OpenMP QuickSort

Por Guilherme Caulada

1. O algoritmo em série

Quicksort é um algoritmo de ordenação popular que, em média, realiza O(n*log(n)) comparações para ordenar n elementos.
O problema pede para escrevermos uma versão paralela do Quicksort que pode utilizar outros algoritmos de ordenação, entretanto Quicksort deve ser a parte principal da solução.

A entrada possui apenas um caso teste com 100000000 elementos, um por linha, cada elemento deve ser uma string de 7 caracteres imprimíveis(0x21~0x7E ASCII) exceto pelo caractere ” ”(0x20 ASCII).

O arquivo de saída produzido possui todos os elementos, um por linha, organizados alfabeticamente.

O algoritmo apresentado utiliza a função qsort e a função strcmp para realizar a ordenação. A implementação da função qsort em C de 1993, é um pouco mais complexa, entretanto a lógica do algoritmo é a mesma, seleciona um elemento do array chamado de pivô, pode ser o primeiro, ou o último, ou de forma aleatória, ordena o array de forma que todos os elementos com valor menor que o pivô fiquei antes dele e todos os elementos maiores que o pivô fiquem depois dele, recursivamente aplica-se a mesma lógica para os sub-arrays de menores e maiores, até que cada sub-array tenha tamanho 0 ou 1. A função strcmp também possui uma implementação um pouco mais complexa, mas ela basicamente compara todos os caracteres de cada string atribuindo-as um valor e retornando um valor menor que 0 caso a primeira seja menor que a segunda, maior que 0 caso a primeira seja maior que a segunda, é igual a 0 se forem idênticas. Utilizando estas duas funç oes, o algoritmo compara e ordena o array de strings.
O algoritmo Quicksort possui uma complexidade O(n*log(n)) apenas no seu melhor caso, quando o pivô escolhido divide o array exatamente no meio, possuindo um valor médio comparado ao resto do array, caso o pivô escolhido seja o elemento de menor ou maior valor o algoritmo possui uma complexidade O(n^2). Vamos dizer que o Quicksort dividiu um array em 2 partes, uma de tamanho k outra de tamanho n-k, logo o tempo para ordenar n elementos será: T(n) = T(k) + T(n-k) + O(n), sendo que O(n) é o tempo para colocarmos o pivô em sua posição e T(n) é o tempo para ordenarmos o vetor por completo, logo se o elemento selecionado for o de menor tamanho temos que:

T(n) = T(n-1) + T(1) + O(n) = [T(n-2) + T(1)] + T(1) + O(n-1 + n) =
= [[T(n-3) + T(1)] +T(1)] +T(1) + O(n-1 + n-2 + n) = ...

Ou seja: T(n) = T(1) + (n-1)*T(1) + O(n*(n-2) - (n-2)*(n-1)/2).
Portanto o algoritmo possui uma complexidade O(n^2) no seu pior caso.
No seu melhor caso o Quicksort divide o array em duas partes com tamanho k = n/2, logo o

tempo para ordenar n elementos será: T(n) = 2T(n/2) + O(n), logo:

T(n)=2*T(n/2) + O(n) = 2*[2*T(n/4) + O(n)/2] + O(n) = 2*[2*[2*T(n/8) + O(n)/4] + O(n)/2] + O(n)...

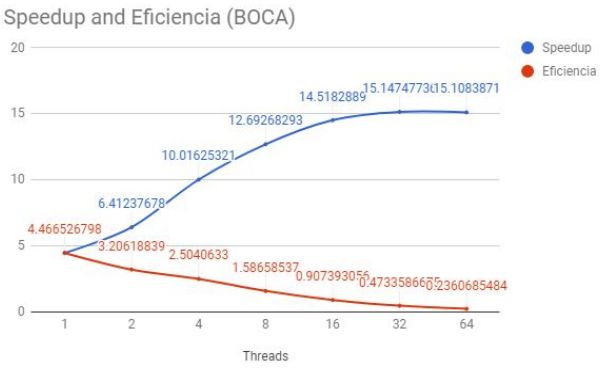
Ou seja: T(n) = (2^k)*T((n/2)^k) + O(k*n)/k.
Esta recursão deve se repetir até que n = 2^k, se não teremos n/2^k < 1. Portanto o valor máximo de k será log n, resultando em T(n) = n*T(1) + O(n)*log(n) que possui uma complexidade O(n*log(n)), o melhor caso do Quicksort.

2. Implementação paralela

A solução paralela desenvolvida busca otimizar o Quicksort realizando as suas recursões em threads paralelas entretanto sincronizando-as de maneira que duas threads não acessem a mesma posição do array. Para isso, define-se um valor de corte, e realiza-se a recurso do algoritmo de partição, até que o corte seja atingido, então cria-se duas novas threads para realizar essa recursão nos próximos blocos de corte.
Utilizando OpenMP, criamos um bloco de código paralelo com *#pragma omp parallel*, e especificamos uma área de código que será executada por apenas uma thread utilizando *#pragma omp single nowait* [1] em seguida realizamos a primeira iteração do Quicksort, que realiza o partition e então divide suas próximas recursões em tarefas paralelas utilizando dois *#pragma omp task* [2].

Ao executar essa implementação em um supercomputador com 64 cores conseguiu-se os seguintes resultados:

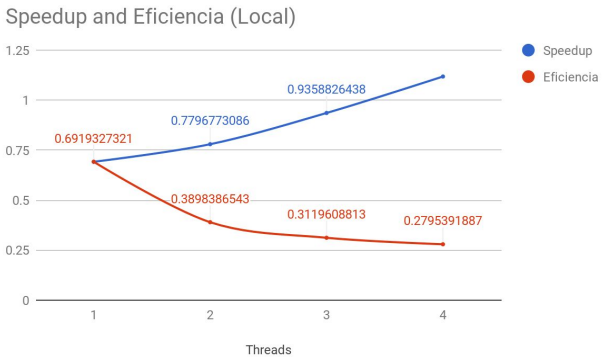
Threads	Speedup	Eficiência
1	4.466526798	4.466526798
2	6.41237678	3.20618839
4	10.01625321	2.5040633
8	12.69268293	1.58658537
16	14.5182889	0.907393056
32	15.14747736	0.4733586675
64	15.1083871	0.2360685484



Consegue-se identificar o paralelismo do algoritmo ao aumentar a quantidade de Threads, pelo aumento de seu Speedup, também identifica-se uma diminuição da Eficiência do paralelismo.

Ao executar este algoritmo em um computador comum, de 4 cores e 16Gb de RAM encontra-se os seguintes resultados:

Threads	Speedup	Eficiência
1	0.6919327321	0.6919327321
2	0.7796773086	0.3898386543
3	0.9358826438	0.3119608813
4	1.118156755	0.2795391887



Identificamos um speedup menor, entretanto temos resultados semelhantes, com o aumento do número de Threads temos o aumento do Speedup e a diminuição da Eficiência do paralelismo.

3. Conclusões

Devido a facilidade em manipular e ordenar dados, a quantidade de dados a ser manipulada tem grande impacto na eficiência do algoritmo paralelo. O algoritmo do Quicksort paralelo foi de fácil implementação entretanto seu speedup tem uma limitação (Gráfico 1). No computador local o algoritmo apresentou menor eficiência do paralelismo devido a limitação a quantidade de dados que poderia ser trabalhada quando comparado ao supercomputador, deixando claro uma relação da eficiência do algoritmo a quantidade de dados a ser trabalhada. Ao escolhermos um algoritmo de ordenação devemos levar em consideração a quantidade de dados a ser trabalhada para decidir se um algoritmo em série ou paralelo será mais eficiente.

4. Bibliografia

[1] IBM Knowledge Center, Pragma directives for parallel processing, prag_omp_single. URL:https://www.ibm.com/support/knowledgecenter/SSGH3R_12.1.0/com.ibm.xlcpp121.aix.doc/compiler_ref/prag_omp_single.html, Acessado em: 18/05/2018

[2] IBM Knowledge Center, Pragma directives for parallel processing, prag_omp_task. URL:https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.cbclx01/prag_omp_task.html, Acessado em 18/05/2018