Códigos e Exercícios: Algoritmos de Ordenação Paralelos

Baseado nos slides e Cap. 9 de Grama et al.

Índice

Índice

Códigos dos Algoritmos de Ordenação

1. Odd-Even Transposition Sort (Versão Bloco)

Este algoritmo ordena n elementos usando p processos. Cada processo detém n/p elementos.

Pseudocódigo

1. **Cada processo P_i (i=0..p-1):** Ordena localmente seus n/p elementos (usando Quicksort/Mergesort).
2. **for fase = 1 to p:**
3. **if fase é ÍMPAR:**
4. **forall processos P_i ÍMPARES em paralelo:**
5. Envia bloco local para P_{i+1}
6. Recebe bloco de P_{i+1}
7. Mescla bloco local e recebido.
8. Mantém os n/p menores elementos (compare-split min).
9. **forall processos P_i PARES (com i > 0) em paralelo:**
10. Recebe bloco de P_{i-1}
11. Envia bloco local para P_{i-1}
12. Mescla bloco local e recebido.
13. Mantém os n/p maiores elementos (compare-split max).
14. **else (fase é PAR):**
15. **forall processos P_i PARES em paralelo:**
16. Envia bloco local para P_{i+1} (se i+1 < p)
17. Recebe bloco de P_{i+1} (se i+1 < p)
18. Mescla bloco local e recebido.
19. Mantém os n/p menores elementos.
20. **forall processos P_i ÍMPARES em paralelo:**
21. Recebe bloco de P_{i-1}
22. Envia bloco local para P_{i-1}
23. Mescla bloco local e recebido.
24. Mantém os n/p maiores elementos.
25. **end for**

Código Serial C (Ilustrativo da lógica sequencial de fases)

Nota: Uma versão puramente sequencial simplesmente usaria qsort ou mergesort no array inteiro. Este código simula as fases sequencialmente para mostrar a lógica que será paralelizada.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // Para memcpy

// Função de comparação para qsort
int compare_int(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

// Função para mesclar e dividir (simula compare-split)
// Retorna os menores em 'local', os maiores em 'other'
void merge_split_low(int local[], int other[], int temp[], int n_local) {
    int i = 0, j = 0, k = 0;
    // Mescla 'local' e 'other' em 'temp'
    while (i < n_local && j < n_local) {
        if (local[i] <= other[j]) {
            temp[k++] = local[i++];
        } else {
            temp[k++] = other[j++];
```

```
        }
    }
    while (i < n_local) temp[k++] = local[i++];
    while (j < n_local) temp[k++] = other[j++];

    // Mantém os menores em 'local'
    memcpy(local, temp, n_local * sizeof(int));
}

// Retorna os maiores em 'local', os menores em 'other'
void merge_split_high(int local[], int other[], int temp[], int n_local) {
    int i = 0, j = 0, k = 0;
     // Mescla 'local' e 'other' em 'temp'
    while (i < n_local && j < n_local) {
        if (local[i] <= other[j]) {
            temp[k++] = local[i++];
        } else {
            temp[k++] = other[j++];
        }
    }
    while (i < n_local) temp[k++] = local[i++];
    while (j < n_local) temp[k++] = other[j++];

    // Mantém os maiores em 'local'
    memcpy(local, temp + n_local, n_local * sizeof(int));
}


// Simulação sequencial das fases
void sequential_odd_even_block(int *data[], int n, int p) {
    int n_local = n / p;
    int *temp = (int *)malloc(2 * n_local * sizeof(int));
    int *recv_buffer = (int *)malloc(n_local * sizeof(int)); // Simula buffer

    // 1. Ordenação local inicial (não mostrada, assume-se feita)
    // for (int i = 0; i < p; ++i) {
    //     qsort(data[i], n_local, sizeof(int), compare_int);
    // }

    // 2. Fases de Compare-Split
    for (int phase = 0; phase < p; ++phase) {
        if (phase % 2 == 0) { // Fase Par
            for (int i = 0; i < p - 1; i += 2) {
                // Simula P_i envia para P_{i+1}, P_{i+1} envia para P_i
                memcpy(recv_buffer, data[i+1], n_local * sizeof(int)); // P_i recebe de P_{i+1}
                merge_split_low(data[i], recv_buffer, temp, n_local); // P_i mantém menores

                memcpy(recv_buffer, data[i], n_local * sizeof(int)); // P_{i+1} recebe de P_i
                merge_split_high(data[i+1], recv_buffer, temp, n_local); // P_{i+1} mantém maiores
            }
        } else { // Fase Ímpar
            for (int i = 1; i < p - 1; i += 2) {
                 // Simula P_i envia para P_{i+1}, P_{i+1} envia para P_i
                memcpy(recv_buffer, data[i+1], n_local * sizeof(int)); // P_i recebe de P_{i+1}
                merge_split_low(data[i], recv_buffer, temp, n_local); // P_i mantém menores

                memcpy(recv_buffer, data[i], n_local * sizeof(int)); // P_{i+1} recebe de P_i
                merge_split_high(data[i+1], recv_buffer, temp, n_local); // P_{i+1} mantém maiores
            }
        }
    }
    free(temp);
    free(recv_buffer);
}

// Função main para teste (simplificada)
// int main() {
//    // Alocar e inicializar 'data' como um array de ponteiros para blocos
//    // Chamar qsort local para cada bloco
//    // Chamar sequential_odd_even_block(data, n, p);
//    // Verificar resultado
//    return 0;
// }
```

2. Rank Sort (Versão n Processadores)

Pseudocódigo

1. **forall processos P_i (i=0..n-1) em paralelo:**
2. `x_i = 0` (rank local)
3. `meu_valor = a[i]`
4. **for j = 0 to n-1:**
5. `outro_valor = READ(a[j])` (Acesso pode requerer comunicação implícita/cache)
6. **if** (`meu_valor > outro_valor`) **or** (`meu_valor == outro_valor` **and** `j < i`) **then**
7. `x_i = x_i + 1`
8. **end for**
9. `b[x_i] = meu_valor`

Código Serial C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // Para clock() e CLOCKS_PER_SEC

/**
 * @brief Performs serial rank sort.
 *
 * Sorts array 'a' into array 'b'. Handles duplicates for stability.
 * Assumes 'a' and 'b' are allocated with size 'n'.
 *
 * @param a Input array (unsorted).
 * @param b Output array (sorted).
 * @param n Number of elements.
 */
void serial_rank_sort(int a[], int b[], int n) {
    int i, j; // Loop counters

    // Loop externo: Itera sobre cada elemento a[i] para encontrar seu rank
    for (i = 0; i < n; i++) {
        int x = 0; // Inicializa o rank para o elemento a[i] atual

        // Loop interno: Compara a[i] com todos os outros elementos a[j]
        for (j = 0; j < n; j++) {
            // Comparação estável: Incrementa o rank se a[j] for menor,
            // ou se a[j] for igual mas aparecer antes no array original.
            if (a[i] > a[j] || (a[i] == a[j] && j < i)) {
                x++;
            }
        }
        // Coloca a[i] na posição correta 'x' no array de saída 'b'
        b[x] = a[i];
    }
}

// Função auxiliar para imprimir arrays (para teste)
void print_array(const char* label, int arr[], int n) {
    printf("%s: [", label);
    for (int i = 0; i < n; i++) {
        printf("%d%s", arr[i], (i == n - 1) ? "" : ", ");
    }
    printf("]\n");
}

// Função principal para teste
int main(int argc, char *argv[]) {
    int n = 1000; // Tamanho padrão se nenhum argumento for passado
    if (argc > 1) {
        n = atoi(argv[1]);
    }
     if (n <= 0) {
        fprintf(stderr, "Tamanho N inválido\n");
        return 1;
```

```
    }

    int *a = (int *)malloc(n * sizeof(int));
    int *b = (int *)malloc(n * sizeof(int));

    if (!a || !b) {
        fprintf(stderr, "Falha na alocação de memória\n");
        free(a); free(b);
        return 1;
    }

    // Inicializa com dados aleatórios (e algumas duplicatas)
    srand(time(NULL)); // Use a semente do tempo atual uma vez
    printf("Gerando %d elementos...\n", n);
    for (int i = 0; i < n; i++) {
        a[i] = rand() % (n / 2); // Gera números no intervalo [0, n/2 - 1] para garantir duplicatas
    }

    // Opcional: Imprime array não ordenado para N pequeno
    // if (n <= 20) print_array("Não Ordenado", a, n);

    printf("Ordenando usando Rank Sort Serial...\n");

    // --- Medição de Tempo Serial ---
    clock_t start_time = clock(); // Marca o início
    serial_rank_sort(a, b, n);
    clock_t end_time = clock();   // Marca o fim
    // --- Fim da Medição ---

    // Calcula o tempo de execução em segundos
    double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    // Opcional: Imprime array ordenado para N pequeno
    // if (n <= 20) print_array("Ordenado", b, n);

    // Verificação (simples)
    int sorted = 1;
    for (int i = 0; i < n - 1; i++) {
        if (b[i] > b[i + 1]) {
            sorted = 0;
            fprintf(stderr, "Erro de ordenação em índice %d: %d > %d\n", i, b[i], b[i+1]);
            break;
        }
    }
    printf("Verificação: %s\n", sorted ? "Passou" : "Falhou");
    printf("Tempo Serial Gasto: %f segundos\n", time_taken);
    printf("Nota: Correção depende da lógica estável se houver duplicatas.\n");

    free(a);
    free(b);
    return 0;
}
```

3. Counting Sort

Pseudocódigo

1. **Fase 1: Histograma Paralelo**
2. Inicializa array global `C[1..m]` com zeros (pode ser feito em paralelo ou serialmente).
3. **forall processos P_k (k=0..p-1) em paralelo:**
4. Inicializa histograma local `C_local[1..m]` com zeros.
5. **for** cada elemento `a_elem` no bloco local de P_k:
6. `C_local[a_elem]++`
7. **end for**
8. Combina histogramas locais em `C` global (usando redução atômica/locks ou uma fase de redução explícita).
9. **Fase 2: Soma de Prefixos** (Pode ser serial no mestre ou paralela)

10. Calcula soma de prefixos no array `C`.
11. **Fase 3: Colocação Paralela**
12. **forall processos P_k (k=0..p-1) em paralelo:**
13. **for** cada elemento `a_elem` no bloco local de P_k (iterando de trás para frente):
14. `pos = C[a_elem]`
15. `b[pos] = a_elem`
16. `ATOMIC_DECREMENT(C[a_elem])` (ou usa posições pré-calculadas exatas).

Código Serial C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // Para memset

void serial_counting_sort(int a[], int b[], int n, int m) {
    // Assume que os valores em 'a' estão na faixa [1, m]
    // Aloca array de contagem (índices 1 a m)
    int *c = (int *)malloc((m + 1) * sizeof(int));
    if (!c) { perror("malloc failed"); exit(1); }

    // 1. Histograma
    memset(c, 0, (m + 1) * sizeof(int)); // Inicializa com 0
    for (int i = 0; i < n; ++i) {
        if (a[i] >= 1 && a[i] <= m) { // Checa se está na faixa
            c[a[i]]++;
        } else {
            fprintf(stderr, "Erro: Elemento %d fora da faixa [1, %d]\n", a[i], m);
            // Tratar erro ou ignorar
        }
    }

    // 2. Soma de Prefixos
    for (int i = 2; i <= m; ++i) {
        c[i] += c[i - 1];
    }

    // 3. Colocação (iterando de trás para frente para estabilidade)
    // Índices de 'a' vão de 0 a n-1, índices de 'b' vão de 1 a n (baseado em c)
    for (int i = n - 1; i >= 0; --i) {
         if (a[i] >= 1 && a[i] <= m) { // Checa novamente
             b[c[a[i]]] = a[i]; // Coloca em b na posição correta (1-based)
             c[a[i]]--;         // Decrementa contador para próxima duplicata
         }
    }
    // Nota: O array 'b' agora contém os dados ordenados nos índices 1 a n.
    // Se for desejado 0-based, ajuste os acessos a b[] e talvez c[].

    free(c);
}

// Main para teste (simplificado)
// int main() {
//     int n = ...; int m = ...; // ex: m = n/2
//     int *a = malloc(n * sizeof(int));
//     int *b = malloc((n + 1) * sizeof(int)); // +1 para indexação 1-based
//     // ... inicializar a ...
//     serial_counting_sort(a, b, n, m);
//     // Imprimir b[1] até b[n]
//     free(a); free(b);
// }
```

4. Radix Sort (LSD - Binário)

Pseudocódigo

1. **for** bit = 0 **to** (num_bits - 1):
2. // — Fase de Ordenação Estável Paralela (baseada em Counting/Rank Sort) —

3. **forall processos P_k em paralelo:**
4. Determina o valor do `bit` atual para todos os elementos locais.
5. **// Passo 1: Contagem Local**
6. Conta localmente quantos elementos têm bit 0 (`count0_local`) e quantos têm bit 1 (`count1_local`).
7. **// Passo 2: Contagem Global e Offsets**
8. Realiza soma paralela (`parallel_sum`) de `count0_local` para obter `total_zeros` global.
9. Realiza soma de prefixos (`prefix_sum`) de `count0_local` para obter `offset0_local` (posição inicial para 0s locais).
10. Realiza soma de prefixos de `count1_local` para obter `offset1_local` (posição inicial para 1s locais).
11. **// Passo 3: Determinar Destinos**
12. **for** cada elemento local `a_elem`:
13. **if** `bit` de `a_elem` é 0 **then**
14. Posição destino = `offset0_local`; `offset0_local++`.
15. **else**
16. Posição destino = `total_zeros` + `offset1_local`; `offset1_local++`.
17. **end if**
18. **end for**
19. **// Passo 4: Redistribuição Global**
20. Envia cada elemento local para sua posição destino calculada (requer comunicação *all-to-all personalized*).
21. Recebe elementos destinados a este processo e os armazena localmente no array temporário.
22. **// Passo 5: Copiar de volta**
23. Copia dados do array temporário de volta para o array de dados principal local.
24. **end forall**
25. // — Fim da Ordenação Estável —
26. **end for** (loop dos bits)

Código Serial C

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h> // Para CHAR_BIT
#include <string.h> // Para memcpy, memset

// Função para obter o k-ésimo bit de um número
int get_bit(int num, int k) {
    return (num >> k) & 1;
}

// Counting sort estável para um bit específico
void counting_sort_for_bit(int a[], int b[], int n, int bit) {
    int count[2] = {0, 0};
    int *output_pos = (int *)malloc(n * sizeof(int)); // Armazena a posição final

    // 1. Contar 0s e 1s
    for (int i = 0; i < n; ++i) {
        count[get_bit(a[i], bit)]++;
    }

    // 2. Calcular posições de início (soma de prefixos simples)
    // count[0] já é a contagem de 0s.
    // count[1] será a posição inicial dos 1s (após os 0s).
    count[1] += count[0];

    // 3. Calcular posições finais e colocar em b[] (de trás para frente p/ estabilidade)
    for (int i = n - 1; i >= 0; --i) {
        int current_bit = get_bit(a[i], bit);
        b[count[current_bit] - 1] = a[i]; // -1 porque count armazena a próxima posição livre
```

```
        count[current_bit]--;
    }

    // 4. Copiar b de volta para a para a próxima iteração
    memcpy(a, b, n * sizeof(int));
    free(output_pos);
}

// Radix Sort LSD Serial
void serial_radix_sort_lsd(int a[], int n) {
    int *b = (int *)malloc(n * sizeof(int)); // Array auxiliar
    if (!b) { perror("malloc failed"); exit(1); }

    int num_bits = sizeof(int) * CHAR_BIT; // Total de bits em um int

    for (int bit = 0; bit < num_bits; ++bit) {
        counting_sort_for_bit(a, b, n, bit);
    }

    free(b);
}

// Funções main e print_array para teste (semelhantes às anteriores)
// ...
```

Atividades de Programação

Você deverá implementar as atividades de ordenação indicadas abaixo; pode seguir as dicas apresentadas, mas é livre para implementar de modo diferente. Entretanto, siga as instruções que forem requisitos obrigatórios.

Entregue um relatório (em PDF) das implementações e execuções, analisandos os tempos de execução, limitações nestas medidas (experimentos), e comentando suas estratégias de implementação.

Atividade Individual

Atividade 1: Odd-Even Transposition Sort Paralelo (Bloco)

Enunciado: Implemente o algoritmo Odd-Even Transposition Sort (versão bloco) para ordenar um array de inteiros. Sua implementação deve usar OpenMP para execução em múltiplos cores (CPU) e OpenACC para execução em GPU.

1. Crie uma função `void odd_even_sort_omp(int *data, int n, int p)` que recebe um ponteiro para o array completo `data` de tamanho `n` e o número de threads `p` a serem usados. Assuma que `n` é divisível por `p`. A função deve ordenar o array `data` *in-place* ou usando um array auxiliar temporário, se necessário. Use `omp_set_num_threads(p)` antes de chamar a função.
2. Crie uma função `void odd_even_sort_acc(int *data, int n, int p)` com os mesmos parâmetros, mas usando diretivas OpenACC. Você precisará gerenciar a cópia dos dados entre a CPU (host) e a GPU (device) usando diretivas `acc data copy(...)` ou similar. Os loops de fase e internos precisarão de diretivas `acc parallel loop` ou `acc kernels`. A comunicação entre "blocos" (partes do array na GPU) pode ser mais desafiadora em OpenACC e pode requerer cópias explícitas ou acesso direto à memória do device, se possível.
3. Implemente a lógica de `compare-split` (mesclar e manter a metade correta) como uma função auxiliar ou *inline*.
4. No seu `main`, gere um array de `n` inteiros aleatórios. Meça o tempo de execução da sua versão serial (pode usar `qsort` da `stdlib.h` ou a simulação sequencial das fases), da versão OpenMP e da versão OpenACC para diferentes valores de `n` e `p`.
5. Calcule e apresente o speedup e a eficiência para as versões paralelas em relação à versão serial (`qsort`).
6. Discuta os resultados: compare o desempenho OpenMP vs. OpenACC, analise o speedup e a eficiência obtidos, e relacione com a complexidade teórica e a otimalidade de custo ($p = O(\log n)$).

Guia para Solução

- **Estrutura Geral:** Ambas as versões terão uma fase inicial de ordenação local e um loop principal com p fases.
- **OpenMP:**
  - Use `#pragma omp parallel` para criar a equipe de threads.
  - Divida o array `data` logicamente entre os threads (cada thread calcula seu ponteiro inicial `local_data`).
  - Use `qsort` para a ordenação local inicial (cada thread no seu bloco).
  - Use `#pragma omp barrier` para sincronizar após a ordenação local e após cada fase par/ímpar.
  - Dentro do loop de fases, determine o `partner` para cada thread `tid`.
  - Simule a troca de dados acessando diretamente a memória (`all_data + partner * n_local`). Use `#pragma omp critical` ou locks se a escrita/leitura precisar ser estritamente serializada para simular message passing, ou confie na coerência de cache (mas esteja ciente de possíveis race conditions se não usar buffers privados para recebimento antes do merge). Usar buffers privados (como no código de exemplo) é mais seguro.

  - Implemente as funções `merge_split_low` e `merge_split_high`.
- **OpenACC:**
  - Use `#pragma acc data copy(data[0:n])` para copiar o array para a GPU e de volta.
  - A ordenação local inicial precisará ser feita na GPU. Isso pode ser desafiador. Uma alternativa é fazer na CPU antes de copiar para a GPU, mas isso não é o ideal para medir o desempenho completo. Uma biblioteca de ordenação na GPU (se disponível) seria ideal, ou implementar um sort paralelo na GPU (complexo). Para simplificar, *talvez* permita fazer a ordenação local na CPU antes.
  - O loop de fases principal executa na CPU.
  - Dentro de cada fase, as operações de `compare-split` precisam ser feitas na GPU. Isso envolverá kernels (`#pragma acc kernels` ou `#pragma acc parallel loop`) para realizar as comparações e trocas/merges.
  - A troca de dados entre parceiros lógicos na GPU é o ponto mais complexo. Você pode precisar de um array auxiliar na GPU e usar kernels para copiar dados entre as posições correspondentes aos "parceiros". Sincronização (`#pragma acc wait` ou barreiras implícitas) será necessária entre os passos de "envio" e "recebimento/merge".
  - Implemente as funções `merge_split_low` e `merge_split_high` como código executável na GPU (talvez com `acc routine seq` ou dentro dos kernels).
- **Medição:** Use `omp_get_wtime()` para OpenMP e as funções de tempo do OpenACC (ou `gettimeofday` cercando as regiões `acc data`) para medir o tempo. Compare com `clock()` para a versão serial.

Atividade 2: Sample Sort Paralelo (CPU e GPU)

Enunciado: Implemente o algoritmo **Sample Sort** paralelo para ordenar um array de inteiros, conforme descrito no Capítulo 9 e nos slides. Sua implementação deve usar OpenMP (multicore) e OpenACC (GPU).

1. Implemente as fases do Sample Sort: ordenação local, seleção de amostras, coleta de amostras (gather), seleção de splitters (centralizada), broadcast de splitters, particionamento local, redistribuição global (all-to-all personalized), e mesclagem/ordenação final local.
2. Crie uma função `void sample_sort_omp(int *data, int n, int p)` para a versão OpenMP. Use `p` threads. O processo 0 pode atuar como raiz para coletar amostras e distribuir splitters. Use operações coletivas simuladas ou implemente-as usando diretivas OpenMP (ex: `critical` para gather, `single` + loop para broadcast). A fase de all-to-all é a mais complexa; você pode precisar de arrays auxiliares e sincronização cuidadosa.
3. Crie uma função `void sample_sort_acc(int *data, int n, int p)` para a versão OpenACC. Gerencie as transferências host-device. A ordenação local e a mesclagem final devem ocorrer na GPU. A coleta de amostras, seleção de splitters e broadcast podem ocorrer na CPU (transferindo apenas amostras/splitters) ou, para maior desafio, na GPU (requer redução/broadcast na GPU). A redistribuição all-to-all na GPU é complexa e pode exigir múltiplos kernels e buffers intermediários.
4. No `main`, gere dados aleatórios, meça o tempo serial (`qsort`), OpenMP e OpenACC para diferentes n e p.
5. Calcule speedup e eficiência.
6. Discuta os resultados: Compare OpenMP e OpenACC. Analise a escalabilidade observada e relacione-a com os gargalos identificados (comunicação all-to-all, processamento centralizado de amostras) e a isoeficiência teórica ($O(p^2 \log p)$ ou $O(p^3 \log p)$).

Guia para Solução

- **Estrutura Geral:** Siga as fases descritas na Questão Discursiva 3.
- **OpenMP:**
  - Use `#pragma omp parallel`.
  - Ordenação local: `qsort` dentro da região paralela (cada thread no seu bloco `data + tid * n/p`).
  - Seleção de amostras: Cada thread seleciona localmente.
  - Gather: Use um array compartilhado `all_samples`. Use `#pragma omp critical` ou `atomic` para que cada thread copie suas amostras para a seção correta de `all_samples`. Sincronize com `#pragma omp barrier`.
  - Seleção/Broadcast de Splitters: Use `#pragma omp single` ou `#pragma omp master` para que apenas uma thread ordene `all_samples`, selecione os `splitters` (array compartilhado) e depois todas as threads leiam os splitters (após uma barreira implícita ou explícita).
  - Particionamento Local: Cada thread particiona seu bloco local usando os splitters. Calcule `scounts` (quantos enviar para cada outro thread) e `sdispls` (deslocamentos de envio).
  - All-to-all: Esta é a parte mais difícil. Uma simulação simples:
    - Calcule `rcounts` (quantos receber de cada thread) usando uma redução global (atomic/critical em um array `global_rcounts` e depois um `single` para calcular prefix sum) ou uma operação MPI_Alltoall simulada se estiver mais avançado.
    - Calcule `rdispls`.
    - Alocar buffer de recebimento `sorted_local_block` do tamanho total a receber.
    - Use `#pragma omp critical` (ou locks separados por destino) para que cada thread copie seus sub-blocos $B_{i,j}$ para a posição correta no `sorted_local_block` do processo destino j. Isso terá alta contenção. Uma abordagem melhor (mas complexa) é ter buffers intermediários.

  - Mesclagem Final: Use `qsort` ou um merge k-way no `sorted_local_block`.
- **OpenACC:**
  - Use `#pragma acc data copy(data)` ou `copyin/copyout`.
  - Ordenação Local: Kernel paralelo na GPU (complexo de implementar, ou use biblioteca se houver, ou faça na CPU *antes* de transferir para GPU).
  - Seleção de Amostras: Kernel para selecionar amostras na GPU e copiá-las de volta para a CPU (`copyout`).
  - Coleta/Seleção/Broadcast de Splitters: Feito na CPU com as amostras recebidas. Copie os splitters finais para a GPU (`copyin`).
  - Particionamento Local: Kernel na GPU para particionar usando os splitters. Precisa calcular `scounts` e `sdispls` na GPU (pode exigir atomics ou reduções/scans paralelos).
  - All-to-all: Extremamente desafiador na GPU sem suporte de biblioteca direta. Requer múltiplos kernels: um para cada processo i ler seus dados e escrevê-los em um buffer intermediário global (grande) na GPU, indexado pelo destino j; outro kernel para cada processo j ler os dados destinados a ele a partir do buffer intermediário e colocá-los em seu array final. Requer gerenciamento cuidadoso de memória e sincronização.
  - Mesclagem Final: Kernel de merge k-way na GPU (complexo) ou use um sort paralelo final no bloco recebido.
- **Medição/Análise:** Similar à Atividade 1.