Projeto A - Sequêncial

Anderson Araujo de Oliveira, N USP: 11371311

1 Parte I - Discussão sobre o código

O código criado para resolver o problema dos m vizinhos próximo pode ser dividi em 4 etapas, sendo:

1. Ler arquivo
2. Calcular distâncias
3. Ordenar os próximos vizinhos
4. Escrever arquivo

A primeira parte do processo envolve a leitura do arquivo contendo as informações dos pontos, o cálculo da distância entre os pontos, a ordenação dos pontos por proximidade e, finalmente, a gravação dessa ordenação em um arquivo.

1.1 Ler o arquivo

A primeira etapa consiste em ler o arquivo e verificar seu tamanho para criar depois o vetor.

```
while ((ch = fgetc(file)) != EOF)
{
  if (ch == '\n')
  {
    numCoordenadas++;
  }
}
```

Na segunda parte, o arquivo é relido usando rewind para os dados serem salvos em um vetor, utilizando uma estrutura que está mais para baixo com alocação dinâmica para um uso mais eficiente da memória.

```
  rewind(file);
// gerando o array com a estrutura

struct Coordenada *coordenadas = malloc(numCoordenadas * sizeof(struct Coordenada));

if (coordenadas == NULL)
{
  fprintf(stderr, "Erro ao alocar mem ria\n");
  fclose(file);
  return 2;
}
// Lendo o arquivo
for (int i = 0; i < numCoordenadas; i++)
{
  int nread = fscanf(file, "%lf %lf %lf", &coordenadas[i].x, &coordenadas[i].y, &
 coordenadas[i].z);
  if (nread != 3)
  {
    fprintf(stderr, "n o conseguiu ler\n");
    return 3;
  }
}
```

Estrutura

```
//  Estrtura para salvar cada coordenada
struct Coordenada
{
  double x, y, z;
};
```

1.2 Calculando distância

A primeira parte do código foi para calcular as distâncias entre os pontos, dado pela seguinte função

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

```
// Fun   o para calcular a dist ncia euclidiana entre dois pontos
double calcularDistancia(Coordenada p1, Coordenada p2)
{
  double dx = p2.x - p1.x;
  double dy = p2.y - p1.y;
  double dz = p2.z - p1.z;
  return sqrt(dx * dx + dy * dy + dz * dz);
}
```

Criamos uma matriz usando alocação dinâmica para aceitar uma quantidade maior de pontos. Otimizamos o cálculo das distâncias utilizando simetrias da distância cartesiana, sendo elas $d_{ij} = d_{ji}$ e $d_{ii} = 0$, onde d_{ij} é a distância do ponto i para j, onde i,j=0,1,2,...,N-1. A quantidade de cálculos realizado para calcular as distâncias foram de N^2 para $\frac{N^2}{2} - N$, assim reduzindo o tempo de execução do código.

```
  // Alocando mem ria para a matriz de dist ncias

  double **distancias = (double **)malloc(numCoordenadas * sizeof(double *));
  for (int r = 0; r < numCoordenadas; ++r)
  {
    distancias[r] = (double *)malloc(numCoordenadas * sizeof(double));
  }
  // Calculando dist ncias
  clock_gettime(CLOCK_MONOTONIC, &inicio);
  for (int i = 0; i < numCoordenadas; i++)
  {
    for (int j = i + 1; j < numCoordenadas; j++)
    {
      distancias[j][i] = calcularDistancia(coordenadas[i], coordenadas[j]);
      distancias[i][j] = distancias[j][i];
    }
  }
```

1.3 Ordenar os próximos vizinhos

Para melhorar o desempenho, o algoritmo de ordenação desenvolvido primeiramente o "selected sort" foi substituído pelo "quick sort". Isso reduziu a complexidade do código de $\mathcal{O}(mn)$ para $\mathcal{O}(nlog(n))$ nessa etapa, assim, tornando o processo de ordenação dos pontos mais eficiente. Para isso, foi necessário adaptar o "quick sort" para salvar os índices de cada ponto, além de alterar o comparador para aceitar o tipo double na função qsort da linguagem C.

1.3.1 Quick Sort

```c
// Estrutura para armazenar um valor e seu  ndice  original
typedef struct
{
  double valor;
  int indice;
} ValorIndice;

// Fun  o para comparar dois ValorIndice (usado no qsort)
int comparar(const void *a, const void *b)
{
  ValorIndice *valorA = (ValorIndice *)a;
  ValorIndice *valorB = (ValorIndice *)b;
  if (valorA->valor < valorB->valor)
    return -1;
  if (valorA->valor > valorB->valor)
    return 1;
  return 0;
}
// Fun  o para ordenar um array de doubles e salvar os  ndices  originais
void quicksort_indices(double vetor[], int indices[], int n)
{
  // Criar um array auxiliar de ValorIndice
  ValorIndice *aux = (ValorIndice *)malloc(n * sizeof(ValorIndice));
  if (aux == NULL)
  {
    fprintf(stderr, "Erro ao alocar memoria\n");
    exit(1);
  }

  // Inicializar o array auxiliar com os valores e  ndices
  for (int i = 0; i < n; i++)
  {
    aux[i].valor = vetor[i];
    aux[i].indice = i;
  }

  // Ordenar o array auxiliar usando qsort
  qsort(aux, n, sizeof(ValorIndice), comparar);

  // Salvar os  ndices  ordenados no array indices
  for (int i = 0; i < n; i++)
  {
    indices[i] = aux[i].indice;
  }

  // Liberar a mem ria do array auxiliar
  free(aux);
}
```

1.3.2 Calculo

Esta parte do código implementa a lógica para encontrar os m vizinhos mais próximos de cada ponto. O algoritmo de ordenação "quicksort indices" é aplicado para ordenar as distâncias entre cada ponto e os demais. Em seguida, os índices dos m vizinhos mais próximos são armazenados na matriz vizinhos. A liberação da memória alocada para índices e distâncias garante a otimização do uso da memória.

```
  // algoritmo de ordena  o
  for (int i = 0; i < numCoordenadas; i++)
  {
    int *indices = (int *)malloc(numCoordenadas * sizeof(int)); // criando o vetor de
   indices
    quicksort_indices(distancias[i], indices, numCoordenadas);
    for (int j = 0; j < m; ++j)
    { // passsando os dados para uma matriz
      vizinhos[i][j] = indices[j];
      // printf("%d ",indices[j]);
    }
    free(indices);
    // printf("\n");
  }
  free(distancias);
```

1.4 Escrever o artigo

Na etapa final, procedemos à abertura de um novo arquivo, no qual armazenamos os pontos devidamente ordenados.

```
for (int i = 0; i < numCoordenadas; ++i)
  {
    for (int k = 0; k < m; ++k)
    {
      // salvando em um arquivo
      int nchar = fprintf(arquivo, "%d ", vizinhos[i][k]);
      if (nchar < 0)
      {
        fprintf(stderr, "erro na saida do arquivo.\n");
        fclose(arquivo);
        return 2;
      }
    }
    fprintf(arquivo, "\n");
  }
  fclose(arquivo);
```

2 Parte II - Comparação de algoritmos

Esta seção do relatório compara o desempenho entre os algoritmos. O primeiro algoritmo analisado foi o "Quick Sort". Conforme implementado, o número de vizinhos mais próximos (m) não exerce influência sobre o desempenho do algoritmo.

Quick Sort	
n	tempo(s)
100	0.00186
1000	0.1458
5000	2.18
10000	8.696

Tabela 1: Desempenho do quick sort no processador i5-13450HX

Ao utilizar o algoritmo "Select Sort" na forma como foi implementado, o tempo de execução é afetado pela quantidade de vizinhos mais próximos (m).

Select Sort		
n	m	tempo(s)
1000	500	0,28
1000	1000	0,48
5000	500	5,71
5000	1000	10,92
10000	500	19,24
10000	1000	37,86
10000	2000	72,70
10000	5000	180,02

Tabela 2: Desempenho Selct Sort no processador i5-13450HX

Finalmente, é apresentado o tempo de execução do algoritmo "Quick Sort", utilizando os recursos computacionais do cluster Basalto.

Quick Sort	
n	tempo (s)
100	0,00161
1000	0,1117
1500	0,2357
2000	0,4357

Tabela 3: Desempenho do quick sort no processador i5-7500(basalto)

Como observado, o "Quick Sort" mostrou um desempenho superior em comparação ao "Select Sort" para esse problema dos m vizinhos mais próximos.

As imagens abaixo mostra o desempenho do mesmo código para duas maquinas diferentes com um processador i5-7500(basalto) e outro o i5-13450HX.

```
a11371311@ametista1:~/paralela$ ./proj-1 100 large/large1.pos
Tempo da leitura: 0.001505 segundos
Tempo do calculo: 0.000025 segundos
Tempo ordenação das particulas: 0.001612 segundos
Tempo de escrita: 0.003819 segundos
a11371311@ametista1:~/paralela$ ./proj-1 1500 large/large15.pos
Tempo da leitura: 0.010254 segundos
Tempo do calculo: 0.007043 segundos
Tempo ordenação das particulas: 0.235762 segundos
Tempo de escrita: 4.749576 segundos
a11371311@ametista1:~/paralela$ ./proj-1 2000 large/large20.pos
Tempo da leitura: 0.004840 segundos
Tempo do calculo: 0.015110 segundos
Tempo ordenação das particulas: 0.435750 segundos
Tempo de escrita: 8.761803 segundos
a11371311@ametista1:~/paralela$ ./proj-1 500 large/large20.pos
Tempo da leitura: 0.004094 segundos
Tempo do calculo: 0.014997 segundos
Tempo ordenação das particulas: 0.434983 segundos
Tempo de escrita: 1.612345 segundos
```

Figura 1: tempo de execução utilizando otimização -O2

Figura 2: tempo de execução utilizando otimização -O2

Por final, a escolha do algoritmo de ordenação "Quick Sort" e a otimização no cálculo das distâncias contribuíram para um bom desempenho, especialmente para grandes conjuntos de pontos. Adicionalmente, a alocação dinâmica de memória garantiu o uso eficiente dos recursos computacionais.