



UNIVERSIDADE FEDERAL FLUMINENSE
LABORATÓRIO DE PROGRAMAÇÃO PARALELA

Gustavo Luppi Siloto_217031126
João Victor Simonassi Farias _ 217031149
Leonardo Coreixas_217031137

QuickSort - Implementação Paralela

Análise dos tempos de execução do algoritmo

Niterói, 2022

Sumário:

1. Introdução;
2. Ambiente de testes;
3. QuickSort sequencial;
 - 3.1 Resultados obtidos;
4. QuickSort com MPI;
 - 4.1 Resultados obtidos;
5. QuickSort com OpenMp;
 - 5.1 Resultados obtidos;
6. Conclusão.
7. Referências bibliográficas

1. Introdução

Este relatório tem como objetivo comparar os resultados obtidos na execução do método Quicksort de forma sequencial e de forma paralela, utilizando o MPI e o OpenMP como formas de paralelismo.

O OpenMP é originalmente voltado para arquiteturas paralelas com memória compartilhada, enquanto MPI foi concebida para arquiteturas com memória distribuída com possíveis otimizações em arquiteturas multiprocessadas. Dito isto, será possível observar as principais vantagens de se usar diversos núcleos de processamento ou ainda quando é mais vantajoso executar uma aplicação de forma sequencial.

2. Ambiente de testes

Para manter a padronização nos experimentos e maior coerência na comparação dos dados, executamos todos os testes na mesma máquina, com arrays de inteiros e tamanho **1.000.000**; Para os testes em MPI, a aplicação foi executada com 8 processos; Para os testes em OpenMp, a aplicação foi executada com 8 threads.

Processador: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz

8ª geração de processadores Intel® Core™ i7

Número de núcleos: 4

Número de Threads: 8

Memória: 16GB

O resultado adicionado a este documento foi obtido na décima execução de cada aplicação.

3. QuickSort Sequencial

O QuickSort em sua implementação sequencial possui um algoritmo semelhante a este (Implementação completa, vide arquivo em anexo a atividade):

```
// Algorithm 10.5
// The sequential Algorithm of Quick Sorting
QuickSort(double A[], int i1, int i2) {
    if ( i1 < i2 ){
        double pivot = A[i1];
        int is = i1;
        for ( int i = i1+1; i<i2; i++ )
            if ( A[i] ≤ pivot ) {
                is = is + 1;
                swap(A[is], A[i]);
            }
        swap(A[i1], A[is]);
        QuickSort (A, i1, is);
        QuickSort (A, is+1, i2);
    }
}
```

Figura 1: Algoritmo de QuickSort sequencial

O Quicksort adota uma estratégia de divisão e conquista. Ela consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o quick sort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada.

Pode ser dividido nos seguintes passos:

1. Escolha um elemento da lista, denominado *pivô*;
2. Rearrange a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua

posição final e haverá duas sub listas não ordenadas. Essa operação é denominada *partição*;

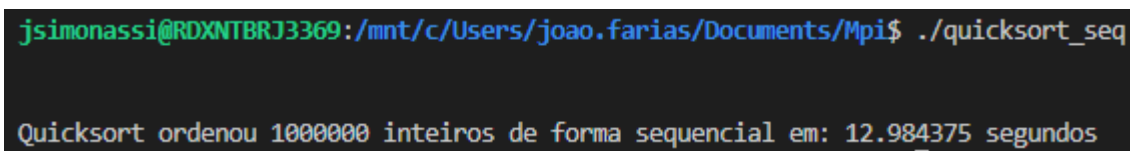
3. Recursivamente ordene a sub lista dos elementos menores e a sublista dos elementos maiores;

Dessa forma, teremos a lista ordenada ao final da execução do programa.

Todo o tempo de execução é contabilizado pelo método `clock()` da lib `time.h`

3.1 Resultados obtidos (QuickSort Sequencial)

Dada as condições de execução listadas na seção 2.0 deste documento, o tempo de execução para o array de 1.000.000 de inteiros foi: 12,984375 segundos



```
jsimonassi@RDXNTBRJ3369:/mnt/c/Users/joao.farias/Documents/Mpi$ ./quicksort_seq  
Quicksort ordenou 1000000 inteiros de forma sequencial em: 12.984375 segundos
```

Figura 2: Tempo de ordenação com QuickSort sequencial

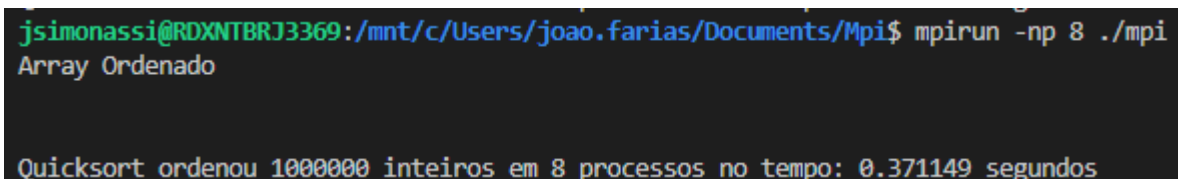
4. QuickSort com MPI

Utilizamos os métodos do MPI para seguir a estratégia de subdividir o array principal no número de processos em execução. Cada processo recebe um “pedaço” do array e aplica o QuickSort de forma sequencial.

Feito a ordenação das subdivisões, o array é mergeado novamente até se tornar uma única lista. Todo o tempo de execução é contabilizado pelo método `MPI_Wtime()`.

4.1 Resultados obtidos (QuickSort com MPI)

Dada as condições de execução listadas na seção 2.0 deste documento, o tempo de execução para o array de 1.000.000 de inteiros foi: **0,371149 segundos**.



```
jsimonassi@RDXNTBRJ3369:/mnt/c/Users/joao.farias/Documents/Mpi$ mpirun -np 8 ./mpi  
Array Ordenado  
Quicksort ordenou 1000000 inteiros em 8 processos no tempo: 0.371149 segundos
```

Figura 3 : Tempo de ordenação com MPI

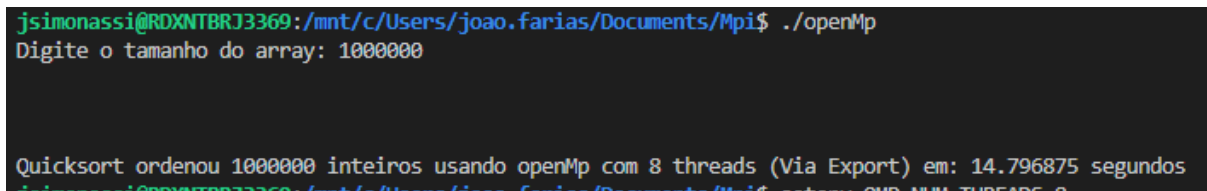
5. QuickSort com OpenMp

Utilizamos as diretivas do openMp para seguir a estratégia de enviar as subdivisões feitas pelo algoritmo do QuickSort, isto é, cada vez que o algoritmo é chamado novamente via recursão, esta execução é feita em uma nova Thread como consequência de: `#pragma omp section`.

Não é preciso realizar quaisquer tipos de merge no array, uma vez que a memória é compartilhada. Todo o tempo de execução é contabilizado pelo método `clock()` da `lib time.h`

5.1 Resultados obtidos (QuickSort com OpenMp)

Dada as condições de execução listadas na seção 2.0 deste documento, o tempo de execução para o array de 1.000.000 de inteiros foi:



```
jssimonassi@RDXTBRJ3369:/mnt/c/Users/joao.farias/Documents/Mpi$ ./openMp
Digite o tamanho do array: 1000000

Quicksort ordenou 1000000 inteiros usando openMp com 8 threads (Via Export) em: 14.796875 segundos
jssimonassi@RDXTBRJ3369:/mnt/c/Users/joao.farias/Documents/Mpi$ cat openMp.c | grep -o "#pragma omp num_threads 8"
```

Figura 4 : Tempo de ordenação com OpenMp

6. Conclusão

Podemos concluir que paralelizar o QuickSort é válido (principalmente usando MPI). Como a aplicação possui um número alto de comparações e iterações, o MPI leva vantagem pois possui a habilidade de segmentar a mesma em diversos processos, fazendo um melhor uso dos recursos e núcleos de processamento disponíveis. Para o experimento realizado, tivemos uma melhora de cerca de 97%. O OpenMp apresentou um resultado similar ao sequencial, contudo, acreditamos que para um número maior de dados, ele seria mais performático, dado as características de memória compartilhada. Para o cenário atual, o overhead de criação de threads o fez gastar mais tempo.

Não conseguimos utilizar um tamanho maior de array, pois ficou inviável para execução em nossa máquina de testes.

7. Referências bibliográficas

Material fornecido como sugestão de aplicação.

<https://www.tutorialspoint.com/explain-the-quick-sort-technique-in-c-language>

https://www.dartmouth.edu/~rc/classes/intro_openmp/compile_run.html

<https://sol.sbc.org.br/journals/index.php/reic/article/download/1072/942/2082>