Projeto A - Paralela

Anderson Araujo de Oliveira, N USP: 11371311

1 Parte I - Discussão sobre o código

O código para resolver o problema das M partículas mais próximas pode ser dividido em 4 etapas:

1. Ler arquivo
2. Calcular distâncias
3. Comunicação
4. Escrever arquivo

Primeiramente, o arquivo contendo as informações dos pontos é lido. Em seguida, a distância entre os pontos é calculada e ordenados por proximidade. Finalmente, essa ordenação é gravada 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 com as coordenadas:

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

Na segunda etapa, o arquivo é lido novamente, utilizando a função rewind para retornar ao seu início. Os dados são então armazenados em um vetor, que utiliza uma estrutura definida mais abaixo no código e alocação dinâmica de memória para otimizar seu uso.

```
  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 da distância

1.2.1 Broadcast

Antes de prosseguir com os cálculos, a quantidade de partículas e suas coordenadas são transmitidas para todos os processos utilizando a função broadcast.

```
MPI_Bcast(&numCoordenadas, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
MPI_Bcast(coordenadas, numCoordenadas * sizeof(struct Coordenada), MPI_BYTE, 0,
    MPI_COMM_WORLD);
```

1.2.2 Cálculo

A primeira parte do código destina-se ao cálculo das distâncias entre os pontos, realizado pela função apresentada a seguir. Para otimizar o código, exploramos a seguinte propriedade: a relação de ordem entre duas distâncias, d_1 **e** d_1 se mantém ao comparar seus quadrados, ou seja, se $d_1 < d_1$, então $d_1^2 < d_1^2$.

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

```
// 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);
}
```

Para armazenar as partículas mais próximas, foi criada uma matriz de dimensões N x M. A cada cálculo de distância, a funçãoadd_vizinhos verifica se a partícula em questão é uma das mais próximas. A otimização do cálculo das distâncias se baseou na propriedade de que $d_{ii} = 0$, 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 $N^2 - N$, assim reduzindo o tempo de execução do código. A justificativa para a não utilização da simetria cartesiana será apresentada posteriormente neste relatório.

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

  double *distancias = (double *)calloc(linhas_por_processo*m,sizeof(double));
  int *indices = (int *)malloc(linhas_por_processo*m*sizeof(int));
  // Calculando dist ncias

  for (int i = inicio_linha; i < fim_linha; i++)
  {
    for (int j = 0; j < numCoordenadas; j++){
        if(i!=j){
        distancia = calcularDistancia(coordenadas[i], coordenadas[j]); // excluido
    valores da diagonal
```

```
        add_vizinho(indices,distancias,distancia,i-inicio_linha,j,m);
        }
    }
  }
```

1.3 Ordenar os próximos vizinhos

Foi criada uma função para gerar uma lista contendo as m partículas mais próximas, armazenando-as ordenadamente. Ao adicionar uma nova partícula, a função verifica se a última partícula na lista possui uma distância maior que a nova partícula. Em caso afirmativo, a função prossegue, identificando a posição correta da nova partícula na lista e realizando o deslocamento das partículas anteriores na lista.

```
void add_vizinho(int *lista ,double *dista , double d, int i, int j, int m) {
    if(d > dista[i*m+(m-1)] && dista[i*m+(m-1)]>0) return;
    for (int k = 0; k < m; ++k) {
        if ((d < dista[i*m + k]) || (dista[i*m + k] == 0.0)) {
            for (int l = m-1; l > k; --l) {
                dista[i*m + l] = dista[i*m + (l-1)];
                lista[i*m + l] = lista[i*m + (l-1)];
            }
            dista[i*m + k] = d;
            lista[i*m + k] = j;
            return;
        }
    }

}
```

1.4 Saida

1.4.1 Gatherv

Após o cálculo das M partículas mais próximas em cada processo, é necessário reunir essas informações em um único processo, no processo raiz para escrita da saída. No entanto, é preciso considerar o caso em que a divisão do número total de partículas (N) pelo número de processos (P) não resulta em um valor inteiro, ou seja, $N\%P \neq 0$. Nessa situação, alguns processos terão mais elementos do que outros, o que pode gerar problemas na junção das matrizes. Para solucionar essa questão, utilizamos a função GatherV, que permite a coleta de dados com diferentes tamanhos de cada processo. Consideramos que o último processo terá uma quantidade maior de elementos, sendo essa diferença dada por $Q = N\%P$.

```
  int *vizinhos = NULL;
  if (rank == 0) {
    vizinhos = (int *)malloc(numCoordenadas * m * sizeof(int));
  }
  int *count = (int *)malloc(size * sizeof(int));
  int *offset = (int *)malloc(size * sizeof(int));
  //printf("a\n");
  particao(numCoordenadas, size, m, count, offset);

  MPI_Gatherv(indices, count[rank], MPI_INT, vizinhos, count,offset, MPI_INT, 0,
   MPI_COMM_WORLD);
  free(indices)
```

1.4.2 Escrever

Na etapa final, procedemos à abertura de um novo arquivo, no qual armazenamos os pontos devidamente ordenados; assim, escrevemos o conteúdo do vizinho.

```
if (rank == 0) {
    //printf("b\n");
    char *arquivo_saida= strcat(argv[1], ".ngb");
    // criando o arquivo de saida
    FILE *arquivo = fopen(arquivo_saida, "w");

    clock_gettime(CLOCK_MONOTONIC, &inicio);

    // salvando em um arquivo
    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 * m + k]);
        if (nchar < 0)
        {
          fprintf(stderr, "erro na saida do arquivo.\n");
          fclose(arquivo);
          MPI_Abort(MPI_COMM_WORLD, 1);
        }
      }
      fprintf(arquivo, "\n");
    }
    fclose(arquivo);
    //printf("c\n");
    clock_gettime(CLOCK_MONOTONIC, &fim);

    tempo_decorrido3 = (fim.tv_sec - inicio.tv_sec) +
                       (fim.tv_nsec - inicio.tv_nsec) / 1e9;

    printf("%f %f %f\n", tempo_decorrido1,tempo_decorrido2,tempo_decorrido3);
  }
```

2 Parte II - Comparação de algoritmos

2.1 Complexidade do código

1. Leitura $O(N)$
2. Calculo/Ordenar $O(N^2M)$
3. Comunicação $O(NM)$
4. Escrita $O(NM)$

A complexidade total do código, considerando todas as etapas, é da ordem de $O(N^2M)$. A etapa que demanda o maior custo computacional é o cálculo da distância entre as partículas e a subsequente inserção na lista dos M vizinhos mais próximos.

2.2 Justificativas

É crucial destacar a razão pela qual a simetria $d_{ij} = d_{ji}$ não foi utilizada neste trabalho. A utilização da simetria implicaria em uma complexidade de tempo da ordem de $O(NM)$,para a função add_vizinhos, enquanto a não utilização da simetria resulta em uma complexidade de $O\left(\frac{NM}{p}\right)$.

Essa diferença se deve ao fato de que, ao utilizar a simetria, a função acessaria todas as coordenadas, tanto d_{ij} e d_{ji} demandando o acesso completo à matriz de vizinhos. Por outro lado, sem a simetria, cada processo acessa apenas as coordenadas que lhe foram atribuídas, reduzindo significativamente o número de chamadas à função add_vizinhos.

Uma alternativa para o algoritmo seria calcular todas as distâncias entre as partículas e, em seguida, ordená-las utilizando o algoritmo Quicksort. No entanto, essa abordagem resultaria em uma complexidade de tempo de $O(N^2log(N))$, em contraste com a complexidade de $O(N^2M)$.

Embora a complexidade assintótica do Quicksort seja superior, é importante considerar que, neste problema específico, $M << N$. Dessa forma, o algoritmo proposto, que explora essa característica do problema, apresenta um desempenho superior.Além disso, a utilização do Quicksort acarretaria um custo de memória de N^2 para armazenar todas as distâncias, o que pode ser um fator limitante para valores muito grandes de N.

2.3 Desempenho do Código

A Figura 1 ilustra o desempenho do código com o parâmetro M fixado em 50, enquanto o valor de N varia.

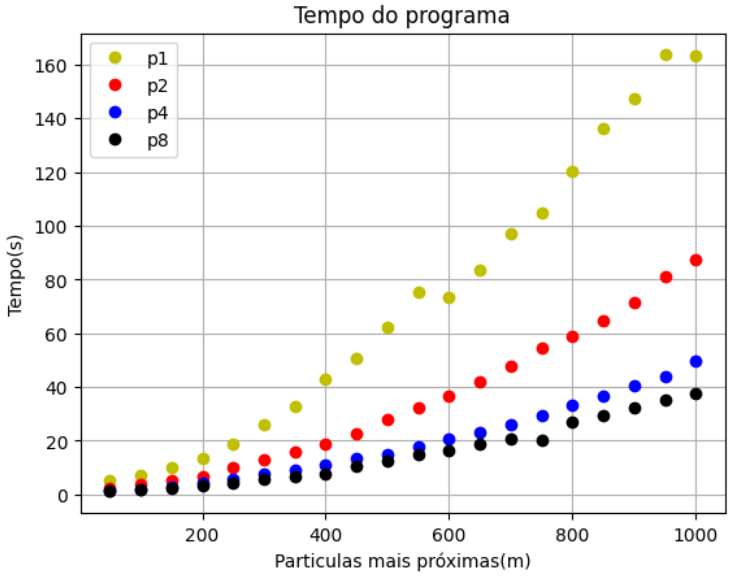


Figura 1: A quantidade de partículas mais próximas é M=50

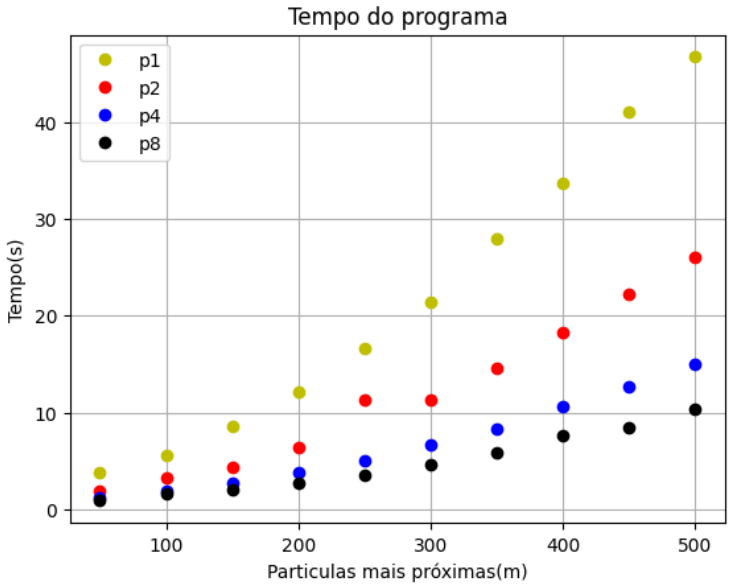
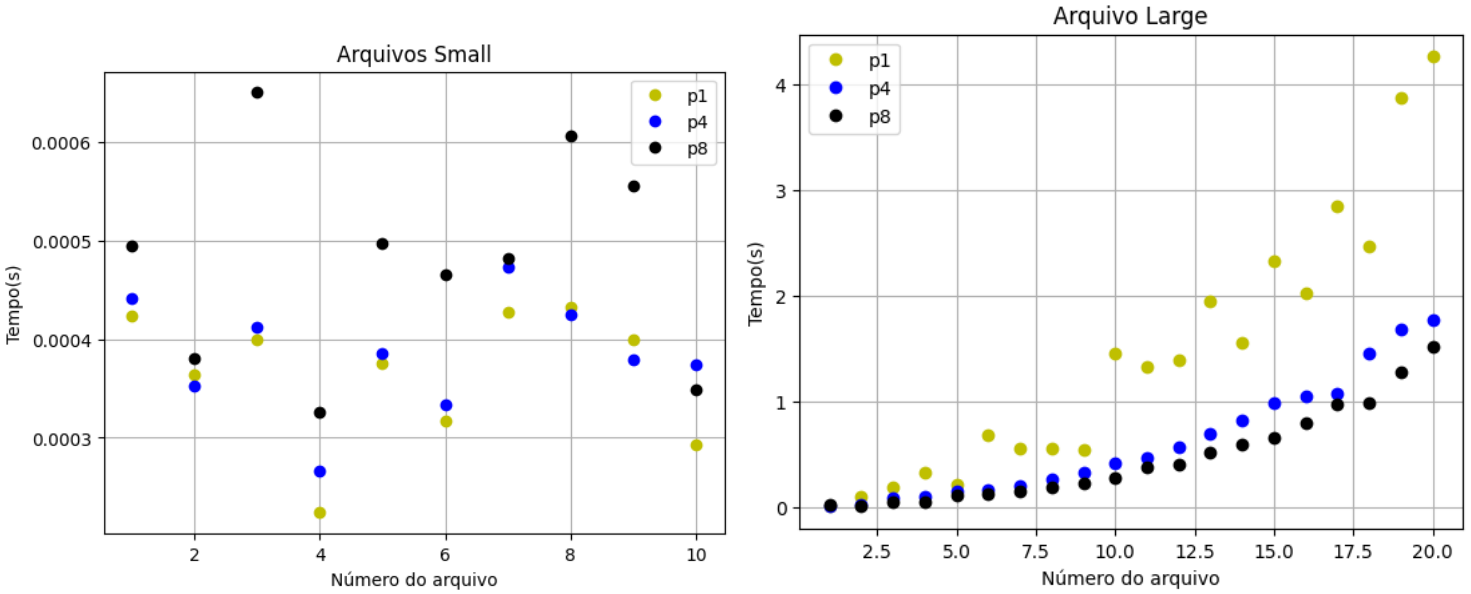


Figura 2: A quantidade de partículas é N=50000

A Figura 2 ilustra o desempenho do código com o número de partículas (N) fixado em 50.000, enquanto o número de partículas mais próximos (M) varia.



(a) Arquivos Small

(b) Arquivos Larges

Figura 3: Desempenho do código com os arquivos de exemplo

A análise de desempenho do código para os arquivos "Large.pos" e "Small.pos" revela um comportamento interessante em relação ao número de processos utilizados. Observa-se que, para uma pequena quantidade de partículas, o aumento no número de processos acarreta um maior custo computacional devido à comunicação entre os processos. Esse fenômeno é evidenciado na figura, onde o código executado com 8 processos apresenta um tempo de execução maior do que com apenas 1 e 4 processos.

Por outro lado, para uma grande quantidade de partículas, o aumento no número de processos resulta em um ganho de desempenho significativo. Isso ocorre porque, nesse caso, o custo computacional da comunicação

entre os processos é compensado pela paralelização do processamento, permitindo que cada processo realize uma parte menor do trabalho de forma simultânea.

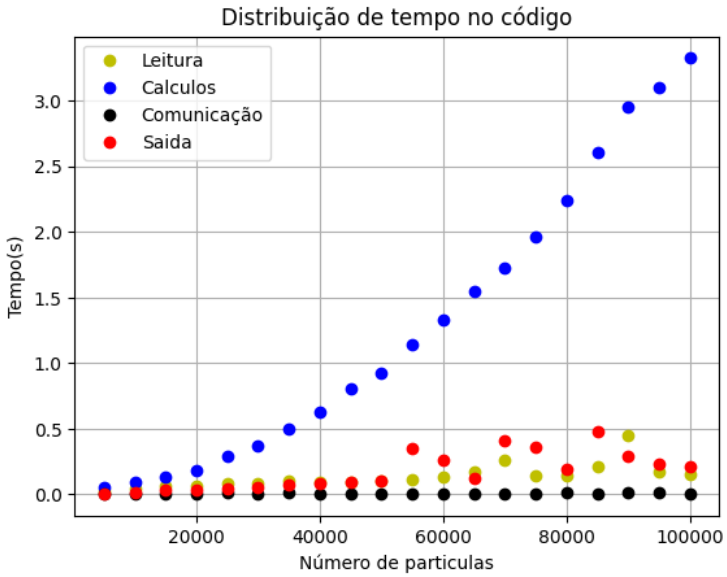


Figura 4: Quantidade de partículas próximas M=50, quantidade de processos P=4

A Figura 4 apresenta a análise do custo computacional de cada etapa do programa, medido em tempo de execução. Os resultados demonstram que o impacto da comunicação entre os processos no tempo total de execução é menos significativo quando a quantidade de cálculos realizados é substancialmente grande.