Instituto Superior Técnico

Computação Paralela e Distribuída

CPD - MPI

Grupo 27:
Manuel Domingues (82437)
Ricardo Santos (90178)
Inês Ferreira (90395)

2020/2021 - 2º Semestre
22.05.2021

Correções na versão sequencial e OpenMP

Foram implementadas algumas melhorias, quer na versão sequencial, quer na versão paralela em OpenMP, para utilizar estas modificações na versão paralela em MPI e comparar os tempos obtidos com a versão MPI com as duas anteriores. As mudanças foram as seguintes:

- Os vetores auxiliares de índices e projeções foram substituídos por uma estrutura, onde também é incluído o ponto correspondente, de modo a minimizar os acessos à memória;
- Caso o número de pontos seja par, em que é necessário encontrar duas medianas, o cálculo da segunda mediana passou a ser feito com recurso a um *for loop* na metade direita do vetor de estruturas, encontrando-se o elemento com menor projeção. Assim, garante-se a linearidade do processo.

MPI

Utilização de memória

Uma das vantagens da utilização de MPI face a OpenMP é que permite utilizar a memória de várias máquinas, permitindo resolver problemas que não cabem numa só. Sendo este o objetivo, os pontos são inicialmente distribuídos pelas várias máquinas. A geração dos pontos com o ficheiro *gen_points.c* foi substituída por código *inline*. Para que os pontos sejam gerados corretamente, é necessário um *for loop* com limite máximo igual ao número de pontos onde se chama a função *random()* mesmo quando os pontos correspondentes não pertencem ao processo em questão. Isto permite distribuir os pontos pelos processos mas implica que a sua geração não seja paralelizada. Tanto a árvore com os nós como o vetor de centros auxiliar vão estar distribuídos entre processos.

É importante referir também que, em várias situações, é necessário os processos saberem quantos elementos têm os outros processos. Para isso utilizaram-se as *macros* *BLOCK_LOW*, *BLOCK_HIGH* e *BLOCK_SIZE* e um vetor *block_size* que contém o número de elementos em cada processo.

Nas chamadas do algoritmo onde é usado MPI é necessário o envio de informação entre processos, nomeadamente de índices, pontos e projeções. Dado que não é possível o envio de estruturas que contenham ponteiros (como era o caso da usada nas versões sequencial e OpenMP), não foi utilizada esta estrutura nas chamadas com MPI, sendo apenas criada imediatamente antes das chamadas sequenciais.

Para manter a ordem relativa dos pontos correta em cada processo, a troca entre dois pontos num mesmo processo não é feita trocando as suas posições de memória, mas sim o conteúdo em memória (usando um ciclo em *n_dims*). O posterior envio dos pontos para outros processos é então facilitado, visto que apenas tem de ser enviado um bloco de memória contínuo. Assim, não é necessário o uso do *pt_array* (vetor de apontadores para os pontos) e usa-se diretamente o vetor dos pontos, de tamanho $n\_dims \times np$.

Por fim, salienta-se que, em cada chamada do algoritmo, apenas o processo com *id* 0 vai guardar o nó correspondente. Mas, dependendo do nível da árvore, podemos ter mais chamadas em paralelo e, consequentemente, ter vários processos com *id* 0 a guardar diferentes nós da árvore.

Paralelização dentro de uma chamada do algoritmo

Índices a e b

O cálculo destes dois índices consiste em três passos dependentes: determinar o ponto com índice mais baixo, calcular o a e calcular o b. Como os pontos correspondentes a estes índices vão ser usados em passos futuros (como o cálculo das projeções), é necessário que todos os processos os tenham. Para o cálculo do índice global mínimo, encontra-se o índice mais baixo de cada processo e guarda-se o mesmo numa estrutura com tipo *MPI_2INT*, sendo que o segundo inteiro corresponde ao *id* do processo. Através da comunicação *MPI_Allreduce* com a operação *MPI_MINLOC*, todos os processos sabem qual o índice global mínimo e qual o processo que o contém. Esta informação é necessária para se especificar qual o *id* do *root* na comunicação *MPI_Bcast*, que permite que o processo que contém o ponto com índice global mínimo o envie para todos os outros.

O cálculo e comunicação dos índices a e b são semelhantes ao anterior. Neste passo, a estrutura é do tipo *MPI_DOUBLE_INT* e a operação no *MPI_Allreduce* é *MPI_MAXLOC*, visto que o que se quer é a distância máxima em relação ao ponto com menor índice global. Também os pontos correspondentes aos índices a e b são enviados com *MPI_Bcast*.

Projeções

As projeções são facilmente calculadas visto que, tendo agora todos os processos os pontos correspondentes a a e b, cada um pode calcular as projeções correspondentes ao seu conjunto de pontos.

Cálculo da mediana

O algoritmo para encontrar a mediana foi baseado no algoritmo *Parallel Sorting by Regular Sampling* (PSRS), segundo os passos a seguir descritos e ilustrados.

- Passo 1 - Cada processo escolhe p *pivots*. O primeiro *pivot* é a menor projeção, sendo os restantes $p-1$ pivots escolhidos de forma espaçada. É utilizado um *quickselect* para os encontrar e ordenar parcialmente o vetor.
- Passo 2 - *MPI_Gather* de todos os *pivots* para o primeiro processo.
- Passo 3 - O primeiro processo seleciona $p-1$ pivots de todos os recebidos, igualmente espaçados, utilizando o *quickselect*.
- Passo 4 - *MPI_Broadcast* dos $p-1$ pivots para todos os processos. Cada processo faz partições dos seus pontos segundo os *pivots* recebidos, guardando o tamanho de cada troço (note-se que os troços estão relativamente ordenados entre si).
- Passo 5 - *MPI_Alltoallv*, onde cada um dos troços é enviado para o processo correspondente à sua posição relativa no processo atual. Assim, cada processo terá todos os pontos inferiores a todos os processos seguintes e superiores a todos os processos anteriores. Note-se que neste passo é necessário que cada processo anuncie as dimensões dos troços que vai enviar (usando *MPI_Alltoall*), sendo posteriormente enviados os pontos, projeções e índices globais correspondentes a cada troço.

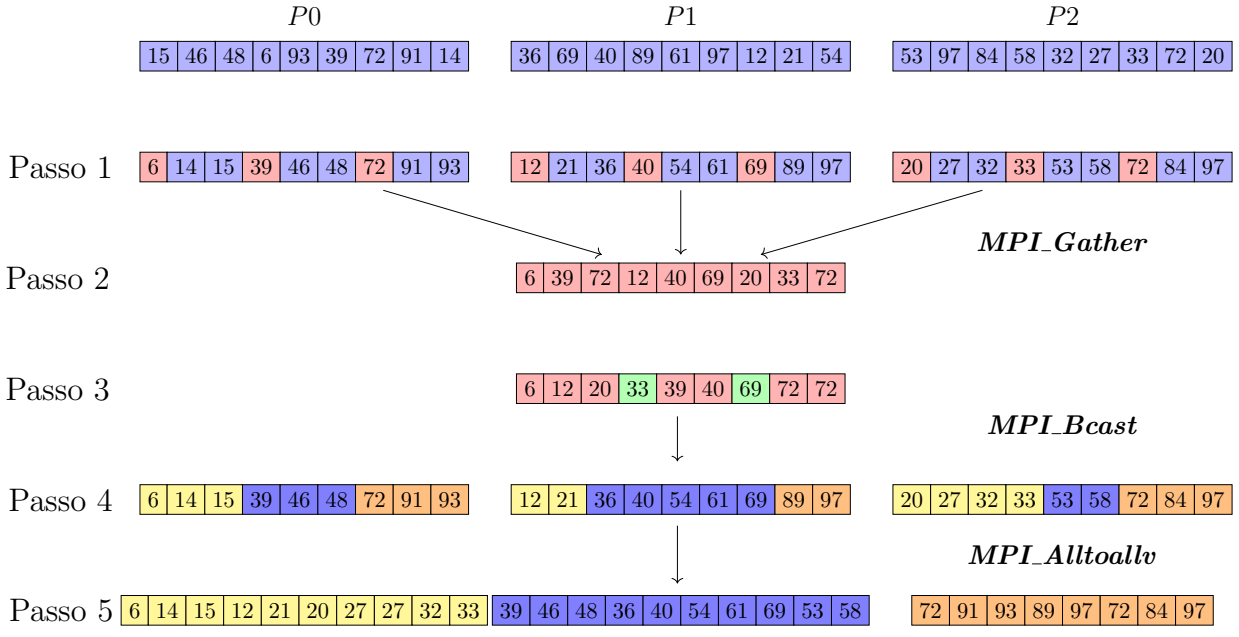


Figura 1: Esquema no algoritmo usado para paralelizar a o cálculo da(s) mediana(s).

No fim deste processo, espera-se que a mediana esteja no processo central, para um número ímpar de processos, ou num dos dois processos centrais, para um número de processos par.

Note-se que o número final de pontos em cada processo poderá não corresponder ao original. Como tal, é alocada memória a mais no início para cada vetor de dados (pontos, projeções, índices e vetores auxiliares), de modo a não serem necessárias realocações. Este valor foi definido como cerca de 2.5 vezes o valor original.

Finalmente, para determinar a mediana propriamente dita, têm de se considerar vários casos. Utilizando uma comunicação *MPI_Allreduce*, todos os processos sabem quantos elementos existem nos outros processos e sabe-se qual/ais o(s) elemento(s) do(s) processo(s) central/ais que se quer encontrar. Como se está a utilizar números de processos que são potências de dois, assume-se que a(s) mediana(s) está/ão num dos dois processos centrais e tem-se um de 3 casos possíveis:

- Mediana(s) no processo central esquerdo - É feito um *quickselect* neste processo para determinar a mediana 1 e fazer a respetiva partição. Se for necessário determinar uma segunda mediana, encontra-se o mínimo à direita da partição anterior. Enviam-se posteriormente as estruturas de dados deste processo para o processo central direito a partir do índice correspondente à mediana (inclusive) - *np* ímpar - ou a partir do índice correspondente à segunda mediana (inclusive) - *np* par.

- Mediana(s) no processo central direito - A(s) mediana(s) são determinadas de maneira análoga ao acima referido. Enviam-se as estruturas de dados deste processo para o processo central esquerdo até ao índice correspondente à mediana (não inclusive) - *np* ímpar - ou até ao índice correspondente à segunda mediana (inclusive).

- Mediana 1 no processo central esquerdo e mediana 2 no processo central direito - cada processo determina a respetiva mediana. Não é necessária qualquer troca de informação adicional.

Utiliza-se aqui o *MPI_Send* e *MPI_Recv*. Note-se que, no segundo caso, para se poder reutilizar as estruturas de dados, e porque é necessário um vetor ser "contínuo"para ser enviado com MPI, é necessário um *shift* após o envio dos dados.

A forma como a mediana é calculada gera preocupações: para processos pares, embora cada metade dos processos tenha o mesmo número de pontos (com uma possível diferença de um), os vários processos podem ter um número diferente de pontos entre si. Isto pode implicar que não seja garantido que a(s) mediana(s) esteja/am nos dois processos centrais (apesar da probabilidade ser bastante elevada).

Para colmatar este desequilíbrio decidiu-se que, no início de cada chamada do algoritmo, o processo com maior número de elementos vai enviar parte desses elementos para o processo com menor número de elementos. Através de um *MPI_Allgather*, todos os processos vão saber se vão enviar ou receber alguma coisa visto que todos eles determinam o *id* dos processos com número máximo e mínimo de pontos. O número de elementos a enviar / receber foi escolhido como a semi-diferença entre o número máximo e mínimo de elementos.

É possível efetuar *load balancing* que uniformize mais as desigualdades entre processos mas o *overhead* associado à comunicação não o compensaria.

Cálculo do centro

Para o cálculo do centro é necessária a(s) projeção/ões correspondente(s) à(s) mediana(s). Estas são copiadas para a variável u nos processos que as contêm e, nos restantes, esta variável é colocada a zero. Se np for par e cada uma das medianas estiver nos processos centrais, cada um contribui com $u/2$. Através de um *MPI_Allreduce* com operação de soma, todos os processos ficam com a variável u, necessária para o cálculo do centro. A vantagem de ter o centro em todos os processos é que agora podemos paralelizar o cálculo do raio.

Cálculo do raio

O cálculo do raio consiste em calcular a distância máxima em todos os processos e depois, através de um *MPI_Reduce*, enviar o máximo para o processo zero, para este guardar o nó na estrutura respetiva.

Paralelização das chamadas do algoritmo

A paralelização do algoritmo consiste em utilizar a versão em paralelo com MPI até ao nível dado por $log_2(p)$, em que p é o número de processos inicial. Após esse nível, utilizamos a versão sequencial.

Note-se que, após cada chamada do algoritmo paralelizado, é necessário separar os processos tal que cada metade comunique entre si. Para isto utilizou-se o comando *MPI_Comm_split* que permite obter o novo número de processos e os *id's* respetivos.

Em relação à numeração dos *id's* da árvore, utilizou-se, até ao último nível em paralelo, $id_L = 2 \times id + 1$ e $id_R = 2 \times id + 2$. Para se poder utilizar a versão sequencial original a partir do nível seguinte, a numeração é alterada. Basta saber que o *id* do primeiro nó dos níveis sequenciais é dado por $2^{max\_level} - 1$ - em que max_level é o número de níveis que são feitos em paralelo - e que o número de nós a ser realizado por cada processo é dado por $2 \times size - 1$ - em que $size$ é o número de elementos em cada processo. Assim, somam-se os tamanhos dos vários blocos ao *id* do primeiro nó previamente calculado para obter os vários *ids*, como se pode ver na figura 4.

MPI + OpenMP

Com o intuito de melhorar a versão paralela usando apenas MPI, criou-se uma segunda versão em paralelo que utiliza MPI e OpenMP, para aproveitar o facto de cada um dos processos em questão ter várias *threads*.

Nos níveis em que se usava MPI pode-se usar também OpenMP para paralelizar alguns dos passos do algoritmo que envolvem *for loops*, nomeadamente: cálculo dos índices a e b; cálculo do primeiro *pivot* - ver Passo 1 do cálculo da mediana; no caso do número de pontos ser par, cálculo da segunda mediana como a projeção mínima à direita da primeira mediana; cálculo do centro e cálculo do raio.

Teve-se o cuidado de colocar todas as comunicações em MPI dentro de um comando *single*.

Após os níveis onde se utiliza MPI, em vez de se chamar a versão sequencial, chama-se agora a versão OpenMP (desenvolvida na primeira parte do projeto, com a otimização do uso da estrutura já referida). Esta, por sua vez, quando atinge o nível máximo até ao qual é possível paralelizar, chama a versão sequencial. Em anexo, na figura 5, encontra-se um exemplo com 4 processos com 4 *threads* cada.

Em relação à versão anterior com MPI apenas foi necessário alterar os índices a partir do nível em que se chama a versão com OpenMP porque mantê-los iguais à versão anterior envolveria ter uma secção crítica em cada chamada do algoritmo. Decidimos então manter a numeração igual àquela que é usada nos primeiros níveis paralelizados com MPI ao longo de toda a árvore. É apenas preciso ter cuidado no caso em que o número de pontos não é potência de dois, visto que o último nível da árvore não vai estar totalmente preenchido. Na versão apenas com OpenMP tinha-se usado um contador dentro de uma região crítica para numerar o último nível mas, neste caso, como os vários sub-ramos da árvore estão em processos diferentes, era difícil sincronizar essa variável entre todos. Cada processo tem então de calcular qual o primeiro *id* do último nível que vai preencher, o que é possível sabendo quantos elementos têm os processos à sua esquerda. Em anexo, na figura 6, encontra-se um exemplo da numeração e respetiva explicação.

Uma vez que agora a posição de um nó na árvore final não corresponde necessáriamente ao seu *id* (visto que a árvore está dividida pelos vários processos), acrescentou-se um atributo na estrututa da árvore para guardar o *id* do nó, e foi necessário usar mais uma região crítica na versão paralela com OpenMP a fim de incrementar corretamente a posição da árvore onde o nó atual é guardado. Isto representa um *overhead* acrescentado.

Resultados

Os resultados, tempos e *speedups* para a versão de MPI encontram-se na tabela 2. Para a versão de OpenMP+MPI, os tempos encontram-se na tabela 3 e os *speedups* encontram-se na tabela 4. Todos os tempos foram retirados apenas com uma *task* por nó / computador.

A fim de comparar estes resultados com as versões sequencial e de OpenMP, também os tempos e *speedups* destes figuram na tabela 1.

Começando pela versão MPI, é possível ver pelos valores da tabela e pela representação gráfica da figura 2 que os *speedups* aumentam, até um certo número de processos, a partir do qual voltam a decrescer. Isto leva a crer que aumentar o número de processos compensa o *overhead* associado à comunicação até cerca de 16 processos (na maioria dos casos), valor a partir do qual o *overhead* passa a ser o fator dominante. Também é possível notar que esta versão é, em geral, melhor do que a versão apenas com OpenMP. Existem duas possíveis razões para isto:

- A versão em OpenMP não paraleliza o cálculo da mediana, ao contrário da versão de MPI. Mesmo implementando um cálculo da mediana paralelizado na versão OpenMP semelhante ao usado em MPI, não se notam melhorias nos tempos, chegando a piorar em vários casos. Como tal, não deve ser esta a razão principal. (este código está anexado com o relatório);
- Acessos diretos a memória - Como os pontos são um bloco contínuo em memória e não são usados ponteiros para os mesmos, e dado que estão distribuídos pelos processos (logo, menos pontos em cada processo), a probabilidade destes estarem em cache é maior, aumentando o número de *cache hits* e a rapidez dos acessos, compensando até o facto de nas trocas de pontos se copiar o conteúdo inteiro da memória do ponto (ciclo em *n_dims*).

Em relação à versão OpenMP+MPI é importante referir que os tempos referentes a 16 processos são muito diferentes do que seria de esperar mas o uso elevado dos computadores de teste também não permitiu explorar se os tempos são de facto de confiança. Nota-se que, para um mesmo número de processos, um aumento no número de *threads*/CPUs usados leva a tempos mais reduzidos e melhores *speedups*, mas as melhorias são muito inferiores às observadas com apenas OpenMP. Contrariamente ao que seria de esperar,

para um mesmo número de processadores (CPUs×processos em OpenMP + MPI e processos em MPI), a versão OpenMP+MPI não apresenta, na generalidade dos casos, grandes melhorias. Esperava-se que assim o fosse, porque haveria uma diminuição na comunicação inter-processos (*overhead* muito superior a uso de memória partilhada). No entanto, algumas razões para tal não se verificar poderão ser, para além do já referido anteriormente, o facto de se acrescentar uma *layer* de sincronização (*omp critical*) nas chamadas de MPI, e o facto de também haver uma região critica na incrementação dos índices da árvore em todas as chamadas OMP.

Na tabela 4 os melhores *speedups* para cada um dos números totais de processos (número de processos × número de *threads*) está indicado com uma cor diferente. A maioria dos melhores *speedups* ocorre para os argumentos 3 20000000 0. Decidiu-se então realizar um gráfico com os *speedups* para este argumento em função do número total de processos para os diferentes números de *threads*. Note-se que o maior *speedup* obtido em todas as versões foi precisamente usando a versão conjunta, o que representa uma vantagem desta versão: permite simular a existência de mais processos, sem se ter o *overhead* associado à comunicação que se verificou para um número elevado de processos na versão exclusivamente MPI.

Foram ainda feitos testes com instâncias de dimensões muito elevadas (centenas de milhões). Note-se que para a instância maior (4 400000000), o problema não cabia na memória de um único computador, mas foi possível realizar tanto com a versão MPI como MPI+OpenMP. Para estas instâncias nota-se uma clara diminuição do tempo de computação com o aumento do número de processos, até um número de processos mais elevado (comparando com as restantes instâncias). Isto ilustra duas vantagens da versão com MPI: a possibilidade de resolver problemas de maiores dimensões e o facto de, para problemas de maiores dimensões, ser possível utilizar um maior número de processos até o *overhead* de comunicação ser dominante.

Anexos

Args	Seq.	*Threads*					
		1		2		4	
		T (s)	S	T (s)	S	T (s)	S
20 1000000 0	4.1	4.1	1.0	2.5	1.6	1.7	2.4
3 5000000 0	7.5	7.6	1.0	4.4	1.7	2.9	2.6
4 10000000 0	18.9	19.1	1.0	11	1.7	7.2	2.6
3 20000000 0	37.0	37.3	1.0	21.3	1.7	13.7	2.7
4 20000000 0	42.8	43.3	1.0	25.1	1.7	16.6	2.6

Tabela 1: Tempos médios e *Speedups* - versão OpenMP modificada

Args	Seq.	Processos													
		1		2		4		8		16		32		64	
		T (s)	S	T (s)	S	T (s)	S	T (s)	S	T (s)	S	T (s)	S	T (s)	S
20 1000000 0	4.1	4.2	1.0	2.5	1.6	2.0	2.1	1.3	3.2	1.4	2.9	1.5	2.7	3.7	1.1
3 5000000 0	7.5	7.3	1.0	4.1	1.8	2.7	2.8	1.8	4.2	1.9	3.9	2.7	2.8	4.5	1.7
4 10000000 0	18.9	18.6	1.0	9.9	1.9	6.3	3.0	4.7	4.0	4.1	4.6	5.6	3.4	9.4	2.0
3 20000000 0	37.0	36	1.0	19	1.9	12.2	3.0	8.4	4.4	8.1	4.6	8.5	4.4	17.2	2.2
4 20000000 0	42.8	41.8	1.0	21.6	2.0	14.4	3.0	10.2	4.2	8.8	4.9	9.8	4.4	18.6	2.3

Tabela 2: MPI - Tempos médios e *Speedups*

Args	Seq.	Processos															
		1			2			4			8			16			32	64
		1	2	4	1	2	4	1	2	4	1	2	4	1	2	4	1	1
20 1000000 0	4.1	4.4	2.6	1.7	3.2	2.4	2.3	1.9	1.7	1.3	1.8	1.5	1.4	4.8	4.7	4.6	1.8	3.1
3 5000000 0	7.5	7.8	4.6	3.2	4.1	3.1	2.6	2.8	2.6	2.3	2.7	2	1.8	6.5	6.2	6.4	2.6	4.3
4 10000000 0	18.9	19.9	11.6	8.0	10.1	7.4	5.5	6.9	6.5	5.3	4.9	4.8	4.0	13.9	13.6	13.8	5.9	10.2
3 20000000 0	37.0	38.5	22.4	14.9	19.0	13.3	9.3	12.2	11.0	9.7	8.8	9.1	7.1	29.5	27.3	27.4	16.1	16.0
4 20000000 0	42.8	45.0	26.1	17.6	22.5	17.6	11.1	15.5	13.3	11.6	11.8	11.4	9.4	33.7	30.1	31.6	19.0	18.6
Total		1	2	4	2	4	8	4	8	16	8	16	32	16	32	64	32	64

Tabela 3: OpenMP+MPI - Tempos médios

Args	Processos																
	1			2			4			8			16			32	64
	1	2	4	1	2	4	1	2	4	1	2	4	1	2	4	1	1
20 1000000 0	0.9	1.6	2.4	1.3	1.7	1.8	2.2	2.4	3.2	2.3	2.7	2.9	0.9	0.9	0.9	2.3	1.3
3 5000000 0	1.0	1.6	2.3	1.8	2.4	2.9	2.7	2.9	3.3	2.8	3.8	4.2	1.2	1.2	1.2	2.9	1.7
4 10000000 0	0.9	1.6	2.4	1.9	2.6	3.4	2.7	2.9	3.6	3.9	3.9	4.7	1.4	1.4	1.4	3.2	1.9
3 20000000 0	1.0	1.7	2.5	1.9	2.8	4.0	3.0	3.4	3.8	4.2	4.1	5.2	1.3	1.4	1.4	2.3	2.3
4 20000000 0	1.0	1.6	2.4	1.9	2.4	3.9	2.8	3.2	3.7	3.6	3.8	4.6	1.3	1.4	1.4	2.3	2.3
Total	1	2	4	2	4	8	4	8	16	8	16	32	16	32	64	32	64

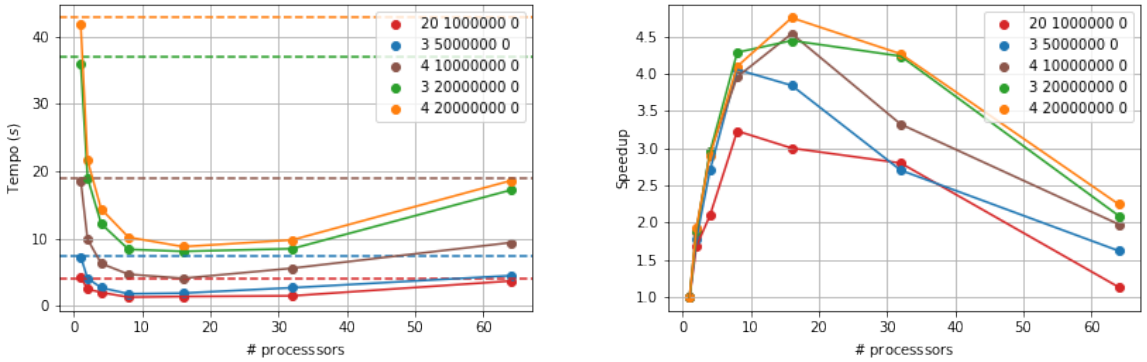
Tabela 4: OpenMP+MPI - *Speedups*

Args	Processos				
	4	8	16	32	64
4 100000000 0	77.1	54,0	47.0	40.8	79.2
4 200000000 0	177.9	113.7	96.6	80.6	172.2
4 400000000 0	-	232.3	209.6	161.1	340.6

Tabela 5: MPI - Tempos médios para maiores instâncias

Args	Processos		
	4	8	16
4 100000000 0	64.5	54	150.7
4 200000000 0	136.4	118.2	325.5
4 400000000 0	-	238.1	636.3

Tabela 6: OpenMP+MPI com 4 *threads* - Tempos médios para maiores instâncias



(a) Tempo em função do número de processos

(b) *Speedup* em função do número de processos

Figura 2: Tempos e *speedups* para a versão de MPI

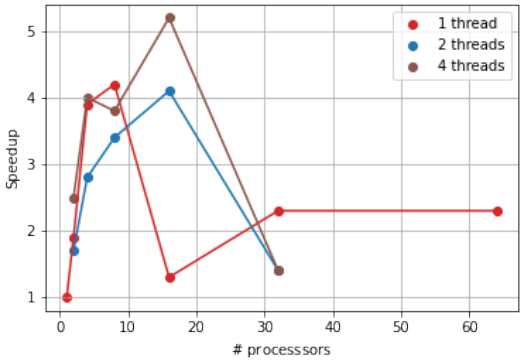


Figura 3: *Speedups* em função do número de processos para diferentes números de threads

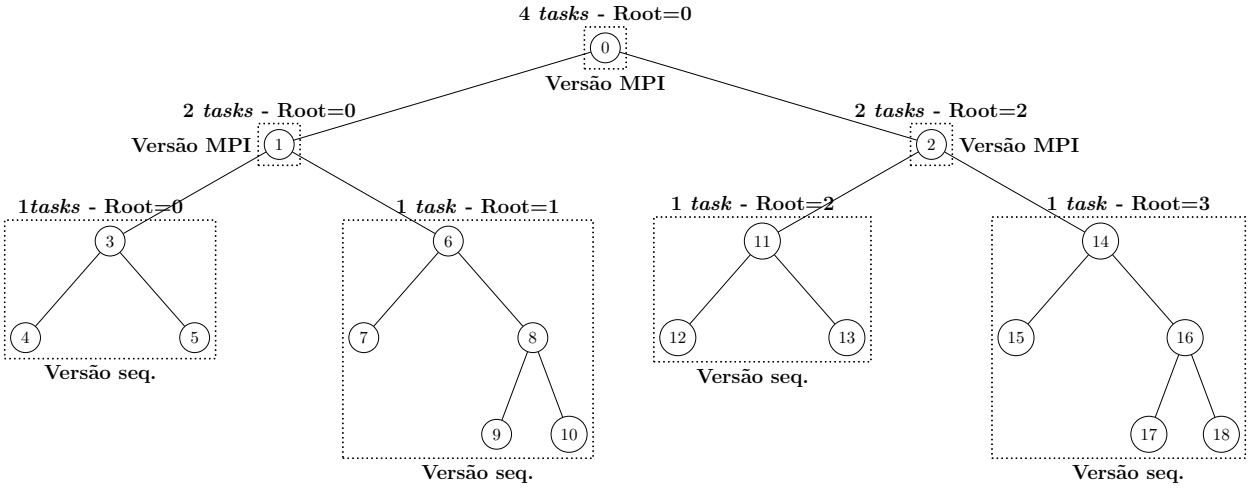


Figura 4: Exemplo de uma árvore utilizando 4 *tasks*.

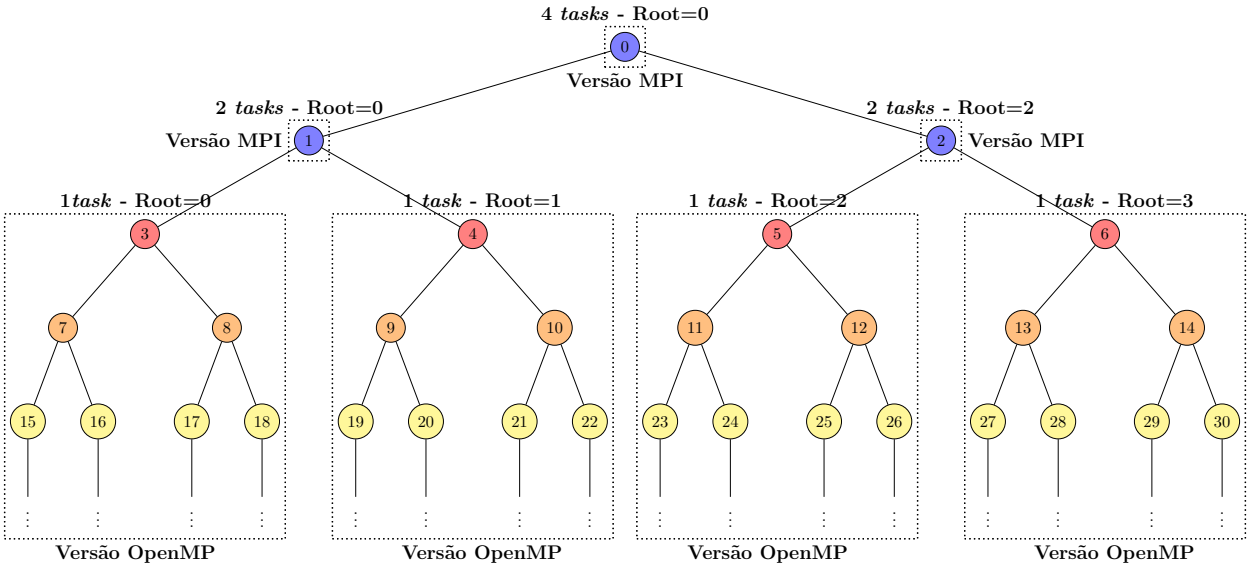


Figura 5: Exemplo de uma árvore utilizando 4 *tasks* com 4 *threads* cada. Os nós a roxo representam chamadas do algoritmo que usam paralelismo em MPI e OpenMP, sendo utilizado o número máximo de *threads* por processo. Os níveis a vermelho, laranja e amarelo representam chamadas do algoritmo paralelizado com OpenMP onde se utilizam 4, 2 e 1 *threads* em cada nó, respetivamente (todos feitos por apenas um processo / *task*).

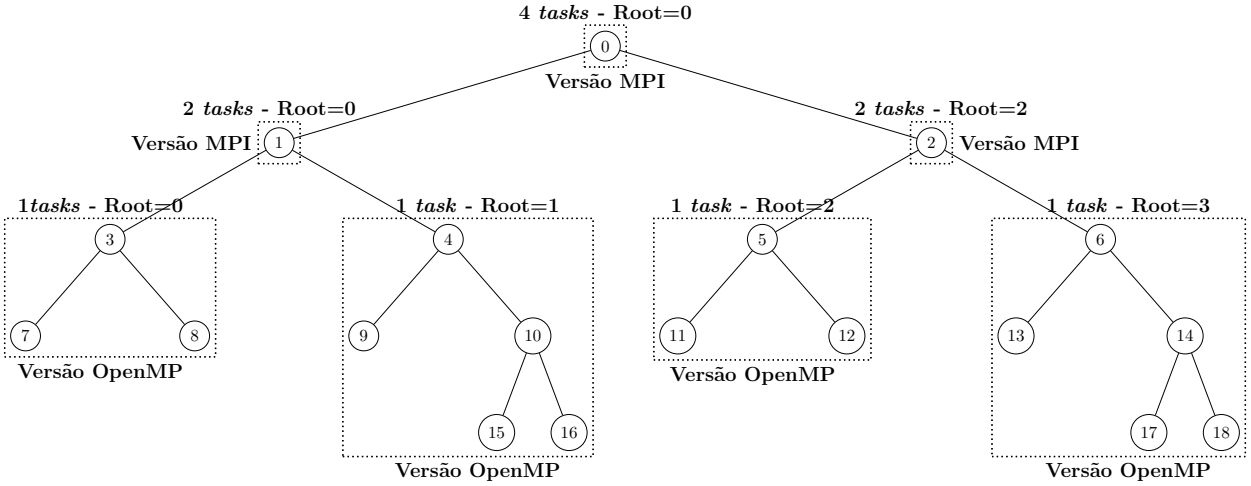


Figura 6: Exemplo da numeração de índices para a versão MPI+OpenMP - O segundo e quarto processos precisam de saber que os primeiros *id's* do último nível são, respetivamente, 15 e 17. Isto é feito da seguinte forma: primeiro, sabe-se que o primeiro *id* do último nível é dado por $2^{\lceil log_2(np) \rceil}$ - 1; a este número vai-se somar o número de elementos no último nível de todos os processos à esquerda daquele para o qual se quer calcular o *id*. Este é dado pelo número de nós total no sub-ramo que é realizado por esse processo, $2 * n\_points - 1$, menos o número de nós presentes em todos os níveis exceto no último, $2^{\lfloor log_2(n\_points) \rfloor + 1} - 1$.