Paralelização de Quicksort com OpenMP

Gustavo Henrique Stahl Müller[1]

[1]Universidade do Vale do Itajaí - UNIVALI

gustavomuller@edu.univali.br

Abstract. *This paper shows the performance difference between two different implementations of Quicksort: one sequential and the other parallel, using OpenMP. Both of them were created using C++.*

Resumo. *Este artigo demonstra a diferença de desempenho entre duas implementações diferentes do algoritmo de ordenação Quicksort: uma sequencial e a outra paralela, usando OpenMP. As duas versões foram criadas em C++.*

1. Introdução

Quicksort é um dos muitos algoritmos de ordenação. Logo, o seu objetivo é ordenar um conjunto de elementos, seja em forma crescente ou decrescente. Foi inventado por C.A.R. Hoare em 1960, quando visitou a Universidade de Moscovo como estudante. Naquela época, Hoare trabalhou em um projeto de tradução de máquina para o National Physical Laboratory. Ele criou o Quicksort ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais fácil e rápido. Foi publicado em 1962 após uma série de refinamentos.

O seu algoritmo é baseado no conceito de Divisão e Conquista *(Divide and Conquer)*, ou seja, procura quebrar um problema em diversas partes, e então aplicar a solução recursivamente nessas partes. Após todas as partes terem sido solucionadas, suas soluções são unidas para obter a solução final.

Dessa forma, podemos pensar em um jeito intuitivo de aplicar paralelismo ao Quicksort: dividimos o conjunto em várias subconjuntos (partes), e fazemos com que cada linha de execução *(thread)* ordene um desses subconjuntos. Como cada *thread* ordena um subconjunto único, uma *thread* nunca irá acessar a mesma área do conjunto que outra, consequentemente eliminando condições de corrida.

2. Fundamentação Teórica

Podemos dividir o algoritmo de Quicksort em duas etapas: partição e chamada recursiva.

Na etapa da partição nós escolhemos um elemento qualquer do conjunto. Chamamos ele de pivô. Após, encontramos todos os elementos menores que o pivô e criamos um subconjunto separado para eles. Depois, encontramos todos os elementos maiores que o pivô e também criamos um subconjunto separado para eles. Não é necessário de fato alocar dois subconjuntos para guardar os elementos menores e maiores, mas dessa maneira o algoritmo fica mais intuitivo. Matematicamente, o melhor elemento para servir de pivô será aquele mais próximo da média aritmética dos elementos do conjunto. Desse modo, teremos uma boa distribuição de carga (o subconjunto menor e maior terão aproximadamente o mesmo tamanho).

Na etapa de chamada recursiva, o Quicksort é executado duas vezes: uma vez sobre o subconjunto menor e outra sobre o subconjunto menor. Então, o processo de partição se repete para cada um deles.

A condição de parada é que o subconjunto sendo ordenado tenha uma quantidade de elementos igual à 1 ou menos, pois ele já estará naturalmente ordenado.

Após a condição de parada ser atingida, a função retorna o próprio subconjunto. A chamada que chamou a função que atingiu a condição de parada irá esperar pelo seu resultado, tanto para o subconjunto menor quanto o maior. Finalmente, o subconjunto menor, o pivô e o subconjunto maior serão concatenados (respectivamente) e esse resultado é retornado. Eventualmente, todas as chamadas retornarão e o resultado final será o conjunto ordenado.

3. Visualização

À seguir, temos uma representação gráfica do Quicksort ordenando um conjunto de 4 elementos, seguindo o método de partição explicado anteriormente.

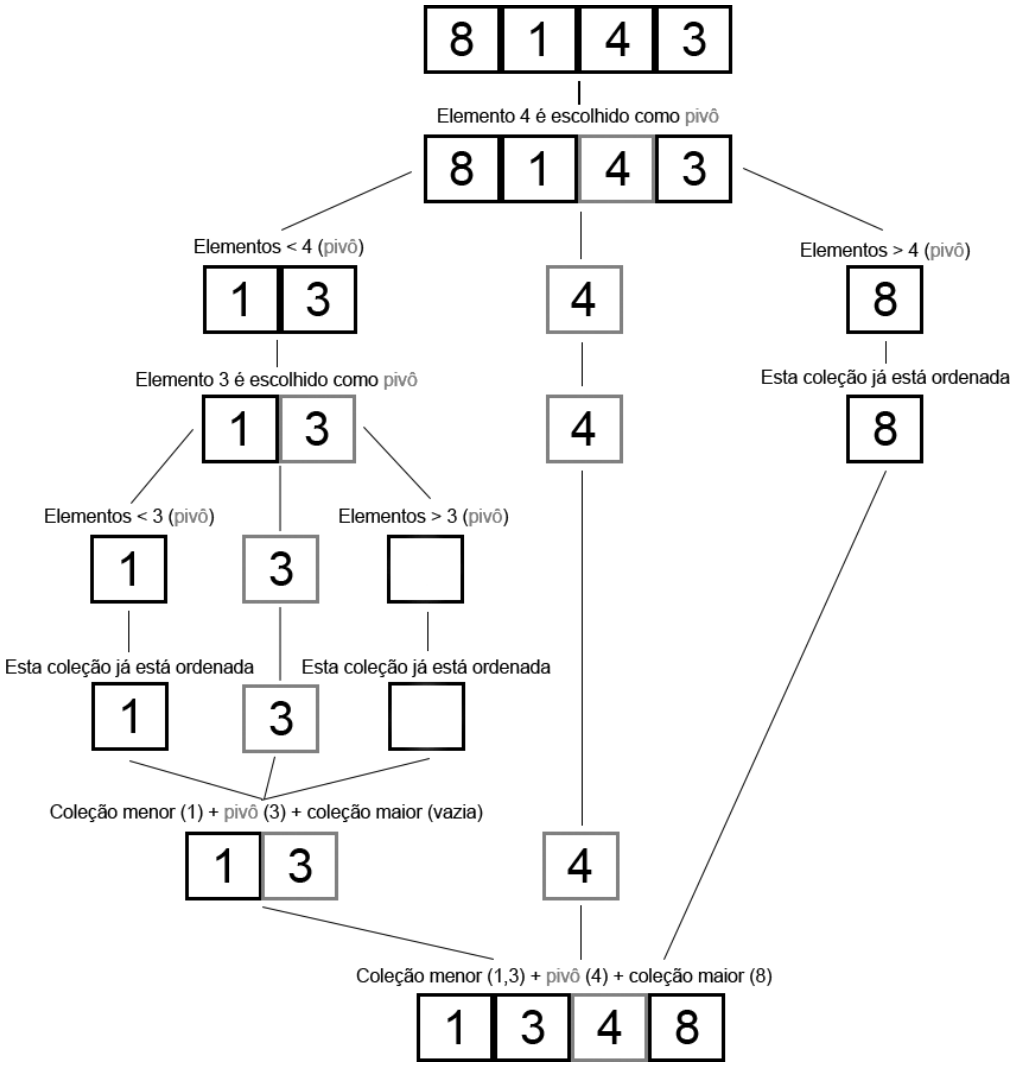

Figura 1. Quicksort aplicado em um conjunto de 4 elementos. Fonte: O autor.

4. Complexidade

Quicksort é utilizado com frequência no mercado de trabalho. Um dos motivos é a sua complexidade de espaço potencialmente pequena, dependendo de como a etapa de partição foi implementada. Como mencionado anteriormente, é possível realizar partições sem alocar subconjuntos adicionais na memória (*i.e* Método de Partição de Lomuto ou Método de Partição de Hoare). Desse modo, o algoritmo fica mais complexo, mas o menor uso de memória é preferenciado.

O pior caso em relação à complexidade de tempo é que o elemento selecionado para servir de pivô seja o maior ou o menor elemento do conjunto. Nesse caso, uma *thread* será responsável por ordenar um subconjunto de tamanho 0, e a outra será responsável por ordenar um subconjunto de tamanho *n-1*, onde *n* é o tamanho do conjunto que foi particionado. Logo, uma das *threads* ficaria ociosa enquanto a outra ficaria sobrecarregada, se assemelhando à execução sequencial.

5. Implementação

À seguir, temos a implementação do Quicksort em paralelo, que segue o Método de Partição de Hoare.

Aqui, são criadas todas as *threads* responsáveis pela ordenação do conjunto (através da diretiva **#pragma omp parallel**), mas somente uma delas irá chamar a função de fato, por causa da diretiva **#pragma omp single**.

```
auto start = omp_get_wtime();
#pragma omp parallel
{
    #pragma omp single
    {
        quicksort(array, 0, arrayLength-1, minSizeToParallelize);
    }
}
auto end = omp_get_wtime();
```

O Método de Partição de Hoare consiste em criar duas variáveis, **i** e **j**, que começam no início e no fim do conjunto, respectivamente. Nessa implementação, o valor pego como pivô sempre será aquele no meio do conjunto. O valor de **i** aumenta enquanto o valor de **j** diminui, até que **i** encontre um valor maior que o pivô e **j** encontre um valor menor. Nesse caso, temos um valor maior à esquerda e um menor à direita, logo, podemos trocá-los. Após fazer todas as trocas necessárias e **i** e **j** terem trocado de lados (**i** está na direita e **j** está na esquerda), ordenamos o vetor do início do conjunto até **j** (pois ele está na esquerda) e de **i** até o fim do conjunto (pois ele está na direita).

Caso **i** passe do fim do conjunto/chegue nele ou **j** passe do início do conjunto/ chegue nele, chegamos em uma condição de parada, pois temos um ou menos elementos à esquerda do pivô e um ou menos elementos à direita do pivô. Nesse caso, os únicos elementos que poderão ser trocados serão esses dois. Após a troca ter acontecido ou não, ele já estará ordenado (um subconjunto de tamanho igual ou menor à 1 é naturalmente ordenado).

Após a partição ocorrer, é checado se o tamanho do subconjunto é maior ou igual à variável **minSizeToParallelize**. Ela representa o tamanho mínimo para que um subcon-

junto crie tarefas paralelas para sua ordenação. Para um subconjunto de tamanho menor que essa variável, a criação de tarefas paralelas pode oferecer redução de desempenho, ao invés de ganho. Se esse for o caso, a própria *thread* que está executando a chamada atual executará tanto a chamada de ordenação do subconjunto menor quanto do maior.

```
int pivot = input[(startIndex + endIndex)/2];
int i = startIndex;
int j = endIndex;

while(i <= j) {
    while (input[i] < pivot) {
        i++;
    }

    while (input[j] > pivot) {
        j--;
    }

    if(i <= j) {
        swap(input[i], input[j]);
        i++;
        j--;
    }
}
```

Figura 2. Método de Partição de Hoare. Fonte: O autor.

```
if(endIndex - startIndex >= minSizeToParallelize){
    if(j > startIndex) {
        #pragma omp task
        {
            quicksort(input, startIndex, j, minSizeToParallelize);
        }
    }

    if(i < endIndex) {
        #pragma omp task
        {
            quicksort(input, i, endIndex, minSizeToParallelize);
        }
    }
} else {
    if(j > startIndex) {
        quicksort(input, startIndex, j, minSizeToParallelize);
    }

    if(i < endIndex) {
        quicksort(input, i, endIndex, minSizeToParallelize);
    }
}
```

Figura 3. Código de chamada recursiva do Quicksort paralelo. Fonte: O autor.

6. Testes

Vários testes foram feitos, tanto em sequencial quanto em paralelo. Ao final, foi medido a média aritmética de cada teste. O processador que executou tanto a versão sequencial quanto a paralela foi o Intel i5-10400F.

Núcleos	Threads	Frequência Base	Frequência Máxima
6	12	2.9GHz	4.3GHz

Tabela 1. Especificações básicas da CPU Intel i5-10400F. Fonte: O autor.

Na ordenação de cem milhões de elementos, a versão paralela demorou apenas 21.32% do tempo que a versão sequencial demorou.

	Teste 1	Teste 2	Teste 3	Média
Sequencial	13.124s	13.049s	12.875s	13.016s
Paralelo	2.55s	2.811s	2.968s	2.7763s

Tabela 2. Tempo gasto para ordenar cem milhões de elementos. Fonte: O autor.

Na ordenação de um bilhão de elementos, a versão paralela demorou apenas 19.454% do tempo que a versão sequencial demorou.

	Teste 1	Teste 2	Teste 3	Média
Sequencial	139.562s	140.021s	141.955s	140.512s
Paralelo	26.656s	27.542s	27.811s	27.3363s

Tabela 3. Tempo gasto para ordenar um bilhão de elementos. Fonte: O autor.

7. Conclusão

É possível concluir que a paralelização do algoritmo Quicksort aumenta seu desempenho consideravelmente, especialmente para as primeiras partições realizadas, e que a diferença de desempenho entre sequencial e paralelo continua aumentando à medida que o tamanho do conjunto também aumenta.

Referências

[1] Wikipédia, a enciclopédia livre. *Quicksort*. 12 de outubro de 2020.