

## **Exercícios Preparatórios para Maratonas de Programação Paralela**

### **Sumário**

<b>1</b>	<b>Instruções gerais</b>	<b>2</b>
<b>2</b>	<b>Programação para sistemas de memória compartilhada com OpenMP</b>	<b>3</b>
<b>3</b>	<b>Programação para sistemas de memória distribuída com MPI</b>	<b>7</b>
<b>4</b>	<b>Programação para GPU com CUDA</b>	<b>9</b>

# 1 Instruções gerais

- Busque a solução com menor tempo de execução
- Incluir os arquivos cmake e shell script para executar em um supercomputador com SLURM

## 2 Programação para sistemas de memória compartilhada com OpenMP

1. [Pacheco and Malensek, 2022] Suponha que lançamos dardos aleatoriamente em um alvo quadrado. Vamos considerar o centro desse alvo como sendo a origem de um plano cartesiano e os lados do alvo medem 2 pés de comprimento. Suponha também que haja um círculo inscrito no alvo. O raio do círculo é 1 pé e sua área é  $\pi$  pés quadrados. Se os pontos atingidos pelos dardos estiverem distribuídos uniformemente (e sempre acertamos o alvo), então o número de dardos atingidos dentro do círculo deve satisfazer aproximadamente a equação

$$\frac{qtd\_no\_circulo}{num\_lancamentos} = \frac{\pi}{4} \quad (1)$$

já que a razão entre a área do círculo e a área do quadrado é  $\frac{\pi}{4}$ .

Podemos usar esta fórmula para estimar o valor de  $\pi$  com um gerador de números aleatórios:

```
qtd_no_circulo = 0;
for (lancamento = 0; lancamento < num_lancamentos; lancamento++) {
    x = double aleatório entre -1 e 1;
    y = double aleatório entre -1 e 1;
    distancia_quadrada = x * x + y * y;
    if (distancia_quadrada <= 1) qtd_no_circulo++;
}
estimativa_de_pi = 4 * qtd_no_circulo/((double) num_lancamentos);
```

Isso é chamado de método "Monte Carlo", pois utiliza aleatoriedade (o lançamento do dardo).

Escreva um programa OpenMP que use um método de Monte Carlo para estimar  $\pi$ . Leia o número total de lançamentos antes de criar as *threads*. Use uma cláusula de *reduction* para encontrar o número total de dardos que atingem o círculo. Imprima o resultado após encerrar a região paralela. Você deve usar `long long ints` para o número de acertos no círculo e o número de lançamentos, já que ambos podem ter que ser muito grandes para obter uma estimativa razoável de  $\pi$ .

2. [Pacheco and Malensek, 2022] *Count sort* é um algoritmo de ordenação serial simples que pode ser implementado da seguinte forma:

```
void Count_sort(int a[], int n) {
    int i, j, count;
    int* temp = malloc(n*sizeof(int));
    for (i = 0; i < n; i++) {
        count = 0;
        for (j = 0; j < n; j++)
            if (a[j] < a[i])
                count++;
    }
}
```

```

        else if (a[j] == a[i] && j < i)
            count++;
        temp[count] = a[i];
    }
    memcpy(a, temp, n*sizeof(int));
    free(temp);
}

```

A ideia básica é que para cada elemento  $a[i]$  na lista  $a$ , contemos o número de elementos da lista que são menores que  $a[i]$ . Em seguida, inserimos  $a[i]$  em uma lista temporária usando o índice determinado pela contagem. Há um pequeno problema com esta abordagem quando a lista contém elementos iguais, uma vez que eles podem ser atribuídos ao mesmo slot na lista temporária. O código lida com isso incrementando a contagem de elementos iguais com base nos índices. Se  $a[i] == a[j]$  e  $j < i$ , então contamos  $a[j]$  como sendo "menor que"  $a[i]$ .

Após a conclusão do algoritmo, sobrescrevemos o *array* original pelo *array* temporário usando a função da biblioteca de *strings* `memcpy`.

- Se tentarmos paralelizar o laço `for i` (o laço externo), quais variáveis devem ser privadas e quais devem ser compartilhadas?
- Se paralelizarmos o laço `for i` usando o escopo especificado na parte anterior, haverá alguma dependência de dados no laço? Explique sua resposta.
- Podemos paralelizar a chamada para `memcpy`? Podemos modificar o código para que esta parte da função seja paralelizável?
- Escreva um programa em C que inclua uma implementação paralela do *Count sort*.
- Como o desempenho da sua paralelização do *Count sort* se compara à classificação serial? Como ela se compara à função serial `qsort`?

3. [Pacheco and Malensek, 2022] Lembre-se de que quando resolvemos um grande sistema linear, frequentemente usamos a eliminação gaussiana seguida de substituição regressiva. A eliminação gaussiana converte um sistema linear  $n \times n$  em um sistema linear triangular superior usando "operações de linha".

- Adicione um múltiplo de uma linha a outra linha
- Troque duas linhas
- Multiplique uma linha por uma constante diferente de zero

Um sistema triangular superior tem zeros abaixo da "diagonal" que se estende do canto superior esquerdo ao canto inferior direito. Por exemplo, o sistema linear

$$\begin{aligned}
 2x_0 - 3x_1 &= 3 \\
 4x_0 - 5x_1 + x_2 &= 7 \\
 2x_0 - x_1 - 3x_2 &= 5
 \end{aligned}$$

pode ser reduzido à forma triangular superior

$$2x_0 - 3x_1 = 3$$

$$x_1 + x_2 = 1$$

$$-5x_2 = 0$$

e este sistema pode ser facilmente resolvido encontrando primeiro  $x_2$  usando a última equação, depois encontrando  $x_1$  usando a segunda equação e finalmente encontrando  $x_0$  usando a primeira equação.

Podemos desenvolver alguns algoritmos seriais para substituição reversa. A versão "orientada a linhas" é

```
for (lin = n-1; lin >= 0; lin--) {
    x[lin] = b[lin];
    for (col = lin+1; col < n; col++)
        x[lin] -= A[lin][col]*x[col];
    x[lin] /= A[lin][lin];
}
```

Aqui, o "lado direito" do sistema é armazenado na matriz  $b$ , a matriz bidimensional de coeficientes é armazenada na matriz  $A$  e as soluções são armazenadas na matriz  $x$ . Uma alternativa é o seguinte algoritmo "orientado a colunas":

```
for (lin = 0; lin < n; lin++)
    x[lin] = b[lin];
for (col = n-1; col >= 0; col--) {
    x[col] /= A[col][col];
    for (lin = 0; lin < col; lin++)
        x[lin] -= A[lin][col]*x[col];
}
```

- Determine se o laço externo do algoritmo orientado a linhas pode ser paralelizado.
- Determine se o laço interno do algoritmo orientado a linhas pode ser paralelizado.
- Determine se o (segundo) laço externo do algoritmo orientado a colunas pode ser paralelizado.
- Determine se o laço interno do algoritmo orientado a colunas pode ser paralelizado.
- Escreva um programa OpenMP para cada um dos loops que você determinou que poderiam ser paralelizados. Você pode achar a diretiva `single` útil - quando um bloco de código está sendo executado em paralelo e um sub-bloco deve ser executado por apenas uma *thread*, o sub-bloco pode ser modificado por uma diretiva `#pragma omp single`. As *threads* serão bloqueadas no final da diretiva até que todas as *threads* a tenha concluído.
- Modifique seu laço paralelo com uma cláusula `schedule(runtime)` e teste o programa com vários escalonamentos. Se o seu sistema triangular superior tiver 10.000 variáveis, qual escalonamento oferece o melhor desempenho?

4. [Pacheco and Malensek, 2022] Use OpenMP para implementar um programa que faça eliminação gaussiana (veja o problema anterior). Você pode assumir que o sistema de entrada não precisa de nenhuma troca de linha.
5. [Pacheco and Malensek, 2022] Use OpenMP para implementar um programa produtor-consumidor no qual algumas *threads* são produtoras e outras são consumidoras. As produtoras leem o texto de uma coleção de arquivos, um por produtor. Elas inserem linhas de texto em uma única fila compartilhada. Os consumidores pegam as linhas do texto e as tokenizam. *Tokens* são "palavras" separadas por espaço em branco. Quando uma consumidora encontra um *token*, ela o grava no `stdout`.

### 3 Programação para sistemas de memória distribuída com MPI

6. [Pacheco and Malensek, 2022] Use o MPI para implementar o programa de histograma discutido na Seção 2.7.1. Faça com que o processo 0 leia os dados de entrada e os distribua entre os processos. Faça também que o processo 0 imprima o histograma.
7. [Pacheco and Malensek, 2022] Escreva um programa MPI que use um método de Monte Carlo para estimar  $\pi$  (Seção 2, questão 1). O processo 0 deve ler o número total de lançamentos e transmiti-lo aos outros processos. Use o `MPI_Reduce` para encontrar a soma global da variável local `qtd_no_circulo` e peça ao processo 0 para imprimir o resultado.
8. [Pacheco and Malensek, 2022] Escreva um programa MPI que calcule uma soma global estruturada em árvore. Primeiro, escreva seu programa para o caso especial em que `comm_sz` é uma potência de dois. Depois que esta versão estiver funcionando, modifique seu programa para que ele possa lidar com qualquer `comm_sz`.
9. [Pacheco and Malensek, 2022] Escreva um programa MPI que calcule uma soma global usando uma borboleta. Primeiro, escreva seu programa para o caso especial em que `comm_sz` é uma potência de dois. Modifique seu programa para que ele lide com qualquer número de processos.
10. [Pacheco and Malensek, 2022] Um *merge sort* paralelo começa com  $n/\text{comm\_sz}$  chaves atribuídas a cada processo. Ele termina com todas as chaves armazenadas no processo 0 em ordem de classificação. Para conseguir isso, ele usa a mesma comunicação estruturada em árvore que usamos para implementar uma soma global. No entanto, quando um processo recebe as chaves de outro processo, ele mescla as novas chaves em sua lista de chaves já ordenadas. Escreva um programa que implemente o *merge sort* paralelo. O processo 0 deve ler  $n$  e transmiti-lo para os outros processos. Cada processo deve usar um gerador de números aleatórios para criar uma lista local de  $n/\text{comm\_sz}$  inteiros. Cada processo deve então classificar sua lista local e o processo 0 deve reunir e imprimir as listas locais. Em seguida, os processos devem usar a comunicação estruturada em árvore para mesclar a lista global no processo 0, que imprime o resultado.
11. [Pacheco and Malensek, 2022] Escreva um programa que possa ser usado para determinar o custo de alterar a distribuição de uma estrutura de dados distribuída. Quanto tempo leva para mudar de uma distribuição em bloco de um vetor para uma distribuição cíclica? Quanto tempo leva a redistribuição reversa?
12. [Silva et al., 2022] Escreva um programa para calcular o produto escalar de dois vetores. Utilize rotinas `MPI_Send` e `MPI_Recv` para comunicação entre os processos. Considere cada vetor com  $N$  posições e divida a operação entre  $P$  processos distintos. Considere que a divisão de  $N$  por  $P$  não tem resto.
13. [Silva et al., 2022] Considere um anel com `num_procs` processos. Escreva um programa utilizando rotinas `MPI_Send` e `MPI_Recv` para comunicação entre os processos que faça circular uma mensagem contendo um inteiro positivo ao longo desse canal. O processo com ranque igual a 0 é quem inicia a transmissão e cada vez que a mensagem passa por

ele novamente o valor contido na mensagem deve ser decrementado de um até chegar ao valor 0. Quando um processo receber a mensagem com valor 0 ele deverá passá-la adiante e então terminar a sua execução.

14. [Silva et al., 2022] Escreva uma rotina para calcular o produto escalar de dois vetores, utilizando rotinas de comunicação coletiva do MPI para envio do vetor inicial e recepção dos resultados parciais pelo processo raiz. Considere cada vetor com  $N$  posições e divida a operação entre  $P$  processos distintos. Considere que a divisão de  $N$  por  $P$  não tem resto. Assuma que o valor de  $N$  não é conhecido e deve ser enviado pelo processo raiz para os demais processos com uso de uma rotina comunicação coletiva.
15. [Silva et al., 2022] A Conjectura de Collatz foi desenvolvida pelo matemático alemão Lothar Collatz. Nela, escolhendo-se um número natural inicial  $N$ , onde  $N > 0$ , os seguintes critérios serão obedecidos: Se  $N$  for par o seu sucessor será a metade e se  $N$  for ímpar o seu sucessor será o triplo mais um, gerando então um novo número. Esse processo repete-se até que eventualmente se atinja o número 1.

Use o modelo “saco de tarefa” para resolver esse problema em paralelo e verificar se os números inteiros no intervalo de 1 a 100.000.000 atendem a esses critérios e o maior número de passos necessários para chegar até o número 1. Atribua a um dos processos o papel de “mestre” (gerencia a distribuição de tarefas) e aos demais processos o papel de “trabalhadores” (executam a tarefa de avaliar se um número está dentro da conjectura). Utilize mensagens de envio e recepção não bloqueantes de forma a paralelizar a computação com o envio de mensagens.



## 4 Programação para GPU com CUDA

16. [Pacheco and Malensek, 2022] Em um algoritmo de regra trapezoidal, se houver um total de  $t$  *threads*,  $n$  trapézios e  $n$  for divisível por  $t$ , podemos atribuir  $n/t$  trapézios a cada *thread*. Escreva um programa que implemente corretamente a regra trapezoidal quando o número de trapézios for um múltiplo do número de *threads*.
17. [Pacheco and Malensek, 2022] Implemente uma ordenação bitônica na qual cada *thread* é responsável por dois blocos de elementos. Se o *array* tiver  $n$  elementos e houver  $\text{blk\_ct}$  blocos de *threads* e  $\text{th\_per\_blk}$  *threads* por bloco, considere que o número total de *threads* é uma potência de dois e que  $n$  é divisível pelo número de *threads*. Assim

$$\text{chunk\_sz} = \frac{n}{\text{blk\_ct} \times \text{th\_per\_blk}} \quad (2)$$

é um inteiro.

Cada *thread* é responsável por uma sublista contígua de  $\text{chunk\_sz}$  elementos, e cada *thread* inicialmente classificará sua sublista em ordem crescente. Então, se as *threads*  $t$  e  $u$  forem pareadas para uma divisão e mesclagem,  $t < u$ , e  $t$  e  $u$  estiverem trabalhando em uma sequência crescente, elas mesclarão suas sublistsas em uma sequência crescente, com  $t$  mantendo a metade inferior e  $u$  mantendo a metade superior. Se estiverem trabalhando em uma sequência decrescente,  $t$  manterá a metade superior e  $u$  manterá a metade inferior. Portanto, após cada divisão e mesclagem, cada *thread* sempre terá uma sublista crescente.

Primeiro implemente a ordenação bitônica usando um único bloco de *threads*. Em seguida, modifique o programa para que ele possa lidar com um número arbitrário de blocos de *threads*.

18. Implemente uma soma de duas matrizes que utilize todas GPUs disponíveis em nó. Este exercício deve ser executado em uma máquina com ao menos duas GPUs.

## Referências

- Peter S. Pacheco and Matthew Malensek. *An Introduction to Parallel Programming*. Elsevier, 2 edition, 2022. ISBN 9780128046050. doi: 10.1016/C2015-0-01650-1. URL <https://linkinghub.elsevier.com/retrieve/pii/C20150016501>.
- G. P. Silva, C. P. Bianchini, and E. B. Costa. *Programação paralela e distribuída: com MPI, OpenMP e OpenACC para computação de alto desempenho*. Casa do Código, 2022.