Universidade de São Paulo
Instituto de Física de São Carlos

SFI5822 - Introdução à Programação Paralela (2024)
Prof. Gonzalo Travieso

Trabalho A - Sequencial

Lucas Constante Mosquini - **NºUSP** 11858258

October 20, 2024

Contents

1 Introdução

Como descrito no projeto, temos que fazer um programa que recebe as coordenadas de N partículas, em um espaço tridimensional, linha por linha. Em seguida, temos que calcular a distância euclidiana entre elas e assim, analisar os m vizinhos mais próximos. Por fim, imprimi-se um arquivo, com as informações N, m, tempo de leitura de entrada, tempo de cálculo e tempo de escrita.

Como diretrizes principais: respeitar boas práticas, como visto em aula e especialmente pensar em código que possa ser paralelizado facilmente futuramente, ou seja, evitar usar estruturas complexas como árvores de armazenamento, mesmo que elas tenham um desempenho superior, como visto na secção de comparação de desempenho 3 de diferentes códigos.

Por fim, esse pdf se resume em justificar as escolhas visando o desempenho, assim como explicar o funcionamento e ideias bases por trás do código.

Como o trabalho foi feito em diferentes dias para que ideias de otimização pudessem florir, estruturarei o pdf em diferentes versões evolutivas do código proposto.

A versão de cada evoluçaõ do código pode ser vista completa em 5. Aqui deixo claro que para a disciplina o código enviado foi 5.3, sem aproximações e estruturas complexas, conforme requerido no projeto.

2 Version 1.0

Primeiramente, o uso das bibliotecas padrões essenciais, especialmente a math. Para o uso de potências, vem a discusão sobre desempenho de pow() vs **. Pelo que eu testei, depende muito do compilador, e acredito que talvez da arquitetura em si (só testei em uma máquina), então resolvi deixar pow() pela maior simplicidade . Algo que pensei inicialmente, seria o tamanho do buffer que eu deveria definir para ler o arquivo de entrada, ou seja, as partículas. Então, na disciplina foi fornecido alguns arquivos de testes que tinham no máximo 18 casas decimais com sinais (float). Assim, sabendo previamente dessa informação poderíamos fazer a conta e precisar exatamente o tamanho necessário. No entanto para fins gerais, deixarei o tamanho do buffer suficiente para ler as 18 casas propostas e alguma sobra. Como a maioria dos hardwares tem memória o suficiente para essa questão não fazer diferença, no fim das contas isso não afeta muito o desempenho para poucas partículas, mas deixo aqui essa reflexão caso o usuário queira testar os códigos para muitas partículas, sem comprometer a memória do sistema.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>

#define MAX_LINE_LENGTH 300
```

Aqui a declaração da struct Particle que armazena x,y,z de cada partícula. Como veremos mais para frente, pode-se otimizar esse aspecto.

```c
typedef struct {
    double x, y, z;
} Particle;
```

```
//  Funo  para calcular a distncia euclidiana entre duas partculas
double euclidean_distance(Particle a, Particle b) {
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2) + pow(a.z - b.z, 2));
}

//  Funo  para ler partculas do arquivo
Particle* read_particles(const char* filename, int* n, double* read_time) {
    clock_t start = clock();
    FILE* file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "Erro ao abrir o arquivo %s\n", filename);
        exit(EXIT_FAILURE);
    }
```

Aqui é onde funciona a minha alocação de memória. Como vimos em aula, temos a possibilidade de alocar estáticamente ou dinamicamente (pilha vs heap). Como o código tem n, apenas conhecido em tempo de execução e não compilação a alocação dinâmica foi escolhida devido ao melhor desempenho. No entanto, é importante ressaltar a necessidade de free() na memória adequado, como veremos no fim. Repare também no tracker de time, requerido no projeto.

```
    // Contar o nmero de linhas (partculas)
    *n = 0;
    char line[MAX_LINE_LENGTH];
    while (fgets(line, sizeof(line), file)) {
        (*n)++;
    }

    // Alocar memria para as partculas
    Particle* particles = (Particle*)malloc(*n * sizeof(Particle));
    if (!particles) {
        fprintf(stderr, "Erro ao alocar memria\n");
        exit(EXIT_FAILURE);
    }

    // Rewind do arquivo para ler as partculas
    rewind(file);
    for (int i = 0; i < *n; i++) {
        if (fscanf(file, "%lf %lf %lf", &particles[i].x, &particles[i].y, &particles[i].z) !=
            3) {
            fprintf(stderr, "Erro ao ler a linha %d do arquivo\n", i);
            exit(EXIT_FAILURE);
        }
    }
    fclose(file);

    *read_time = (double)(clock() - start) / CLOCKS_PER_SEC;
    return particles;
}
```

Para a versão mais básica do programa requerido, foi pensado inicialmente na estrutura de armazenamento de dados mais intuitiva e simples para o problema: uma matriz. Assim, fazemos funções de encontrar os m vizinhos e escreve-los no arquivo como requerido. Sempre mantendo o tracking de time.

Uma discussão possível é sobre como poderia ser feita a inserção das distâncias: podemos fazer linearmente (como foi feito), ou dispor do uso de algoritimos de ordenação como quicksort ou heap. Para o primeiro caso, rodei alguns testes e o quicksort parecia funcionar melhor só quando temos um m grande 500. Isso faz sentido, pois quando fazemos uma análise de complexidade, constatamos que teríamos $O(n^2 \log n)$ contra $O(n^2 * m)$ do método linear, ou seja, temos que ter m próximo de n para vermos uma melhora significativa no desempenho usando quicksort. Outra possibilidade era usar heap, uma estrutura de dados semelhante à lógica de árvores, o que me fez evitá-los, uma vez que, de acordo com o projeto, não podemos usar tais estruturas.

Dessa forma, como na disciplina foi fornecido testes, com todos os m pequenos, para avaliação, foi enviado o algoritimo linear, cumprindo os requisitos ao mesmo tempo que otimiza o desempenho.

```
    //  Funo  para encontrar os m vizinhos mais prximos de cada partcula
void find_neighbors(Particle* particles, int n, int m, int** neighbors, double*
    computation_time) {
    clock_t start = clock();

    for (int i = 0; i < n; i++) {
        double* distances = (double*)malloc(n * sizeof(double));
        if (!distances) {
            fprintf(stderr, "Erro ao alocar memria para distncias\n");
            exit(EXIT_FAILURE);
        }

        for (int j = 0; j < n; j++) {
            if (i != j) {
                distances[j] = euclidean_distance(particles[i], particles[j]);
            } else {
                distances[j] = INFINITY; // Define a distncia da partcula para si mesma como
                    infinita
            }
        }

        // Encontre os m vizinhos mais prximos
        for (int k = 0; k < m; k++) {
            double min_dist = INFINITY;
            int min_index = -1;
            for (int j = 0; j < n; j++) {
                if (distances[j] < min_dist) {
                    min_dist = distances[j];
                    min_index = j;
                }
            }
            neighbors[i][k] = min_index;
            distances[min_index] = INFINITY; // Marque como j utilizado
        }

        free(distances);
```

```
    }

    *computation_time = (double)(clock() - start) / CLOCKS_PER_SEC;
}

//  Funo  para escrever os vizinhos no arquivo de sada
void write_neighbors(const char* input_filename, int** neighbors, int n, int m, double*
    write_time) {
    clock_t start = clock();

    char output_filename[FILENAME_MAX];
    snprintf(output_filename, sizeof(output_filename), "%s.ngb", input_filename);

    FILE* file = fopen(output_filename, "w");
    if (!file) {
        fprintf(stderr, "Erro ao abrir o arquivo de sada %s\n", output_filename);
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            fprintf(file, "%d ", neighbors[i][j]);
        }
        fprintf(file, "\n");
    }
    fclose(file);

    *write_time = (double)(clock() - start) / CLOCKS_PER_SEC;
}
```

Assim, por fim é feita a main(), onde são executadas as funções requeridas no projeto, gerando o arquivo final. Note que no final é feita a liberação de memória da struct particle e de cada elemento vizinho, liberando espaço na memória, otimizando o desempenho.

```
    //  Funo  principal
int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Uso: %s <arquivo de entrada> <nmero de vizinhos>\n", argv[0]);
        return EXIT_FAILURE;
    }

    const char* input_filename = argv[1];
    int m = atoi(argv[2]);

    int n;
    double read_time, computation_time, write_time;

    // Ler as partculas do arquivo
    Particle* particles = read_particles(input_filename, &n, &read_time);
```

```
    // Alocar memria para os vizinhos
    int** neighbors = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        neighbors[i] = (int*)malloc(m * sizeof(int));
    }

    // Encontrar os vizinhos mais prximos
    find_neighbors(particles, n, m, neighbors, &computation_time);

    // Escrever os vizinhos no arquivo de sada
    write_neighbors(input_filename, neighbors, n, m, &write_time);

    // Imprimir os resultados na sada padro
    printf("%d %d %.6lf %.6lf %.6lf\n", n, m, read_time, computation_time, write_time);

    // Liberar memria
    free(particles);
    for (int i = 0; i < n; i++) {
        free(neighbors[i]);
    }
    free(neighbors);

    return EXIT_SUCCESS;
}
```

2.1 Version 1.1

A primeira otimização foi clara: logo de cara podemos notar uma simetria clara na matriz de armazenamento: o elemento i,j é simétrico com j,i, devido à natureza de distância euclidiana. Asim podemos bolar um jeito de simplesmente usar uma matriz superior, ou seja, metade do custo de computação.

Desta forma, o código foi reestruturado. Assim, para não ocupar tanto espaço, vou apenas comentar sobre cada mudança, onde o leitor pode acompanhar em 5.2: a função computedistances calcula e armazena as distâncias em uma matriz triangular superior evitando os cálculos repetidos e também fornecendo previamente os dados para findneighbors, permitindo o acesso direto. Por fim há uma liberação de memória associada à matriz de distâncias, uma vez que ela é criada a parte.

2.2 Version 1.2

Em seguida recorri às notas de aulas e ao que foi estudado, tentando explorar localidades, mais especialmente a espacial. Pensei em ao inves de montar uma struct de Particles, contendo x,y e z de cada partícula (Array of Struct), eu poderia inverter e criar uma struct de array, ou seja, 3 arrays com sua respectiva coordenada espacial, assim mantendo uma maior localidade espacial. Assim, ajustei as outras funções que dependem dessa organização, como visto em 5.3.

2.3 Version 1.3

Como a partir de agora não via um caminho muito claro para otimizações, então comecei a pensar em aproximações, ou seja, não teremos resultados totalmente corretos no arquivo de saída, mas teremos

maior desempenho. Para o cálculo dos vizinhos mais próximos é verificado iterativamente, a distância da particula em relação a todas as outras. Assim, podemos inicialmente calcular uma distância média iterando todas as distâncias, ou seja, descobrimos a distância média do sistema todo. Dessa forma, ao verificarmos os vizinhos, simplesmente podemos restringir os laços de acordo com essa distância média = distância máxima, economizando custo computacional. No entanto, agora temos o custo computacional do cálculo da média. Ademais, para sistemas muito espaçados, ou com uma distribuição espacial desigual, o cálculo da média fica ruim de ser usado como métrica, assim como regiões de fronteiras(extremos) ficam problemáticas. Assim surge um novo problema, tentar achar quando esse algoritimo será vantajoso em relação ao tradiconal.

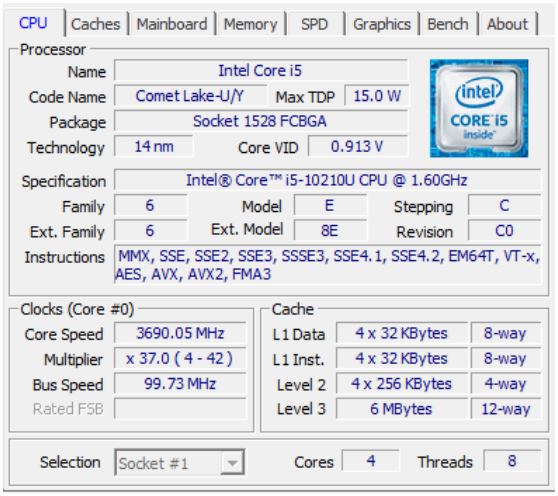
O código pode ser visto em 5.4.

2.4 Version 1.4

Como curiosidade, adaptei alguns algoritimos existentes de K-D Tree ao problema, para comparar o desempenho, como visto em 5.5. Como discutiremos na próxima secção, o ganho de desempenho foi consideravel, colaborando com a ideia vista em aula, que antes de sairmos paralelizando um código, podemos adotar ideias anteriores e boas práticas, resultando em possivelmente mais desempenho.

3 Comparações de Desempenho

Nesta secção, vamos comparar o desempenho das diferentes versões do programa. Para os cálculos, foi usado o processador abaixo:



Vale notar que na linha de saida, os valores de tempo de leitura, tempo de cálculo e tempo de escrita sempre variam. Isso acontece por diversos fatores, como visto em aula: diferente tempos de latência entre outros hardwares com a CPU, localidades (principalmente temporal para as memórias na escrita e leitura), aquecimento e varição de frequência da CPU, distribuição de uso de outras aplicações e etc.

Assim, inicialmente é feito um "aquecimento" do código, rodando ele algumas vezes, com o intuito de minimizar esses efeitos. Em seguida, é tirado os valores da linha de saída 5 vezes. Disso, é feita a média dessas variáveis e então mostradas na tabela abaixo .

Primeiro vou mostrar a diferença entre os tempos: compilando o primeiro programa de acordo com o que foi dito em aula vs uma compilação simples.

	Compilação Normal	Compilação Instruída
Tempo de leitura	0.0006324	0.000575
Tempo de cálculo	0.034592	0.00992
Tempo de escrita	0.0005078	0.000404

Table 1: Tabela comparando compilações normal e instruída. Para cada caso, foi gerado um arquivo de 1000 partículas aleatórias e calculado os 5 vizinhos mais próximos. O programa usado é o 5.1

Como vemos, quando usamos as instruções de compilação, normalmente o código ganha desempenho. É importante ressaltar que a escolha de compilação é dependende de diversos fatores, cabendo ao usuário definir qual a melhor para o seu programa.

Em seguida, vamos comparar as três primeiras versões do programa, ou seja, as versões que produzem resultados exatos e não usam estruturas complexas.

	$t1_{mt}$	$t1$	$t1_{SoA}$
Tempo de leitura	0.000575	0.000623	0.000601
Tempo de cálculo	0.00992	0.00310	0.00291
Tempo de escrita	0.000404	0.000437	0.000450

Table 2: Para cada caso, foi gerado um arquivo de 1000 partículas aleatórias e calculado os 5 vizinhos mais próximos.

De cara, vemos que a ideia de utilizar uma simetria na matriz acelerou o código consideravelmente. Ademais, a mudança nas structs feita, acelerou levemente o nosso programa. Com isso o código final $t1_{SoA}$, apresentou o melhor desempenho, sendo escolhido como final para entrega.

Agora vamos comparar as outras evoluções com $t1_{SoA}$: começemos pelo 5.4, usando a ideia de limitação da distância máxima dentro dos laços.

	$\mathrm{t1}_{SoA}$	$\mathrm{t1}_{dist-max}$
Tempo de leitura	0.000601	0.000781
Tempo de cálculo	0.002910	0.002430
Tempo de escrita	0.000450	0.000430

Table 3: Para cada caso, foi gerado um arquivo de 1000 partículas aleatórias e calculado os 5 vizinhos mais próximos.

A ideia aqui é que para certo número de partículas N, e vizinhos m o algoritimo se torne menos ou mais eficiente, pois há a necessidade de cálculo da média das distâncias.

Por fim, vamos comparar o desempenho do código $t1_{SoA}$ com o do algoritimo usando Árvores d-dimensionais (no nosso caso d=3).

	$\mathrm{t1}_{SoA}$	$\mathrm{t1}_{kd-tree}$
Tempo de leitura	0.000601	0.000627
Tempo de cálculo	0.002910	0.000278
Tempo de escrita	0.000450	0.008632

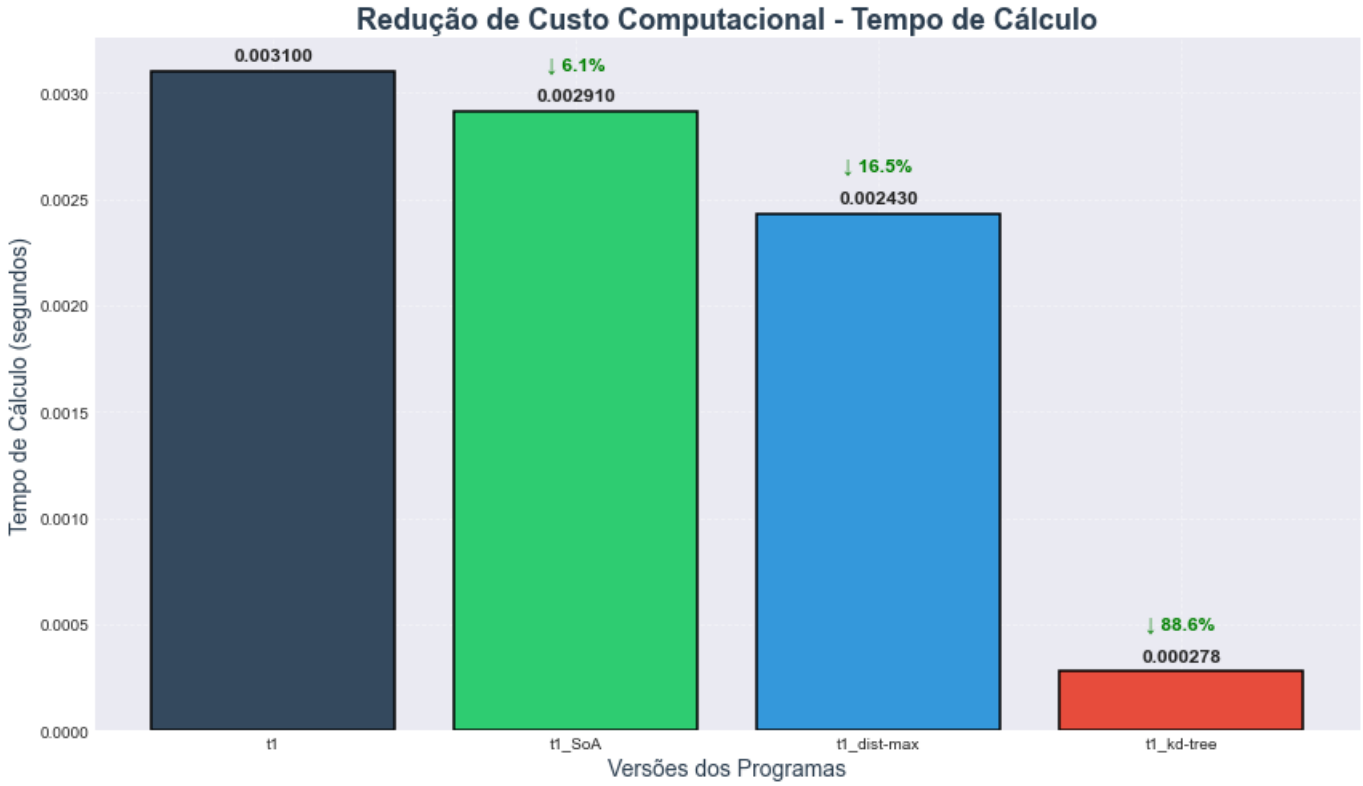
Table 4: Para cada caso, foi gerado um arquivo de 1000 partículas aleatórias e calculado os 5 vizinhos mais próximos.

Dessa forma, vemos que o tempo de cálculo foi drasticamente reduzido, so pela forma de organizar/armazenar os dados usando árvoes. Só como curiosidade rodei o programa $t1_{kd-tree}$ para um arquivo com 100 000 partículas, e impressionantemente, o tempo de cálculo foi de 0.053953. No entanto, como temos que escrever os vizinhos no final e devido ao gargalo na velocidade de escrita, demorou 177 segundos para ser realizado, o que ressalta a eficiência do algoritmo com árvores.

Deixo claro aqui, que isso foi uma parte adicional ao trabalho, pois como foi descrito no projeto, não podemos usar algoritimos como as árvores KD, pois elas seriam dificeis de paralelizar. No entanto, acho que seria legal voltar para esse trabalho, quando fizermos a versão paralela desse no próximo trabalho e comparar o que valeu mais a pena: paralelizar ou usar algoritimos complexos, mesmo que sequenciais.

4 Resultados

Como facilitação visual, deixo abaixo um gráfico para comparação das diferentes versões dos programas.



5 Raw codes

5.1 Version 1.0 - $t1_{mt}$

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
```

```c
#define MAX_LINE_LENGTH 300

typedef struct {
    double* x;
    double* y;
    double* z;
} ParticlesSoA;

// Calcular a distncia
double calc_distancia(double ax, double ay, double az, double bx, double by, double bz) {
    double dx = ax - bx;
    double dy = ay - by;
    double dz = az - bz;
    return sqrt(dx * dx + dy * dy + dz * dz);
}

// Ler partculas de um arquivo de entrada
ParticlesSoA ler_particulas(const char* nome_arquivo, int* n, double* tempo_leitura) {
    clock_t inicio = clock();
    FILE* arquivo = fopen(nome_arquivo, "r");

    if (!arquivo) {
        perror("Erro ao abrir o arquivo");
        exit(EXIT_FAILURE);
    }

    // Contar o nmero de partculas
    *n = 0;
    char linha[MAX_LINE_LENGTH];

    while (fgets(linha, sizeof(linha), arquivo)) {
        (*n)++;
    }

    // Alocar memria para as partculas
    ParticlesSoA particulas;
    particulas.x = (double*)malloc(*n * sizeof(double));
    particulas.y = (double*)malloc(*n * sizeof(double));
    particulas.z = (double*)malloc(*n * sizeof(double));

    if (!particulas.x || !particulas.y || !particulas.z) {
        fprintf(stderr, "Erro na alocao de memria\n");
        exit(EXIT_FAILURE);
    }

    // Rewind do arquivo para ler os dados
    rewind(arquivo);
    for (int i = 0; i < *n; i++) {
        if (fscanf(arquivo, "%lf %lf %lf", &particulas.x[i], &particulas.y[i],
            &particulas.z[i]) != 3) {
```

```
            fprintf(stderr, "Erro ao ler a linha %d do arquivo\n", i);
            exit(EXIT_FAILURE);
        }
    }
    fclose(arquivo);

    *tempo_leitura = (double)(clock() - inicio) / CLOCKS_PER_SEC;
    return particulas;
}

// Calcular distncias e armazen-las em uma matriz triangular
double** calcular_distancias(ParticlesSoA particulas, int n, double* tempo_calculo) {
    clock_t inicio = clock();
    double** distancias = (double**)malloc(n * sizeof(double*));

    for (int i = 0; i < n; i++) {
        distancias[i] = (double*)malloc((n - i - 1) * sizeof(double));
    }

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            distancias[i][j - i - 1] = calc_distancia(particulas.x[i], particulas.y[i],
                particulas.z[i],
                                                      particulas.x[j], particulas.y[j],
                                                          particulas.z[j]);
        }
    }

    *tempo_calculo = (double)(clock() - inicio) / CLOCKS_PER_SEC;
    return distancias;
}

//  Funo  para obter a distncia entre duas partculas usando a matriz triangular
double obter_distancia(double** distancias, int i, int j) {
    if (i < j) {
        return distancias[i][j - i - 1];
    }
    return distancias[j][i - j - 1];
}

//  Funo  para encontrar os m vizinhos mais prximos
void encontrar_vizinhos(ParticlesSoA particulas, int n, int m, double** distancias, int**
    vizinhos) {
    for (int i = 0; i < n; i++) {
        double* distancias_minimas = (double*)malloc(m * sizeof(double));
        int* indices_minimos = (int*)malloc(m * sizeof(int));

        // Inicializar distncias mnimas
        for (int k = 0; k < m; k++) {
            distancias_minimas[k] = INFINITY;
```

```
            indices_minimos[k] = -1;
        }

        for (int j = 0; j < n; j++) {
            if (i == j) continue;

            double distancia = obter_distancia(distancias, i, j);

            // Inserir a distncia na lista dos m menores
            for (int k = 0; k < m; k++) {
                if (distancia < distancias_minimas[k]) {
                    for (int l = m - 1; l > k; l--) {
                        distancias_minimas[l] = distancias_minimas[l - 1];
                        indices_minimos[l] = indices_minimos[l - 1];
                    }
                    distancias_minimas[k] = distancia;
                    indices_minimos[k] = j;
                    break;
                }
            }
        }

        // Copiar os vizinhos encontrados
        for (int k = 0; k < m; k++) {
            vizinhos[i][k] = indices_minimos[k];
        }

        free(distancias_minimas);
        free(indices_minimos);
    }
}

//  Funo  para escrever os vizinhos em um arquivo
void escrever_vizinhos(const char* nome_entrada, int** vizinhos, int n, int m, double*
    tempo_escrita) {
    clock_t inicio = clock();
    char nome_saida[FILENAME_MAX];
    snprintf(nome_saida, sizeof(nome_saida), "%s.ngb", nome_entrada);

    FILE* arquivo = fopen(nome_saida, "w");
    if (!arquivo) {
        perror("Erro ao abrir o arquivo de sada");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            fprintf(arquivo, "%d ", vizinhos[i][j]);
        }
        fprintf(arquivo, "\n");
```

```
    }
    fclose(arquivo);

    *tempo_escrita = (double)(clock() - inicio) / CLOCKS_PER_SEC;
}


int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Uso: %s <arquivo de entrada> <nmero de vizinhos>\n", argv[0]);
        return EXIT_FAILURE;
    }

    const char* nome_entrada = argv[1];
    int m = atoi(argv[2]);

    int n;
    double tempo_leitura, tempo_calculo, tempo_escrita;

    ParticlesSoA particulas = ler_particulas(nome_entrada, &n, &tempo_leitura);

    double** distancias = calcular_distancias(particulas, n, &tempo_calculo);

    // Alocar memria para os vizinhos
    int** vizinhos = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        vizinhos[i] = (int*)malloc(m * sizeof(int));
    }


    encontrar_vizinhos(particulas, n, m, distancias, vizinhos);
    escrever_vizinhos(nome_entrada, vizinhos, n, m, &tempo_escrita);
    printf("%d %d %.6lf %.6lf %.6lf\n", n, m, tempo_leitura, tempo_calculo, tempo_escrita);

    // Liberar memria alocada
    free(particulas.x);
    free(particulas.y);
    free(particulas.z);

    for (int i = 0; i < n; i++) {
        free(vizinhos[i]);
    }
    free(vizinhos);

    for (int i = 0; i < n; i++) {
        free(distancias[i]);
    }
    free(distancias);

    return EXIT_SUCCESS;
```

```
}
```

5.2 Version 1.1 - t1

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>

#define MAX_LINE_LENGTH 100

typedef struct {
    double x, y, z;
} Particle;

//  Funo  para calcular a distncia euclidiana entre duas partculas
double euclidean_distance(Particle a, Particle b) {
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2) + pow(a.z - b.z, 2));
}

//  Funo  para ler partculas do arquivo
Particle* read_particles(const char* filename, int* n, double* read_time) {
    clock_t start = clock();
    FILE* file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "Erro ao abrir o arquivo %s\n", filename);
        exit(EXIT_FAILURE);
    }

    // Contar o nmero de linhas (partculas)
    *n = 0;
    char line[MAX_LINE_LENGTH];
    while (fgets(line, sizeof(line), file)) {
        (*n)++;
    }

    // Alocar memria para as partculas
    Particle* particles = (Particle*)malloc(*n * sizeof(Particle));
    if (!particles) {
        fprintf(stderr, "Erro ao alocar memria\n");
        exit(EXIT_FAILURE);
    }

    // Rewind do arquivo para ler as partculas
    rewind(file);
    for (int i = 0; i < *n; i++) {
```

```
        if (fscanf(file, "%lf %lf %lf", &particles[i].x, &particles[i].y, &particles[i].z) !=
            3) {
            fprintf(stderr, "Erro ao ler a linha %d do arquivo\n", i);
            exit(EXIT_FAILURE);
        }
    }
    fclose(file);

    *read_time = (double)(clock() - start) / CLOCKS_PER_SEC;
    return particles;
}

//  Funo  para calcular e armazenar as distncias na matriz triangular superior
double** compute_distances(Particle* particles, int n, double* computation_time) {
    clock_t start = clock();

    // Alocar matriz triangular superior
    double** distances = (double**)malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        distances[i] = (double*)malloc((n - i - 1) * sizeof(double));
    }

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            distances[i][j - i - 1] = euclidean_distance(particles[i], particles[j]);
        }
    }

    *computation_time = (double)(clock() - start) / CLOCKS_PER_SEC;
    return distances;
}

//  Funo  para obter a distncia entre duas partculas utilizando a matriz triangular
double get_distance(double** distances, int i, int j) {
    if (i < j) {
        return distances[i][j - i - 1];
    } else {
        return distances[j][i - j - 1];
    }
}

//  Funo  para encontrar os m vizinhos mais prximos de cada partcula
void find_neighbors(Particle* particles, int n, int m, double** distances, int** neighbors) {
    for (int i = 0; i < n; i++) {
        double* min_distances = (double*)malloc(m * sizeof(double));
        int* min_indices = (int*)malloc(m * sizeof(int));

        // Inicializar distncias mnimas com infinito
        for (int k = 0; k < m; k++) {
            min_distances[k] = INFINITY;
```

```
            min_indices[k] = -1;
        }

        for (int j = 0; j < n; j++) {
            if (i == j) continue;

            double dist = get_distance(distances, i, j);

            // Inserir a distncia na lista dos m menores
            for (int k = 0; k < m; k++) {
                if (dist < min_distances[k]) {
                    // Desloca os valores para dar espao
                    for (int l = m - 1; l > k; l--) {
                        min_distances[l] = min_distances[l - 1];
                        min_indices[l] = min_indices[l - 1];
                    }
                    min_distances[k] = dist;
                    min_indices[k] = j;
                    break;
                }
            }
        }

        // Copiar os vizinhos encontrados
        for (int k = 0; k < m; k++) {
            neighbors[i][k] = min_indices[k];
        }

        free(min_distances);
        free(min_indices);
    }
}

//  Funo  para escrever os vizinhos no arquivo de sada
void write_neighbors(const char* input_filename, int** neighbors, int n, int m, double*
    write_time) {
    clock_t start = clock();

    char output_filename[FILENAME_MAX];
    snprintf(output_filename, sizeof(output_filename), "%s.ngb", input_filename);

    FILE* file = fopen(output_filename, "w");
    if (!file) {
        fprintf(stderr, "Erro ao abrir o arquivo de sada %s\n", output_filename);
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            fprintf(file, "%d ", neighbors[i][j]);
```

```
        }
        fprintf(file, "\n");
    }
    fclose(file);

    *write_time = (double)(clock() - start) / CLOCKS_PER_SEC;
}

//  Funo  principal
int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Uso: %s <arquivo de entrada> <nmero de vizinhos>\n", argv[0]);
        return EXIT_FAILURE;
    }

    const char* input_filename = argv[1];
    int m = atoi(argv[2]);

    int n;
    double read_time, computation_time, write_time;

    // Ler as partculas do arquivo
    Particle* particles = read_particles(input_filename, &n, &read_time);

    // Calcular e armazenar as distncias na matriz triangular superior
    double** distances = compute_distances(particles, n, &computation_time);

    // Alocar memria para os vizinhos
    int** neighbors = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        neighbors[i] = (int*)malloc(m * sizeof(int));
    }

    // Encontrar os vizinhos mais prximos
    find_neighbors(particles, n, m, distances, neighbors);

    // Escrever os vizinhos no arquivo de sada
    write_neighbors(input_filename, neighbors, n, m, &write_time);

    // Imprimir os resultados na sada padro
    printf("%d %d %.6lf %.6lf %.6lf\n", n, m, read_time, computation_time, write_time);

    // Liberar memria
    free(particles);
    for (int i = 0; i < n; i++) {
        free(neighbors[i]);
    }
    free(neighbors);

    for (int i = 0; i < n; i++) {
```

```
        free(distances[i]);
    }
    free(distances);

    return EXIT_SUCCESS;
}
```

5.3 Versions 1.2 - $t1_{SoA}$

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>

#define MAX_LINE_LENGTH 100

typedef struct {
    double* x;
    double* y;
    double* z;
} ParticlesSoA;

//  Funo  para calcular a distncia euclidiana entre duas partculas
double euclidean_distance(double ax, double ay, double az, double bx, double by, double bz) {
    return sqrt(pow(ax - bx, 2) + pow(ay - by, 2) + pow(az - bz, 2));
}

//  Funo  para ler partculas do arquivo
ParticlesSoA read_particles(const char* filename, int* n, double* read_time) {
    clock_t start = clock();
    FILE* file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "Erro ao abrir o arquivo %s\n", filename);
        exit(EXIT_FAILURE);
    }

    // Contar o nmero de linhas (partculas)
    *n = 0;
    char line[MAX_LINE_LENGTH];
    while (fgets(line, sizeof(line), file)) {
        (*n)++;
    }

    // Alocar memria para as partculas
    ParticlesSoA particles;
    particles.x = (double*)malloc(*n * sizeof(double));
    particles.y = (double*)malloc(*n * sizeof(double));
```

```
    particles.z = (double*)malloc(*n * sizeof(double));

    if (!particles.x || !particles.y || !particles.z) {
        fprintf(stderr, "Erro ao alocar memria\n");
        exit(EXIT_FAILURE);
    }

    // Rewind do arquivo para ler as partculas
    rewind(file);
    for (int i = 0; i < *n; i++) {
        if (fscanf(file, "%lf %lf %lf", &particles.x[i], &particles.y[i], &particles.z[i]) !=
            3) {
            fprintf(stderr, "Erro ao ler a linha %d do arquivo\n", i);
            exit(EXIT_FAILURE);
        }
    }
    fclose(file);

    *read_time = (double)(clock() - start) / CLOCKS_PER_SEC;
    return particles;
}

//  Funo  para calcular e armazenar as distncias na matriz triangular superior
double** compute_distances(ParticlesSoA particles, int n, double* computation_time) {
    clock_t start = clock();

    // Alocar matriz triangular superior
    double** distances = (double**)malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        distances[i] = (double*)malloc((n - i - 1) * sizeof(double));
    }

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            distances[i][j - i - 1] = euclidean_distance(particles.x[i], particles.y[i],
                particles.z[i],
                                                         particles.x[j], particles.y[j],
                                                             particles.z[j]);
        }
    }

    *computation_time = (double)(clock() - start) / CLOCKS_PER_SEC;
    return distances;
}

//  Funo  para obter a distncia entre duas partculas utilizando a matriz triangular
double get_distance(double** distances, int i, int j) {
    if (i < j) {
        return distances[i][j - i - 1];
    } else {
```

```
        return distances[j][i - j - 1];
    }
}

//  Funo  para encontrar os m vizinhos mais prximos de cada partcula
void find_neighbors(ParticlesSoA particles, int n, int m, double** distances, int**
    neighbors) {
    for (int i = 0; i < n; i++) {
        double* min_distances = (double*)malloc(m * sizeof(double));
        int* min_indices = (int*)malloc(m * sizeof(int));

        // Inicializar distncias mnimas com infinito
        for (int k = 0; k < m; k++) {
            min_distances[k] = INFINITY;
            min_indices[k] = -1;
        }

        for (int j = 0; j < n; j++) {
            if (i == j) continue;

            double dist = get_distance(distances, i, j);

            // Inserir a distncia na lista dos m menores
            for (int k = 0; k < m; k++) {
                if (dist < min_distances[k]) {
                    // Desloca os valores para dar espao
                    for (int l = m - 1; l > k; l--) {
                        min_distances[l] = min_distances[l - 1];
                        min_indices[l] = min_indices[l - 1];
                    }
                    min_distances[k] = dist;
                    min_indices[k] = j;
                    break;
                }
            }
        }

        // Copiar os vizinhos encontrados
        for (int k = 0; k < m; k++) {
            neighbors[i][k] = min_indices[k];
        }

        free(min_distances);
        free(min_indices);
    }
}

//  Funo  para escrever os vizinhos no arquivo de sada
void write_neighbors(const char* input_filename, int** neighbors, int n, int m, double*
    write_time) {
```

```
    clock_t start = clock();

    char output_filename[FILENAME_MAX];
    snprintf(output_filename, sizeof(output_filename), "%s.ngb", input_filename);

    FILE* file = fopen(output_filename, "w");
    if (!file) {
        fprintf(stderr, "Erro ao abrir o arquivo de sada %s\n", output_filename);
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            fprintf(file, "%d ", neighbors[i][j]);
        }
        fprintf(file, "\n");
    }
    fclose(file);

    *write_time = (double)(clock() - start) / CLOCKS_PER_SEC;
}

//  Funo  principal
int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Uso: %s <arquivo de entrada> <nmero de vizinhos>\n", argv[0]);
        return EXIT_FAILURE;
    }

    const char* input_filename = argv[1];
    int m = atoi(argv[2]);

    int n;
    double read_time, computation_time, write_time;

    // Ler as partculas do arquivo
    ParticlesSoA particles = read_particles(input_filename, &n, &read_time);

    // Calcular e armazenar as distncias na matriz triangular superior
    double** distances = compute_distances(particles, n, &computation_time);

    // Alocar memria para os vizinhos
    int** neighbors = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        neighbors[i] = (int*)malloc(m * sizeof(int));
    }

    // Encontrar os vizinhos mais prximos
    find_neighbors(particles, n, m, distances, neighbors);
```

```
    // Escrever os vizinhos no arquivo de sada
    write_neighbors(input_filename, neighbors, n, m, &write_time);

    // Imprimir os resultados na sada padro
    printf("%d %d %.6lf %.6lf %.6lf\n", n, m, read_time, computation_time, write_time);

    // Liberar memria
    free(particles.x);
    free(particles.y);
    free(particles.z);
    for (int i = 0; i < n; i++) {
        free(neighbors[i]);
    }
    free(neighbors);

    for (int i = 0; i < n; i++) {
        free(distances[i]);
    }
    free(distances);

    return EXIT_SUCCESS;
}
```

5.4 Version 1.3 - t1$_{dist-max}$

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <time.h>

typedef struct {
    double* x;
    double* y;
    double* z;
} ParticleArray;

//  Funo  para calcular a distncia mdia
double calculate_average_distance(ParticleArray particles, int N) {
    double total_distance = 0.0;
    int count = 0;

    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            double dx = particles.x[i] - particles.x[j];
            double dy = particles.y[i] - particles.y[j];
            double dz = particles.z[i] - particles.z[j];
            double distance = sqrt(dx * dx + dy * dy + dz * dz);
```

```
            total_distance += distance;
            count++;
        }
    }

    return total_distance / count; // Retorna a distncia mdia
}

//  Funo  para encontrar os vizinhos mais prximos
void find_nearest_neighbors(ParticleArray particles, int N, int m, double max_distance, int**
    neighbors) {
    for (int i = 0; i < N; i++) {
        int count = 0;

        for (int j = 0; j < N; j++) {
            if (i != j) {
                double dx = particles.x[i] - particles.x[j];
                double dy = particles.y[i] - particles.y[j];
                double dz = particles.z[i] - particles.z[j];
                double distance = sqrt(dx * dx + dy * dy + dz * dz);

                // Verifica se a distncia est dentro do limite
                if (distance <= max_distance) {
                    // Se ainda no temos m vizinhos, armazena o ndice
                    if (count < m) {
                        neighbors[i][count] = j;
                        count++;
                    } else {
                        // Verifica se o novo vizinho  mais prximo do que o mais distante atual
                        double max_dist = 0;
                        int max_index = -1;
                        for (int k = 0; k < m; k++) {
                            if (neighbors[i][k] != -1) {
                                double dx_max = particles.x[i] - particles.x[neighbors[i][k]];
                                double dy_max = particles.y[i] - particles.y[neighbors[i][k]];
                                double dz_max = particles.z[i] - particles.z[neighbors[i][k]];
                                double dist_max = sqrt(dx_max * dx_max + dy_max * dy_max +
                                    dz_max * dz_max);
                                if (dist_max > max_dist) {
                                    max_dist = dist_max;
                                    max_index = k;
                                }
                            }
                        }

                        // Substitui o vizinho mais distante se o novo vizinho for mais prximo
                        if (max_index != -1 && distance < max_dist) {
                            neighbors[i][max_index] = j;
                        }
                    }
```

```
            }
        }
    }
}

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <input_file> <number_of_neighbors>\n", argv[0]);
        return 1;
    }

    const char* input_file = argv[1];
    int m = atoi(argv[2]);

    // Marca o tempo de incio da leitura
    clock_t start_time = clock();

    // Leitura dos dados
    FILE* file = fopen(input_file, "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    int N = 0;
    ParticleArray particles;
    particles.x = (double*)malloc(0);
    particles.y = (double*)malloc(0);
    particles.z = (double*)malloc(0);

    while (!feof(file)) {
        particles.x = (double*)realloc(particles.x, (N + 1) * sizeof(double));
        particles.y = (double*)realloc(particles.y, (N + 1) * sizeof(double));
        particles.z = (double*)realloc(particles.z, (N + 1) * sizeof(double));
        fscanf(file, "%lf %lf %lf", &particles.x[N], &particles.y[N], &particles.z[N]);
        N++;
    }
    fclose(file);

    // Calcula o tempo de leitura
    clock_t end_read_time = clock();
    double read_time = (double)(end_read_time - start_time) / CLOCKS_PER_SEC;

    // Calcular a distncia mxima automaticamente
    double max_distance = calculate_average_distance(particles, N);

    // Aloca matriz de vizinhos
    int** neighbors = (int**)malloc(N * sizeof(int*));
    for (int i = 0; i < N; i++) {
```

```
    neighbors[i] = (int*)malloc(m * sizeof(int));
    for (int j = 0; j < m; j++) {
        neighbors[i][j] = -1; // Inicializa como -1
    }
}

// Marca o incio do tempo de clculo
clock_t start_calc_time = clock();

// Executar a busca de vizinhos
find_nearest_neighbors(particles, N, m, max_distance, neighbors);

// Calcula o tempo de clculo
clock_t end_calc_time = clock();
double calc_time = (double)(end_calc_time - start_calc_time) / CLOCKS_PER_SEC;

// Marca o incio do tempo de escrita
clock_t start_write_time = clock();

// Escreve os resultados no arquivo de sada
char output_file[256];
snprintf(output_file, sizeof(output_file), "%s.ngb", input_file);
FILE* out_file = fopen(output_file, "w");
if (!out_file) {
    perror("Error creating output file");
    return 1;
}

for (int i = 0; i < N; i++) {
    for (int j = 0; j < m; j++) {
        if (neighbors[i][j] != -1) {
            fprintf(out_file, "%d ", neighbors[i][j]);
        }
    }
    fprintf(out_file, "\n");
}
fclose(out_file);

// Calcula o tempo de escrita
clock_t end_write_time = clock();
double write_time = (double)(end_write_time - start_write_time) / CLOCKS_PER_SEC;

//  Sada  dos resultados no formato desejado
printf("%d %d %f %f %f\n", N, m, read_time, calc_time, write_time);

// Liberar memria
for (int i = 0; i < N; i++) {
    free(neighbors[i]);
}
free(neighbors);
```

```
    free(particles.x);
    free(particles.y);
    free(particles.z);

    return 0;
}
```

5.5 Versions 1.4 - t1$_{kd-tree}$

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <string.h>
#include <time.h>

#define MAX_LINE_LENGTH 100

typedef struct KDNode {
    double x, y, z;
    struct KDNode *left, *right;
} KDNode;

typedef struct Neighbor {
    double distance;
    KDNode* node;
} Neighbor;

//  Funo  para calcular a distncia euclidiana entre duas partculas
double euclidean_distance(double ax, double ay, double az, double bx, double by, double bz) {
    return sqrt(pow(ax - bx, 2) + pow(ay - by, 2) + pow(az - bz, 2));
}

//  Funes  de  comparao  para ordenar partculas por cada eixo
int compare_x(const void* a, const void* b) {
    double x1 = *(double*)a;
    double x2 = *(double*)b;
    return (x1 > x2) - (x1 < x2);
}

int compare_y(const void* a, const void* b) {
    double y1 = *(double*)a;
    double y2 = *(double*)b;
    return (y1 > y2) - (y1 < y2);
}

int compare_z(const void* a, const void* b) {
    double z1 = *(double*)a;
```

```
    double z2 = *(double*)b;
    return (z1 > z2) - (z1 < z2);
}

//  Funo  recursiva para construir a K-D Tree
KDNode* build_kd_tree(double* x, double* y, double* z, int start, int end, int depth) {
    if (start > end) return NULL;

    int axis = depth % 3; // Alterna entre as 3 dimenses
    int mid = (start + end) / 2;

    // Ordena as partculas pelo eixo corrente
    if (axis == 0) {
        qsort(x + start, end - start + 1, sizeof(double), compare_x);
    } else if (axis == 1) {
        qsort(y + start, end - start + 1, sizeof(double), compare_y);
    } else {
        qsort(z + start, end - start + 1, sizeof(double), compare_z);
    }

    KDNode* node = (KDNode*)malloc(sizeof(KDNode));
    node->x = x[mid];
    node->y = y[mid];
    node->z = z[mid];

    // Recurso para os filhos esquerdo e direito
    node->left = build_kd_tree(x, y, z, start, mid - 1, depth + 1);
    node->right = build_kd_tree(x, y, z, mid + 1, end, depth + 1);

    return node;
}

//  Funo  para ler partculas do arquivo
void read_particles(const char* filename, double** x, double** y, double** z, int* n, double*
    read_time) {
    clock_t start = clock();
    FILE* file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "Erro ao abrir o arquivo %s\n", filename);
        exit(EXIT_FAILURE);
    }

    // Contar o nmero de linhas (partculas)
    *n = 0;
    char line[MAX_LINE_LENGTH];
    while (fgets(line, sizeof(line), file)) {
        (*n)++;
    }

    // Alocar memria para as partculas
```

```
    *x = (double*)malloc(*n * sizeof(double));
    *y = (double*)malloc(*n * sizeof(double));
    *z = (double*)malloc(*n * sizeof(double));

    if (!*x || !*y || !*z) {
        fprintf(stderr, "Erro ao alocar memria\n");
        exit(EXIT_FAILURE);
    }

    // Rewind do arquivo para ler as partculas
    rewind(file);
    for (int i = 0; i < *n; i++) {
        if (fscanf(file, "%lf %lf %lf", &(*x)[i], &(*y)[i], &(*z)[i]) != 3) {
            fprintf(stderr, "Erro ao ler a linha %d do arquivo\n", i);
            exit(EXIT_FAILURE);
        }
    }
    fclose(file);

    *read_time = (double)(clock() - start) / CLOCKS_PER_SEC;
}

//  Funo  para buscar os m vizinhos mais prximos
void find_k_nearest_neighbors(KDNode* root, double x, double y, double z, Neighbor*
    neighbors, int m, int depth) {
    if (!root) return;

    double dist = euclidean_distance(x, y, z, root->x, root->y, root->z);

    // Verifica se a distncia  menor que a distncia mxima no array de vizinhos
    if (dist < neighbors[m - 1].distance) {
        neighbors[m - 1].distance = dist;
        neighbors[m - 1].node = root;

        // Reordena o array de vizinhos pelo menor
        for (int i = m - 1; i > 0 && neighbors[i].distance < neighbors[i - 1].distance; i--) {
            Neighbor temp = neighbors[i];
            neighbors[i] = neighbors[i - 1];
            neighbors[i - 1] = temp;
        }
    }

    int axis = depth % 3;
    double diff = (axis == 0) ? (x - root->x) : (axis == 1) ? (y - root->y) : (z - root->z);

    KDNode* near_subtree = (diff < 0) ? root->left : root->right;
    KDNode* far_subtree = (diff < 0) ? root->right : root->left;

    find_k_nearest_neighbors(near_subtree, x, y, z, neighbors, m, depth + 1);
```

```
    // Verificar se a outra subrvore pode conter vizinhos mais prximos
    if (fabs(diff) < neighbors[m - 1].distance) {
        find_k_nearest_neighbors(far_subtree, x, y, z, neighbors, m, depth + 1);
    }
}

//  Funo  principal
int main(int argc, char* argv[]) {
    if (argc != 4) {
        fprintf(stderr, "Uso: %s <nome_do_arquivo> <numero_de_vizinhos>
            <nome_do_arquivo_saida>\n", argv[0]);
        return 1;
    }

    double* x;
    double* y;
    double* z;
    int n;
    double read_time, calc_time, write_time;

    // Ler partculas
    read_particles(argv[1], &x, &y, &z, &n, &read_time);

    // Construir a K-D Tree
    clock_t start_calc = clock();
    KDNode* root = build_kd_tree(x, y, z, 0, n - 1, 0);
    calc_time = (double)(clock() - start_calc) / CLOCKS_PER_SEC;

    //  Nmero  de vizinhos
    int m = atoi(argv[2]);
    if (m > n) {
        fprintf(stderr, "O nmero de vizinhos no pode ser maior que o nmero de partculas.\n");
        return 1;
    }

    FILE* output_file = fopen(argv[3], "w");
    if (!output_file) {
        fprintf(stderr, "Erro ao abrir o arquivo de sada %s\n", argv[3]);
        return 1;
    }

    // Para cada partcula, encontrar os m vizinhos mais prximos
    clock_t start_write = clock();
    for (int i = 0; i < n; i++) {
        Neighbor* neighbors = (Neighbor*)malloc(m * sizeof(Neighbor));
        for (int j = 0; j < m; j++) {
            neighbors[j].distance = DBL_MAX;
            neighbors[j].node = NULL;
        }
```

```
        find_k_nearest_neighbors(root, x[i], y[i], z[i], neighbors, m, 0);

        // Escrever os resultados no arquivo de sada
        fprintf(output_file, "Partcula %d: (%f, %f, %f)\n", i, x[i], y[i], z[i]);
        for (int j = 0; j < m; j++) {
            if (neighbors[j].node) {
                fprintf(output_file, "  Vizinho %d: (%f, %f, %f) Distncia: %f\n", j,
                        neighbors[j].node->x, neighbors[j].node->y, neighbors[j].node->z,
                        neighbors[j].distance);
            }
        }
        free(neighbors);
    }
    write_time = (double)(clock() - start_write) / CLOCKS_PER_SEC;

    fclose(output_file);

    //  Liberao  de memria
    free(x);
    free(y);
    free(z);
    free(root);

    // Impresso dos tempos e resultados
    printf("N: %d, m: %d, Tempo de Leitura: %f segundos, Tempo de Clculo: %f segundos, Tempo
        de Escrita: %f segundos\n",
           n, m, read_time, calc_time, write_time);

    return 0;
}
```