**RELATÓRIO TÉCNICO**

**QUICK SORT COM OPENMP**

**Felipe Carvalho Funck**

**Ciência da Computação**

**Programação Paralela e Distribuída**

**Prof(ª). Dr(ª). Giani Carla Ito**

CAMPUS SANTA HELENA, 2019

**RELATÓRIO TÉCNICO**

1. **Introdução**

O presente trabalho apresenta uma abordagem sobre a biblioteca utilizada em programação paralela OpenMP. Será apresentado uma comparação de uma implementação realizada do algoritmo de ordenação Quick Sort paralelemente e sequencial. Uma analise do tempo de execução de ambas as implementações será realizada para apontar as principais vantagens da utilização da computação paralela neste tipo de trabalho.

1. **Metodologia utilizada**

O processo de instalação do OPENMP é bem simples utilizando a linguagem de programação C++, basta “Linkar” a biblioteca OPENMP com o compilador utilizado durante o processo de compilação de um programa em C++. O seguinte comando apresenta o processo de compilação de um programa com OPENMP em C++ utilizando o compilador gcc:

**gcc <nome\_arquivo.cpp> -o <nome\_saida> -fopenmp**

O comando acima ira compilar um programa com OPENMP e gerar um arquivo executável com o programa. O Anexo 1 apresenta a implementação sequencial do algoritmo Quick Sort utilizado neste trabalho.

O algoritmo Quick Sort segue o paradigma de programação “Dividir para conquistar” ou “divide and conquer”. Esse paradigma é uma abordagem recursiva em que a entrada do algoritmo é ramificada múltiplas vezes a fim de quebrar o problema maior em problemas menores da mesma natureza. O algoritmo utilizado a técnica de escolher um pivô e particiona o vetor dado ao redor do pivô escolhido. A técnica geralmente utilizada para se escolher o pivô é a escolha do elemento central do vetor. A partir da escolha do pivô uma varredura é feita no vetor, e todos os elementos menores que o pivô são posicionados a sua esquerda, e todos os maiores são colocados à sua direita. Após esta iteração acontecer todos elementos menores que o pivô estarão a sua esquerda e todos maiores a sua direita. A próxima etapa agora será ordenar o vetor resultante dos números menores que pivô anterior e o vetor resultante dos números maiores que o pivô aplicando a mesmo técnica, escolhendo um pivô para cada um desses vetores. Todo processo é realizado de forma recursiva até que todo vetor seja ordenado. Na forma sequencial do algoritmo todo esse processo é realizado de forma sequencial, ou seja, após a divisão dos vetores entre o pivô primeiro é aplicado a recursão para realizar a ordenação em um dos vetores e depois em outro. A paralelização neste algoritmo pode ser aplicada após a divisão dos vetores entre o pivô, quando for a aplicado o mesmo processo nos vetores resultantes, este processo pode acontecer de maneira paralela.

A Figura 1 apresenta as iterações que ocorrem no algoritmo Quick Sort, no exemplo na imagem o pivô escolhido é sempre o último número do vetor, apesar do aconselhável ser que seja escolhido o número do meio esta técnica também funciona.

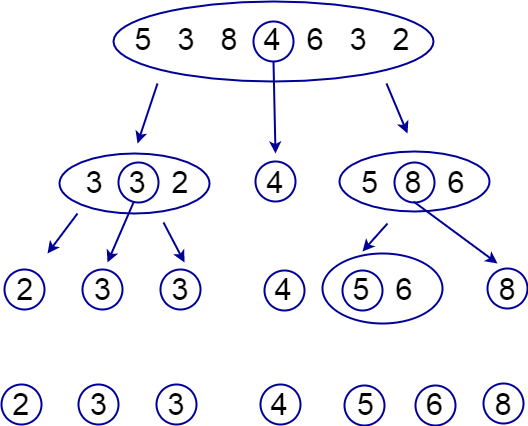


Figura 1. Iterações do algoritmo Quick Sort

Como ilustrado na Figura, o algoritmo vai criando várias ramificações, parecidas com uma arvore. A paralelização entrara em cada uma dessas ramificações. O algoritmo paralelizado do Quick Sort com a biblioteca OpenMP é representado no anexo 2.

Para a avaliação de desempenho de ambas implementações realizadas do algoritmo Quick Sort foram aplicadas variações de testes envolvendo o tamanho do vetor a ser ordenado e o número de threads utilizado pelo OpenMP. O mesmo conjunto de dados utilizado para testar o algoritmo sequencia foi utilizado para testar o algoritmo paralelo, para que a comparação realizada fosse junta.

A Tabela 1 representa o conjunto de variações aplicados durantes os testes realizados em ambas implementações do Quick Sort.

Tabela 1. Variações aplicadas em testes.

|  |  |
| --- | --- |
| Número de threads | Quantidade de números |
| 4 | 10.000 |
| 4 | 50.000 |
| 4 | 100.000 |
| 4 | 150.000 |
| 4 | 200.000 |
| 8 | 10.000 |
| 8 | 50.000 |
| 8 | 100.000 |
| 8 | 150.000 |
| 8 | 200.000 |

1. **Resultados Alcançados**

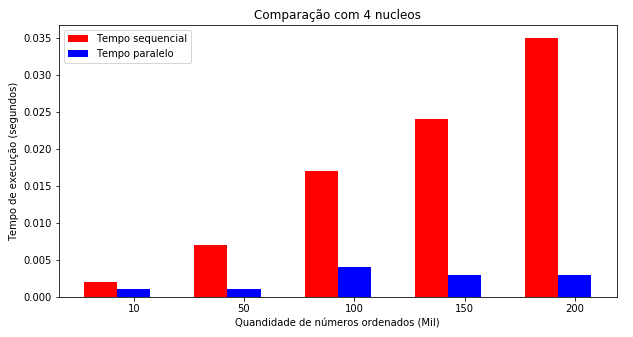
****

Figura 2. Resultados obtidos em comparação com 4 threads

1. **Informações Adicionais**

* Informações e detalhes não previstos neste modelo.
* Anexos e apêndices (se for o caso)

1. **Conclusões**

* Contribuição e conclusão do seu trabalho
* Sugestão para trabalhos futuros
* Recomendações

1. **Referências Bibliográficas**

* Livros e artigos utilizados

**Anexos**

**Anexo 1 – Implementação sequencial do algoritmo Quick Sort**

void quick\_sequencial(int \*vetor, int inicio, int fim) {

int pivo, aux, i, j, meio;

i = inicio;

j = fim;

meio = (int) ((i + j) / 2);

pivo = vetor[meio];

do {

while (vetor[i] < pivo) i = i + 1;

while (vetor[j] > pivo) j = j - 1;

if(i <= j) {

aux = vetor[i];

vetor[i] = vetor[j];

vetor[j] = aux;

i = i + 1;

j = j - 1;

}

} while(j > i);

if(inicio < j) quick\_sequencial(vetor, inicio, j);

if(i < fim) quick\_sequencial(vetor, i, fim);

}

**Anexo 2 – Implementação paralela do algoritmo Quick Sort**

void quick\_paralelo(int \*vetor, int inicio, int fim) {

int pivo, aux, i, j, meio;

i = inicio;

j = fim;

meio = (int) ((i + j) / 2);

pivo = vetor[meio];

do {

while (vetor[i] < pivo) i = i + 1;

while (vetor[j] > pivo) j = j - 1;

if(i <= j) {

aux = vetor[i];

vetor[i] = vetor[j];

vetor[j] = aux;

i = i + 1;

j = j - 1;

}

} while(j > i);

if (fim - inicio < LIMITE\_PARALELO) {

if(inicio < j) quick\_paralelo(vetor, inicio, j);

if(i < fim) quick\_paralelo(vetor, i, fim);

} else {

#pragma omp task

{ if(inicio < j) quick\_paralelo(vetor, inicio, j); }

#pragma omp task

{ if(i < fim) quick\_paralelo(vetor, i, fim); }

}

}