Exercício Prático 2 de Programação Paralela

Seleção Aleatória

Douglas Rodrigues de Almeida
douglasralmeida@live.com

1. Introdução

O objetivo do trabalho foi a implementação de um algoritmo que calcule, de um conjunto de números, o i-ésimo menor número deste conjunto.

O algoritmo utilizado foi o Seleção Aleatória (do inglês Randomized-Select), que é baseado no Quicksort, que utiliza partições randômicas para separar os números do conjunto de acordo com um pivô escolhido aleatoriamente.

2. Implementação sequencial

O Seleção Aleatória se baseia no paradigma de dividir e conquistar. Ele é divido em dois passos para escolher o iésimo elemento de um vetor A[e..d].

Dividir: O vetor A[e..d] é particionado em dois subvetores A[e..p-1] e A[p+1..d] tais que cada elemento de A[e..p-1] é menor ou igual a A[p], aqui chamado de pivô, e todos os elementos de A[p+1..d] é maior ou igual ao pivô. O pivô é escolhido aleatoriamente.

Conquistar: Caso o pivô não seja o iésimo elemento, um dos subvetores, A[e..p-1] ou A[p+1..d], aquele que contém o iésimo elemento, será particionado recursivamente pela mesma regra da divisão.

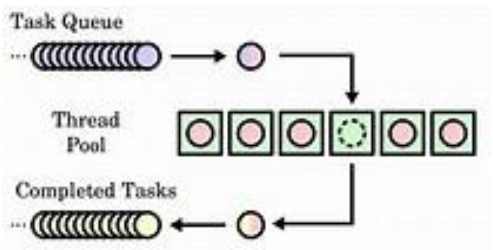
Em algum momento, ou pivô será o iésimo elemento, ou o subvetor resultante da divisão terá apenas um único elemento que será o iésimo procurado.

89	32	75	55	49	69	50

50 <55	49 <55	32 <55	55	89 >55	69 >55	75 >55

3. Implementação paralela

A versão paralela do algoritmo foi implementada usando a biblioteca Pthreads. Foi criado um pool, ou piscina de threads, uma estrutura de dados que mantém uma quantidade fixa de threads alocadas previamente que recebem tarefas para serem executadas.



O paralelismo ocorre na partição do vetor quando os valores são copiados para o subvetor que está sendo criado após a checagem com o pivô. Cada tarefa de transferência de elemento para a futura posição no subvetor será uma tarefa a ser executada pela piscina de threads. Após a execução de todas as threads da fila, um novo subvetor estará formado. Daí o pivô escolhido aleatoriamente poderá ser o iésimo elemento, ou ocorrerá um novo particionamento do subvetor gerado.

A cada chamada recursiva que gerar um novo subvetor, será criada uma fila de tarefas para a piscina de threads.

4. Avaliação

Os testes foram feitos em uma máquina da Google Cloud com 8 CPUs e 32GB de RAM.

Entrada N	10^3	10^4	10^5	10^6	10^7
Tempo (s) (sequencial)	0,000013	0,000087	0,003857	0,321868	31,6157
Tempo (s) (paralelo) p/ 2 threads	0,002400	0,008666	0,124731	1,498747	17,314892
Tempo (s) (paralelo) p/ 4 threads	0,006456	0,066019	0,410576	6,672222	
Tempo (s) (paralelo) p/ 8 threads	0,014050	0,076790	1,156507	15,33921	
Tempo (s) (paralelo) p/ 16 threads	0,017732	0,252677	2,97247	16,1832	

Percebe-se que a implementação sequencial é mais rápida que a versão paralela. A versão paralela realiza uma várias alocações de memória que a deixa mais lenta.

O *speedup* entre a versão sequencial e a paralela para é sempre abaixo de 1,0.

A implementação não é fortemente escalável pois a medida que aumentamos o número de CPUs, mantendo o número da entrada ocorre perda de eficiência. Ela também não é fracamente escalável pois a medida que variamos a entrada juntamente com o número de CPUs também ocorreu perda de eficiência.