Relatório Programação paralela

Professora: Luciana Drummond
Alunos: Allan Patrick de Freitas Santana, Arthur Vieira Silva, Victor Ferreira Teixeira

Objetivo

O objetivo deste laboratório foi a implementação de 2 algoritmos de ordenação de forma paralela utilizando **OpenMP** e **MPI**, o caso de estudo é denotado por uma lista de **1024 chaves**, com **4 processos** (MPI) e com **4 threads** (OpenMP).

Metodologia

Foi escolhida a utilização do **Bitonic-Sort** com **OpenMP**, durante nossas pesquisas descobrimos que ele é um algoritmo com aproveitamento de paralelização de quase 100%, enquanto que possui uma complexidade de *O(log²(n)) - um pouco elevada em relação aos demais candidatos* - e foi escolhido o **Merge Sort** com **MPI**, pois ele é um algoritmo de baixa complexidade O(*n*log(n)*) com maior taxa de paralelização que o algoritmo do **Quicksort**, chegando a possuir até **39%** menos comparações que o **Quicksort** enquanto que o **Bubblesort** possui uma alta complexidade O(n²) comparado às demais opções apresentadas.

Montagem

Para executarmos o algoritmo **Mergesort** com **MPI** utilizamos:

1. *Compilar: mpicc -o merge_paralelo merge_paralelo.c*
2. *Executar: mpiexec -np 4 ./merge_paraleloclear*

Para executarmos o algoritmo **bitonic-sort** com **OpenMp** utilizamos:

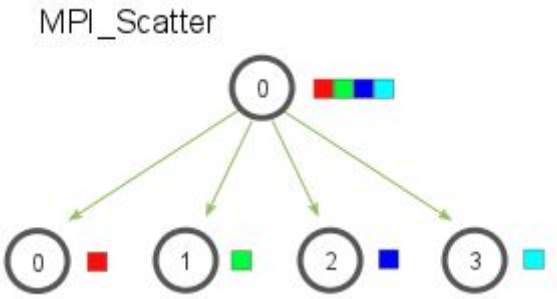
1. *set OMP_NUM_THREADS=4*
2. *gcc -fopenmp bitonic_openmp.c -o bitonic*

Repositório com o código: https://github.com/victorl2/algoritmos-paralelos-ordenacao

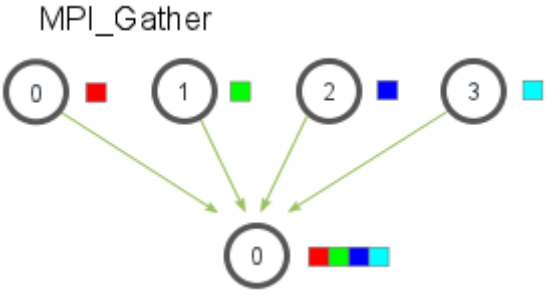
Experimento

Mergesort

Reproduzimos um algoritmo **Mergesort** padrão em C, encapsulado pelas rotinas **MPI** para podermos paralelizar o nosso código, além das rotinas padrões utilizamos também a **MPI_Scatter** para enviar de forma simples os valores do array as várias threads, facilitando assim a realização do merge sort



e a **MPI_Gather** para reunir os valores que foram dispersados pelo **MPI_Scatter** e reuni-los de volta ao array.



Bitonicsort

Reproduzimos um algoritmo **Bitonic-sort** padrão em C, apesar das nossas pesquisas não tivemos êxito em realizar a paralelização completa do algoritmo. Conseguimos apenas paralelizar uma das iterações internas. Ou paralelizávamos a criação da sequência bitônica ou a fase de trocas. Optamos por deixar a paralelização na criação da sequência.

Resultados e análise

Utilizamos além do tamanho pedido do array, arrays de tamanho 256 e 512 para fins comparativos e testamos também com apenas uma thread para comparar com algoritmos não paralelizados.

Array size	**Time**			
	Merge Sort	**Parallel MergeSort**	**Bitonic-sort**	**Bitonic-sort Parallel**
256	**0,002s**	**0,004s**	**0,003s**	**0,004s**
512	**0,003s**	**0,004s**	**0,003s**	**0,004s**
1024	**0,003s**	**0,007s**	**0,004s**	**0,006s**
1048576	**0,734s**	**0,558s**	**0,864s**	**0,476s**

Conclusão

Concluímos pela tabela que o **Mergesort** não paralelizado é mais eficiente que o **Bitonic-sort** contudo quando paralelizamos o **Bitonic-sort** se torna uma alternativa mais eficiente visto que o algoritmo é muito paralelizável. Além disso pode ser observado que para listas pequenas paralelizar não é uma boa idéia visto que o gerenciamento de threads é algo custoso.