Instituto Superior Técnico

Computação Paralela e Distribuída

CPD - OpenMP

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

2020/2021 - 2º Semestre
24.04.2021

Versão Sequencial

O algoritmo usado para a criação da árvore *Ball Tree* está descrito no enunciado do projeto. É, no entanto, importante realçar alguns aspetos relativos à sua implementação e a otimizações desenvolvidas:

- Vetores auxiliares de índices e de projeções - Todo o processamento e indexação do vetor inicial de pontos e do vetor das projeções é feito sobre um vetor de índices. Um índice inferior (l) e um superior (r) delimitam o troço do vetor sobre o qual se está a operar na chamada atual.
- Alocação de memória - Toda a memória é alocada antes da primeira chamada do algoritmo, dado que se sabe que a árvore terá um número total de $2 \times np - 1$ nós. (onde np é o número de pontos total)
- Numeração dos *ID's* na árvore - Os *ID's* dos nós da árvore são determinados de acordo com a ordem das chamadas recursivas da função: primeiro o nó atual, depois a sub-árvore esquerda e por fim a sub-árvore direita (varrimento pré-fixado).
- Vetor auxiliar com centros - A árvore final será constituída por $np - 1$ nós que representam centros e np folhas, correspondentes aos pontos iniciais. Assim, de modo a não copiar estes últimos para a estrutura da árvore, é apenas usado o seu índice. Os centros propriamente ditos são guardados num outro vetor auxiliar, de dimensão $np - 1$.
- Índices a e b - Para a determinação dos pontos a e b, começa-se por determinar o índice mais baixo do troço atual do vetor de índices, que poderá não estar na primeira posição (devido a reordenações aquando do cálculo da mediana).
- Cálculo das projeções - O critério usado para comparação aquando da determinação da mediana é o produto interno $(p-a)\cdot(b-a)$ (onde p é o ponto cuja projeção sobre ab se quer determinar), ao invés da primeira coordenada da projeção. Para garantir a reprodutibilidade dos resultados obtidos pelo corpo docente, a é sempre escolhido como o ponto com primeira coordenada mais baixa relativamente a b.
- Cálculo do índice correspondente à mediana - Para determinação da mediana optou-se pela seguinte abordagem: faz-se uma partição do vetor de acordo com um *pivot* - primeiro elemento do conjunto de dados considerado - sendo que à esquerda da sua posição final estarão apenas elementos não superiores e à direita elementos superiores. Se a posição final do *pivot* for o índice pretendido, está encontrada a mediana. Se for inferior (ou superior), repete-se o procedimento apenas para a partição à esquerda (ou direita) do *pivot*. Note-se que com este método apenas se tem uma ordenação parcial do vetor, sendo sempre mais eficiente que um *quicksort*, uma vez que apenas ocorre numa das partições. Como tal, determinar a mediana tem uma complexidade média $\mathcal{O}(n)$ (no pior caso pode ser quadrático, algo irrealista para dados aleatórios). Garante-se assim uma complexidade total do algoritmo de $\mathcal{O}(n\ log(n))$, o que foi comprovado experimentalmente (ver figura 2 (b)). Experimentou-se implementar um segundo algoritmo - *Median of Medians* - para determinação da mediana, de complexidade linear tanto média como de pior caso. No entanto, o *overhead* associado à escolha do *pivot* era muito elevado, levando a tempos bastante superiores, optando-se pela primeira opção.
- Mediana para número par de pontos - Se o número de pontos de um conjunto for par, é necessário encontrar duas "medianas"(pontos centrais). A primeira mediana é encontrada da maneira acima referida. A segunda mediana, ou seja, o primeiro elemento da partição à direita, é encontrada com o mesmo algoritmo chamado apenas para a parte direita do troço.
- Cálculo do centro - Para o cálculo do centro é necessário o cálculo de uma projeção vetorial, onde se usa a projeção escalar (produto interno), calculada anteriormente. Quando o número de pontos no conjunto é par, ao invés de se calcular duas projeções vetoriais e fazer a sua média, calcula-se apenas uma, usando para tal a média das projeções escalares de ambas as medianas.
- Cálculo do raio - Para efeitos de comparação utilizou-se o quadrado das distâncias entre pontos, calculando-se a raiz apenas quando a distância máxima é obtida.

Verificou-se experimentalmente a complexidade do algoritmo implementado, tanto em função das dimensões como em função do número de pontos. Observou-se que o algoritmo é linear nas dimensões do problema e $\mathcal{O}(n\ log(n))$ no número de pontos. Em ambos os casos, este é o comportamento esperado. Estes resultados podem ser observados nas figuras 2 (a) e 2 (b), em anexo.

OpenMP

Paralelização dentro de uma chamada do algoritmo

Começou-se por notar que iterações individuais do algoritmo não são muito paralelizáveis, visto que cada passo do mesmo depende do anterior: o cálculo do raio depende do centro, que depende da mediana, que depende das projeções que, por sua vez, dependem do cálculo de a e b. Devido a estas dependências foram colocadas barreiras no fim de cada uma destas etapas para garantir que não haja nenhuma *thread* que avança para o passo seguinte do algoritmo sem os anteriores estarem completamente concluídos. Alguns dos passos são paralelizáveis mas outros não. Note-se também que, em relação ao código sequencial, muitas das chamadas de funções foram substituídas por código *inline* para facilitar a paralelização.

Índices a e b

O cálculo destes dois índices consiste em três passos dependentes: calcular o índice mais baixo, calcular o a e calcular o b. Como estes três passos têm todos um padrão semelhante - um *for loop* com uma condição *if*, onde se vai substituindo uma variável que vai ser o máximo / mínimo - explicar-se-á apenas a paralelização do segundo passo, o cálculo do índice a.

Se se quiser paralelizar o *loop* tem de se garantir que a variável *max_d*, neste caso correspondente à distância máxima, é partilhada entre todas as *threads* e que estas não a substituem ao mesmo tempo. Tal pode ser feito definindo a variável antes de se entrar na região paralela e utilizando uma região crítica para a condição *if*. Isto envolve ter uma região crítica por iteração do *loop*, resultando em paralelismo quase nulo. Assim, decidiu-se seguir uma segunda abordagem (ilustrada no troço de código ao lado), onde se calcula o máximo em cada *thread* e depois, através de uma região crítica, obtém-se o máximo global através dos máximos de cada uma das *threads*. Assim tem-se apenas uma região crítica por *thread*. Note-se que foi necessário a colocação da barreira visto que o cálculo do b depende do valor de a e que, ao contrário do *omp single* e *omp for*, o *omp critical* não tem uma barreira implícita no fim.

```
1  long a = -1; // Indice global
2  double max_d = 0.0; // Maximo global
3  #pragma omp parallel
4  {
5    double d;
6    double max_d_t = 0.0; // Maximo privado de
       cada thread
7    long a_t = -1; // Indice privado de cada
       thread
8    #pragma omp for
9      for (int i = l; i < r; ++i) {
10       d = dist(pt_array[idx0], pt_array[idx[i
       ]]);
11       if (d > max_d_t) {
12         max_d_t = d;
13         a_t = idx[i];
14       }
15     }
16   #pragma omp critical // Todas as threads
       executam esta seccao mas nunca ao mesmo
       tempo
17   {
18     if (max_d_t > max_d) {
19       max_d = max_d_t;
20       a = a_t;
21     }
22   }
23   #pragma omp barrier
24 }
```

Projeções

O cálculo das projeções é paralelizado colocando o *loop* exterior em paralelo. Como cada uma das projeções é independente das outras não existem quaisquer problemas de sincronização.

Cálculo da mediana

O cálculo da mediana manteve-se sequencial, uma vez que o algoritmo usado não é paralelizável: é necessário fazer uma partição do vetor, algo que tem de ser feito sequencialmente, e o processo apenas é repetido para uma das partições, logo também não é possível a paralelização de chamadas do algoritmo.

Cálculo do centro

Todos os *loops* do código sequencial são paralelizados, mas foi necessário ter em conta que a variável onde é acumulado o valor do produto interno $(b-a)\cdot(b-a)$ é partilhada por todas as *threads*. Usou-se então o comando *reduction* com a operação de soma para esta variável.

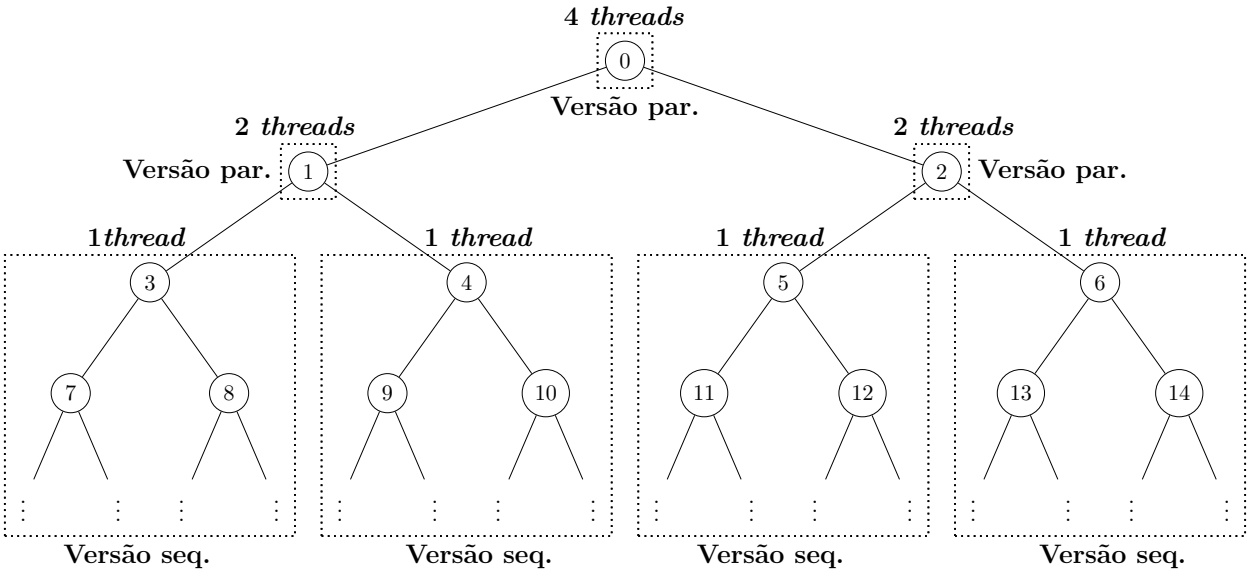
Cálculo do raio

A paralelização deste passo é em tudo semelhante à do cálculo de a e b.

Paralelização das chamadas do algoritmo

Como já foi referido, chamadas individuais do algoritmo não são altamente paralelizáveis. É então mais vantajoso ter várias chamadas do algoritmo sequencial a ocorrer em paralelo, do que apenas uma chamada de cada vez do algoritmo paralelizado. Tendo em conta esta ideia, numa primeira fase de implementação utilizava-se o algoritmo paralelizado (explicado na secção anterior) apenas para o primeiro nó, chamando-se recursivamente a versão sequencial do algoritmo para os nós filhos, utilizando o comando *task*. Também as chamadas subsequentes eram feitas dentro de *tasks*. Visto que a árvore é balanceada, a quantidade de trabalho das chamadas num mesmo nível é semelhante. No entanto, se se utilizar quatro *threads*, como cada chamada é executada apenas por uma *thread*, no segundo nível da árvore utilizar-se-iam apenas duas das quatro *threads* disponíveis.

Para maximizar a utilização de todas as *threads* em cada um dos níveis decidiu-se estender a utilização da versão paralela do algoritmo para além do primeiro nó. Definiu-se um nível máximo até ao qual se usa a versão paralelizada do algoritmo, dado por $\lceil log_2\,(n\_threads)\rceil$, em que $n\_threads$ é o número de *threads* definido pelo utilizador. Ainda, na versão paralela, o número de *threads* usadas num nível é metade do número usado no nível anterior a fim de garantir a existência de verdadeiro paralelismo entre chamadas do mesmo nível. Também se obtém um *overhead* menor em relação à criação de *threads* desta forma. Apresentamos então um exemplo para quatro *threads*:



Para usar mais do que uma *thread* em chamadas subsequentes do algoritmo é necessário permitir o uso de paralelismo *nested*. Isto é feito usando a função *omp_set_nested()*, chamada no início do programa.

É também de referir que se utilizou a numeração de índices BFS (Breadth First Search), conforme indicado no esquema acima, em vez da numeração pré-fixada usada na versão sequencial. Assim, o *id* dos filhos de um nó é dado pelo dobro do *id* do pai mais um, para o lado esquerdo, ou mais dois, para o lado direito. Como esta regra não é válida para o último nível da árvore, foi necessário definir uma variável global correspondente ao *id* deste nível, que é incrementada numa região crítica. Desta forma apenas se tem

uma região crítica no último nível da árvore, ao invés de se ter uma região crítica em todas as chamadas do algoritmo, o que seria necessário se se mantivesse o mesmo sistema de *id's* que na versão sequencial.

É importante referir que esta segunda abordagem foi pensada para um número de processadores que seja potência de dois, permitindo que o *load balancing* da árvore seja equilibrado. Caso contrário, é mais vantajoso usar a primeira abordagem (primeira chamada paralelizada e chamadas subsequentes sequenciais, usando *tasks*).

Assim, a versão final tem três funções que realizam o algoritmo recursivo: *ballAlg_par*, *ballAlg* e *ballAlg_tasks*. A primeira é utilizada quando se quer utilizar paralelismo dentro de uma chamada do algoritmo; a segunda é uma versão sequencial e a terceira é idêntica à segunda mas as chamadas recursivas do algoritmo para os nós filhos são feitas utilizando *tasks*. Se o número inicial de processadores a usar for potência de dois, utiliza-se o *ballAlg_par* para os primeiros níveis e depois o *ballAlg* a partir do momento em que todas os processadores já estão ocupados. Caso contrário utiliza-se o *ballAlg_par* para o primeiro nó e nas chamadas subsequentes utiliza-se a versão sequencial com *tasks*, *ballAlg_tasks*.

Por fim, é de referir que os vetores auxiliares de índices, centros e projeções podem ser mantidos como variáveis globais visto que as chamadas do algoritmo num mesmo nível operam sobre partes disjuntas dos mesmos, não havendo problemas de sincronização.

Resultados

Os tempos de execução para cada um dos conjuntos de argumentos foram obtidos através de uma média de dez execuções nos computadores do laboratório, e encontram-se representados na tabela 1. T representa o tempo, em segundos e S o *Speedup*. A representação gráfica dos dados obtidos encontra-se nas figuras 1 (a) e 1 (b), em anexo.

Args	Seq.	Processadores									
		1		2		3		4		8	
		T (s)	S	T (s)	S	T (s)	S	T (s)	S	T (s)	S
20 1000000 0	6.0	6.2	1.0	3.6	1.7	2.9	2.1	2.3	2.7	2.3	2.7
3 5000000 0	13.2	13.3	1.0	7.4	1.8	5.7	2.3	4.6	2.9	4.7	2.8
4 10000000 0	34.8	35.3	1.0	19.4	1.8	15.0	2.4	12.0	2.9	12.4	2.9
3 20000000 0	71.7	72.6	1.0	40.0	1.8	30.2	2.4	24.4	3.0	24.9	2.9
4 20000000 0	82.4	83.8	1.0	45.9	1.8	35.7	2.3	28.7	2.9	29.0	2.9

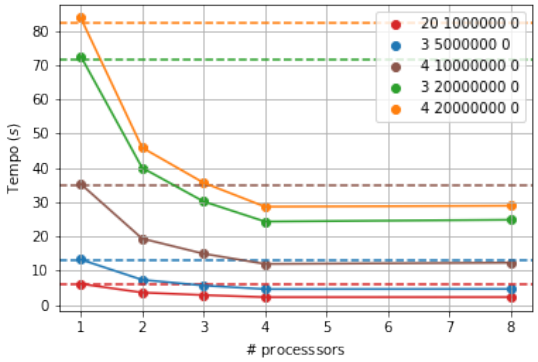
Tabela 1: Tempos médios e *Speedups*

Note-se que na figura 1 (a), as linhas horizontais representam os tempos sequenciais de cada um dos exemplos. É esperado que sejam ligeiramente inferiores aos tempos obtidos com uma *thread* visto que a criação desta tem um *overhead* associado.

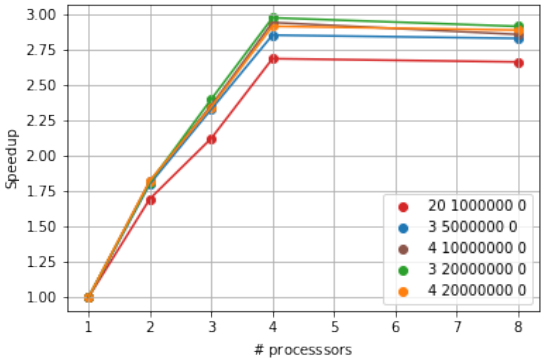
A estagnação dos tempos / *speedups* após as 4 *threads* leva à conclusão que este é o número máximo de processadores nos computadores do laboratório.

Na figura 1 (b) vê-se que os *speedups* seguem uma tendência quase linear, que é o comportamento ideal. Note-se que a abordagem usada para um número de processadores potência de dois é ligeiramente mais eficiente que a abordagem usada nos restantes casos: faz-se uso de paralelismo em mais chamadas do algoritmo e não se tem o *overhead* associado à criação de novas *tasks* quando já todos os processadores estão ocupados (que, apesar de ser mínimo, existe). Tal é especialmente notório no exemplo "20 1000000 0"na figura 1 (b), onde o *speedup* para três processadores está ligeiramente abaixo da linha formada pelos *speedups* de dois e quatro processadores.

Anexos

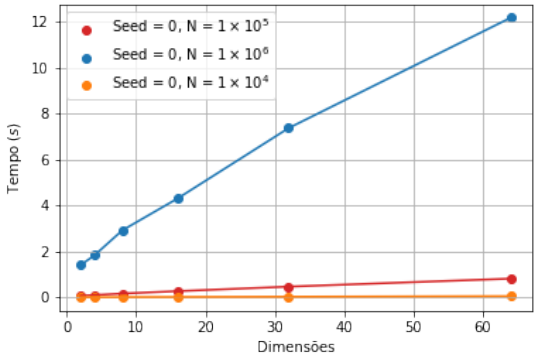


(a) Tempo em função do número de processadores

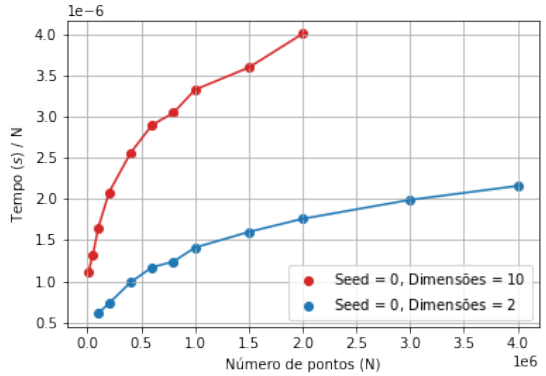


(b) *Speedup* em função do número de processadores

Figura 1: Tempos e *speedups*



(a) Tempo em função do número de dimensões



(b) Tempo em função do número de pontos

Figura 2: Variação dos tempos com as dimensões e o número de pontos