



Universidade do Vale do Itajaí

**Curso:** Bacharelado em Ciência da Computação

**Professor:** Felipe Viel

**Disciplina:** Sistemas Operacionais

## **Avaliação 02 – Threads e Paralelismo**

**Acadêmicos:** Alexandre Machado de Azevedo,

Matheus Passold Carelli,

Vinícius dos Santos Moreira.

Itajaí, Setembro de 2023

## Projeto 1

Realize uma implementação em sua linguagem de preferência de uma multiplicação entre matrizes utilizando o sistema single thread e multithread (pelo menos duas threads), no qual o último deve ser feito usando as bibliotecas thread suportada na linguagem escolhida. Realize uma análise comparativa no quesito tempo de processamento utilizando bibliotecas como time.h (como o exemplo fornecido no material ou biblioteca equivalente na linguagem escolhida). A operação de multiplicação deve usar duas abordagens, a multiplicação matricial e a posicional, e deve ser entre, no mínimo, matrizes quadráticas de 3X3, como no exemplo apresentado e os números deve estar em float (ponto flutuante): Matrizes a serem multiplicadas (exemplo):

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad e \quad B = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

Multiplicação matricial:

$$AB = \begin{bmatrix} 1 \times 9 & 2 \times 8 & 3 \times 7 \\ 4 \times 9 & 5 \times 8 & 6 \times 7 \\ 7 \times 9 & 8 \times 8 & 9 \times 7 \end{bmatrix}$$

Multiplicação posicional:

$$A@B = \begin{bmatrix} 1 \times 9 & 2 \times 8 & 3 \times 7 \\ 4 \times 6 & 5 \times 5 & 6 \times 4 \\ 7 \times 3 & 8 \times 2 & 9 \times 1 \end{bmatrix}$$

Responda: Você conseguiu notar a diferença de processamento? O processamento (multiplicação) foi mais rápido com a implementação single thread ou multithread? Explique os resultados obtidos. Você é livre para implementar estratégias diferentes para conseguir processar bem como usar recursos de aceleração em hardware das bibliotecas.

### Resolução projeto 1:

Este código é um programa em C que realiza multiplicação de matrizes de duas maneiras diferentes: multiplicação matricial e multiplicação posicional. Ele pode executar estas operações de forma única ou simultânea (usando múltiplas threads) para acelerar o processo, especialmente para matrizes grandes. O programa também permite ao usuário definir o tamanho da matriz através da linha de comando e mede o tempo que leva para completar as operações de multiplicação, oferecendo uma visão clara da eficiência do processo multithread em comparação com o single thread.

## Resultados:

As simulações apresentadas aqui foram executadas em um notebook com Windows 11 utilizando o processador Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz.

Simulação na qual o multithreads(4 threads) foi mais eficiente que o single thread:

```
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500
Tempo gasto em uma matriz(500x500) com multiThread(Desativado): 0.944398 segundos, 0.944390 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500
Tempo gasto em uma matriz(500x500) com multiThread(Desativado): 0.995428 segundos, 0.995437 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500
Tempo gasto em uma matriz(500x500) com multiThread(Desativado): 0.958754 segundos, 0.958774 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500
Tempo gasto em uma matriz(500x500) com multiThread(Desativado): 0.955029 segundos, 0.955046 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500
Tempo gasto em uma matriz(500x500) com multiThread(Desativado): 0.946209 segundos, 0.946218 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500 multithread
Tempo gasto em uma matriz(500x500) com multiThread(Ativo): 0.470270 segundos, 1.379433 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500 multithread
Tempo gasto em uma matriz(500x500) com multiThread(Ativo): 0.366334 segundos, 1.185667 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500 multithread
Tempo gasto em uma matriz(500x500) com multiThread(Ativo): 0.365849 segundos, 1.187423 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500 multithread
Tempo gasto em uma matriz(500x500) com multiThread(Ativo): 0.364913 segundos, 1.188475 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$ ./projeto1 matriz=500 multithread
Tempo gasto em uma matriz(500x500) com multiThread(Ativo): 0.428306 segundos, 1.567698 segundos clock.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1$
```

Single thread com matriz 500x500, média 0.9599636 segundos:

0.944398 segundos  
0.995428 segundos  
0.958754 segundos  
0.955029 segundos  
0.946209 segundos

Multithreads com matriz 500x500, média 0.39913 segundos:

0.470270 segundos  
0.366334 segundos  
0.365849 segundos  
0.364913 segundos  
0.428306 segundos

## Códigos:

Nesse trecho abaixo cada operação está sendo realizada em uma variável da matriz diferente para não ocorrer contenção de recursos por estarem acessando a mesma memória. Também podemos notar que no multithread é passado o parâmetro para identificar qual parte da matriz deve ser executada. Podemos ver também que não foi passado o parâmetro usando o tipo dele, pois na linguagem C o pthread\_create, deve ser capaz de aceitar um único argumento do tipo void \* e retornar um void \*. A razão para aceitar um argumento void \* é para proporcionar flexibilidade. void \* é um ponteiro para um tipo de dado não especificado, o que significa que você pode passar um ponteiro para qualquer tipo de dado para a sua função de thread.

```

1 void *multiplicacao_matricial_multithread(void *param)
2 {
3     Parametros *p = (Parametros *)param;
4     for (int i = p->inicio; i < p->fim; i++)
5     {
6         for (int j = 0; j < tamanhoMatriz; j++)
7         {
8             resultado_matricial[i][j] = A[i][j] * B[i][j];
9         }
10    }
11    return NULL;
12 }
13
14 void *multiplicacao_posicional_multithread(void *param)
15 {
16     Parametros *p = (Parametros *)param;
17     for (int i = p->inicio; i < p->fim; i++)
18     {
19         for (int j = 0; j < tamanhoMatriz_posicional; j++)
20         {
21             resultado_posicional[i][j] = 0;
22             for (int k = 0; k < tamanhoMatriz_posicional; k++)
23             {
24                 resultado_posicional[i][j] += A_posicional[i][k] * B_posicional[k][j];
25             }
26         }
27     }
28    return NULL;
29 }

```

Já no trecho abaixo, fizemos a inicialização do clock antes de qualquer das operações específicas do multiThread ou single thread para ser justo com cada execução, uma parte interessante é que utilizamos dois clock, no clock() tradicional reparamos uma falta de precisão ao utilizar multiThread por isso o uso adicional do clock\_gettime().

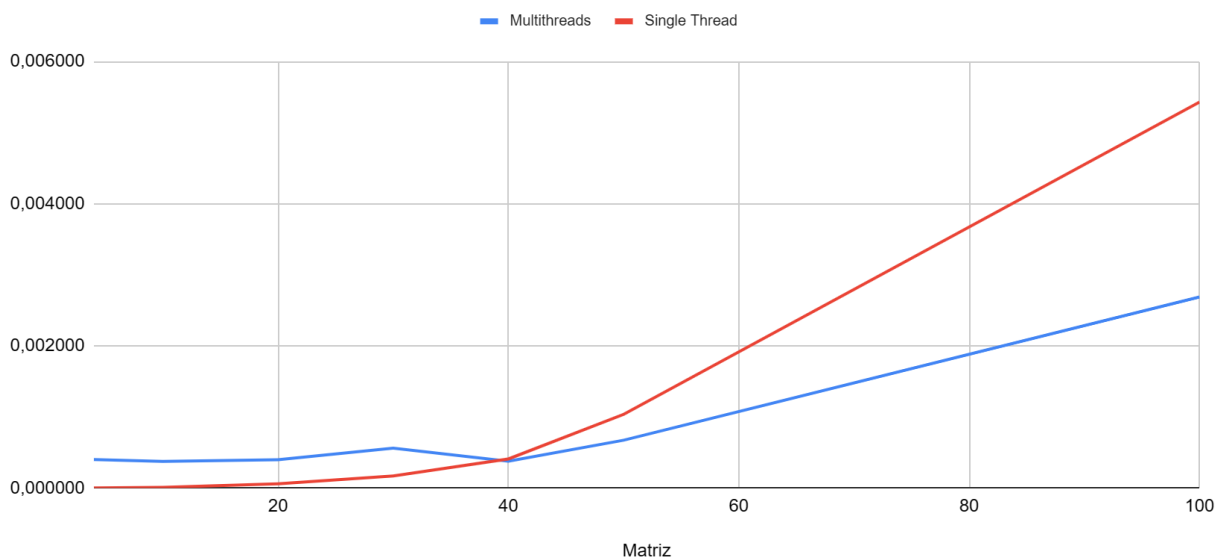
```

1  inicio_clock = clock();
2  clock_gettime(CLOCK_REALTIME, &inicio);
3
4  if (multiThread)
5  {
6      pthread_t thread1, thread2, thread3, thread4;
7      Parametros p1, p2, p3, p4;
8
9      p1.inicio = 0;
10     p1.fim = tamanhoMatriz / 4;
11
12     p2.inicio = tamanhoMatriz / 4;
13     p2.fim = (tamanhoMatriz / 4) * 2;
14
15     p3.inicio = (tamanhoMatriz / 4) * 2;
16     p3.fim = (tamanhoMatriz / 4) * 3;
17
18     p4.inicio = (tamanhoMatriz / 4) * 3;
19     p4.fim = tamanhoMatriz;
20
21     duplicarMatrizes();
22
23     pthread_create(&thread1, NULL, multiplicacao_multithread, &p1);
24     pthread_create(&thread2, NULL, multiplicacao_multithread, &p2);
25     pthread_create(&thread3, NULL, multiplicacao_multithread, &p3);
26     pthread_create(&thread4, NULL, multiplicacao_multithread, &p4);
27
28     pthread_join(thread1, NULL);
29     pthread_join(thread2, NULL);
30     pthread_join(thread3, NULL);
31     pthread_join(thread4, NULL);
32 }
33 else
34 {
35     multiplicacao_matricial();
36     multiplicacao_posicional();
37 }
38
39 fim_clock = clock();
40 clock_gettime(CLOCK_REALTIME, &fim);

```

## Gráfico:

Multithread(4 threads) e Single Thread (Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz)



## Resultados finais:

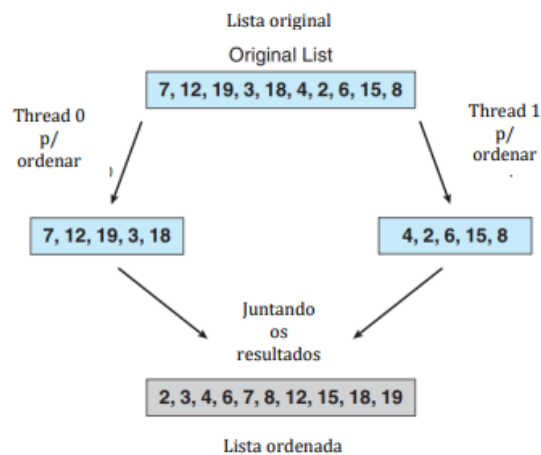
Nos primeiros testes, com matrizes menores de tamanho 4 e 10, observamos que o método Single thread, onde a operação é realizada de uma só vez, se mostrou mais eficiente, apresentando tempos mais curtos. Isso mostra que para operações mais simples, a eficiência está justamente na simplicidade, sem a necessidade de criar múltiplos processos simultâneos, o que pode acabar gerando um overhead desnecessário.

Conforme aumentamos o tamanho das matrizes, a partir do tamanho 40, por exemplo, começamos a ver uma inversão na velocidade. O método Multithread, que divide a tarefa em várias threads rodando ao mesmo tempo, começa a mostrar sua velocidade. Esse método foi claramente mais rápido ao lidar com matrizes maiores, demonstrando que, para tarefas mais complexas, a distribuição do trabalho pode realmente acelerar as coisas.

Isso nos leva a concluir que enquanto o método Single thread é mais eficiente para matrizes menores, evitando o overhead de criar várias threads, o método Multithread torna-se mais eficiente conforme a complexidade da tarefa aumenta, otimizando significativamente o tempo de processamento.

## Projeto 2

Realize uma implementação em sua linguagem de preferência de algoritmo de ordenação de vetores Bubble sort. O vetor deverá ser de pelo menos 300 posições e deverá ser comparado um sistema singlethread com um sistema multithread (com pelo menos 2 threads). Além disso, a ordenação deverá ser na ordem crescente (do maior para o menor) e o vetor de valores deve ser iniciado do maior para o menor (descendente) não importando o valor das posições, desde que respeita essa regra. Isso gerará o pior caso de uso do bubble sort. Exemplo do bubble sort e outros algoritmos: exemplos.



Responda: Você conseguiu notar a diferença de processamento? O processamento foi mais rápido com a implementação single thread ou multithread? Explique os resultados obtidos. Você é livre para implementar estratégias diferentes para conseguir processar bem como usar recursos de aceleração em hardware das bibliotecas.

## Resolução projeto 2:

Neste programa em C, o objetivo principal é ordenar uma sequência de números em ponto flutuante utilizando o método Bubble Sort, que é um simples algoritmo de ordenação. Primeiramente, o usuário tem a liberdade de determinar o tamanho do array e o método de preenchimento do mesmo, podendo ser uma sequência em ordem decrescente, valores aleatórios ou até mesmo um conjunto fixo de números.

Uma particularidade do código é a sua capacidade de realizar a ordenação utilizando duas threads em paralelo, uma técnica chamada multithreading. Neste processo, o array é dividido em duas partes iguais, sendo cada uma ordenada de forma independente e simultânea, para posteriormente serem ordenadas conjuntamente, garantindo assim a ordenação correta do array inteiro.

Durante a execução do programa, é calculado o tempo gasto para realizar a ordenação, fornecendo assim uma métrica de desempenho. Além disso, o programa oferece a opção de visualizar o array antes e depois da ordenação, permitindo verificar a eficácia do algoritmo. Assim, o programa não apenas realiza uma ordenação eficiente através do uso do algoritmo Bubble Sort e do multithreading, mas também permite uma análise detalhada do desempenho e correção do processo de ordenação.

## Resultados:

As simulações apresentadas aqui foram executadas em um notebook com Windows 11 utilizando o processador Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz.

Simulação na qual o multithreads(2 threads) foi mais eficiente que o single thread:

```
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000
Tempo gasto: 0.077627
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000
Tempo gasto: 0.077073
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000
Tempo gasto: 0.078060
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000
Tempo gasto: 0.080947
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000
Tempo gasto: 0.097834
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000 multithread
Tempo gasto: 0.059757
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000 multithread
Tempo gasto: 0.059225
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000 multithread
Tempo gasto: 0.080915
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000 multithread
Tempo gasto: 0.078619
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_2$ ./projeto2 array=5000 multithread
Tempo gasto: 0.081469
```

Single thread com array 5000, média 0.082308 segundos:

0.077627 segundos  
0.077073 segundos  
0.078060 segundos  
0.080947 segundos  
0.097834 segundos

Multithreads com matriz 5000, média 0.071997 segundos:

0.059757 segundos  
0.059225 segundos  
0.080915 segundos  
0.078619 segundos  
0.081469 segundos

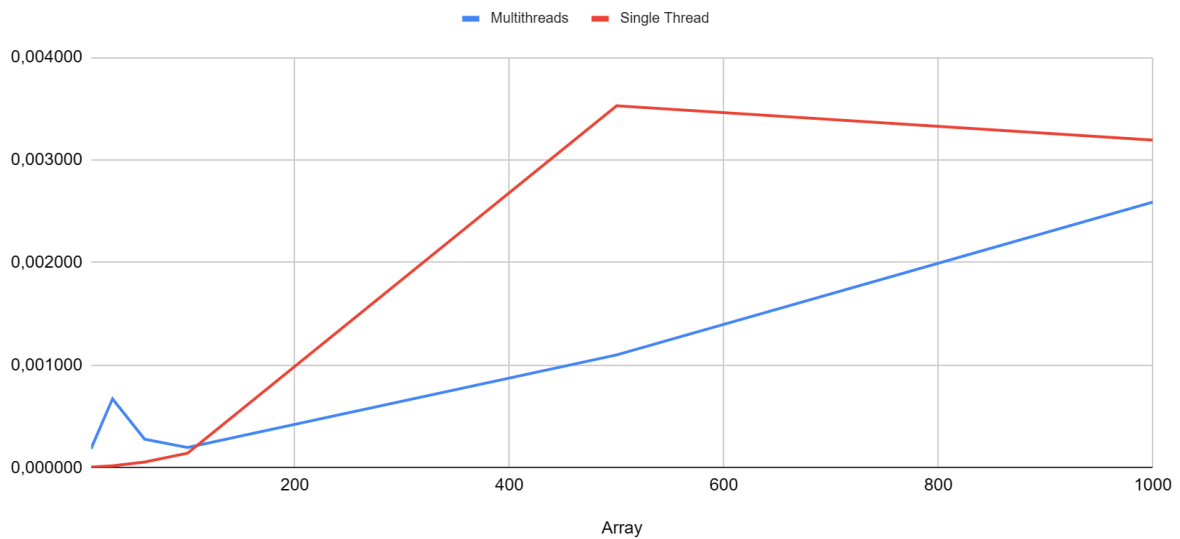


## Códigos:

```
1 // Iniciar o timer
2 inicio_clock = clock();
3 clock_gettime(CLOCK_REALTIME, &inicio);
4
5 if (multiThread)
6 {
7     pthread_t thread1, thread2;
8     Parametros p1, p2;
9     pthread_attr_t attr; /* set of attributes for the thread */
10
11     pthread_attr_init(&attr);
12
13     int meio = tamanhoGlobal / 2;
14     p1.array = meuArray;
15     p1.inicio = 0;
16     p1.fim = meio;
17
18     p2.array = meuArray;
19     p2.inicio = meio;
20     p2.fim = tamanhoGlobal;
21
22     pthread_create(&thread1, &attr, bubbleSort_multithread, &p1);
23     pthread_create(&thread2, &attr, bubbleSort_multithread, &p2);
24
25     pthread_join(thread1, NULL);
26     pthread_join(thread2, NULL);
27
28     if (isPrintArray)
29     {
30         printf("Array antes do merge:\n");
31         printArray(meuArray, tamanhoGlobal);
32     }
33     // bubbleSort(meuArray, meio - 1, meio + 1);
34     bubbleSort(meuArray, 0, tamanhoGlobal);
35 }
36 else
37 {
38     bubbleSort(meuArray, 0, tamanhoGlobal);
39 }
```

## Gráfico:

Multithread(4 threads) e Single Thread (Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz)



## Resultados finais:

O código fornecido demonstra a implementação de um algoritmo de ordenação bubble sort, em versões singlethread e multithread, visando comparar os desempenhos de ambos em diferentes tamanhos de arrays. Inicialmente, é evidente que a versão singlethread demonstra uma eficiência considerável para arrays menores, apresentando tempos de execução bastante reduzidos. No entanto, à medida que o tamanho do array cresce, o tempo de execução aumenta consideravelmente, destacando uma limitação significativa deste método, especialmente para conjuntos de dados maiores.

A versão multithread, por sua vez, apesar de apresentar um tempo de execução um pouco maior para arrays menores, provavelmente devido ao overhead inicial de configuração das threads, mostra-se significativamente mais eficiente para arrays de maior tamanho, ilustrando o potencial do processamento paralelo em melhorar o desempenho da ordenação.

Em resumo, a escolha da abordagem de ordenação pode depender muito do tamanho do conjunto de dados em questão. A abordagem singlethread pode ser mais indicada para conjuntos de dados menores, devido à sua simplicidade e menor overhead, enquanto a abordagem multithread surge como uma opção mais viável para conjuntos de dados maiores, proporcionando uma melhoria notável no tempo de execução e, consequentemente, uma eficiência geral aprimorada.

### Projeto 3

A fim de estimular a ampliação dos seus conhecimentos sobre paralelismo, concorrência e IPC em Sistemas Operacionais, será concedido de 0,5 à 1,5 ponto extra na nota da prova para os trabalhos que apresentarem uma implementação de um dos dois projetos usando a abordagem de comunicação de processos, podendo ser memória compartilhada ou troca de mensagens por pipe. Como é uma pontuação extra, não será descontado caso não realize essa implementação. Além disso, o professor, no momento da defesa não irá solicitar a apresentação, ficando a cargo dos alunos indicarem se implementaram e mostrar o funcionamento.

- Explicação e contexto da aplicação para compreensão do problema tratado pela solução.
- Resultados obtidos com as simulações.
- Códigos importantes da implementação.
- Resultados obtidos com a implementação (tabelas, gráficos e etc).
- Análise e discussão sobre os resultados finais.

Dicas para os projetos: Observe que é exigido pelo menos duas threads (excluindo a que cria as threads), mas você pode utilizar mais. Além disso, você também tem que mensurar o custo de criação da thread, já que a comparação deverá ser justa com um código original. Outra observação é que você provavelmente não irá capturar o mesmo tempo entre diferentes testes, logo, repita pelo menos 5 vezes o processamento e você tem duas opções: (1) exibir a média ou (2) exibir o maior tempo de processamento. Além disso, você pode utilizar algoritmos prontos da literatura, onde o desafio do trabalho é a paralelização da operação. Para a implementação do código, você pode fazer um fork do repositório da disciplina no github e usar o codespaces do github para a implementação, onde o mesmo irá executar o Visual Code em uma distro Linux Ubuntu com 2 núcleos e 8 GB de memória principal. Mas onde será executado seus códigos fica a critério do(s) aluno(s)