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:

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

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:

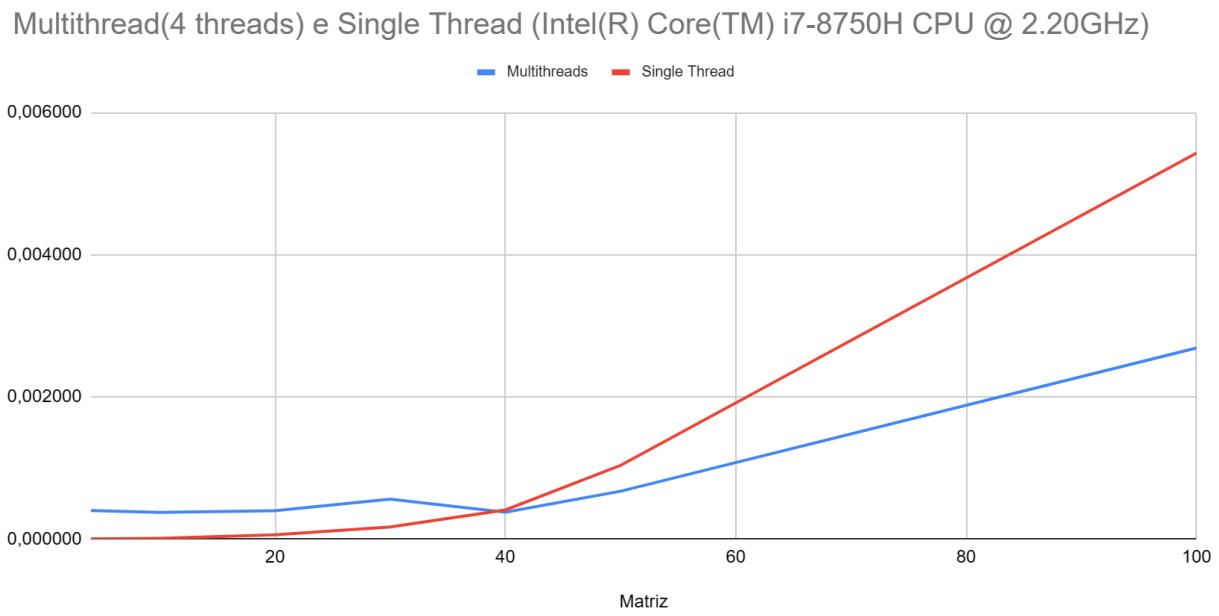
Nesse trecho abaixo cada operação está sendo realizada em uma variável da matriz diferente para não ocorrer contenção de recursos por estarem acessando a mesma memória. Também podemos notar que no multithread é passado o parâmetro para identificar qual parte da matriz deve ser executada. Podemos ver 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.

```c
void *multiplicacao_matricial_multithread(void *param)
{
    Parametros *p = (Parametros *)param;
    for (int i = p->inicio; i < p->fim; i++)
    {
        for (int j = 0; j < tamanhoMatriz; j++)
        {
            resultado_matricial[i][j] = A[i][j] * B[i][j];
        }
    }
    return NULL;
}

void *multiplicacao_posicional_multithread(void *param)
{
    Parametros *p = (Parametros *)param;
    for (int i = p->inicio; i < p->fim; i++)
    {
        for (int j = 0; j < tamanhoMatriz_posicional; j++)
        {
            resultado_posicional[i][j] = 0;
            for (int k = 0; k < tamanhoMatriz_posicional; k++)
            {
                resultado_posicional[i][j] += A_posicional[i][k] * B_posicional[k][j];
            }
        }
    }
    return NULL;
}
```

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().

```c
inicio_clock = clock();
clock_gettime(CLOCK_REALTIME, &inicio);

if (multiThread)
{
    pthread_t thread1, thread2, thread3, thread4;
    Parametros p1, p2, p3, p4;

    p1.inicio = 0;
    p1.fim = tamanhoMatriz / 4;

    p2.inicio = tamanhoMatriz / 4;
    p2.fim = (tamanhoMatriz / 4) * 2;

    p3.inicio = (tamanhoMatriz / 4) * 2;
    p3.fim = (tamanhoMatriz / 4) * 3;

    p4.inicio = (tamanhoMatriz / 4) * 3;
    p4.fim = tamanhoMatriz;

    duplicarMatrizes();

    pthread_create(&thread1, NULL, multiplicacao_multithread, &p1);
    pthread_create(&thread2, NULL, multiplicacao_multithread, &p2);
    pthread_create(&thread3, NULL, multiplicacao_multithread, &p3);
    pthread_create(&thread4, NULL, multiplicacao_multithread, &p4);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    pthread_join(thread4, NULL);
}
else
{
    multiplicacao_matricial();
    multiplicacao_posicional();
}

fim_clock = clock();
clock_gettime(CLOCK_REALTIME, &fim);
```

Gráfico:



Resultados finais:

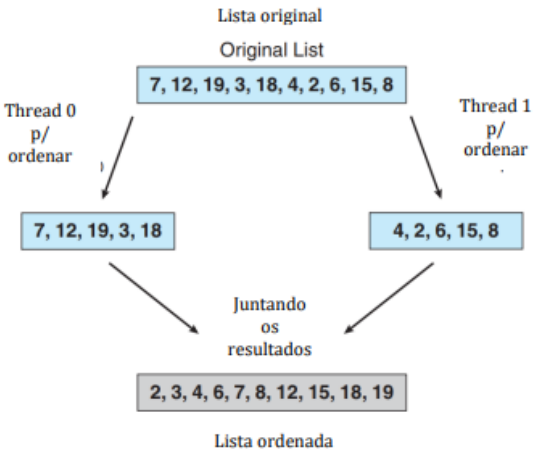
Nos primeiros testes, com matrizes menores de tamanho 4 e 10, observamos que o método Single thread, onde a operação é realizada de uma só vez, se mostrou mais eficiente, apresentando tempos mais curtos. Isso mostra que para operações mais simples, a eficiência está justamente na simplicidade, sem a 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 (descrente) 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. 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.

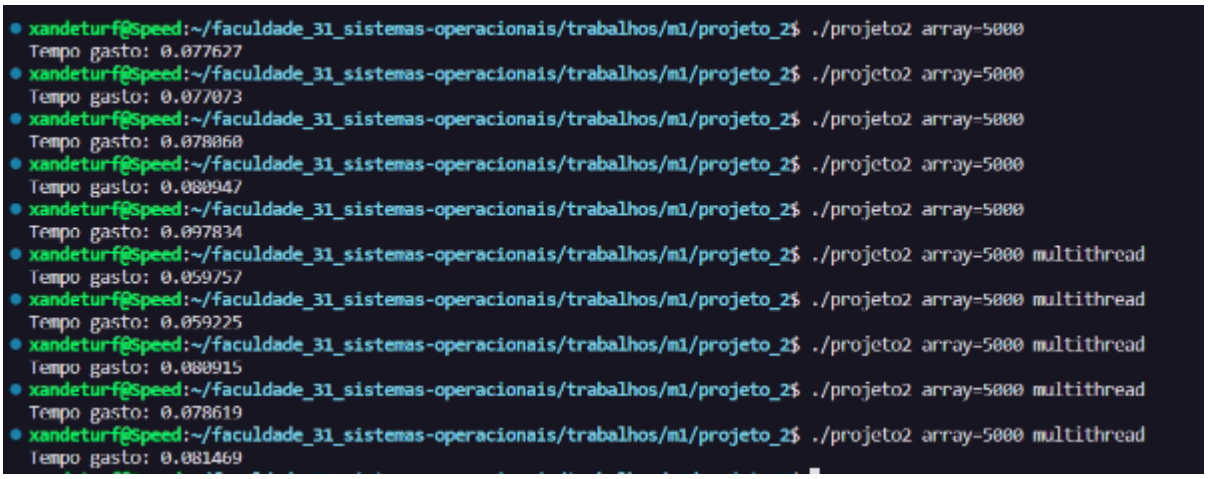
Uma particularidade do código é a sua capacidade de realizar a ordenação utilizando duas threads em paralelo, uma técnica chamada multithreading. Neste processo, o array é dividido em duas partes, sendo cada uma ordenada de forma independente e simultânea, para posteriormente serem ordenadas conjuntamente,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



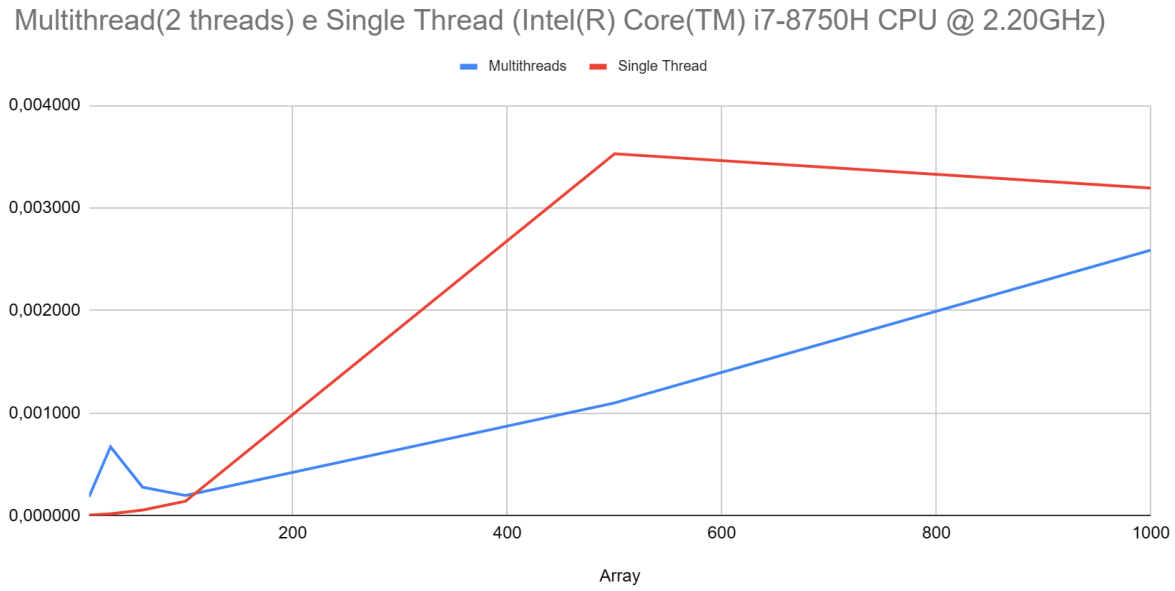
Single thread com array 5000, média 0.082308 segundos:
0.077627 segundos
0.077073 segundos
0.078060 segundos
0.080947 segundos
0.097834 segundos

Multithreads com matriz 5000, média 0.071997 segundos:
0.059757 segundos
0.059225 segundos
0.080915 segundos
0.078619 segundos
0.081469 segundos

Códigos:

```c
// Iniciar o timer
inicio_clock = clock();
clock_gettime(CLOCK_REALTIME, &inicio);

if (multiThread)
{
    pthread_t thread1, thread2;
    Parametros p1, p2;
    pthread_attr_t attr; /* set of attributes for the thread */

    pthread_attr_init(&attr);

    int meio = tamanhoGlobal / 2;
    p1.array = meuArray;
    p1.inicio = 0;
    p1.fim = meio;

    p2.array = meuArray;
    p2.inicio = meio;
    p2.fim = tamanhoGlobal;

    pthread_create(&thread1, &attr, bubbleSort_multithread, &p1);
    pthread_create(&thread2, &attr, bubbleSort_multithread, &p2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    if (isPrintArray)
    {
        printf("Array antes do merge:\n");
        printArray(meuArray, tamanhoGlobal);
    }
    // bubbleSort(meuArray, meio - 1, meio + 1);
    bubbleSort(meuArray, 0, tamanhoGlobal);
}
else
{
    bubbleSort(meuArray, 0, tamanhoGlobal);
}
```

Gráfico:



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.

Resolução projeto 3:

O código apresentado é uma aplicação de ordenação de arrays em C, que oferece flexibilidade na forma como o array é inicializado e como é ordenado. Os usuários podem optar por preencher o array com números em ordem decrescente, valores aleatórios ou um conjunto fixo de números. Adicionalmente, a aplicação permite aos usuários escolher se desejam ordenar o array utilizando um único processo ou dois processos (pai e filho), facilitando assim a exploração de diferentes abordagens de ordenação e a avaliação do seu impacto no desempenho. A aplicação também mede e exibe o tempo necessário para completar a ordenação, proporcionando uma maneira direta de comparar a eficiência de diferentes métodos 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(4 threads) foi mais eficiente que o single thread:

```
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3
Tempo gasto: 0.527012
Tempo gasto em um array(10000) com processo filho(Desativado): 0.527012 segundos.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3
Tempo gasto: 0.522807
Tempo gasto em um array(10000) com processo filho(Desativado): 0.522807 segundos.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3
Tempo gasto: 0.525862
Tempo gasto em um array(10000) com processo filho(Desativado): 0.525862 segundos.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3
Tempo gasto: 0.520124
Tempo gasto em um array(10000) com processo filho(Desativado): 0.520124 segundos.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3
Tempo gasto: 0.523013
Tempo gasto em um array(10000) com processo filho(Desativado): 0.523013 segundos.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3 processos
Tempo gasto: 0.399224
Tempo gasto em um array(10000) com processo filho(Ativo): 0.399224 segundos.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3 processos
Tempo gasto: 0.398699
Tempo gasto em um array(10000) com processo filho(Ativo): 0.398699 segundos.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3 processos
Tempo gasto: 0.399844
Tempo gasto em um array(10000) com processo filho(Ativo): 0.399844 segundos.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3 processos
Tempo gasto: 0.403265
Tempo gasto em um array(10000) com processo filho(Ativo): 0.403265 segundos.
xandeturf@Speed:~/faculdade_31_sistemas-operacionais/trabalhos/m1/projeto_3$ ./projeto3 processos
Tempo gasto: 0.396419
```

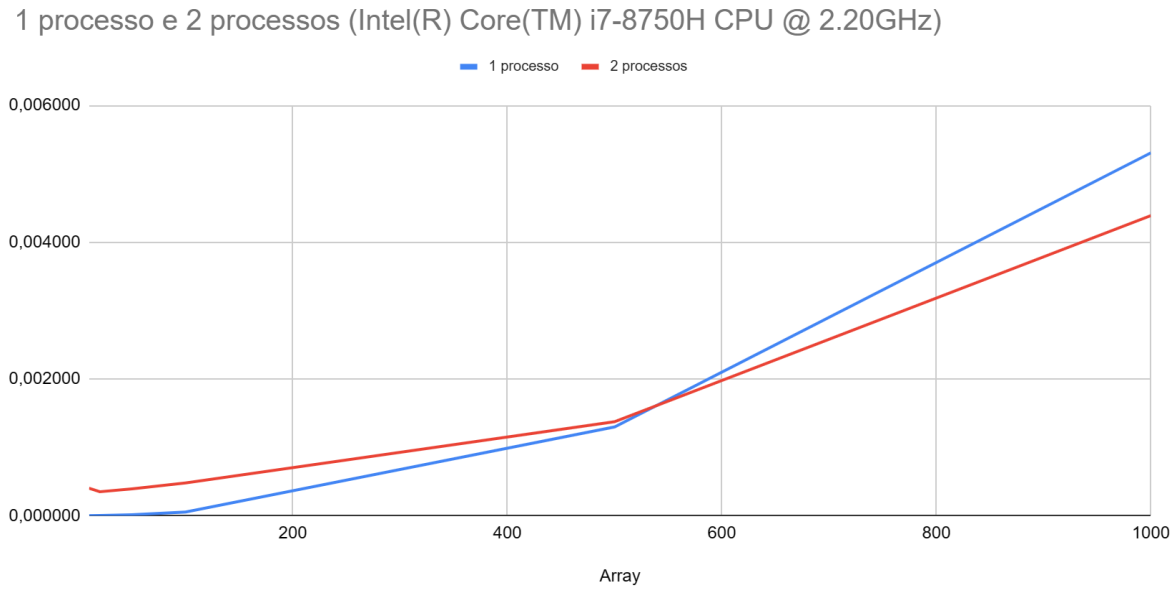
Single processo com array 10000, média 0.5237636 segundos:
0.527012 segundos
0.522807 segundos
0.525862 segundos
0.520124 segundos
0.523013 segundos

2 processos com array 10000, média 0.3994902 segundos:
0.399224 segundos
0.398699 segundos
0.399844 segundos
0.403265 segundos
0.396419 segundos

Códigos:

```c
struct timespec inicio, fim;
    clock_gettime(CLOCK_REALTIME, &inicio);

    if (processos)
    {

        int pipe_fds[2];
        pipe(pipe_fds);
        pid_t pid;

        if ((pid = fork()) == 0)
        {
            // Processo filho
            // printf("Executando o processo filho\n");
            close(pipe_fds[0]);
            bubbleSort(meuArray, 0, tamanhoGlobal / 2);
            write(pipe_fds[1], meuArray, sizeof(float) * tamanhoGlobal / 2);
            close(pipe_fds[1]);
            exit(0);
        }
        else
        {
            // Processo pai
            close(pipe_fds[1]);
            bubbleSort(meuArray, tamanhoGlobal / 2, tamanhoGlobal);

            wait(NULL); // Espera o processo filho terminar

            read(pipe_fds[0], meuArray, sizeof(float) * tamanhoGlobal / 2);
            close(pipe_fds[0]);

            // Realizando uma segunda ordenação com bubbleSort no array inteiro
            bubbleSort(meuArray, 0, tamanhoGlobal);
        }
    }
    else
    {
        bubbleSort(meuArray, 0, tamanhoGlobal);
    }

    // Finalizar o timer
    clock_gettime(CLOCK_REALTIME, &fim);
```

Gráfico:



Resultados finais:

O código em questão, implementado em C, demonstra uma técnica de ordenação de arrays através de uma estratégia paralela que utiliza processos pai e filho, além de oferecer a opção de execução em um único processo.

Na prática, a estratégia de usar processos separados para ordenar partes distintas do array não se mostrou muito eficiente em grandes arrays. No entanto, é preciso considerar que a comunicação entre processos através de pipes pode introduzir uma sobrecarga considerável. Além disso, a etapa adicional de realizar uma segunda passagem de ordenação após a combinação dos segmentos ordenados individualmente afeta o tempo total de execução.