Relatório Trabalho 2 - Programação Paralela

Pedro Amaral Chapelin
Pedro Willian Aguiar

Introdução

Para esse trabalho, foi pedido para que implementássemos o algoritmo dos **K vizinhos mais próximos**. Esse problema da programação consiste no seguinte: São duas matrizes de entrada, que representam pontos e suas coordenadas. Cada linha representa um ponto, e cada coluna os valores de suas dimensões (*x, y, z…*), sendo a matriz **Q** de tamanho **nq** x **d** (*nq* = número de pontos em q, *d* = quantidade de dimensões dos pontos) e a matriz **P** de tamanho **np** x **d** (*np* = número de pontos em p) as matrizes de entrada, vale ressaltar que só as dimensões de cada matriz foram passadas como entradas, o conteúdo de cada uma é preenchido aleatóriamente. Além das dimensões de cada, o programa também recebe o número **k** como argumento, indicando quantos vizinhos mais próximos serão calculados para cada ponto.

A ideia principal dos k vizinhos mais próximos funciona da seguinte maneira: O programa deve devolver uma matriz **R** de tamanho **nq** x **k** que contenha as distâncias e os índices dos **k** pontos de **P** que são mais próximos de cada ponto em **Q**. Agora explicado como funciona o algoritmo, partimos para a implementação do mesmo. Basicamente, para cada ponto de **Q** foi necessário varrer todos os pontos de **P**, e para ir armazenando as menores distâncias calculadas utilizamos a estrutura **Heap** cedida pelo professor. Ao final de cada varredura por todos os pontos de **P** e com a Heap pronta, tudo que tínhamos que fazer era copiar o conteúdo dela para linha da matriz **R**.

Feita a implementação, agora o desafio era paralelizar esse processo todo usando a biblioteca *mpi.h*, para isso, toda a execução era baseada em nodos e processos que realizariam o trabalho dividido. Primeiramente, antes de entrar na função principal, definimos que o preenchimento das matrizes com números aleatórios seria realizado apenas no nodo 0, juntamente com a medição do tempo, utilizando a biblioteca do professor *chrono.h*. Logo após isso, realizamos a chamada da função principal:

```
// Execução principal do programa
        calculaDistancias(Q, nq, P, npp, d, k, R, nq / nproc,
MPI_COMM_WORLD, processId);
```

Passamos todos os dados das matrizes juntamente com os dados necessários para paralelizar com o MPI. **nq** / **nproc** significa o que cada nodo pegará da matriz **Q** para calcular, assim realizando uma divisão do processo todo pegando o número total de pontos em q (**nq**) pelo número total de processos passados na compilação (**nproc**). O MPI é responsável por dividir e matriz **Q** entre os nodos(fução *MPI_Scatter*), enviar a matriz **P** completa para todos (função *MPI_Broadcast*), e no fim da função, juntar todos os resultados calculados paralelamente (*MPI_Gather*).

Depois de alcançado o passo do paralelismo, agora é hora de testar e anotar a diferença observada nos testes. Para testarmos, foi disponibilizado para nós um cluster de computadores, todos com as mesmas características a seguir de processamento:

Experiência: Achar K menores com MPI									
	INFOS obtidas do programa lscpu sobre o hardware rodado no cluster								
Processador:	Intel(R) Xeon(R) CPU			E5462					
CPU MHz	2792,84								
L1d cache	256 KiB								
L1i cache	256 KiB								
L2 cache	24 MiB								

Para todas as execuções, os parâmetros internos do programa foram os mesmos:

*knn-mpi 128(**nq**) 400000(**np**) 300(**d**) 1024(**k**)*

Entretanto, foram realizadas 3 experiências diferentes com parâmetros do MPI e do Cluster:

1- **Rodar o programa para APENAS 1 processo MPI e medir o tempo da computação de knn**

sbatch –exclusive -N 1 knn-mpi-slurm.sh

*mpirun -np 1 knn-mpi 128(**nq**) 400000(**np**) 300(**d**) 1024(**k**)*

2 - **Rodar o programa para 4 processos MPI no mesmo host e medir o tempo da computação de knn**

sbatch –exclusive -N 1 knn-mpi-slurm.sh

*mpirun -np 4 knn-mpi 128(**nq**) 400000(**np**) 300(**d**) 1024(**k**)*

*3 - **Rodar o programa para 4 processos MPI em hosts diferentes e medir o tempo da computação de knn***

sbatch –exclusive -N 4 knn-mpi-slurm.sh

*mpirun -np 4 knn-mpi 128(**nq**) 400000(**np**) 300(**d**) 1024(**k**)*

Alunos:
Pedro Amaral
Pedro Willian

Primeiro Teste	
Tempo (Segundos)	Vazão (MB/s)
27.611743	556.285060
27.998788	548.595171
28.479297	539.339155
28.277272	543.192426
27.491169	558.724869
28.44815	539.929656
27.64	555.792407
27.977651	549.009628
28.497226	538.999827
27.494236	558.662549

sbatch --exclusive -N 1 knn-mpi-slu
mpirun -np 1 knn-mpi 128 400000 300 1024

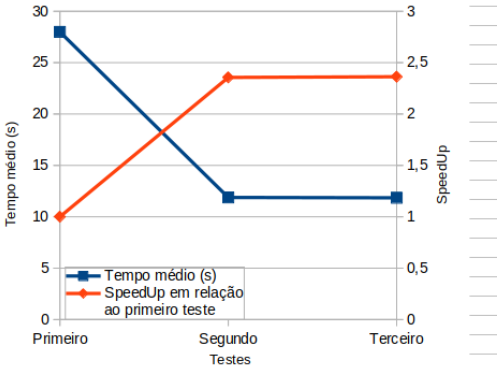
Segundo Teste	
Tempo (Segundos)	Vazão (MB/s)
11.821159	1299.36499
11.86764	1294.2759
11.835271	1297.815608
11.909201	1289.759039
12.156639	1263.507143
11.725739	1309.938758
11.88	1293.14952
11.778158	1304.108803
11.87175	1293.827781
11.967188	1283.509577

sbatch --exclusive -N 1 knn-mpi-slurm.sh
mpirun -np 4 knn-mpi 128 400000 300 1024

Terceiro Teste	
Tempo (Segundos)	Vazão (MB/s)
11.923369	1288.226474
11.803004	1301.363596
11.825171	1298.924142
11.868306	1294.203247
11.888636	1291.990147
11.894246	1291.380771
11.802498	1301.419450
11.771034	1304.898091
11.770748	1304.929821
11.924169	1288.140035

sbatch --exclusive -N 4 knn-mpi-slurm.sh
mpirun -np 4 knn-mpi 128 400000 300 1024

Testes	Tempo médio (s)	SpeedUp em relação ao primeiro teste
Primeiro	27,991175	1
Segundo	11,8810722	2,35594688162908
Terceiro	11,8471181	2,36269907700169



Conclusão

Como podemos perceber pelos resultados obtidos acima, quando ligamos de fato a paralelização com 4 nodos, o tempo de execução praticamente cai pela metade. Já a vazão quase triplica, o que faz sentido quando pensamos que estamos realizando mais operações, mesmo que de modo paralelo. Já quando dividimos a execução para vários hosts, o resultado permanece o mesmo em relação ao segundo teste, não variando praticamente nada, mas continua melhor que o sequencial também, executado no primeiro teste.