Programação Paralela
Trabalho II

Giovanni Cupertino, Matthias Nunes, *Usuário pp12820*

I. Introdução

O objetivo do trabalho é desenvolver uma solução que ordene um vetores, em casos de teste diferentes, utilizando o algoritmo do bubble sort. Os vetores utilizados possuem dois tamanhos que são de cem mil elementos e outro com um milhão que estão na ordem inversa de valores e devido as explicações durante a realização do trabalho não foram encontrados problemas para a implementação.

Para abordagem paralela do trabalho, utilizou-se o modelo de divisão e conquista criando uma estrutura de árvore binária. Para esta abordagem cada nó da árvore decide se vai ser dividido ou conquistado por meio de um valor fixo denominado delta, no qual, caso o vetor seja maior que o delta, o processo divide o vetor para seus dois filhos, em partes iguais, e os seus filhos repetem o processo até que o vetor seja menor ou igual ao delta, optando assim por conquista-lo. Para conquistar o processo executa o algoritmo de ordenação e depois devolve o pedaço do vetor já ordenado para o seu pai que terá de fazer o método de intercalação- que consiste em juntar vetores ordenados e gerar um novo também ordenado- com os dois vetores que irá receber e repetir o processo a raiz onde se terá o valor ordenado, após a intercalação.

Para otimizar o algoritmo, que tem o trabalho de ordenação somente nas folhas da árvore, e para não manter os processos, que não são folhas, esperando foi criada uma versão otimizada que consiste em ter a ordenação de parte do trabalho no processo local e a outra parte ser passada dividida para os dois filhos diminuindo o tempo que os processos ficam em espera.

II. Análise dos Resultados Obtidos

A analise dos dados foi feita baseada no pior caso, do problema apresentado, que é o vetor de um milhão de posições. Para o caso do vetor de cem mil posições também foram coletados os tempos e podem ser observados na tabela.

Em primeira analise é possível observar que no caso sem otimização o tempo de resposta diminui e o speed-up aumentou, ultrapassando até mesmo o speed-up ideal com o aumento do número de processos, isso ocorre devido não só ao fato de ter mais processos em execução mas também pelo algoritmo de ordenação possuir uma notação $O(n^2)$ que permite a cada divisão no vetor uma melhora quadrática no tempo para a ordenação da nova parte. É possível observar também que a eficiência também aumenta com o aumento do número de processos em execução, entretanto de 15 para 31 threads a eficiência diminui um pouco já que devido a estrutura ser de uma árvore binária o número de vezes que o método de intercalação vai precisar ser executado e a quantidade de mensagens enviadas vai aumentando bastante por altura da árvore o que reduz um pouco o benefício ganho com a divisão. Outra questão e que devido ao trabalho de ordenação estar somente nas folhas, neste caso, é necessário eles esperar toda a ordenação por elas para depois realizar as intercalações necessárias o que pode deixar os processos esperando por bastante tempo sem realizar nenhuma tarefa.

Versão Normal					
Núcleos	Tempo(s)	Speed-Up	Speed-Up Ideal	Eficiência	100k elementos(s)
1	4200	1,0	1	1.0	-
3	1118,725	3.8	3	1,3	11,197
7	283,095	15	7	2,1	2,864
15	73,926	56,8	15	3,8	0,911
31	36,979	113,6	31	3,7	0,379
Versão Otimizada					
1	4200	1,0	1	1,0	-
3	504,497	8,3	3	2,8	5,030
7	92,283	46	7	6,5	0,924
15	21,100	199,0	15	13,3	0,360
31	9,868	425,6	31	13,7	0,115

Tabela I: Resultados obtidos para 1000000 e tempo para 100000

Observando que ao executar em paralelo, sem otimizar, vários processos ficavam esperando para realizar uma tarefa, a versão otimizada tem como modo de resolução para este problema ter uma parte do vetor para ordenar localmente e realizar a divisão do resto dele para para seus dois filhos(mesma quantidade para cada um e caso sobre um pouco ele ordena esta parte) até um ponto em que o vetor passado é menor ou igual ao delta. Para determinar o delta, que é a quantidade do vetor que será ordenado localmente para o caso otimizado, utilizamos o tamanho do vetor original dividido pelo número de processos, depois de pegar sua parte o nodo pai distribui igualmente o resto da tarefa. Com isso foi possível obter resultados muito melhores do que o algoritmo sem otimização visto que ele começa a distribuir parte menores de trabalhos nas divisões e utiliza o tempo que os outros processos demoram para realizar a sua ordenação local, mesmo ainda existindo uma perda de tempo pelas trocas de mensagens e para o método de intercalação.

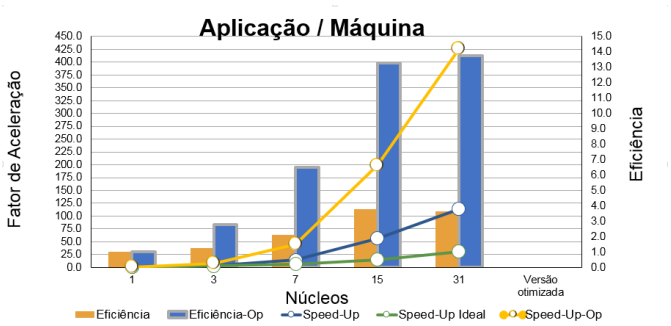

Figura 1: Gráfico gerado a partir da tabela

O fato de se estar utilizando a biblioteca MPI e dois nós da maquina atlantica permitiu um balanceamento da carga entre os núcleos e threads destes nós e há uma possível perda nos tempos de resposta para a comunicação que não teve relevância para a análise realizada. A utilização de mais de 16 processos(hyper-threading, já que passa da soma dos núcleos dos dois nós) apresentou uma melhoria significativa nos tempos de resposta e no speed-up devido a permitir dividir o problema em mais pedaços e tirar proveito do algoritmo de ordenação. A utilização de outro algoritmo de ordenação mais rápido permitiria um tempo de execução menor mas não seria observado tamanha diferença entre seus valores para diferentes números de processos como foi possível observar com o bubble sort.