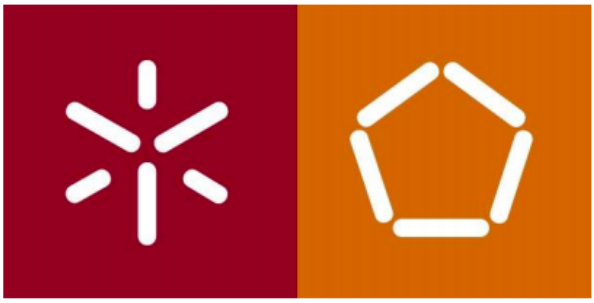


Universidade do Minho

Mestrado Integrado em Engenharia Informática

Paradigmas de Computação Paralela

Paralelização de um Algoritmo de Esqueletização de Imagens
Trabalho Prático 2 - MPI

Eduardo Lourenço da Conceição (A83870)
Rui Nuno Borges Cruz Oliveira (A83610)

22/01/2021
Braga

Índice

1 Introdução

Neste segundo trabalho de Paradigmas de Computção Paralela implementamos uma otimização ao algoritmo de esqueletização de imagens utilizando a Message Passing Interface (MPI). Iremos expor a decomposição do algoritmo, identificar os pontos cruciais que justificam paralelização, explicar como a implementamos e comparar a sua performance sob vários ambientes de teste, e contrastar a mesma com a versão de OpenMP, explicando quais as vantagens e desvantagens desta nova abordagem.

Uma nota introdutória que queremos adicionar é que, sempre que mencionamos um cálculo de tempo, estamos a referir-nos a um cálculo da **complexidade** de cada uma das partes do algoritmo.

2 Algoritmo de Esqueletização

2.1 Descrição do Algoritmo

O algoritmo irá fazer dois varrimentos a uma matriz de inteiros (zeros e uns) e, para cada elemento da matriz, aplicará três operações: N (contar o número de vizinhos a um), S (somar as transições de zero para um nos vizinhos) e um *checksum*, diferente entre cada uma das passagens. Quando os resultados destas operações respeitam quatro condições, alteramos o valor do elemento da matriz para um. Este processo é aplicado repetidamente até que não haja mais mudanças na matriz, e aqui a imagem está esqueletizada.

2.2 Análise do Algoritmo

Assumindo *i* como o número de iterações do algoritmo, *linhas* como o número de linhas da matriz e *colunas* como o número de colunas, o peso computacional do algoritmo pode ser representado da seguinte forma (não contando com possíveis acessos à memória):

$$T_{comp} = i * 56 * linhas * colunas$$

Para podermos paralelizar um algoritmo, teremos primeiro que o caracterizar as suas fases e ver quais se apresentam como melhores candidatas para paralelização. No entanto, este algoritmo não pode ser decomposto em fases. A segunda passagem não pode ser realizada depois da primeira, e não faz sentido preocuparmo-nos com obrigar a que cada uma das três operações (N, S e o *checksum*) sejam feitas em paralelo, uma vez que o seu tamanho é tão diminuto que qualquer comunicação entre processos terá um custo muito maior do que as computações em si. Como tal, apenas poderemos dividir os elementos da matriz, mapeando-os nos processos.

Para podermos calcular o potencial ganho máximo obtido com a paralelização, recorremos à **Lei de Amdahl**. Como concluímos no primeiro trabalho, a paralelização que podemos fazer nível das linhas será mais eficiente, pelo que o trabalho sequencial que cada processo terá de fazer, em função do número de processos (*P*), será:

$$56 * i * colunas * \frac{linhas}{P}$$

Pelo que concluímos que a fração sequencial corresponde a:

$$frac\_seq = \frac{1}{P}$$

Como tal, aplicando a Lei de Amdahl, temos que o *speedup* máximo que poderemos obter com a paralelização (S_P) será:

$$S_P \leqslant \frac{P}{2-\frac{1}{P}}$$

3 Paralelização com MPI

3.1 Partição

No que toca à **partição dos dados** pelos processos, teremos de olhar para como dividiremos a matriz. A nosso ver, a forma de fazermos isto que implique o menor número de comunicações será dividir a matriz por blocos de linhas (*chunks*). Assim, o esperado é que, se temos *P* processos e *linhas* linhas, cada processo receberá aproximadamente $\frac{linhas}{P}$ linhas. No entanto, o algortimo apresenta **dependências de dados**, pois, para processar cada elemento, temos de verificar todos os seus vizinhos, incluindo os que se encontram na linha superior e na inferior. Assim, não podemos enviar apenas as linhas que vão ser processadas a cada processo. Temos também de mandar **duas linhas extra** para cada processo (exceto o primeiro e o último).

Podemos observar uma ilustração desta partição na imagem seguinte:

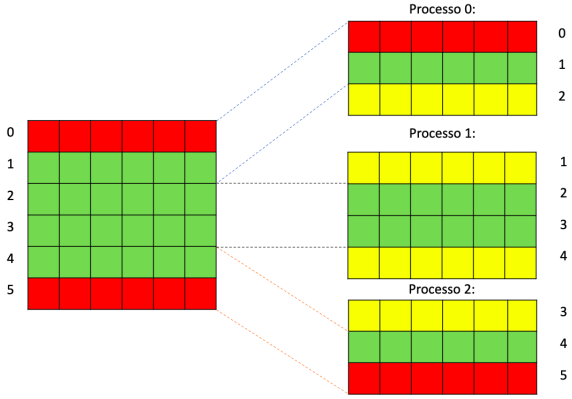


Figure 1: Partição de Dados

Neste caso, utilizamos para exemplificar 3 processos, em que a verde vemos as linhas que são realmente processadas por cada processo, a amarelo aquelas que são utilizadas apenas para obter os valores das linhas adjacentes (por exemplo, para o processo 1, a linha 2 não pode ser processada sem termos a linha 1 (a amarelo) e 3 (a verde)), e a vermelho a primeira e a segunda linha, que não são percorridas.

Para a **partição funcional**, esta está diretamente relacionada com a partição de dados, uma vez que as funções executadas são sobre cada elemento da matriz. Como tal, se a matriz é dividida equitativamente por *P* processos, também o *overhead* computacional. Como tal, traduz-se na seguinte igualdade, relacionada com a que mostramos anteriormente:

$$T_{comp} = \frac{i*56*linhas*colunas}{P}$$

3.2 Comunicação

A comunicação entre processos neste algoritmo tem várias fases. Em primeiro lugar, apenas o processo *ROOT* (normalmente o processo com *rank* 0) lerá os valores da imagem do ficheiro onde se encontra. Como tal, terá de fazer ***broadcast* do tamanho da imagem** para os outros processos, bem como ***scatter* dos valores da matriz** que guarda a imagem. Esta operação de *scatter* vai obrigar a uma adição no *overhead* de computação do programa, uma vez que cada processo vai ter de preparar as estruturas de dados onde guardarão os valores das matriz que terão de tratar, o que implica várias alocações de memória adicionais, bem como certos cálculos para cada processo saber quantos valores vai tratar e quais. Este tempo extra de computação não foi contabilizado mais tarde nas comparações com *OpenMP*, uma vez que tornaria a comparação injusta, pois tratam-se de cálculos que a implementação em *OpenMP* não foi obrigada a fazer.

O passo de comunicação mais complexo vem depois. Um processo de *rank* K tem de **enviar a sua penúltima linha ao processo K+1 e a sua segunda ao processo K-1**, bem como **receber a sua primeira linha do processo K-1 e a sua última do processo K+1**. O processo de *rank* 0 não enviará a sua segunda linha, nem receberá a primeira, e o processo de *rank P-1* não terá de enviar a sua penúltima linha nem receber a última. Tudo isto é feito para garantir que, sempre que, na versão sequencial, a matriz acabaria de ser percorrida, todos os valores das linhas em todos os participantes estariam atualizados para os últimos calculados. Podemos visualizar estas trocas de linhas na seguinte imagem:

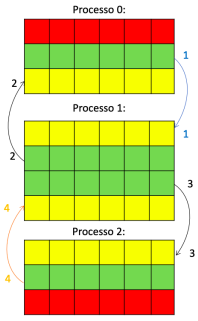


Figure 2: Troca de Linhas entre Processos

De notar que, se, não fizermos estas trocas, **o esqueleto resultante será na mesma**

válido. Isto deve-se ao facto que, quando estas trocas não acontecem, as linhas que não são processadas em cada processo simplesmente não mudam, o que acaba por funcionar para cada processo como a primeira e a última linha (que sob nenhuma circunstância mudam) localmente para cada processo, fazendo com que cada submatriz com que cada processo lida funcione como uma matriz completa isolada em cada processo.

Figure 3: Imagem Original, Esqueleto com Comunicação Extra, Esqueleto sem Comunicação extra (respetivamente)

No entanto, a primeira versão descrita está muito mais em linha com o algoritmo sequencial, pelo que decidimos testar as duas versões.

Depois disto, temos também de fazer uma ***all reduction*** da *flag* que verifica se houveram alterações na matriz, de modo a sabermos quando o algoritmo acabou (para efeitos de teste, para normalizar os resultados, limitamos estas iterações a 5). Por fim, há que fazer um ***gather*** dos valores guardados em cada matriz. Assim, para a versão deste algoritmo que inclui todas as comunicações, temos que:

$$T_{comm} = 2log_2(P) + 2(P * N) + (log_2(P) + (4P - 4) * colunas * \frac{linhas}{P}) * i$$

Sendo *N* o número de elementos a enviar/receber em cada operação de envio/receção
Ou, para o caso em que não fazemos a comunicação extra, ficamos com:

$$T_{comm} = (2 + i) * log_2(P) + 2 * P * N$$

Como podemos notar, a maior parte da comunicação involvida é **local**, visto que o maior *overhead* comunicacional está nas trocas de linhas entre processos vizinhos. No entanto, existem vários passo de comunicação **global**, como o *scatter*, o *gather*, o *all reduce* e o *broadcast*, uma vez que estas operações envolvem todos os processos.

3.3 Aglomeração

Como dissemos anteriormente, os valores da matriz vão ser aglomerados em *chunks*, constituídos por várias linhas. Esta será a forma de diminuir ao máximo a comunicação necessária entre processos.

3.4 Mapeamento

Cada *chunk* da matriz será mapeado ao processo com o mesmo *rank* que o número do *chunk*. Ou seja, o primeio *chunk* vai ser mapeado no processo 0, o segundo no processo 1, etc.

3.5 Trabalho Redundante

A versão em MPI, uma vez que utiliza memória partilhada, necessita de alocar, em cada processo, memória extra para guardar a porção da matriz que o processo vai tratar. Para esse efeito, existe uma porção do código dedicada a alocar memória para que o *scatter* seja possível. Isto terá de ser executado em todos os processos, e adiciona um *overhead* maior à execução global do algoritmo. Como tal, decidimos não contabilizar esta porção nos tempos de execução, de modo a podermos ver qual é o tempo que demora a porção do código que realmente trata do algoritmo, da mesma forma que não contabilizamos tempo de I/O.

4 Testes

4.1 Condições de Teste

Como estipulado no enunciado, utilizamos o nó r641[1] do *cluster* SeARCH6, bem como o nó r662[2]. Utilizamos 2 *nodes* e 32 *processors per node*. Mais ainda, utilizamos dois tipos de mapeamento: por nó e por *core*, como pedido no enunciado.

4.1.1 *Datasets*

Os *datasets* de teste são os mesmos que tivemos no primeiro trabalho.

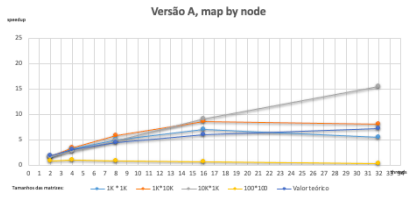
- 100x100: um tamanho que cabe na *cache L2*
- 1000x1000: um tamanho que cabe na *cache L3*
- 10000x1000: tamanho que não cabe nas *caches* e não é quadrado, tendo muitas mais linhas que colunas.

A utilização dos mesmos tamanhos permite também comparação direta com os valores obtidos no primeiro trabalho.

É de notar que testamos também com *1000x10000* e *10000x10000*, mas estes valores nunca permitiram que os testes acabassem no *cluster*, pois excediam o *walltime* de duas horas que os nós de interesse possuem.

4.2 Análise de Resultados

Uma vez que concluímos os testes, obtivemos os seguintes resultados para a versão com comunicação extra, dita **versão A**, para o *speedup* relativo à versão sequencial do primeiro trabalho:



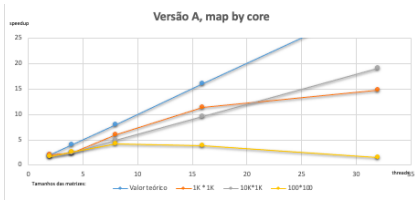


Figure 4: *Speedup* relativo à versão sequencial na versão A no r641

Nesta versão, como seria de esperar, o *speedup* na versão mapeada por *core* é maior, uma vez que na versão mapeada por *nodes* os processos têm de comunicar com outros nodos, o que demora consideravelmente mais tempo. Mais ainda, a versão no r641 é **mais lenta** que a no nodo r662. Isto deve-se ao facto que, no nodo r662 temos um processador diferente, possivelmente mais avançado e adequado para este problema.

Por fim, esta versão é rápida, mas a versão B (sem a comunicação extra) é, como seria de esperar, mais rápida ainda, uma vez que o *overhead* comunicacional é menor.

Os gráficos relativos à versão B e ao nodo r662 encontram-se em anexo.

Um facto que é interessante notar em quase todas as versões é que o *speedup* máximo ocorre com 16 processos. Isto poderá ser explicado pelo facto que, quando pedimos 32 processos, estamos a obrigar cada *core* a tratar de mais do que um processo, obrigando *hyperthreading*. Até 16 *cores* isto não acontece.

4.3 Tempos de Comunicação

O tempo de comunicação das duas implementações foi medido apenas no processo *ROOT*, que realiza o mesmo número de comunicações que os outros processos na versão B e ligeiramente menos comunicação que os outros na versão A. Escolhemos medir apenas neste processo pois permite-nos ter uma ideia do peso da comunicação em relação ao número de processos e tamanho da matriz.

Tendo isto em conta, podemos olhar para o tempo de comunicação da versão A na seguinte tabela, em microssegundos, no nodo r641:

	Nº threads	2	4	8	16	32
Node	100*100	1	2	2	4	8
	1K * 1K	37	46	49	53	69
	10K*1K	353	429	435	403	398
Core	100*100	0	0	0	0	4
	1K * 1K	3	18	10	9	51
	10K*1K	24	179	103	71	390

Como seria de esperar, e para corroborar as conclusões a que tínhamos chegado antes, a comunicação feita na versão mapeada nos *nodes* é consideravelmente mais demorada do que na mapeada nos *cores*, devido à necessidade de comunicar através de canais que introduzem maior latência. Quando comparamos com o tempo de comunicação na versão B:

	Nº threads	2	4	8	16	32
Node	100*100	1	1	2	3	4
	1K * 1K	49	44	44	45	48
	10K*1K	350	358	413	387	377
Core	100*100	0,100851	0,258923	0,307083	0,276804	4
	1K * 1K	3	14	10	8	51
	10K*1K	22	172	100	66	386

Podemos então provar que este é comunicacionalmente mais leve, como seria de esperar, apesar da diferença não ser tão notória quanto esperado.

Mais uma vez, os tempos de comunicação são menores no r662. As tabelas encontram-se em anexo.

4.4 Perfil de Execução

Podemos finalmente, com as informações retiradas das últimas secções, traçar um **perfil de execução** da nova implementação com MPI.

Como pudemos confirmar, o programa tem um comportamente melhor quando os processos são mapeados por *core*, como pudemos ver no tempo de comunicação e no *speedup*. Mais ainda, o tempo de computação, quando excluímos os tempos de comunicação, comporta-se de forma semelhante ao da versão OpenMP, mas o *overhead* comunicacional adicionado, juntamente com o tempo que é passado por cada processo a alocar a memória para receber os valores das matrizes (que, com os devidos testes, concluímos que estava entre 5 microssegundos e 30 microssegundos para todos os casos) adicionam mais tempo ao tempo de execução, mas, mesmo com estes em consideração, **os ganhos são significativos**, pois a maior porção de tempo de execução continua a ser o tempo de computação que é semelhante ao do OpenMP.

4.5 Comparação com OpenMP

Por fim, comparando esta versão com a de OpenMP, vemos que ambas apresentam vantagens e *speedups* significativos, mas esta tem algumas desvantagens a ter em consideração.

No que toca aos *speedups*, ambas as versões mostram ganhos promissores na execução do corpo principal do algoritmo, sendo que o ganho máximo ronda para os dois as 23 vezes.

O OpenMP provou ser mais difícil na gestão das *data races*, sendo que o esqueleto que era produzido numa execução era diferente do produzido na próxima, e este problema não existe aqui, uma vez que não existem *data races* devido ao facto que a memória é distribuída. No entanto, a versão em MPI introduz um novo *overhead* de memória, uma vez que tratamos o sistema como vários bancos de memória separados em vez de um só, e temos de fazer mais alocações de memória, de computação, que estas alocações têm de ser feitas em todos os processos, e de comunicação, uma vez que a passagem de mensagens é explícita e, para muitos dos casos, bastante pesada.

Em suma, diríamos que o OpenMP apresenta mais vantagens, apesar de MPI também apresentar algumas, nomeadamente o controlo explícito da localização de memória onde guardamos certos dados, o que não é uma *feature* oferecida pelo OpenMP, ou pelo C standart em geral.

5 Conclusão

Por fim, acreditamos ter feito o que era esperado. Conseguimos uma implementação em MPI que nós acreditamos ser boa, tendo bons ganhos em relação à versão sequencial. Mais ainda, pudemos comparar com os resultados do primeiro trabalho e daí concluir quais as vantagens e desvantagens da utilização do MPI em detrimento do OpenMP, e chegamos a conclusões lógicas do porquê dos resultados que obtivemos.

6 Referências

[1] *Intel Xeon E5-2670 v2 specifications*. URL: https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%5C%20E5-2670%5C%20v2.html.

[2] *Intel Xeon E5-2965 v2 specifications*. URL: https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%5C%20E5-2695%5C%20v2.html.

7 Anexos

7.1 Dedução do T_{comp}

- N = 7 somas;
- S = 7 somas + 8 operações AND lógico;
- *Checksum* = 3 somas;
- 1ª/2ª Passagem = *linhas* * *colunas* * (7 + 7 + 8 + 3 + 3);
- T_{comp} = T(Passagem) * 2 * *i* = *linhas* * *colunas* * 28 * 2 * *i*

7.2 Dedução da Fração Sequencial

$frac\_seq = \frac{56*i*colunas*\frac{linhas}{P}}{56*i*colunas*linhas} = \frac{linhas/P}{linhas} = 1/P$

7.3 Dedução do valor da *Lei de Amdahl* obtido

$S_P \leqslant \frac{1}{frac\_seq+(1-frac\_seq)/P} = \frac{1}{\frac{1}{P}+\frac{1-\frac{1}{P}}{P}} = \frac{1}{\frac{2+1/P}{P}} = \frac{P}{2-\frac{1}{P}}$

7.4 Dedução do T_{comm}

Complexidade das operações de envio de mensagem do MPI em função do número de processos envolvidos (P) e do número de items envolvidos (N):

- *Scatter*: $P * N$ [1]
- *Gather*: $P * N$ [1]
- *AllReduce*: $log_2(P)$ [2]
- *Broadcast*: $log_2(P)$ [2]
- *Send*: N [3]
- *Receive*: N [3]

$T_{comm} = (2 * (colunas * \frac{linhas}{P}) + (P-2) * 4 * colunas * \frac{linhas}{P} + 2 * (colunas * \frac{linhas}{P}) + log_2(P)) * i + 2 * log_2(P) + P * N + P * N$

$\Leftrightarrow T_{comm} = ((2 + (P-2) * 4 + 2) * (colunas * \frac{linhas}{P}) + log_2(P)) * i + 2 * log_2(P) + 2 * P * N$

$\Leftrightarrow T_{comm} = ((4P-4) * (colunas * \frac{linhas}{P}) + log_2(P)) * i + 2 * log_2(P) + 2 * P * N$

[1] Neste caso, *N* é o tamanho do *chunk* que é enviado a um processo, logo $N = colunas * \frac{linhas}{P}$

[2] Para um número pequeno de elementos, está mais próximo de *N*, mas, quando P tende para $+\infty$, a complexidade do *AllReduce* e do *Broadcast* é $log_2(P)$, uma vez que as mensagens são propagadas em cascata, de certa forma, como numa árvore binária.

[3] Em ambos os casos, o *N* é igual ao número de colunas, uma vez que enviámos uma linha de cada vez, logo $N_{send} = N_{receive} = colunas$

7.5 Gráficos adicionais

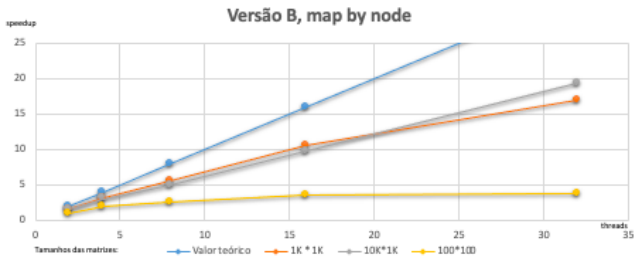


Figure 5: *Speedup*, versão B, *map by node*, r641

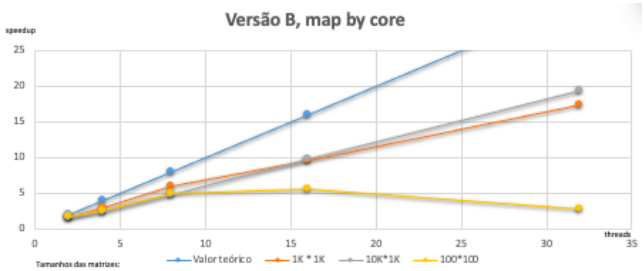


Figure 6: *Speedup*, versão B, *map by core*, r641

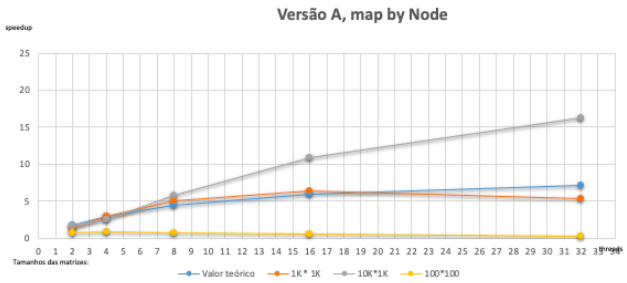


Figure 7: *Speedup*, versão A, *map by node*, r662

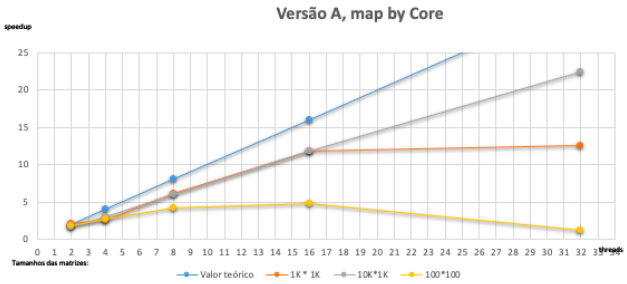


Figure 8: *Speedup*, versão A, *map by core*, r662

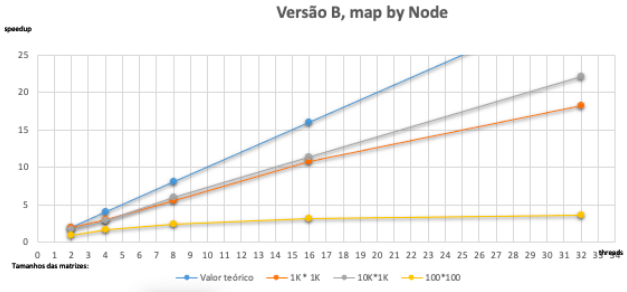


Figure 9: *Speedup*, versão B, *map by node*, r662

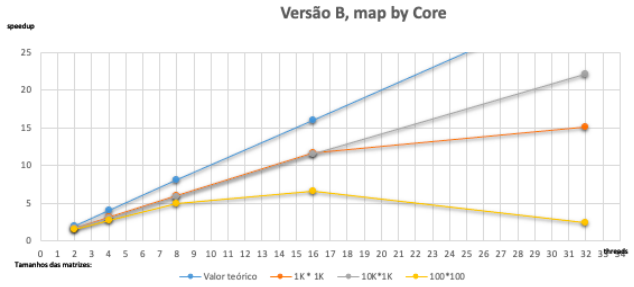


Figure 10: *Speedup*, versão B, *map by core*, r662

	Nº threads	2	4	8	16	32
Node	100*100	1	2	3	5	8
	1K * 1K	38	41	43	54	70
	10K*1K	401	480	371	371	389
Core	100*100	0	0	0	0	3
	1K * 1K	3	3	4	6	31
	10K*1K	27	31	31	32	205

Figure 11: Tempo de Comunicação, versão A, r662

	Nº threads	2	4	8	16	32
Core	100*100	0,132084	0,181913	0,315189	0,292063	0
	1K * 1K	3	4	4	6	29
	10K*1K	25	31	31	33	204
Node	100*100	1	1	2	3	4
	1K * 1K	37	38	39	42	48
	10K*1K	392	438	359	359	360

Figure 12: Tempo de Comunicação, versão B, r662