Trabalho de implementação
Computação concorrente (MAB-117) - 2021/2
Implementação de algoritmo merge sort multithreaded

Carolina Naccarato Nunes - 117220395
Lucas Guimarães Batista - 114136076

1

1. Descrição do problema

O problema escolhido por nós foi a implementação do algoritmo de Merge Sort de forma concorrente. Merge Sort é um algoritmo conhecido de ordenação que recebe um vetor não ordenado como entrada e o retorna ordenado. No caso da nossa implementação, a entrada consiste em um vetor de inteiros. A implementação padrão desse algoritmo é sequencial e baseada no princípio de ”dividir e conquistar”, ou seja, dividir um problema maior em problemas menores até que o problema seja pequeno o suficiente para ser resolvido diretamente. No caso do algoritmo em questão a divisão é feita particionando o vetor ao meio sucessivamente até chegarmos no caso base em que a ordenação seja elementar (comparar um inteiro com outro). Por apresentar essa lógica natural de divisão de trabalho, o Merge Sort se mostra um bom candidato a melhora de performance com uma implementação concorrente, já que esses problemas menores podem ser resolvidos por threads diferentes. A nossa decisão de implementar paralelismo no Merge Sort se dá por alguns fatores. Primeiro, problemas de ordenação são constantes no dia a dia de um desenvolvedor. Inúmeras aplicações voltadas para usuários finais utilizam métodos de ordenação, tornando esse tema um de extrema relevância. Além disso, optamos por ele também pelo fato de ser um assunto muito bem documentado online, o que fez com que a pesquisa e implementação do tema se tornem mais fáceis quando comparadas a um tema mais obscuro.

2. Projeto e implementação da solução concorrente

A tarefa principal que o algoritmo quer realizar é ordenar um vetor de inteiros e as subtarefas que ele realiza são duas: separar o vetor ao meio em dois novos vetores e comparar esses dois vetores. Uma forma bem evidente de implementar paralelismo nesse processo é deixar que a comparação de vetores menores seja realizada por mais de uma thread. Inicialmente tentamos lançar threads que faziam o processo inteiro, ou seja, a divisão e a ordenação de porções do vetor, mas nos deparamos com vários problemas. A divisão de um vetor em duas metades é uma tarefa mais simples que a tarefa de ordenar um vetor, então optamos por separar o vetor na função principal, mantendo essa parte do processamento sequencial, e entregar posteriormente entregar o vetor divido às threads para elas realizarem a ordenação.

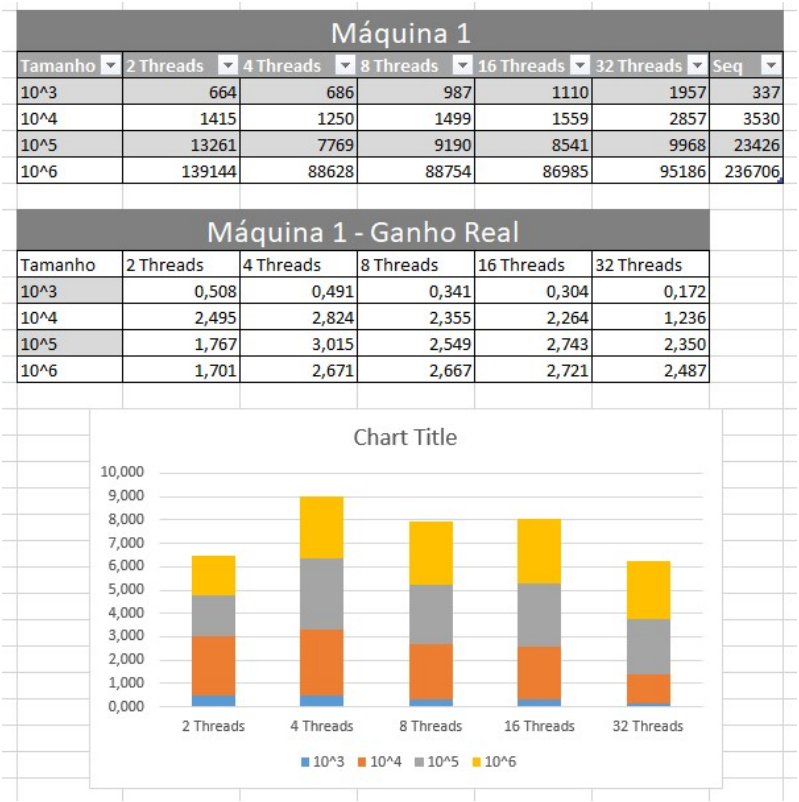
3. Casos de teste

A avaliação da corretude do algoritmo é feita utilizando uma função implementada por nós e rodada junto ao programa principal. Essa função itera sobre o vetor após a aplicação do algoritmo de ordenação e acusa caso ele não esteja ordenado, confirmando de fato

se a execução foi bem sucedida ou não. Essa função é executada dentro do programa principal a cada chamada da main, então cada execução do programa testará a corretude após a execução do algoritmo de ordenação. O conjunto de casos de testes elaborados envolve repetidas execuções do programa variando tamanho de vetor e quantidade de threads utilizadas somente. Como o vetor é criado aleatoriamente a cada execução do programa, não vimos razão para variar ainda mais o conteúdo desse vetor de entrada. Na seção seguinte há um detalhamento maior do casos de teste aliados aos resultados apresentados.

4. Avaliação do desempenho

A bateria de testes foi rodada em duas máquinas diferentes para fins de validação de resultados. As características da máquina 1 são: CPU Intel i7 3630QM @ 2.4GHz - 4 Cores/8 Threads - OS Kubuntu 20.04 64-bits. As características da máquina 2 são: CPU AMD Ryzen 7 2700U - 4 Cores/8 Threads - OS Manjaro Linux 21.1.6. Ambas baterias de teste foram feitas repetindo cada combinação de parâmetros 5 vezes e tomando como medida de tempo para cálculo de aceleração o menor tempo dentre os obtidos. A unidade de tempo da primeira tabela de cada imagem é microssegundo. Os valores mostrados na segunda tabela de cada imagem é um número absoluto que representa o ganho de desempenho, calculado segundo a fórmula $Ganho = t_{seq}/t_{conc}$ O cálculo de aceleração As tabelas com resultados, cálculos e gráficos relevantes podem ser vistos abaixo:



Máquina 1

Tamanho	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads	Seq
10^3	664	686	987	1110	1957	337
10^4	1415	1250	1499	1559	2857	3530
10^5	13261	7769	9190	8541	9968	23426
10^6	139144	88628	88754	86985	95186	236706

Máquina 1 - Ganho Real

Tamanho	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
10^3	0,508	0,491	0,341	0,304	0,172
10^4	2,495	2,824	2,355	2,264	1,236
10^5	1,767	3,015	2,549	2,743	2,350
10^6	1,701	2,671	2,667	2,721	2,487

Figure 1. Resultados dos testes realizados na máquina 1

Máquina 2						
Tamanho	2	4	8	16	32	Seq
10^3	288	397	542	853	1256	151
10^4	1601	1222	1224	1787	2320	1940
10^5	15826	10347	9452	10023	12484	22025
10^6	165496	121529	98030	115035	124962	277203

Máquina 2 - Ganho Real					
Tamanho	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
10^3	0,524	0,380	0,279	0,177	0,120
10^4	1,212	1,588	1,585	1,086	0,836
10^5	1,392	2,129	2,330	2,197	1,764
10^6	1,675	2,281	2,828	2,410	2,218

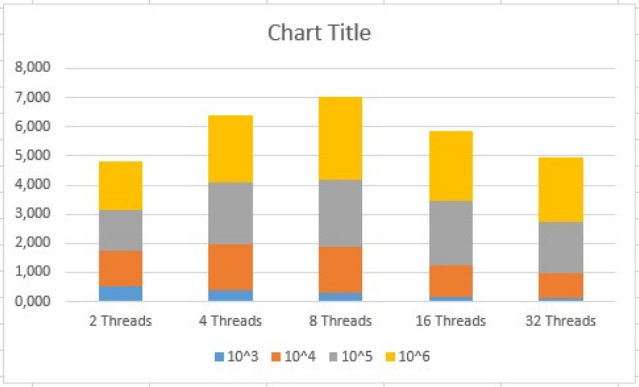


Figure 2. Resultados dos testes realizados na máquina 2

5. Discussão

Houve um ganho claro de desempenho na execução concorrente quando comparamos os tempos dela com os da versão sequencial. Esse ganho escala de forma linear com a quantidade de threads em execução, o que é o comportamento esperado, mas só até uma certa quantidade de threads. Na máquina 1, o melhor desempenho veio utilizando 4 threads, enquanto que na máquina 2 o melhor desempenho foi utilizando 8 threads. A provável explicação para isso é a diferença de hardware entre as máquinas, uma vez que a máquina 1 possui um processador bem mais antigo. Acreditamos que a implementação está em um ponto ótimo e que não há mais muito espaço para melhorias no que tange código. O problema de ordenação é um problema resolvido já e com algoritmos conhecidos para tal. Caso quiséssmos buscar uma melhoria de performance para esse problema, a melhor opção seria escolher algum outro algoritmo de ordenação que possua desempenho melhor ou manter a utilização do algoritmo atual e melhorar as configuyações do hardware utilizado. Sobre dificuldades encontradas ao longo do trabalho, acreditamos que a maior deles se deu logo no início. Como reportado anteriormente, o plano inicial de disparar threads diferentes no handler de threads era claramente equivocado mas demoramos a notar, o que consumiu uma quantidade de tempo razoável da nossa implementação. A solução foi bem simples e similar à utilizada por nós no Lab 1 da disciplina: dividir o array em metades no fluxo principal e atribuir essas metades as threads para que elas executem a carga pesada do trabalho.

6. Referências

Vrajitoru, D. *B424 - Parallel and distributed programming*, Curso de programação paralela e distribuída da Indiana University, disponível em *https://www.cs.iusb.edu/~danav/teach*