Programação Paralela Trabalho 2

Bruno Brandelli, Rodrigo Pacheco, Usuário pp12804

I. Introdução

Este trabalho tem como principal objetivo analisar a eficiência, speed-up e alguns outros fatores da execução de um algoritmo de divisão e conquista realizando o ordenamento de um vetor utilizando processos paralelos. Tal analise será feita apartir da observação da execução do algoritmo em forma sequencial, com três variações de carga, 10000, 100000 e 1000000 elementos organizados de forma inversa de contagem dentro de um vetor, de forma a utilizar o pior caso no algoritmo de ordenamento bubblesort. Outro cenário que será observado é a diferença de eficiência quanto ao ordenamento apenas nas ultimas folhas da árvore, e a utilização de ordenamento local junto do envio de carga aos filhos.

II. Divisão e Conquista

Serão desenvolvidas duas versões do algoritmo executando de forma paralela, uma mais simples que utiliza o conceito básico de divisão e conquista, onde o trabalho é dividido, em algum ponto conquistado, e por fim intercalado os resultados da divisão. E uma versão que busca o menor tempo ocioso dos processos, realizando uma ordenação local enquanto espera o resultado da conquista.

Na versão mais simplória do algoritmo desenvolvido, os processos que ficam na camada mais abaixo da arvore realizam o bubblesort, enquanto os processos em que estão mais acima apenas fazem a combinaçao destes resultados, já na versão que busca um aproveitamento maior dos processos, todos os processos farão o bubblesort em uma fração do vetor. A técnica de divisão e conquista aplicada neste trabalho consegue tirar muito proveito da programação paralela, visto que a divisão das tarefas que ocorre é repassada entre os processos.

III. Análise de Resultados

Os algoritmos foram executados usando 2 núcleos do cluster Grad, onde cada núcleo tem 8 processadores, e são capazes de rodar 16 processos utilizando o Hyper-Threading. Na figura 1 temos os resultados encontrados utilizando o vetor com 1000000 elementos, tal decisão foi tomada devido à maior necessidade de processamento, embora os testes com as outras configurações de vetores tenham sido realizadas.

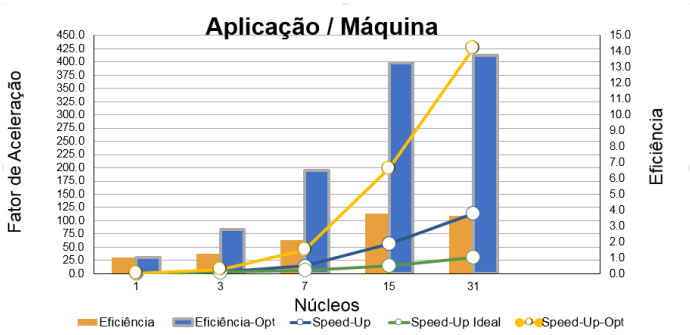

Figure 1. Gráfico de eficiência/speed-up

Este trabalho mostrou um comportamento interessante, porque devido ao fato da ordenação ter uma complexidade quadrática, ao dividir o problema entre os processos, o ganho de desempenho também é quadrático. Isto leva a execução dos algoritmos em paralelo à um speed-up maior do que o calculado como ideal.

Ao utilizar o algoritmo sem otimização, a carga de trabalho mais pesado encontravam-se nas folhas do ultimo nível da árvore, onde era realizado o ordenamento dos pedaços menores do vetor, e as camadas superiores apenas intercalavam estes pedaços de vetores, que é uma tarefa muito mais facil computacionalmente. No caso do algoritmo otimizado, o pai sempre ficava com metade do vetor para ordenar localmente, e após receber o trabalho dos seus filhos, ele intercalava os 3 vetores, ou seja, quanto mais ao topo da árvore, mais carga de trabalho o processo recebia, mas apesar disto, como pode ser visto na figura 1, ele tem um ganho de eficiência devido ao fato de haver menos processos ociosos, porque com uma divisão de metade da carga de trabalho entre pai e filhos , o pai gasta quase a mesma quantidade de tempo ordenando sua porção do vetor que os filhos que talvez tenham que dividir seu trabalho novamente.

O hyper-threading demonstra um ganho de eficiência até um certo ponto, já que para ativa-lo é necessário chamar mais processos. E uma grande quantidade de processos pode gerar muitas camadas para a árvore, que reduz a eficiência devido ao processo de intercalação que ocorre em cada camada.