TRABALHO 2 PROGRAMAÇÃO PARALELA

João Caboclo GRR20221227 Luíza Diapp GRR20221252

Objetivo do Trabalho

O objetivo principal deste trabalho é desenvolver uma solução paralela utilizando o Message Passing Interface (MPI) para o problema do K-Nearest Neighbors (KNN). O programa deve receber um conjunto de pontos de consulta Q (nq pontos) e um conjunto de pontos de base P (npp pontos), ambos em um espaço de D dimensões. Para cada ponto em Q, a função deve determinar seus k vizinhos mais próximos contidos em P. O resultado final é uma matriz R de nq linhas por k colunas, onde cada elemento armazena o índice do vizinho correspondente em P.

Estratégia de Paralelização com MPI

A implementação adota o modelo Single Program, Multiple Data (SPMD), distribuindo a carga de trabalho de computação de distâncias entre os processos MPI. A estratégia consiste em paralelizar o loop externo, onde cada processo se responsabiliza por calcular o KNN para um subconjunto dos pontos de consulta em Q.

Fluxo de Execução Principal

O programa é conduzido pelo processo root (rank 0), que é responsável pela inicialização dos dados, distribuição das consultas e coleta dos resultados.

Inicialmente, os parâmetros nq, npp, d, e k são lidos e a inicialização do ambiente MPI ocorre. A alocação de memória para as matrizes P (dataset), Q (consultas globais) e R (resultado global) é realizada, sendo Q e R alocados apenas no rank 0. Todos os processos alocam memória para suas porções locais Q_local e R_local, para otimizar o acesso à memória, buscando alinhamento de 64 bytes.

No rank 0, as matrizes P e Q são geradas com dados aleatórios, conforme a especificação do trabalho.

Comunicação de Dados

A fase de comunicação é composta por quatro etapas principais: broadcast do dataset P, scatter das consultas Q, computação local dos vizinhos e gather dos resultados. Os tempos de cada uma dessas fases são medidos separadamente com ̀MPI_Wtime().

1. Broadcast de P (t_{bcast}): distribuição do dataset completo P para todos os processos.
2. Scatter de Q (t_{scatt}): envio das porções de Q para cada processo.

3. Cálculo Local (t_{comp}): execução paralela do KNN sobre os subconjuntos locais.
4. Gather de R (t_{gath}): reunião dos resultados parciais R_local no processo root.

Cálculo Local e Coleta de Resultados

Cada processo executa o cálculo do KNN em sua porção local de consultas Q_local, utilizando a função `knn_1query`, que emprega uma estrutura Max-Heap para armazenar os k vizinhos mais próximos. Os resultados são então coletados pelo processo root por meio do `MPI_Gather`.

Medição de Tempo

Os tempos individuais de cada etapa são registrados e salvos em `tempo.txt`, permitindo a análise detalhada de comunicação e computação.

Lógica do Algoritmo KNN (Função `knn_1query`)

A função `knn_1query` realiza a busca dos k vizinhos mais próximos de um ponto q dentro do conjunto P. Para isso, utiliza-se a distância quadrática:

$$\text{dist}^2(q, p) = \sum_{i=1}^{D} (q_i - p_i)^2$$

Essa métrica evita o uso da raiz quadrada e preserva a ordem das distâncias.

Uso da Estrutura Max-Heap

A estrutura Max-Heap é utilizada para manter os k vizinhos com as menores distâncias. Durante o processamento, sempre que uma nova distância é menor que o maior valor do heap, ela substitui a raiz, garantindo que apenas os k menores valores sejam mantidos. No final, os elementos são extraídos em ordem crescente.

Função de Verificação (`verificaKNN.c`)

A função `verificaKNN` valida os resultados do KNN paralelo, reconstruindo os k vizinhos mais próximos e comparando-os com os retornados pelo programa principal. Ela é executada apenas pelo processo root (rank 0) quando a flag `-v` é usada.

A verificação é feita usando exclusivamente a API do `maxheap.c`. Para cada ponto de consulta:

1. Calculam-se as distâncias quadráticas entre o ponto e todos os elementos de P.
2. Mantêm-se no heap as k menores distâncias, com desempate determinístico por índice menor.
3. Após ordenar os elementos do heap, compara-se o resultado obtido com o armazenado em R.

São checados três aspectos: validade dos índices, ordenação não decrescente das distâncias e igualdade exata com a verdade-terra. Em caso de divergência, a função imprime um relatório indicando a linha com erro e o contexto da diferença.

A `verificaKNN` garante que o paralelismo não altere o resultado esperado e assegura a corretude e estabilidade determinística do KNN paralelo.

Arquivos de Suporte e Execução

- `maxheap.c / maxheap.h`: implementam a estrutura Max-Heap.
- `verificaKNN.c / verificaKNN.h`: função de verificação e impressão do resultado.
- `knn_mpi.c`: implementação principal paralela.
- `rodar1.sh`, `rodar2.sh`, `rodar3.sh`: scripts de execução para diferentes configurações de número de processos.
- `makefile`: automação da compilação via `mpicc` com otimização `-O3`.

Execução no Cluster e Experimentos

Os experimentos foram executados em um ambiente de computação de alto desempenho (HPC) gerenciado pelo SLURM (Simple Linux Utility for Resource Management), um sistema de filas utilizado para alocar recursos de processamento em clusters.

Cada experimento foi submetido ao SLURM por meio de um script de submissão (`rodar1.sh`, `rodar2.sh`, `rodar3.sh`) que define quantos nós e quantos processos MPI serão utilizados na execução. O SLURM aloca automaticamente os nós físicos e distribui os processos MPI conforme as diretivas declaradas no início de cada script.

Scripts de Submissão e Configurações

A seguir, estão descritas as configurações específicas de cada script utilizado nos experimentos:

Script `rodar1.sh` — Execução Sequencial

Executa o programa com 1 processo MPI em 1 nó. Modo totalmente sequencial, utilizado como baseline para comparação. - A opção `-exclusive` garante uso exclusivo do nó.

Script `rodar2.sh` — Execução Paralela em Um Único Nó

- Executa com 8 processos MPI distribuídos dentro de um único nó físico. - Cada núcleo do nó executa um processo MPI, totalizando 8 núcleos utilizados. - Esse experimento representa o primeiro cenário de paralelismo efetivo, sem comunicação entre nós. - O foco aqui é medir o ganho de desempenho devido ao paralelismo local.

Script `rodar3.sh` — Execução Distribuída em Vários Nós do Cluster

- Executa 8 processos MPI distribuídos em 4 nós do cluster (2 processos por nó). - Representa o cenário de execução paralela distribuída, com comunicação pela rede do cluster. - Essa configuração avalia a escalabilidade e o impacto da comunicação inter-nó.

Resultados dos Experimentos

Os experimentos foram realizados com os mesmos parâmetros de entrada (`nq=128`, `npp=400000`, `d=300`, `k=1024`), variando apenas a configuração de paralelismo. Os resultados de tempo foram extraídos do arquivo `resultados.txt`.

Experimento 1

- Tempo Total: 28.305 s
- Broadcast(P): 0.000002 s
- Scatter(Q): 0.000152 s
- Compute: 28.3049 s
- Gather(R): 0.000525 s

Experimento 2

- Tempo Total: 11.806 s
- Broadcast(P): 1.5809 s
- Scatter(Q): 0.0019 s
- Compute: 10.2218 s
- Gather(R): 0.0010 s

Experimento 3

- Tempo Total: 8.969 s
- Broadcast(P): 5.440 s
- Scatter(Q): 0.0063 s
- Compute: 3.501 s
- Gather(R): 0.0215 s

Conclusões

A análise dos experimentos mostra que a paralelização do KNN com MPI apresentou ganhos de desempenho significativos quando executada em um ambiente de cluster com SLURM.

O tempo total caiu de aproximadamente 28 s (execução sequencial) para menos de 9 s (execução distribuída em 8 processos), evidenciando um speedup de cerca de 3x. O uso de múltiplos nós permitiu que a carga de cálculo fosse dividida eficientemente, embora o custo de comunicação (especialmente o broadcast) tenha se tornado mais relevante nas execuções distribuídas.

Os resultados confirmam que:

- A paralelização reduz drasticamente o tempo de computação do KNN.

- A fase de broadcast é a principal responsável pelo overhead em execuções distribuídas.
- O SLURM gerencia de forma eficiente a alocação de recursos, permitindo boa escalabilidade.

Como melhorias futuras, propõe-se:

- Uso de comunicação assíncrona (`MPI_Isend`/`MPI_Irecv`) para sobrepor comunicação e computação;
- Divisão de P em blocos distribuídos em vez de replicação total;
- Testes com diferentes números de nós para avaliar escalabilidade forte e fraca.

Esses resultados demonstram que o KNN paralelo implementado é funcional, escalável e adequado para execução em clusters HPC com SLURM.

O trabalho atingiu seus objetivos de implementar um KNN paralelo eficiente com MPI, utilizando uma estrutura Max-Heap para otimização da busca e demonstrando aceleração real com múltiplos processos. A estrutura modular do código facilita a manutenção e futuras otimizações.

Apêndice A — Características da CPU (Saída do `lscpu`)

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   38 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              1
Core(s) per socket:              4
Socket(s):                       2
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           23
Model name:                      Intel(R) Xeon(R) CPU E5462 @ 2.80GHz
Stepping:                        6
CPU MHz:                         2793.307
L1d cache:                       256 KiB
L2 cache:                        24 MiB
NUMA node(s):                    1
NUMA node0 CPU(s):               0-7
```

Apêndice B — Topologia da Máquina (Saída do `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)
```