Relatório do Trabalho 2 de Programação Paralela

Túlio de Pádua Dutra
tpd20@inf.ufpr.br
GRR20206155

Gabriel Pimentel Dolzan
gpd20@inf.ufpr.br
GRR20209948

O código criado é um exemplo de um programa em C que utiliza MPI (Message Passing Interface) para realizar uma operação de K-Nearest Neighbors (KNN) em um ambiente distribuído. Vamos explicar cada parte do código passo a passo.

Instruções de compilação

Basta digitar `make` no terminal e será gerado um executável chamado **knn_mpi**.

Para rodar o programa basta digitar no terminal:

```
mpirun -np 4 knn_mpi nq npp d k
```

Onde:

nq = número de pontos em Q (tamanho do conjunto de pesquisa)

npp = número de pontos em P (dataset)

d = número de dimensões dos pontos (dimensionalidade)

k = tamanho de cada conjunto de vizinhos (k vizinhos por ponto)

Exemplo:

```
mpirun -np 4 knn_mpi 128 400000 300 1024
```

Calcula os 1024 vizinhos mais próximos de 128 pontos de 300 dimensões em base de dados de 400 mil pontos.

Definição da função main e verificação dos argumentos

Inicialização do MPI:
`MPI_Init(&argc, &argv);`
Esta chamada inicializa o ambiente MPI.

Obtenção do tamanho e rank do mundo MPI:
```
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

1. `MPI_Comm_size` obtém o número total de processos no comunicador `MPI_COMM_WORLD`. `MPI_Comm_rank` obtém o rank (identificador) do processo atual.
2. **Processamento dos argumentos da linha de comando**: Os argumentos são convertidos de strings para inteiros e atribuídos às variáveis `nq`, `npp`, `d` e `k`, que representam, respectivamente, o número de pontos em Q, o número de pontos em P, o número de dimensões e o número de vizinhos mais próximos.
3. **Alocação de memória e geração de dados**: O programa aloca memória para os conjuntos de dados P e Q (este último apenas no processo com rank 0) e os preenche com dados.
4. **Distribuição dos dados entre os processos**: O programa calcula quantos elementos cada processo deve tratar (`sendcounts` e `displs`) e então distribui partes do conjunto Q para cada processo usando `MPI_Scatterv`.
5. **Sincronização e medição de tempo**: `MPI_Barrier` sincroniza todos os processos. O processo com rank 0 inicia a medição do tempo.
6. **Execução do KNN**: Cada processo executa a operação KNN localmente nos dados (`local_Q_flattened`).
7. **Coleta dos resultados**: Após o processamento, os resultados são coletados de todos os processos para o processo com rank 0 usando `MPI_Gatherv`.
8. **Cálculo do tempo e verificação dos resultados**: O processo com rank 0 para o cronômetro, calcula o tempo total e o throughput, e verifica os resultados do KNN.
9. **Liberação de recursos e finalização do MPI**: O programa libera toda a memória alocada e finaliza o ambiente MPI.

O programa é uma aplicação paralela utilizando MPI para distribuir o trabalho de cálculo do KNN entre vários processos, medindo o desempenho do processamento distribuído.

Definição da função geraConjuntoDeDados e verificação dos argumentos

A função `geraConjuntoDeDados` é usada para gerar um conjunto de dados aleatórios. Ela preenche um array de pontos flutuantes com valores aleatórios, que podem ser usados, por exemplo, em simulações ou testes de algoritmos. Vamos analisar cada parte desta função:

1. **Parâmetros da Função**:
    - `float *C`: Um ponteiro para um array de pontos flutuantes. Este array é onde os dados gerados serão armazenados.
    - `int nc`: Número de pontos que o conjunto de dados deve conter.
    - `int D`: Número de dimensões para cada ponto.

2. **Geração de Dados Aleatórios**:

```
for (int i = 0; i < nc * D; i++)

{

    C[i] = (float)rand() / RAND_MAX;

}
```

1. **Geração pseudo aleatória de floats:**
    - A função itera sobre cada elemento do array `C`, que tem um tamanho total de `nc * D`. Este tamanho é calculado multiplicando-se o número de pontos pelo número de dimensões, pois cada ponto terá `D` valores associados a ele.
    - Dentro do loop, a função `rand()` é chamada para gerar um número inteiro aleatório.

- Este número é então dividido por `RAND_MAX`, que é o valor máximo que `rand()` pode retornar, normalizando-o para o intervalo [0, 1]. O resultado é um número flutuante representando um valor aleatório entre 0 e 1.
- Cada elemento do array `C` é preenchido com um desses valores aleatórios.

2. **Finalização**:
   - Após o loop, todos os elementos do array `C` terão sido preenchidos com valores aleatórios flutuantes.

A função `geraConjuntoDeDados` cria um conjunto de dados multidimensionais com valores aleatórios. É importante notar que a qualidade e a distribuição dos dados gerados dependem da função `rand()`, que gera números pseudo aleatórios e pode ter suas limitações em termos de aleatoriedade e distribuição.

Definição da função squaredDistance e verificação dos argumentos

A função `squaredDistance` calcula a distância ao quadrado entre dois pontos multidimensionais. Essa função é um componente crucial para o K-Nearest Neighbors (KNN). Vamos analisar cada parte da função:

1. **Parâmetros da Função**:
   - `const float *p1`: Um ponteiro para o primeiro ponto.
   - `const float *p2`: Um ponteiro para o segundo ponto.
   - `int D`: Número de dimensões dos pontos.
2. **Inicialização da Variável de Distância**:

A variável `distance` é inicializada com 0.0. Ela acumulará a soma das diferenças ao quadrado entre as coordenadas correspondentes dos dois pontos.

Cálculo da Diferença ao Quadrado para Cada Dimensão:

```
for (int i = 0; i < D; i++)

{
```

```
    float diff = p1[i] - p2[i];

    distance += diff * diff;

}
```

1. **Cálculo da Distância ao Quadrado**: A função itera sobre cada dimensão dos pontos (`i < D`). Para cada dimensão:
    - Calcula-se a diferença entre as coordenadas correspondentes dos dois pontos (`p1[i] - p2[i]`).
    - Essa diferença é então elevada ao quadrado (`diff * diff`) e adicionada à variável `distance`.
2. **Retorno da Distância ao Quadrado**: Após somar as diferenças ao quadrado de todas as dimensões, a função retorna o valor de `distance`.

Essencialmente, a função `squaredDistance` implementa a fórmula da distância euclidiana ao quadrado, mas sem a raiz quadrada final. Segundo o enunciado do trabalho, trabalhar com a distância ao quadrado é suficiente e evita o cálculo da raiz quadrada, que é mais custoso computacionalmente.

Definição da função knn e verificação dos argumentos

A função `knn` (K-Nearest Neighbors) é uma implementação do algoritmo KNN para encontrar os `k` vizinhos mais próximos de cada ponto em um conjunto de dados `Q`, considerando um segundo conjunto de dados `P`. Vamos detalhar cada parte desta função:

1. **Parâmetros da Função**:
    - `float *Q`: Um array de pontos, representando o conjunto de dados `Q`.
    - `int nq`: Número de pontos em `Q`.
    - `float *P`: Um array de pontos, representando o conjunto de dados `P`.
    - `int npp`: Número de pontos em `P`.
    - `int D`: Número de dimensões de cada ponto.

    - `int k`: Número de vizinhos mais próximos a serem encontrados.
    - `int **resultIndices`: Uma matriz para armazenar os índices dos `k` vizinhos mais próximos de cada ponto em `Q`.
2. **Processamento de cada ponto em Q**: Para cada ponto `i` em `Q`:

```
for (int i = 0; i < nq; i++)
```

- É criada uma maxHeap (`neighborsHeap`) para armazenar os vizinhos mais próximos.

Encontrando os vizinhos mais próximos para cada ponto em Q: Para cada ponto `j` em `P`:

```
for (int j = 0; j < npp; j++)
```

- Calcula-se a distância ao quadrado (`squaredDistance`) entre o ponto `i` de `Q` e o ponto `j` de `P`.
- Cria-se um par (`pair_t`) contendo a distância e o índice `j`.
- Se o tamanho da maxHeap é menor que `k`, o par é inserido na maxHeap (`insert`).
- Se a distância é menor que a maior distância na maxHeap , o maior elemento da maxHeap é substituído pelo novo par (`decreaseMax`).

Armazenamento dos Resultados: Após processar todos os pontos em `P`, os índices dos `k` vizinhos mais próximos são armazenados em `resultIndices`:

```
for (int j = 0; j < k; j++)
{
    resultIndices[i][j] = neighborsHeap[j].val;
}
```

1. **Informações obtidas**: Cada linha de `resultIndices` corresponde a um ponto em `Q` e contém os índices dos `k` vizinhos mais próximos em `P`.
2. **Liberação de Recursos**: A maxHeap é liberada da memória após o processamento de cada ponto em `Q`.

Resumindo, a função `knn` itera sobre cada ponto em `Q`, calculando os `k` vizinhos mais próximos em `P` utilizando uma maxHeap para manter os vizinhos mais próximos durante o processo. O resultado é uma matriz onde cada linha contém os índices dos `k` vizinhos mais próximos de um ponto específico em `Q`.

Definição da função verificaKNN e verificação dos argumentos

A função `verificaKNN` serve para validar os resultados de um algoritmo K-Nearest Neighbors (KNN). Ela compara os vizinhos mais próximos encontrados pelo algoritmo KNN com os vizinhos mais próximos calculados diretamente pela função. Vamos detalhar cada parte da função:

1. **Parâmetros da Função**:
    - `float *Q`: Um array de pontos representando o conjunto de dados Q.
    - `int nq`: Número de pontos em Q.
    - `float *P`: Um array de pontos representando o conjunto de dados P.
    - `int n`: Número de pontos em P.
    - `int D`: Número de dimensões de cada ponto.
    - `int k`: Número de vizinhos mais próximos a serem encontrados.
    - `int *R`: Um array contendo os índices dos k vizinhos mais próximos para cada ponto em Q, conforme calculado pelo algoritmo KNN.

2. **Verificação dos Resultados para Cada Ponto em Q**: Para cada ponto `i` em `Q`:

```
for (int i = 0; i < nq; i++)
```

1. **Detalhamento de inicialização:**
    - Aloca-se um array `distances` para armazenar as distâncias dos k vizinhos mais próximos.

    - Inicializa `distances` com o valor máximo (`FLT_MAX`).
2. **Cálculo das Distâncias dos Vizinhos Mais Próximos**: Para cada ponto `j` em `P`:
    - Calcula-se a distância entre o ponto `i` de `Q` e o ponto `j` de `P`.
    - Atualiza-se o array `distances` com as k menores distâncias encontradas.
3. **Comparação com os Resultados do Algoritmo KNN**: Para cada vizinho mais próximo encontrado pelo algoritmo KNN (`R[i * k + j]`):
    - Calcula-se a distância entre o ponto `i` de `Q` e este vizinho.
    - Verifica-se se esta distância está presente no array `distances`.
    - Se alguma distância não corresponder, a variável `correto` é definida como 0, indicando um erro.
4. **Liberação de Recursos e Conclusão**:
    - Libera-se a memória alocada para `distances`.
    - Se a variável `correto` permanecer 1, todos os vizinhos mais próximos foram corretamente identificados pelo algoritmo KNN, e a função imprime "CORRETO".
    - Se `correto` é 0, a função imprime "ERRADO" e termina.

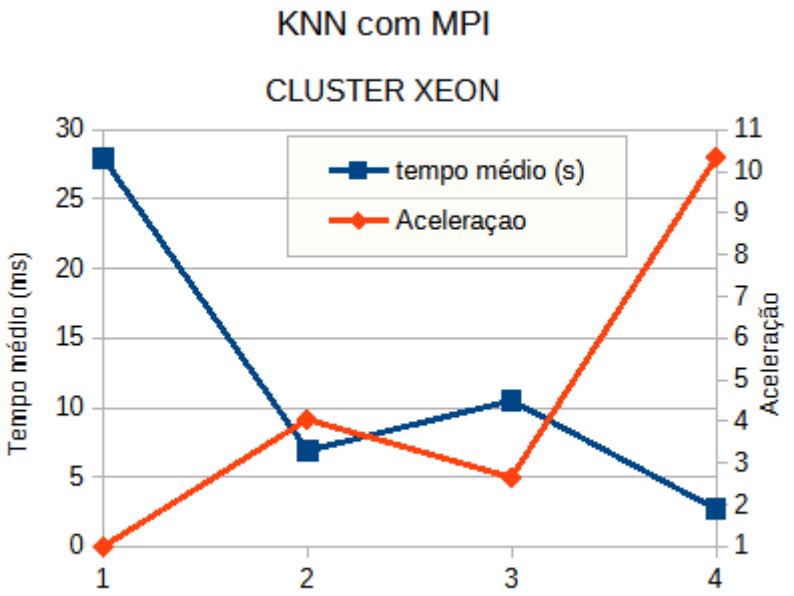
Resumindo, `verificaKNN` é uma função de validação que compara os resultados de um algoritmo KNN com os resultados de uma verificação direta e exaustiva (o que não é muito eficiente computacionalmente), garantindo que os vizinhos mais próximos identificados pelo algoritmo sejam realmente os corretos.

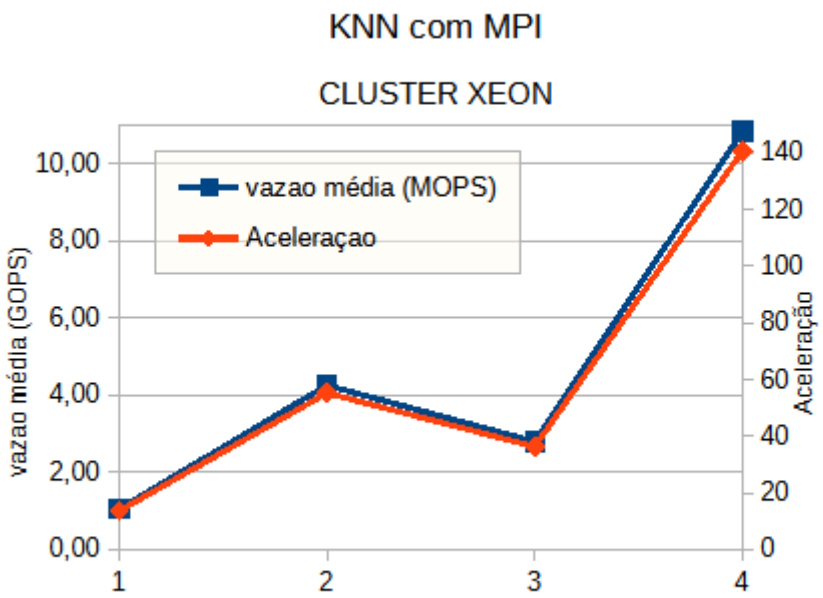
Resultados obtidos

Segundo o enunciado, realizamos alguns experimentos (todos foram executados no cluster XEON), aqui segue os resultados obtidos:

- 1 Processo em 1 Nodo (SEQUENCIAL):
  - Tempo médio: 27,97 segundos
  - Vazão média: 14,30 MOPs por segundo
  - Speedup: 1,00
- 1 Processo em 4 Nodos:
  - Tempo médio: 6,90 segundos
  - Vazão média: 57,97 MOPs por segundo
  - Speedup: 4,05
- 4 Processos em 1 Nodo:
  - Tempo médio: 10,51 segundos
  - Vazão média: 38,06 MOPs por segundo
  - Speedup: 2,66
- 4 Processos em 4 Nodos:
  - Tempo médio: 2,71 segundos
  - Vazão média: 147,60 MOPs por segundo
  - Speedup: 10,32

Gráficos gerados pela planilha



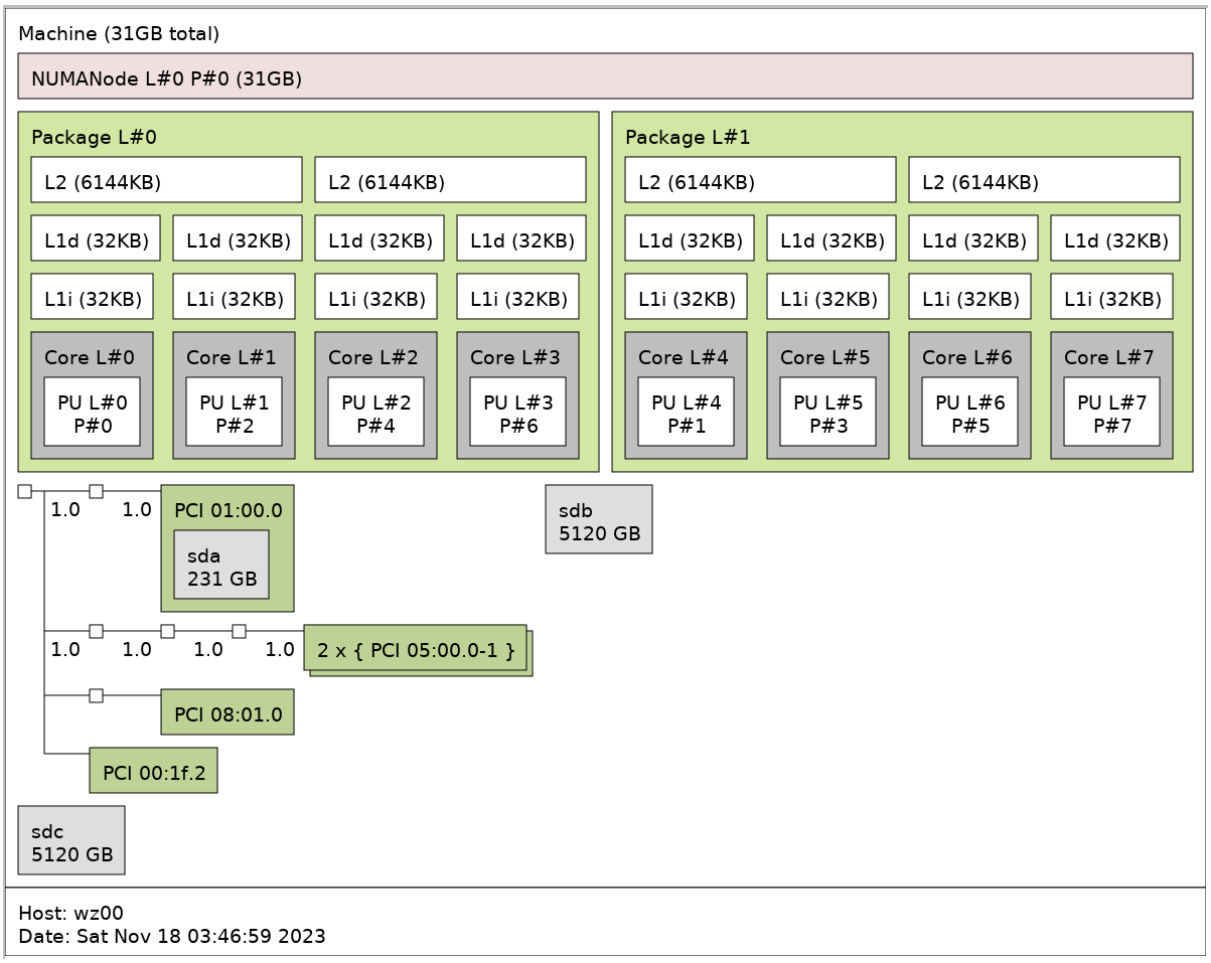


Melhoria com Computação Paralela: Os resultados mostram uma melhoria substancial no desempenho com a computação paralela. O tempo de execução diminui e a vazão aumenta à medida que mais processos e nodos são utilizados.

Eficiência de Escalabilidade: O maior speed-up e a maior vazão são observados quando os processos são distribuídos em vários nodos, indicando que nosso programa se beneficia da escalabilidade horizontal (mais nodos) além da escalabilidade vertical (mais processos em um nodo).

Esses resultados evidenciam que o nosso programa tira bom proveito da computação paralela, tanto em termos de paralelismo de processos quanto na utilização de múltiplos nodos para melhor distribuição e processamento dos dados. Em resumo, nós conseguimos obter um resultado melhor com a computação paralela, maximizando a eficiência ao usar múltiplos processos em múltiplos nodos.

Saída do comando **lstopo**:



Saída do comando **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
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
```

```
Model:                           23
Model name:                      Intel(R) Xeon(R) CPU
E5462  @ 2.80GHz
Stepping:                        6
CPU MHz:                         2792.839
BogoMIPS:                        5585.67
Virtualization:                  VT-x
L1d cache:                       256 KiB
L1i cache:                       256 KiB
L2 cache:                        24 MiB
NUMA node0 CPU(s):               0-7
Vulnerability Itlb multihit:     KVM: Mitigation: VMX
disabled
Vulnerability L1tf:              Mitigation; PTE
Inversion; VMX EPT disabled
Vulnerability Mds:               Vulnerable: Clear CPU
buffers attempted, no microcode; SMT disabled
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1:        Mitigation;
usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic
retpoline, STIBP disabled, RSB filling
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr
pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
dts
                                 acpi mmx fxsr sse sse2
ht tm pbe syscall nx lm constant_tsc arch_perfmon pebs
bts re
                                 p_good nopl cpuid
aperfmperf pni dtes64 monitor ds_cpl vmx est tm2 ssse3
cx16 xtpr p
                                 dcm dca sse4_1 lahf_lm
pti tpr_shadow vnmi flexpriority vpid dtherm
```