2.1 Profiling da versão sequencial

% time	cumulative seconds	seconds	calls	self ms/call	total ms/call	name
100.06	0.17	0.17	1	170.09	170.09	skeletonize_serial
0	0.17	0	1	0	0	output_ppm
0	0.17	0	1	0	0	__static_initialization_and_destruction_0
0	0.17	0	1	0	0	new_img
0	0.17	0	1	0	0	free_img
0	0.17	0	1	0	0	read_ppm
0	0.17	0	1	0	0	read_inp_image
0	0.17	0	1	0	0	write_out_image
0	0.17	0	1	0	0	verify_command_line

Tabela 1: Profiling em Gprof

Pela seguinte tabela obtida com gprof, executanto o programa para uma imagem de reduzidas dimensões, verificamos que a totalidade do tempo é consumido pela função skeletonize. Pela lei de Amdahl sabemos que existe um grande potencial para um speedup, se melhorar-mos os tempos de execução de skeletonize, uma vez que o programa não é limitado pelas restantes funções. Skeletonize será então o nosso foco.

2.2 Análise teórica das oportunidades de paralelização

Para paralelizar este algoritmo, a forma mais simples de atacar o problema será começar por inicialmente dividir equitativamente os pixeis da imagem pelos diversos processos. No entanto, para cada pixel a ser processado, será necessário conhecer os seus pixeis vizinhos, nomeadamente os pixeis imediatamente acima e abaixo do pixel a ser caracterizado. Para colmatar esta dependência de dados, cada processo terá de ter uma visão mais abrangente da sua porção de imagem, ou seja, além dos pixeis que se encontra a trabalhar/processar, precisará de conhecer as bordas da sua porção de imagem.

Após este passo inicial de divisão do trabalho, poderá dar-se início ao processamento de cada um dos pedaços da imagem.

Sabemos à partida que, devido à natureza do algoritmo, cada iteração depende sempre da anterior, logo de imediato deduzimos que ao fim de cada iteração haverá sempre algum tipo de sincronização entre processos intervenientes.

Uma das formas mais simples de o fazer seria cada processo, após finalizada a sua iteração sobre a porção de imagem, enviaria o seu trabalho para o processo principal, este reunia a totalidade da imagem e enviaria de novo um pedaço da imagem para cada processo avançar para a próxima iteração.

Esta operação de comunicação, no entanto, seria extremamente cara, pois peca pela excessiva centralização dos dados. Teríamos um único processo principal que receberia, em cada iteração, várias mensagens de tamanho significativo sequencialmente. Este bloqueio momentâneo para comunicações em cada iteração representaria um custo significativo nos tempos finais de execução, não só para tamanhos de imagens grandes, mas ainda mais quando trabalhamos com um grande número de processos. Daí ser necessário uma outra abordagem.

Duas soluções para este problema serão, 1) em vez de os processos comunicarem com um processo principal, comunicarem apenas com os processos vizinhos, na prática, em cada iteração o processo P apenas precisa de trocar mensagens com o processo de rank P-1 e P+1; 2) em vez de enviarmos a porção da imagem toda entre processos, trocamos apenas parte da informação que interessa aos processos vizinhos, ou seja, envia-se apenas as bordas da porção da imagem.

Esta abordagem permite-nos comunicações em paralelo entre processos, o que potencia bastante o número de processos que poderão ser usados simultaneamente.

Uma outra questão que se levanta é saber quando é que cada um dos processos termina o seu trabalho, sabendo à partida que o seu trabalho depende do que foi feito pelos seus processos vizinhos.

Sabemos do algoritmo de esqueletização que o trabalho dá-se por terminado quando mais nenhum pixel for apagado. Por isso, em cada iteração, cada processo terá de informar os outros se ainda tem pixeis para apagar ou não. Um broadcast do tipo All to All será necessário, ao final de cada iteração.

Quando todos os processos terminarem, cada processo fará dispose do seu trabalho para o processo principal que se encarregará depois de guardar o resultado em disco.

2.3 Análise teórica dos tempos de comunicação

Cada processo ficará com uma fatia da imagem e apenas as bordas de cada fatia serão comunicadas em cada iteração. Denominemos tempo de comunicação de uma mensagem como $tl + tb * nbytes$, ou seja, tempo de latência mais o tempo que cada byte demora a ser transmitido. Cada fatia da imagem terá de comunicar duas mensagens por iteração da imagem, exceptuando o processo de rank 0 e o último processo que ficarão com o bloco do topo e do fundo da imagem, respetivamente, tendo estes apenas que trocar uma mensagem por iteração. Se o total de fatias for dado por P, o tamanho da linha for W, e o número de iterações i, teremos

$$Tcom = i * (2(P-2)(tl + tb * W) + 2(tl + tb * W))$$

No entanto, como todos os processos comunicarão em paralelo ao fim de cada iteração, os tempos de comunicação de cada processo não são acumuláveis. Logo simplificamos para

$$Tcom = 2 * i * (tl + tb * W)$$

Teremos também, em cada iteração, de fazer um reduce All to All, que implica fazer um reduce para o rank 0 (com custo log P) seguido de um broadcast (também com custo log P), logo

$$Tcom = i * (2(tl + tb * W) + 2 * tl * logP)$$

Ao valor gasto em comunicações durante o processamento da imagem teremos de somar os gastos com o scatter inicial e o gather final. O scatter inicial será feito paralelamente do processo principal com rank 0 para os demais, o que se traduz como

$$\begin{cases} blockSize = W * W/P \\ Tsca = blockSize * tb + tl \end{cases}$$

O gather final será feito sequencialmente, do rank 1 até rank P-1, o que implica que os gastos comunicativos de cada processo serão cumulativos.

$$Tgat = (blockSize * tb + tl) * (P-1)$$

Teremos também que ter em consideração os tempos de preparação das interfaces de rede para escrita/leitura, ao qual chamaremos tempo de setup, Tset.

Sabemos que, para o caso particular em que a imagem a ser processada representa um círculo, o número de iterações i poderá ser deduzido a partir de W, logo poderemos transformar a equação de cima, num sistema de 3 variáveis, cujo comportamento pode ser observado pelo seguinte gráfico.

$$\begin{cases} i = W/sqrt(2) \\ Tcom = i * (2 * (tl + tb * W) + 2 * tl * logP) + Tsca + Tgat + Tset \end{cases}$$

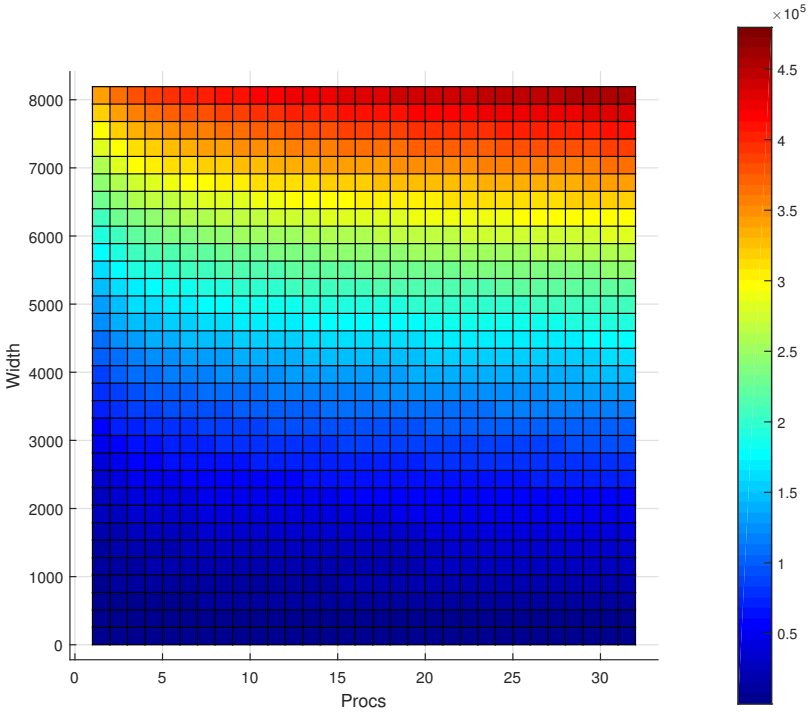


Figura 2: Análise teórica dos tempos de comunicação (em micro-segundos). Impacto nos tempos finais com o aumento de número de processadores e do aumento do tamanho de imagem. Pode-se observar que, como as mensagens são na sua maioria trocadas paralelamente, o aumento no número de processadores não terá impacto significativo nos tempos finais de comunicação. Os dados apresentados foram calculados tendo como base o tecido de ligação myrinet, com largura de banda 10Gbps e latência 3 ms.

2.4 Implementação do algoritmo paralelo

O algoritmo paralelo terá na sua essência quatro fases:

1. Scatter da imagem
2. Troca de mensagens das linhas das bordas
3. Broadcast All to All variável 'cont'
4. Gather da imagem

O scatter é feito apenas no início, e o gather no final. A troca de linhas e o broadcast de 'cont' é feito por cada iteração da imagem.

Começamos por distribuir pedaços da imagem pelos processos. Inicialmente será o rank 0 a ler do disco, e será este processo a fazer scatter da imagem para os outros processos. Terá de se ter em atenção as dependências de dados, pelo que cada processo receberá não só os pixeis onde tem que trabalhar mas também duas linhas de 1 pixel de espessura e W de largura, para as bordas do pedaço da imagem de cima e de baixo, respetivamente.

Estando os dados distribuídos, dá-se início ao processamento de cada uma das fatias da imagem. Para o processamento, em cada iteração, utilizamos uma versão modificada do algoritmo skeletonize usado em memória partilhada. Esta versão simplificada, apenas processa a imagem uma única iteração e devolve um apontador para uma nova imagem já processada. A funcão main decidirá se o processamento continua ou não e chamará a skeletonize de novo, se a imagem não estiver finalizada.

Quando o processo P termina uma iteração no seu pedaço da imagem, enviará as linhas superior e inferior da sua porção da imagem ao processo de rank P-1 e rank P+1, respetivamente.

Após esta troca de mensagens entre processos vizinhos será feito o broadcast da variável 'cont', que informará todos os outros processos se já terminou. O algoritmo só terminará quando todos os processos terminarem, daí neste broadcast ter-se usado um Reduce All to All, de forma a que todos os processos tenham conhecimento de que todos já terminaram.

Trocados por todos os seus valores de 'cont', o processo decide se continua ou não. Quando termina será feito então o envio do seu trabalho para o processo principal rank 0. Esta operação de gathering é feita sequencialmente à medida que os processos vão terminando a sua parte do processamento vão enviando o resultado final ao processo principal.

No final teremos a imagem processada na rank 0.

3 Ambiente de testes

3.1 Plataformas de teste

Para os testes utilizamos uma das nossas máquinas pessoais (portátil) e vários nós do Cluster do DI, do tipo 641. Equipados com dois Xeon E5-2650v2, arquitectura Ivy Bridge, 16 cores físicos no total, 32 virtuais. Em termos de cache, tem 20MB de L3 partilhada, 256K de L2 exclusiva a cada core (totalizando 4MB por chip) e 64K de cache L1, dividida em 32K para dados e 32K para código. Cada linha de cache tem 64 Bytes. Em anexo na tabela 3 podemos ver uma descrição mais pormenorizada de cada máquina.

3.2 Métricas

A metodologia utilizada para o cálculo de resultados foi o k-best, com k=3 de 10 amostras, e feita a média desses 3 resultados, que permite eliminar os valores excessivamente desviados do resultado real/expectável num cenário real de uso.

As medicões não incluem I/O de leitura e escrita de ficheiros. Para medir os tempos foram usadas as funções disponibilizadas pelo openMP para cronometragem. Todos os tempos foram registados em micro-segundos.

3.3 Inputs utilizados

Um script de submissão para o Cluster correu extensivamente cada uma das versões do kernel de forma iterativa. Para cada uma das suas versões correu imagens de diferentes tamanhos, várias vezes, para extracção do k-best score. As imagens de input utilizadas foram círculos brancos de fundo preto, de diâmetro igual à largura da imagem.

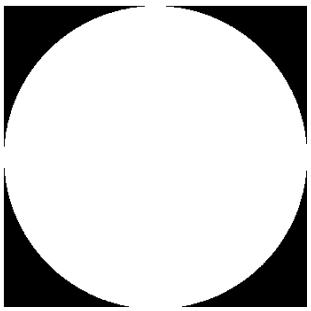


Figura 3: Exemplo de um input. Um círculo de 1024 de diâmetro será representado por uma imagem 1024x1024 de 1048576 pixels. Assumindo 1 pixel = 4 bytes em memória, necessitaremos de 4MB para armazenar a sua matriz.

3.3.1 Tabela inputs

disk size	Width*Height	malloc space
128K	horse 402*330	1MB
256K	circle 512*512	2MB
1M	circle 1024*1024	8MB
2M	circle 1536*1536	18.5MB
4M	circle 2048*2048	32MB
16M	circle 4096*4096	131MB
32M	circle 6144*6144	300MB
64M	circle 8192*8192	520MB

Tabela 2: Lista dos inputs utilizados para teste. Cada tamanho de imagem foi escolhida de forma a encaixar em diferentes níveis de cache. O espaço calculado para malloc engloba o espaço alocado para a leitura da imagem do disco mais o espaço necessário para alocar a matriz auxiliar para onde os pixeis serão transferidos em cada iteração

3.4 Compilador utilizado e optimizações

Foram compiladas várias versões dos executáveis com g++ 5.3 e 6.5, ambos com nível de optimização O3. As flags de compilação foram:

-O3 -std=c++11 -march=native -fopenmp

Foi também utilizada a versão 5.4.1 do papi para extração de algumas métricas extra na execução de cada kernel.

4 Análise dos resultados

4.1 Resultados em Nodes 641 do cluster com myrinet

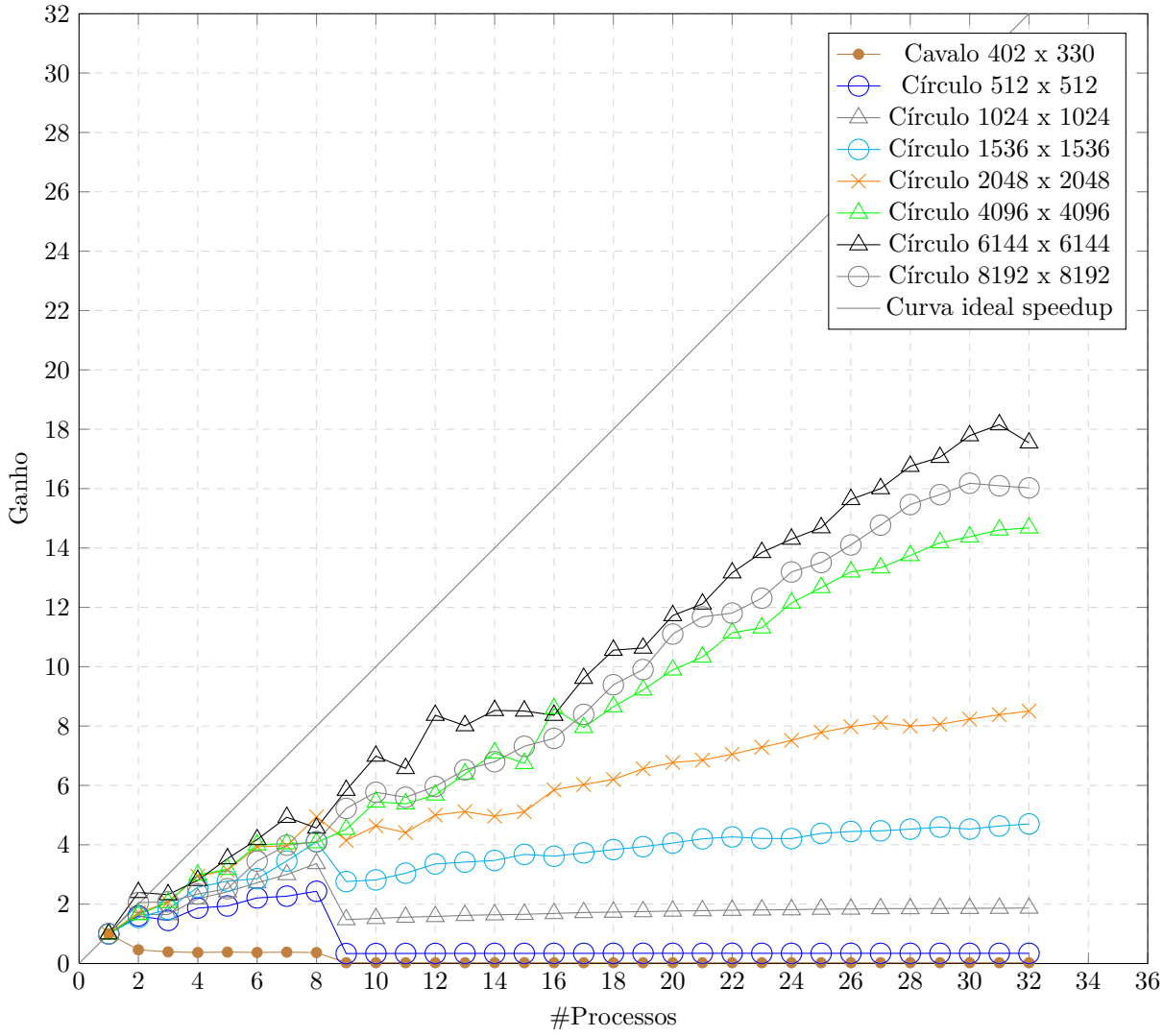


Figura 4: Resultados apresentados foram obtidos com a versão 1.6.3 do open-mpi, com bindings by-node e com as comunicações a decorrerem sobre a rede myrinet. Apenas foram utilizados cores físicos, sem recurso a hyper-threading

Em anexo podem ser consultados os resultados dos tempos da versão sequencial.

Pode-se observar desde logo que para imagens de dimensão inferior a 1536 pixeis os ganhos em tempo de processamento não compensam o tempo gasto em comunicação. Quando duplicamos o número de cores para dois, quase todas as imagens experienciam ganhos, mas as imagens de grande dimensão obtêm ganhos ligeiramente acima de 2. Isto resulta do facto de estarem agora a correr em 2 máquinas distintas, tendo mais cache disponível, acabam por poupar nas leituras/escritas em memória.

De realçar as perdas em performance observadas acima dos 8 cores, abrangente a todas as imagens testadas. Aqui estamos perante alguma limitação na rede myrinet ou dos drivers em si que apenas permitem comunicação

paralela e eficiente entre 8 processos em simultâneo. Esta questão é explorada no capítulo seguinte.

4.2 Tempos de comunicação em Nodes 641 com myrinet

Analisamos agora a porção do tempo total de execução gasto em comunicações, para cada input e para cada número de processos.

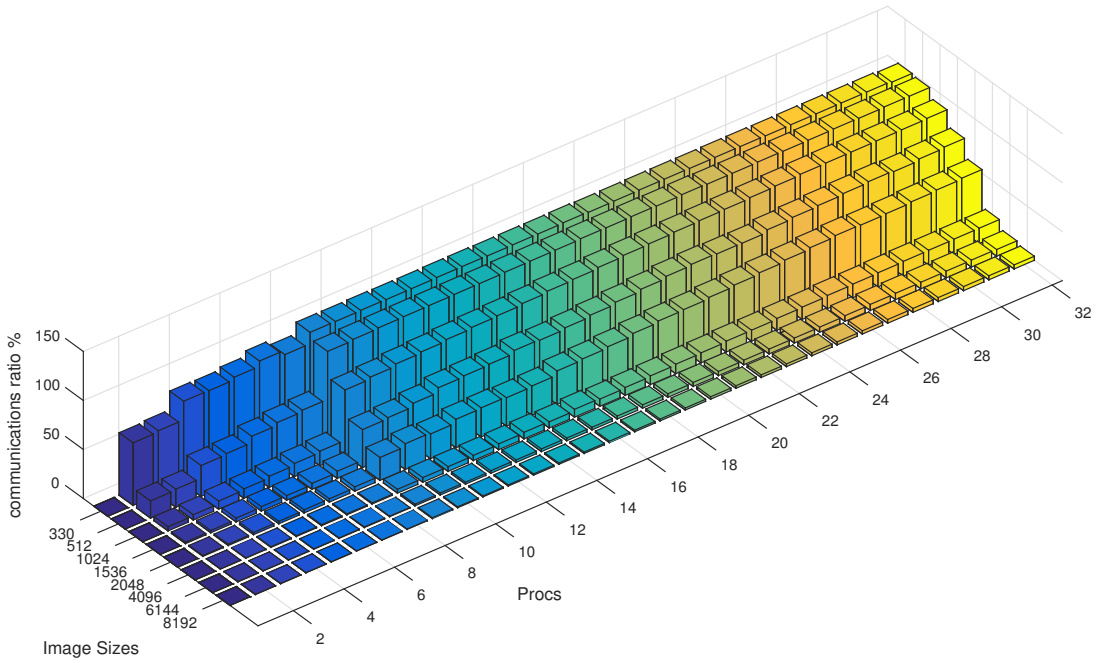


Figura 5: Resultados para os tempos de comunicação em nodes 641. O racio representa a percentagem do tempo total gasto a enviar e receber mensagens, não contabilizando tempos de processamento.

Pode-se observar para imagens de pequena dimensão o overhead de comunicação. Para imagens de dimensão superior a 2048 o tempo gasto a comunicar já não se faz notar tanto, pois os tempos gastos a processar por iteração são bem superiores.

Também de realçar o aumento súbito em tempos de comunicação quando temos 9 ou mais processos em simultâneo, que como foi referido no capítulo anterior, será alguma limitação ao nível do hardware de rede.

De notar que aqui neste gráfico não se encontra contabilizado o tempo de espera/sincronização que ocorre quando um processo acaba mais cedo do que os outros, apenas as trocas de mensagens foram contabilizadas para o tempo aferido.

4.3 Resultados no portátil pessoal

Apesar de, neste caso, o programa correr com comunicações sobre memória partilhada, pretende-se aqui analisar como é que este algoritmo se comportaria a correr numa máquina pessoal, e se traria ganhos de desempenho numa utilização mais comum.

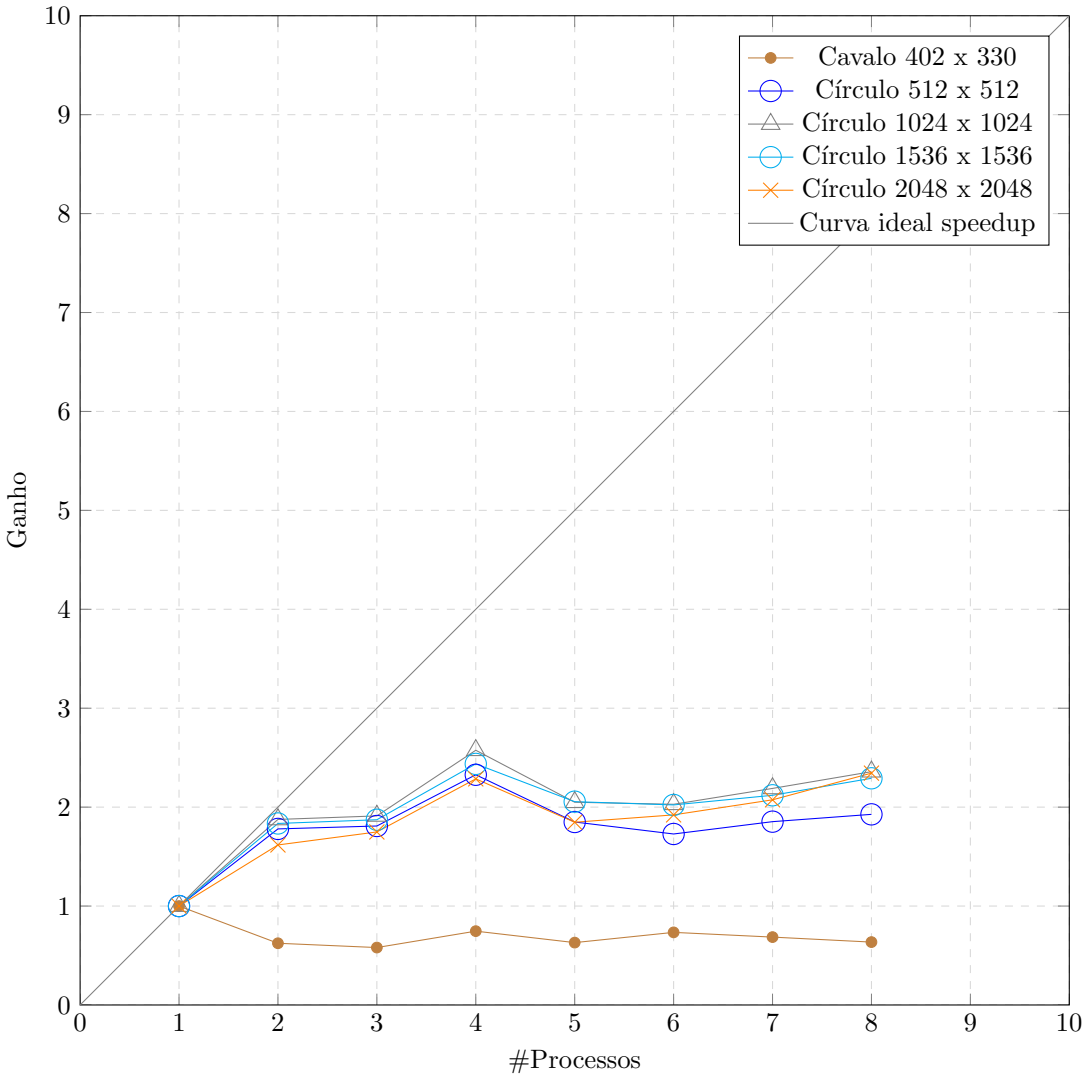
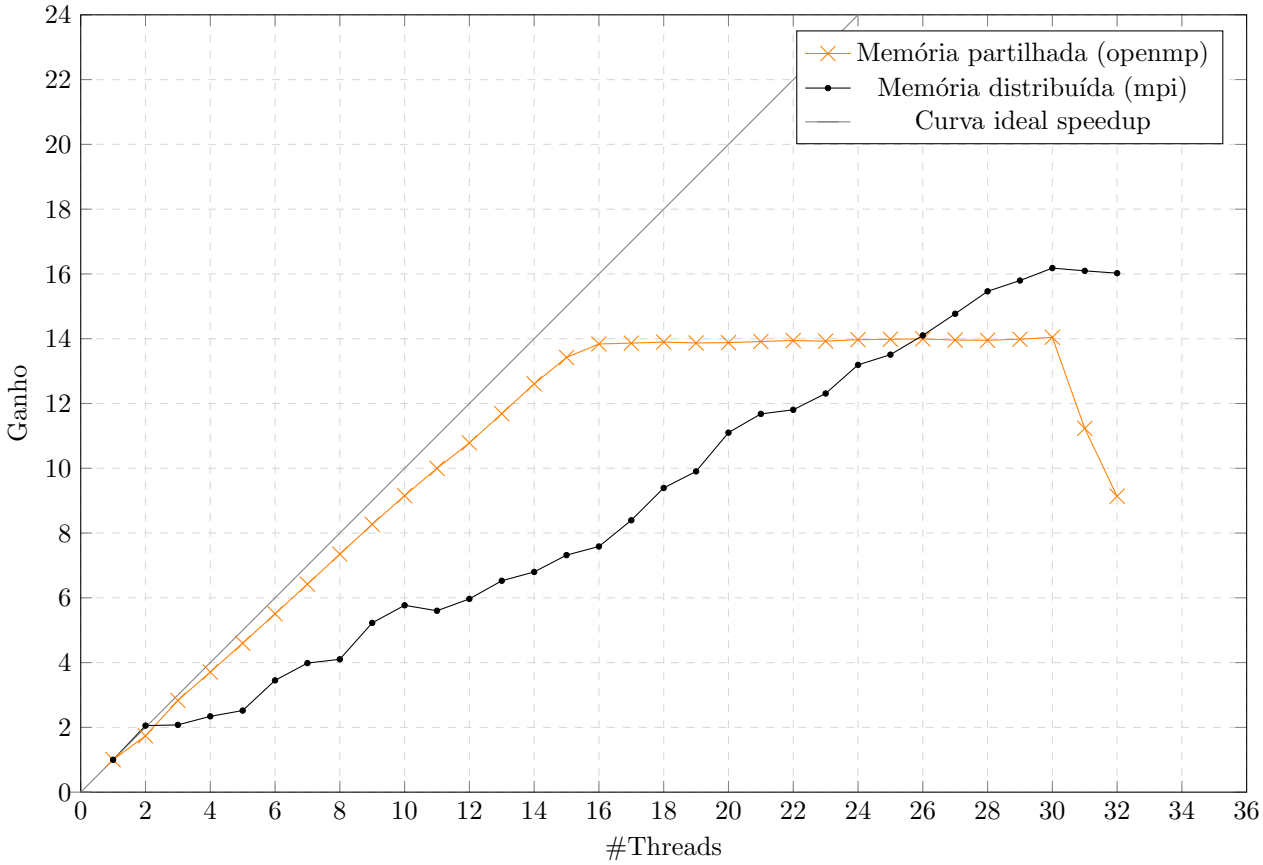


Figura 6: Resultados apresentados foram obtidos com a versão 1.6.3 do open-mpi. Foram utilizados cores físicos até 4 processos, com recurso a hyper-threading de 5 para cima

Assim como no nó 641, aqui imagens de pequena dimensão também não trazem ganhos de desempenho, observando-se, aliás, perdas na performance significativas para a imagem mais pequena. Para os outros tamanhos, quando duplicamos o número de cores disponíveis para dois observamos ganhos quasi-lineares, mas que depois se esbatem quando passamos para três cores, que poderá ser fruto de uma divisão mais injusta do trabalho, pois a divisão por 3 processos acarreta sempre que algum processo vai ter mais linhas para trabalhar do que os demais, gerando uma espera suplementar entre cada iteração para sincronização. Acima de 4 cores, já não compensa investir na paralelização pois estamos a utilizar hyper-threading, com custos adicionais devido à contenção de recursos.

5 Comparação relativa ao algoritmo memória partilhada OpenMP

Comparamos agora os ganhos obtidos em memória distribuída com a versão implementada em memória partilhada.



Ambas as curvas foram calculadas fazendo a média dos ganhos das 3 imagens de maior dimensão, para cada uma das versões. De notar que a versão em openmp sofreu diversas melhorias em termos de optimização, aproximando-se agora de um comportamento mais linear em termos de ganhos. Entre outras otimizações, foi adicionado loop unrolling e o chunk size do escalonador passou a ser calculado dinamicamente, variando com o tamanho da imagem, de forma a optimizar o uso da cache.

De realçar o facto de o openmp estar limitado a uma só máquina e daí consiga apenas expandir em termos de performace até os 16 cores, que são os cores físicos disponíveis no node 614.

Daqui poderemos partir para uma versão híbrida futura que concilie as vantagens do paradigma de memória distribuída com o paradigma de memória partilhada.

6 Conclusão

O algoritmo desenvolvido mostrou-se capaz de obter um *speedup* na ordem das 18 vezes em determinadas condições. Este ganho parece-nos razoável, se olharmos à quantidade de mensagens trocadas e tempos de sincronização necessários entre iterações.

A criação deste algoritmo foi um processo interessante tendo-se primeiro criado uma versão mais centralizada, mais simples, que depois foi expandida e melhorada explorando o paralelismo intrínseco do algoritmo, tirando também partido das capacidades da rede.

Comparativamente ao algoritmo em memória partilhada, pensamos haver vantagens em procurar uma solução híbrida que concilie a localidade dos dados oferecida pelo paradigma distribuído com a extensão de recursos que advêm com o paradigma distribuído.

A Tabela com caracterização das máquinas utilizadas nos testes

System	compute-641	personal-laptop
#Sockets	2	1
CPU	Xeon E5-2650v2	Core i7-7700HQ
#Cores per CPU	8	4
#Threads per CPU	16	8
Architecture	Ivy Bridge	Skylake
Lithography	22nm	14nm
Base Frequency	2.60 GHz	2.80 GHz
Turbo-Boost	3.40 GHz	3.80 GHz
Extensions	SSE4.2, AVX	SSE4.2, AVX2
Peak FP Performance	400 GFlops/s	36.8 GFlops/s
L1 Cache	32KB per core	32KB per core
L2 Cache	256KB per core	256KB per core
L3 Cache	20MB	6MB
Total memory	64GB	16GB
#memory channels	4	2
Vendor Peak Mem. BW	59.7 GB/s	21.3 GB/s

Tabela 3: Descrição de um dos nós do cluster 641, e do portátil pessoal utilizados nos testes.

B Tempos versão sequencial :: algoritmo swap da matriz

disk size	Input W*H	total iterations	exec time (in ms)	time p/ iteration (in ms)
128K	horse 402*330	94	28620	304
256K	circle 512*512	362	387082	1069
1M	circle 1024*1024	724	2480423	3426
2M	circle 1536*1536	1086	8125303	7482
4M	circle 2048*2048	1448	20983525	14491
16M	circle 4096*4096	2896	166379970	57452
32M	circle 6144*6144	4344	632746869	145660
64M	circle 8192*8192	5792	1344659802	232158

Tabela 4: Tempos da versão sequencial, para cada input. Tempos obtidos no node 641 do Cluster.