Cálculo de K-Nearest-Neighbors em Cluster MPI

Gabriel Pucci Bizio

11 de novembro de 2025

1 Introdução

1.1 Objetivo

Usar o problema dos *K nearest neighbors* para testar o desempenho do cluster XEON com diferentes nodos usando MPI.

1.2 K-Nearest-Neighbors

Dado um vetor **Q** de 128 elementos e outro vetor **P** com 400000 pontos de 300 dimensões, devemos retornar, para cada ponto de **Q**, os 1024 pontos mais próximos no vetor **P** (*nearest neighbors*).

1.3 Max-Heap

Para calcular os vizinhos, foi utilizada uma *heap* de valor máximo, de forma que valores menores são inseridos na heap, garantindo que sempre os 1024 vizinhos mais próximos sejam mantidos.

2 MPI e Paralelismo

2.1 MPI

O programa utiliza MPI para implementar o algoritmo de forma paralela em múltiplas máquinas do cluster.

- **MPI_Bcast**: envio do vetor P.
- **MPI_Scatter**: envio de faixas do vetor Q para as threads.

2.2 Paralelismo

Cada thread processa uma parte do vetor **Q**, iterando por todos os elementos de **P**.

- Exemplo: 4 threads → cada thread processa 32 elementos de Q, iterando sobre todos os 400000 elementos de P.
- Observação: números de threads que geram faixas desequilibradas não são permitidos ($128 \bmod t > 0 \rightarrow$ erro).

3 Testes e Resultados

3.1 Configurações de Testes

- Teste 1: 1 processo em 1 host. Base para medir aceleração do paralelismo.
- Teste 2: Vários processos em 1 host. Espera-se overhead devido ao compartilhamento de recursos.
- Teste 3: 8 processos em 4 hosts, 2 por host. Melhor performance apesar do overhead de sincronização.

3.2 Resultados

3.2.1 Teste 1

Rodada	Tempo (s)
1	22.247589
2	22.250900
3	22.252647
4	22.268508
5	22.230591
6	22.272807
7	22.220403
8	22.265369
9	22.244418
10	22.259823
Total	222.51
Média	22.25

Tabela 1: Resultados do Teste 1

3.2.2 Teste 2

3.2.3 Teste 3

4 Conclusões e Otimizações

4.1 Conclusões

- Ganho de 1.9x entre Teste 1 e 2.
- Ganho de 2.3x entre Teste 1 e 3.
- Teste 2 acelera devido ao paralelismo, mas não proporcional ao número de threads.
- Teste 3 aproveita melhor os recursos, evitando compartilhamento excessivo.

4.2 Otimizações

A função inline `distanciaEuclidiana` utiliza flags OpenMP (unroll e AVX) e as flags do Makefile, reduzindo significativamente o tempo de execução.

Rodada	Tempo (s)
1	11.553767
2	11.568632
3	11.529738
4	12.073475
5	11.618497
6	11.535967
7	11.549467
8	11.543729
9	11.591065
10	11.963668
Total	116.53
Média	11.65

Tabela 2: Resultados do Teste 2

Rodada	Tempo (s)
1	9.639977
2	9.675023
3	9.674482
4	9.663414
5	9.658094
6	9.615736
7	9.654807
8	9.604667
9	9.696062
10	9.645677
Total	96.53
Média	9.65

Tabela 3: Resultados do Teste 3

4.3 Otimizações Futuras

Como o vetor **P** é muito maior que **Q**, trocar a ordem dos loops que percorrem Q e P pode reduzir falhas de cache e aumentar a performance consideravelmente.

5 Arquitetura

5.1 Saída lscpu

```
Architecture: x86_64
CPU(s): 8, Cores por socket: 4, Threads por core: 1
Modelo: Intel Xeon E5462 @ 2.80GHz
L1d cache: 256 KiB
L1i cache: 56 KiB
L2 cache: 24 MiB
NUMA node0 CPU(s): 0-7
```

...

5.2 Saída lstopo

```
Machine (63GB total)
  NUMANode L#0 (P#0 63GB)
  Package L#0
    L2 L#0 (6144KB)
      L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
      L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#2)
    L2 L#1 (6144KB)
      L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2 + PU L#2 (P#4)
      L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3 + PU L#3 (P#6)
  Package L#1
    L2 L#2 (6144KB)
      L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4 + PU L#4 (P#1)
      L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5 + PU L#5 (P#3)
    L2 L#3 (6144KB)
      L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6 + PU L#6 (P#5)
      L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7 + PU L#7 (P#7)
```

...