

IFES CAMPUS SERRA
SISTEMAS DE INFORMAÇÃO
SISTEMAS OPERACIONAIS

RELATÓRIO

Introdução à Programação Multithread com PThreads

Isabella Sampaio
Ludmila Dias

Serra, novembro de 2022

SUMÁRIO

SUMÁRIO	2
INTRODUÇÃO	3
PROBLEMA PROPOSTO	4
SOLUÇÃO DESENVOLVIDA	4
TESTES	10
Comparação de desempenho dos Computadores	10
O que aconteceu ao rodar sem mutexes?	13
Quantidade de threads x Desempenho	15
Quantidade de macroblocos x Desempenho	18
CONCLUSÃO	20

INTRODUÇÃO

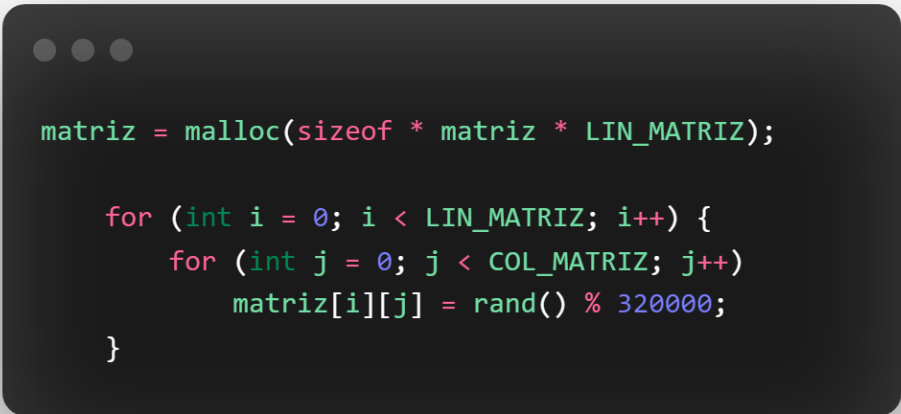
Este presente relatório apresenta as análises do trabalho da disciplina de Sistemas Operacionais, lecionada pelo professor Flávio Giraldelli, com o objetivo de levar os alunos a aplicarem os conceitos aprendidos em sala de aula sobre programação concorrente com múltiplas threads e tratamento de seções críticas.

PROBLEMA PROPOSTO

O programa deve gerar uma matriz de números naturais aleatórios (intervalo 0 a 31999) e contabilizar quantos números primos existem nela e o tempo necessário para isso ao fazer de duas formas: serial e paralela. Para calcular de modo serial, a contagem dos primos deve ser feita um após o outro. Esse será o seu tempo de referência. Enquanto, no modo paralelo, para verificar cada número e contabilizá-lo caso for primo, deverá-se subdividir a matriz em “macroblocos” (submatrizes), baseando-se apenas nos índices. A matriz não é necessariamente quadrada e os macroblocos terão tamanhos que podem variar de desde um único elemento até a matriz toda (equivalente ao caso serial).

SOLUÇÃO DESENVOLVIDA

O primeiro passo foi criar um trecho de código que gerasse para nós uma matriz com os tamanhos definidos no define. Alocamos memória e utilizamos loops *for* para criar a matriz, gerando valores aleatórios com a função *rand()* que sempre gerará sempre a mesma matriz (Figura 1).

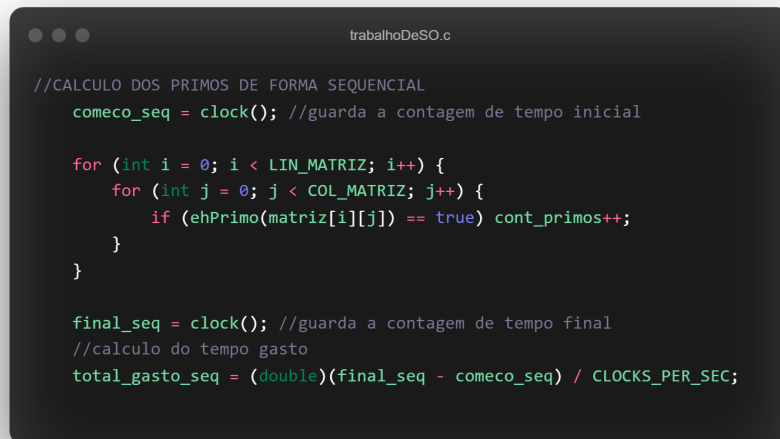


```
matrix = malloc(sizeof * matrix * LIN_MATRIZ);

for (int i = 0; i < LIN_MATRIZ; i++) {
    for (int j = 0; j < COL_MATRIZ; j++)
        matrix[i][j] = rand() % 32000;
}
```

Figura 1 - Trecho de código onde é criada a matriz.

A seguir, desenvolvemos o cálculo do número de primos de forma sequencial. Percorremos a matriz usando loop *for* com a mesma lógica do código da criação da matriz. Dentro do loop, para cada item da matriz chamamos a função *ehPrimo()* (figura 3) que criamos para identificar o número, caso o número for primo, uma variável *cont_primos* soma 1 a ela mesma (Figura 2).



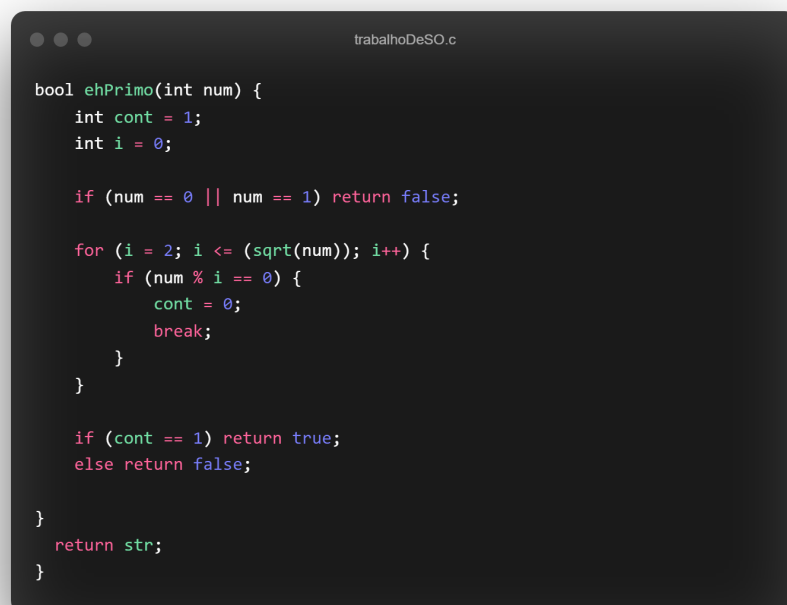
```
trabalhoDeSO.c

//CALCULO DOS PRIMOS DE FORMA SEQUENCIAL
comeco_seq = clock(); //guarda a contagem de tempo inicial

for (int i = 0; i < LIN_MATRIZ; i++) {
    for (int j = 0; j < COL_MATRIZ; j++) {
        if (ehPrimo(matriz[i][j]) == true) cont_primos++;
    }
}

final_seq = clock(); //guarda a contagem de tempo final
//calculo do tempo gasto
total_gasto_seq = (double)(final_seq - comeco_seq) / CLOCKS_PER_SEC;
```

Figura 2 - Busca sequencial por primos.



```
trabalhoDeSO.c

bool ehPrimo(int num) {
    int cont = 1;
    int i = 0;

    if (num == 0 || num == 1) return false;

    for (i = 2; i <= (sqrt(num)); i++) {
        if (num % i == 0) {
            cont = 0;
            break;
        }
    }

    if (cont == 1) return true;
    else return false;
}

return str;
}
```

Figura 3 - Verificação se o número é primo.

Para a busca paralela criamos alguns elementos essenciais para a programação concorrente, assim como foi orientado pelo professor. Criamos mutexes para a definição das seções críticas e pthreads para a alocação das threads.



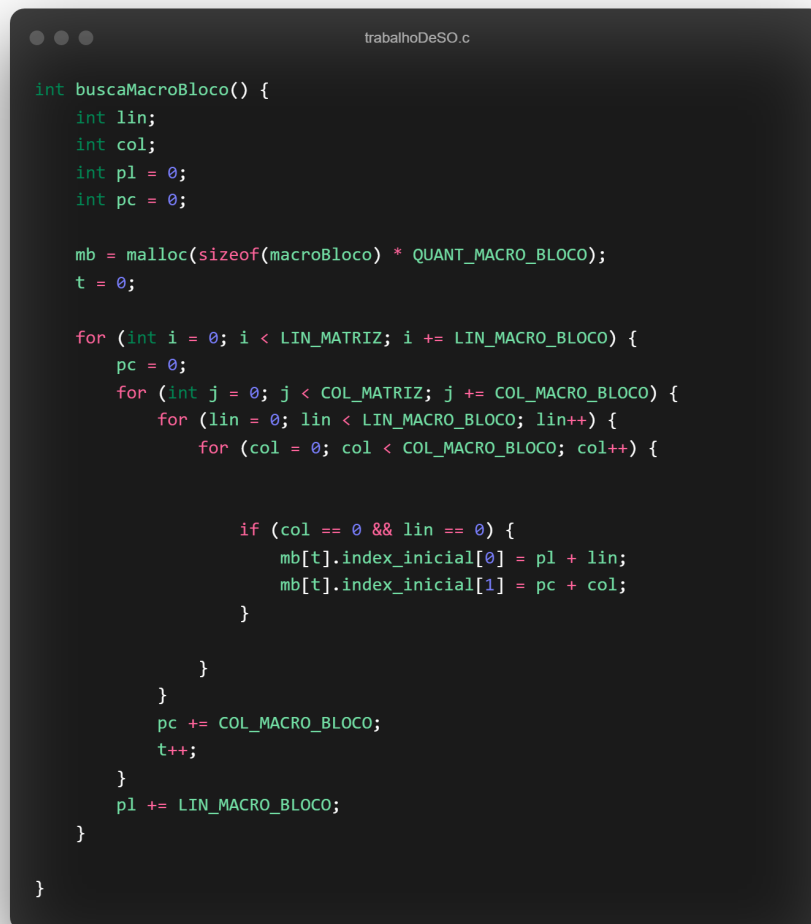
Figura 4 - Declaração dos mutexes como variáveis globais.



Figura 5 - Inicialização dos mutexes e criação das pthreads dentro da função main.

Assim como estabelecido no enunciado do trabalho, dividimos a matriz em macro blocos, utilizando a função *buscaMacroBloco()* que possui um conjunto de loops *for*. Os dois “fors” mais internos percorrem a matriz de acordo com o tamanho do macrobloco, começando a partir de um pivô de linha e de coluna. Esse pivô nos permite saber onde o último macrobloco terminou para o programa poder percorrer

um novo. A partir do pivô mais a posição do contador do for, descobrimos a posição (coluna e linha) do primeiro item do macrobloco, esse valor é salvo numa lista de structs “macrobloco” - a struct é formada por um vetor de tamanho dois, que se refere ao x e y da posição. Os dois “fors” mais externos permitem que esse processo seja executado em toda a matriz (figura 6). Além disso, um contador *t* também é utilizado para identificar quantos macroblocos já foram lidos, e assim poder armazenar o macrobloco que está sendo lido na posição certa do vetor de macroblocos.

A screenshot of a code editor window titled "trabalhoDeSO.c". The code defines a function "buscaMacroBloco()" which iterates through a matrix to find macroblocks. It uses nested for loops for rows and columns, and an inner loop for the macroblock size. When a macroblock is found at the start of a row and column, its starting coordinates are stored in a struct array "mb".

```
int buscaMacroBloco() {
    int lin;
    int col;
    int pl = 0;
    int pc = 0;

    mb = malloc(sizeof(macroBloco) * QUANT_MACRO_BLOCO);
    t = 0;

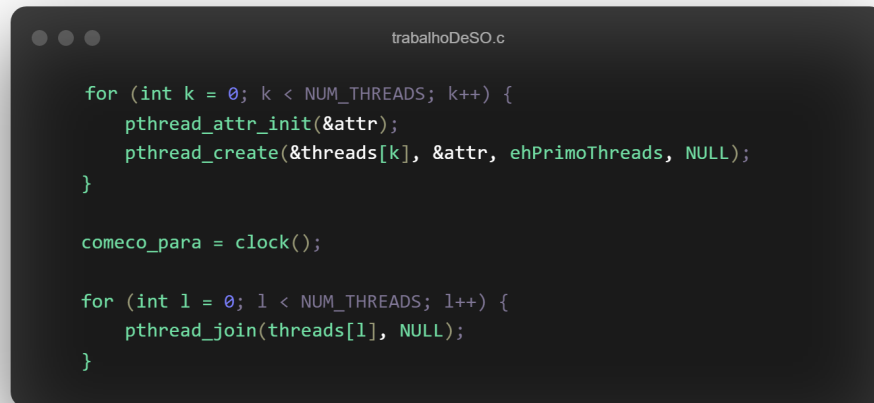
    for (int i = 0; i < LIN_MATRIZ; i += LIN_MACRO_BLOCO) {
        pc = 0;
        for (int j = 0; j < COL_MATRIZ; j += COL_MACRO_BLOCO) {
            for (lin = 0; lin < LIN_MACRO_BLOCO; lin++) {
                for (col = 0; col < COL_MACRO_BLOCO; col++) {

                    if (col == 0 && lin == 0) {
                        mb[t].index_inicial[0] = pl + lin;
                        mb[t].index_inicial[1] = pc + col;
                    }

                }
            }
            pc += COL_MACRO_BLOCO;
            t++;
        }
        pl += LIN_MACRO_BLOCO;
    }
}
```

Figura 6 - Função que busca os macroblocos na matriz.

Utilizando o *pthread_create()*, criam-se as threads, inicializando seus atributos e atribuindo a elas o seu trabalho. Ao final, utiliza-se um *pthread_join()* (figura 7).

A screenshot of a code editor window titled "trabalhoDeSO.c". The code is in C and shows the creation and joining of threads. It includes two for loops: the first loop creates threads using pthread_create, and the second loop joins the threads using pthread_join. The code is as follows:

```
for (int k = 0; k < NUM_THREADS; k++) {
    pthread_attr_init(&attr);
    pthread_create(&threads[k], &attr, ehPrimoThreads, NULL);
}

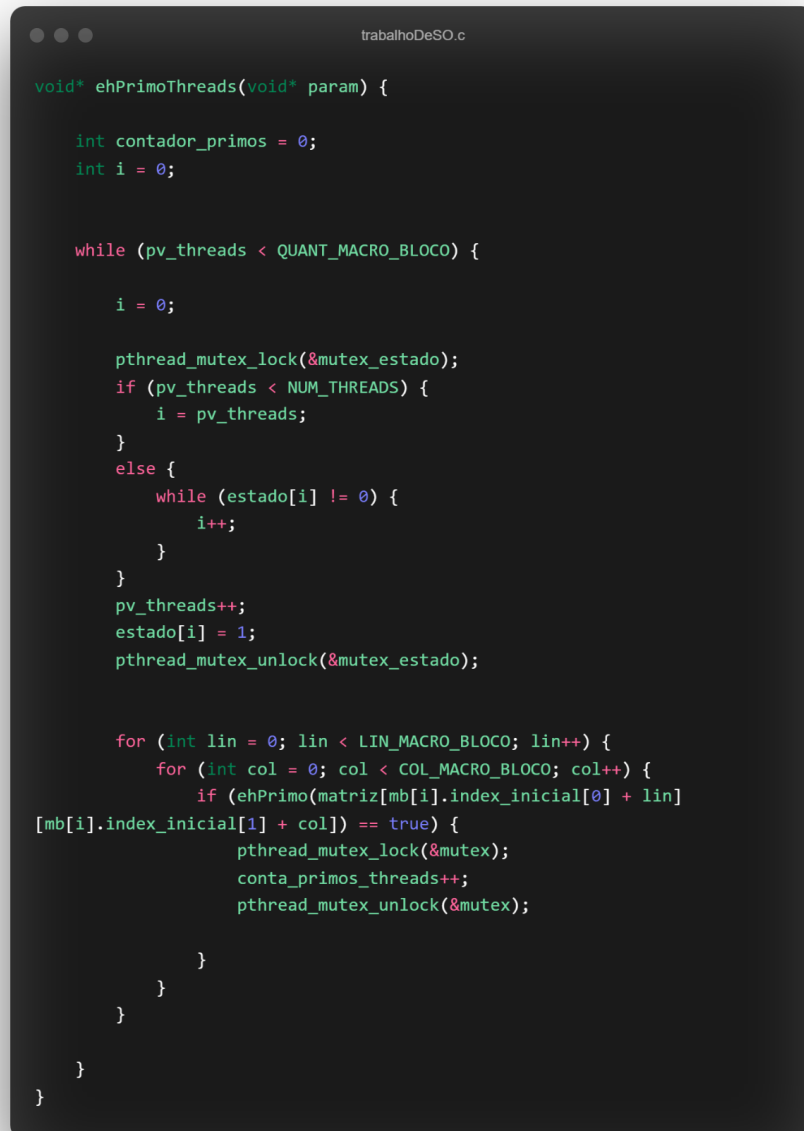
comeco_para = clock();

for (int l = 0; l < NUM_THREADS; l++) {
    pthread_join(threads[l], NULL);
}
```

Figura 7 - Criação e atribuição de threads.

Chamamos para cada thread criada a função *ehPrimoThreads()* que verifica os macroblocos disponíveis e busca nesse a quantidade de números primos que é somada a uma variável global. Cada thread funciona como um carteiro que precisa visitar casas, e visita aquela que não tem carteiro nenhum visitando ou não foi visitada ainda. Para identificar qual macrobloco está disponível, utilizamos um vetor, que armazena na posição respectiva do macrobloco do vetor de structs, 0 para não visitado e 1 para visitado. Utilizamos um condicional if para verificar o vetor e atribuir o macrobloco disponível para a thread, fornecendo a posição do mesmo. Após isso dois loops *for* são executados, de maneira similar a função *buscaMacroBlocos()*, para percorrer a quantidade de valores do macrobloco. No entanto, aqui utilizamos como pivô uma variável “i” que possui o valor da posição fornecido na condicional. A verificação inicia visitando a posição inicial do macrobloco que foi armazenado no vetor de structs na posição “i”, e continua até o fim do macrobloco. Também foi se utilizado a variável *pv_threads* que conta quantos macroblocos já foram visitados, somando mais 1 a ela assim que uma thread é atribuída a um macrobloco. Todo esse processo descrito nessa função até aqui, é executado dentro de um loop *while*

que para quando *pv_threads* é igual ao número total de macroblocos, ou seja, quando todos os macroblocos já foram visitados.



```
trabalhoDeSO.c

void* ehPrimoThreads(void* param) {

    int contador_primos = 0;
    int i = 0;

    while (pv_threads < QUANT_MACRO_BLOCO) {

        i = 0;

        pthread_mutex_lock(&mutex_estado);
        if (pv_threads < NUM_THREADS) {
            i = pv_threads;
        }
        else {
            while (estado[i] != 0) {
                i++;
            }
        }
        pv_threads++;
        estado[i] = 1;
        pthread_mutex_unlock(&mutex_estado);

        for (int lin = 0; lin < LIN_MACRO_BLOCO; lin++) {
            for (int col = 0; col < COL_MACRO_BLOCO; col++) {
                if (ehPrimo(matriz[mb[i].index_inicial[0] + lin]
[mb[i].index_inicial[1] + col]) == true) {
                    pthread_mutex_lock(&mutex);
                    conta_primos_threads++;
                    pthread_mutex_unlock(&mutex);

                }
            }
        }
    }
}
```

Figura 8 - Função que é atribuída a cada thread criada para calcular o número de números primos nos macroblocos.

Para fins de análise de tempo, no início e no final dos trechos de código em que calculamos o número de primos de forma sequencial e paralela, salvamos em variáveis o tempo inicial e final do processo e subtraímos para encontrar o tempo gasto para a execução.

TESTES

Comparação de desempenho dos Computadores

Como estudado na matéria de Arquitetura e Organização de Computadores, computadores com diferentes processadores terão diferentes desempenhos, isso por conta das características do hardware e também do seu números de núcleos físicos e lógicos. Sendo assim, para fins de análise, executamos testes em computadores com diferentes processadores e com diferentes quantidades de threads.

	Processador	Núcleos físicos	Quant. Threads
Computador A	Ryzen 5 1400	4	8
Computador B	Intel core i5-3210M	2	4

Tabela 1 - Computadores utilizados para os testes e seus processadores

Durante os testes, olhando os gráficos gerados no Gerenciador de Tarefas do Windows, foi possível identificar claramente um “padrão” ou algo próximo disso durante a execução do código.

Threads	Uso da CPU
1	10% - 16%
2	20% - 24%
4	40% - 55%
8	80% - 90%
100	100%

Tabela 2 - Uso da CPU no computador A de acordo com a quantidade de threads utilizadas

Threads	Uso da CPU
1	20% - 30%
2	50% - 60%
4	80% - 90%
100	100%

Tabela 3 - Uso da CPU no computador B de acordo com a quantidade de threads utilizadas

As tabelas acima mostram o uso da CPU durante a execução do código utilizando um matriz de 20000 x 10000 e com macrobloco de tamanhos 100x100 e 50x50. Como é possível ver, o uso da CPU se manteve dentro do esperado para as buscas paralelas e também para a busca sequencial, que no PC A apresentou os mesmo valores que os testes feitos com apenas 1 thread (entre 10% e 16%). Além disso, durante o processo de teste, o uso de discos (hd e ssd) se mantiveram bem baixos e próximos a 0%, ou seja, a memória principal não estava fazendo uso deles armazenamento como algum tipo de extensão própria.

É interessante destacar também que utilizando 100 threads, um valor relativamente absurdo, existe um uso constante da CPU em 100%. O SO é que é o responsável por dizer qual thread irá rodar e em qual ordem, gerando um troca e essa troca tem overhead, ou seja, com 100 threads o overhead é muito grande pois acontece uma troca constante. Além disso, analisando o desempenho no PC A, percebemos que o tempo de execução também não varia tanto comparado com o teste com 8 threads, ou seja, não tem um ganho de desempenho, na verdade, neste caso, quanto mais threads (acima do limite existente) pior é para o desempenho. Enquanto o PC B não conseguiu terminar de executar, pois quando o uso da CPU chegou a 100% o computador travou e a execução do programa foi interrompida pelo Visual Studio.

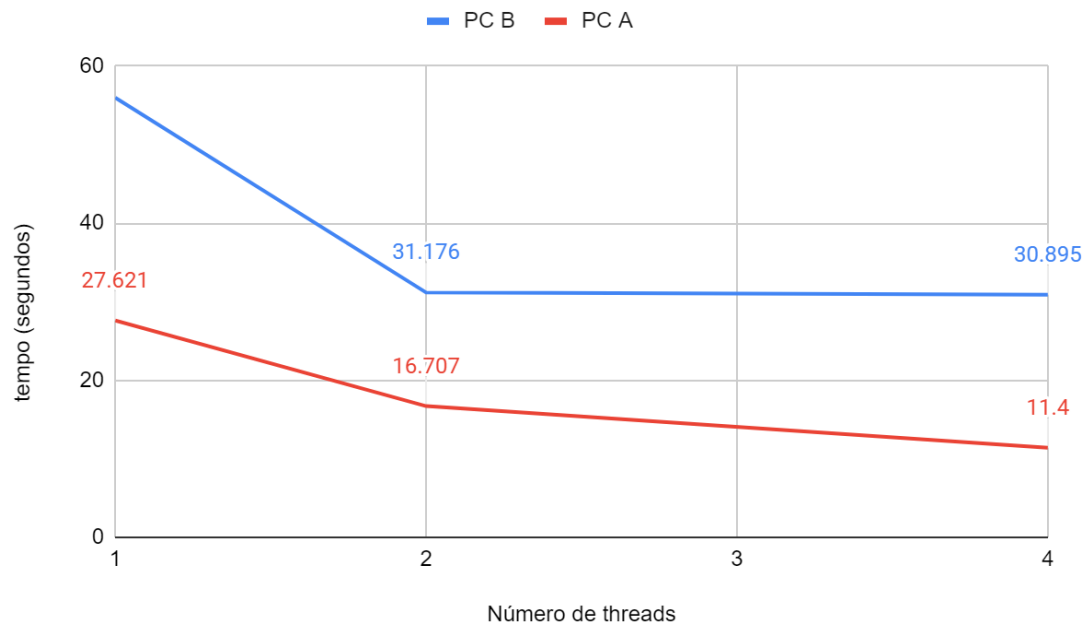


Gráfico 1 - Computador A e Computador B executando matriz 20000 x 10000 com macroblocos de 100 x 100.

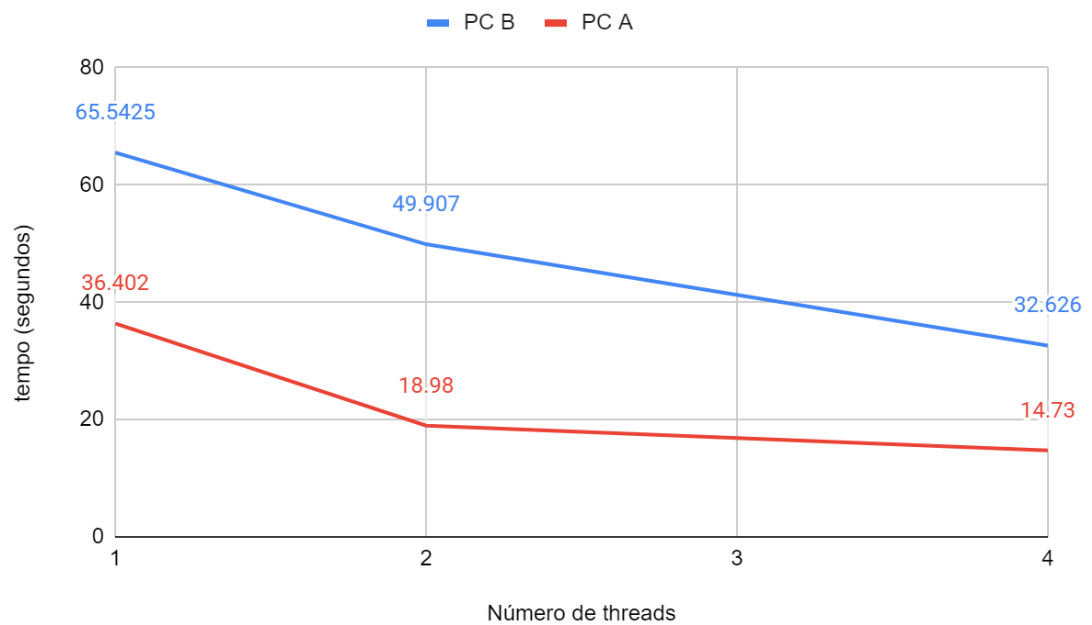


Gráfico 2 - Computador A e Computador B executando matriz 20000 x 10000 com macroblocos de 50 x 50.

O computador A, possui 8 núcleos lógicos e o computador B possui 4, como é possível ver na tabela 1. Apesar de parecer uma “pequena diferença” entre as threads presente nos dois computadores, ao analisarmos o gráfico que dispõe de tempos de execução e números de threads entre PC A e PC B é possível ver uma significativa diferença, mesmo utilizando o mesmo número de threads nos testes em ambos os computadores. Os tempos de execução do PC A se diferem bastante em relação aos do PC B e em todos números de threads, demonstrando que o PC A possui um processador que soube fazer melhor uso do multithreading e assim conseguiu ganhar em desempenho, consequentemente obtendo melhores tempos de execução. Essa grande diferença se dá, principalmente à diferença de threads máxima que cada um possui além de claro, outros aspectos de suas arquiteturas como o tamanho dos níveis de cache L3, no PC A é de 8 mb e no PC B é 3 mb, frequência de clock e tipo de memória , PC A possui 3.2 ghz e é DDR4 e o PC B possui 2,5 ghz e é DDR3.

O que aconteceu ao rodar sem mutexes?

Ao rodar os teste sem mutexes, muitas vezes houve erro durante a contagem de primos e, ou, o próprio visual studio interrompeu já que o acesso às variáveis compartilhadas estava gerando insegurança na memória compartilhada. Além disso, muitas vezes as threads pareciam executar de maneira “desordenada” e o vetor de estado dos macroblocos não era “setado” de maneira correta, já que o mutex delimitador de estado não estava presente, ou seja, as threads não entravam na região crítica e assim, uma fazia o seu trabalho “em cima” da outra, alterando de maneira errada o vetor e também a quantidade de primos contados.

Matriz A 20000 x 10000 100 x 100 sem mutex			
Num de threads	Busca sequencial	Busca paralela	SpeedUp
1	27.4	28.15	0.973
2	27.4	16.13	1.699
4	27.4	10.35	2.647
8	27.4	6.722	4.076

Tabela 3 - Teste em macrobloco de tamanho 100 x 100 sem mutex feitos no PC A

Matriz A 20000 x 10000 50 x 50 sem mutex			
Num de threads	Busca sequencial	Busca paralela	SpeedUp
1	27.4	35.308	0.776
2	27.4	20.848	1.314
4	27.4	12.706	2.156
8	27.4	7.05	3.887

Tabela 4 - Teste em macrobloco de tamanho 50 x 50 sem mutex feitos no PC A

Ao analisar os resultados do código com macroblocos de tamanho 100 x 100 e 50 x 50 sem mutexes, observou-se que houve um aumento de speedup se compararmos com os testes feitos também em macroblocos 100 x 100 e 50 x 50, porém com mutexes. Além disso, a quantidade de primos que foram contados nos testes também sofreu alterações, porque, como explicado acima, mutexes são delimitadores de regiões críticas e auxiliam no controle do acesso a recursos compartilhados entre múltiplas threads, quando esse delimitador não está presente as regiões críticas deixam de ser tratadas e então o acesso aos recursos compartilhados perde a sua ordem , assim a contagem de primos também é afetada já que não se pode mais garantir que apenas uma thread está utilizando um determinado recurso e sim, provavelmente, várias outras também estão, ou seja, várias outras estão adicionando ao contador de primos e verificando os macroblocos e o vetor de estados de maneira desorganizada, gerando a inconsistência no total de primos calculado ao final da execução do código.

Quantidade de threads x Desempenho

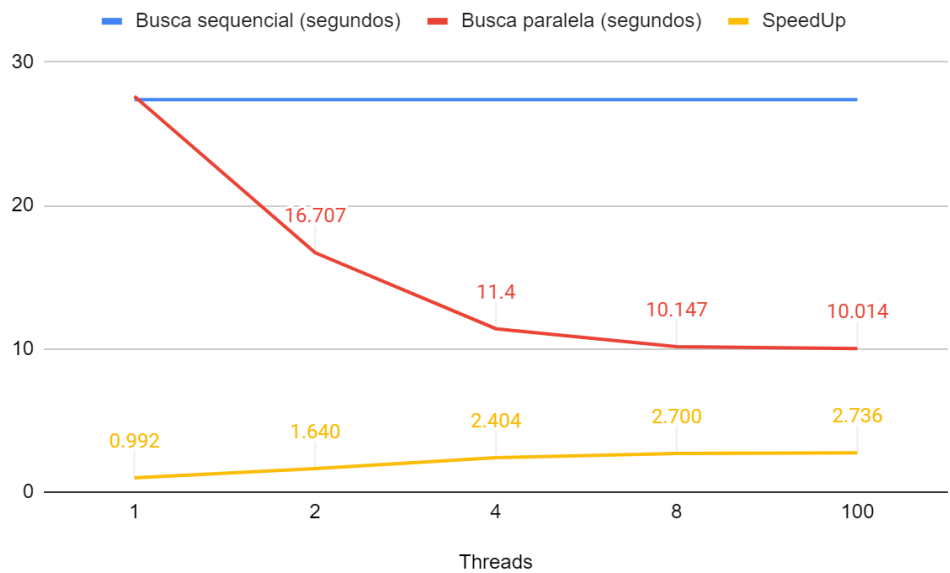


Gráfico 3 - Resultados do computador A rodando uma matriz de 20000 x 10000 com macroblocos de 100 x 100.

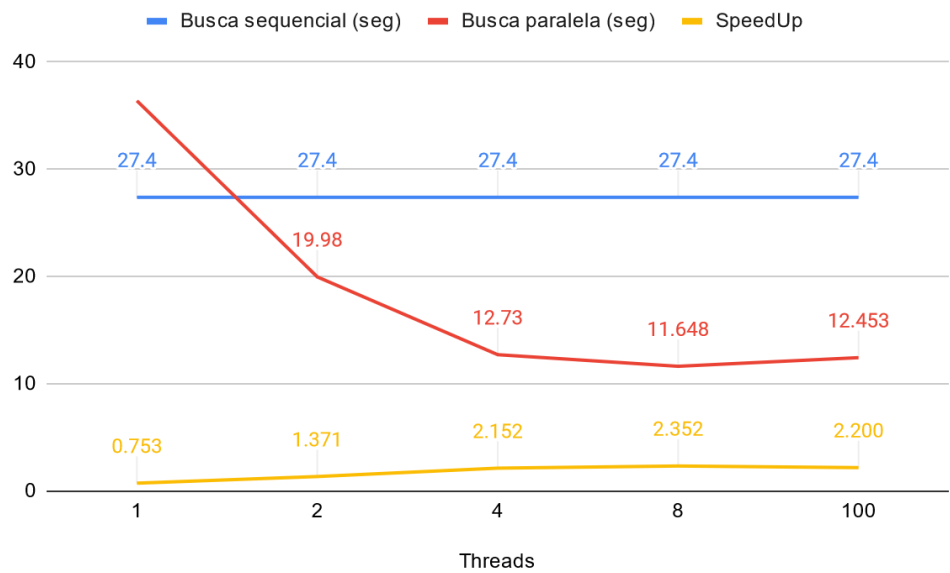


Gráfico 4 - Resultados do computador A rodando uma matriz de 20000 x 10000 com macroblocos de 50 x 50.

Ao analisarmos os dois gráficos referentes ao PC A, gráficos 3 e 4, é possível identificar uma coerência entre a diminuição do tempo de execução e o ganho de speedup, assim como o aumento de desempenho. Em ambos os macroblocos O speedup se manteve próximo de 3 em testes em 4, 8 e 8 threads estando coerente com a lei de amdahl, ou seja, quanto mais próximo a quantidade de núcleos físicos, melhor e neste caso, como o PC A possui 4 núcleos físicos, os resultados obtidos estão de acordo.

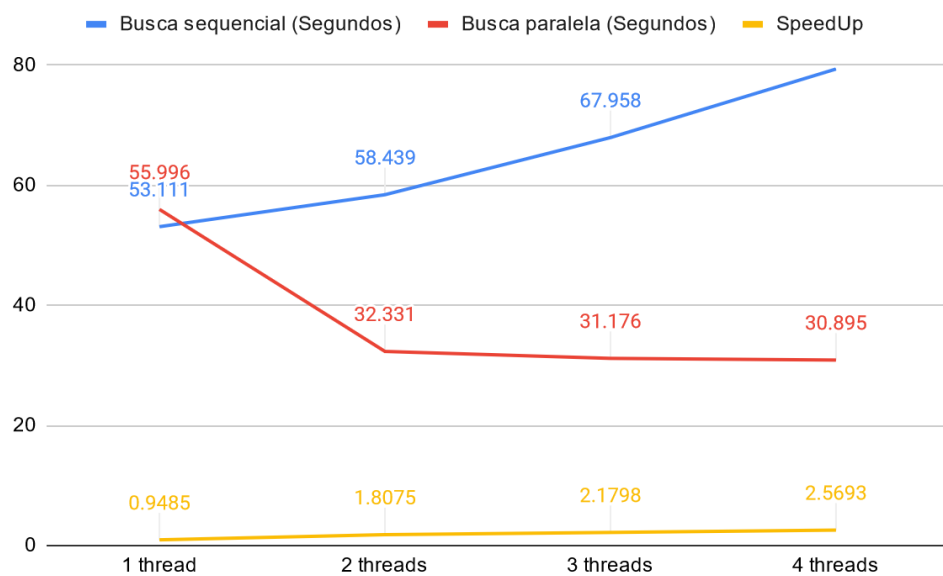


Gráfico 5 - Resultados do computador B rodando uma matriz de 20000 x 10000 com macroblocos de 100 x 100.

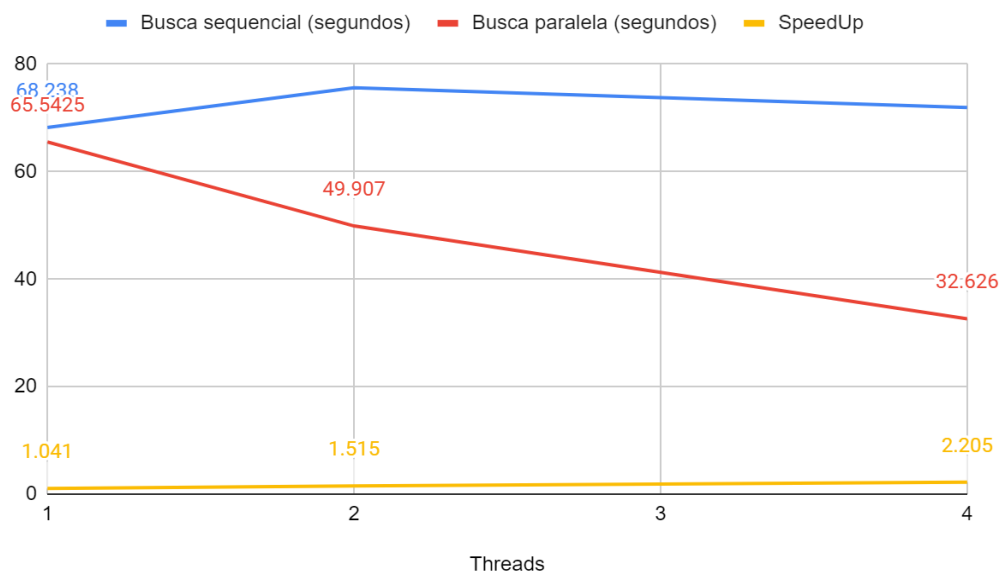


Gráfico 6 - Resultados do computador B rodando uma matriz de 20000 x 10000 com macroblocos de 50 x 50.

Ao analisarmos os dois gráficos referentes ao PC B, gráficos 5 e 6, nota-se que ambos os computadores ao rodarem com 1 thread, apresentaram um tempo de execução da busca paralela próximo ao tempo da busca sequencial. Isso pois a diferença na execução nesse caso está dependendo da forma da busca - sequencial por linha e paralela por macrobloco - podendo ser até mais eficiente a busca sequencial nesse caso, que por exemplo no gráfico 4 mostrou melhor desempenho de tempo. Ao aumentar o número de threads de 1 para 2 o tempo da busca paralela diminui notavelmente, uma queda superior às que ocorrem no aumento de 2 para 4 threads e também de 4 para 8, ou seja, cada vez que se aumenta o número de threads, menor o tempo e a diferença do aumento de desempenho de tempo na mudança do número de threads.

Quantidade de macroblocos x Desempenho

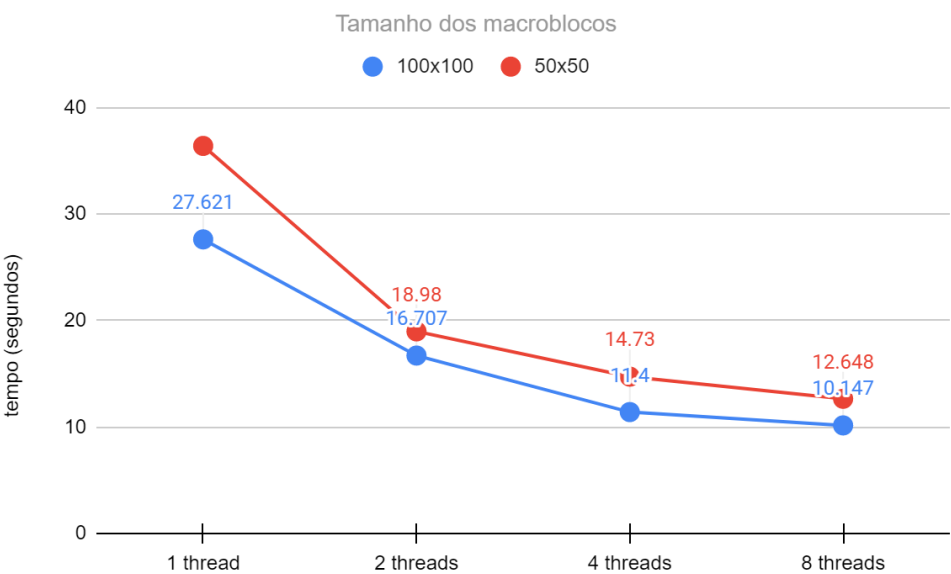


Gráfico 7 - Resultados do computador A comparando tamanhos de macrobloco 100 x 100 e 50 x 50

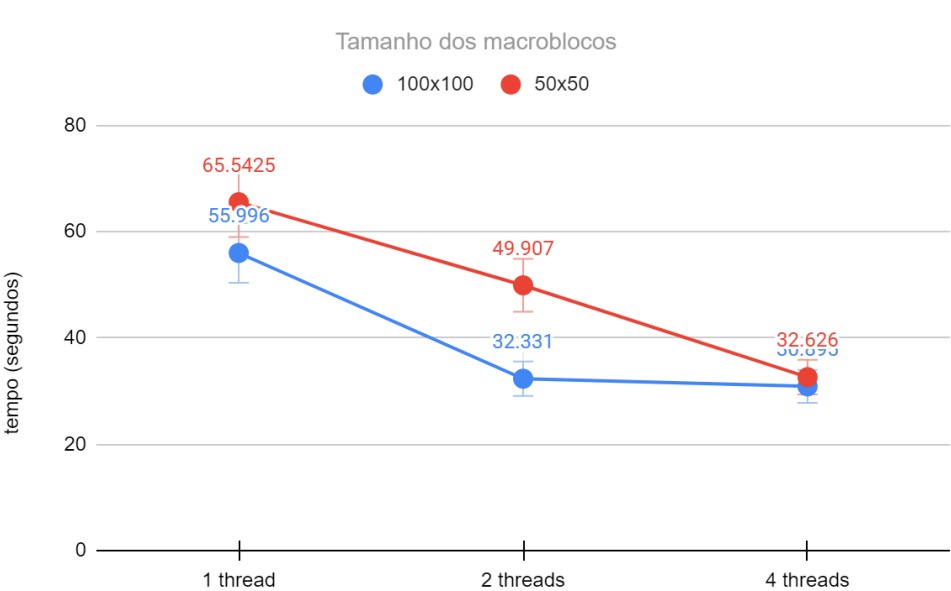


Gráfico 8 - Resultados do computador B comparando tamanhos de macrobloco 100 x 100 e 50 x 50

Ao analisarmos os gráficos 7 e 8, nota-se que tanto o PC A quanto o PC B possuem uma diferença entre tempos de execução entre os macroblocos 100 x 100 e 50 x 50. O tempo de execução tende a cair quando estamos em macroblocos maiores e tempos de execução maiores quando estamos em macroblocos menores. Esse aumento no tempo de execução em ambos os PC 's é compreensível pois, com mais macroblocos as threads têm mais trabalho para fazer e isso influencia diretamente no tempo de execução do código.

CONCLUSÃO

Concluimos, neste presente trabalho - que juntou nossos dois pesadelos: C e Pthreads - que a quantidade de threads agiliza processos, mas saber como utilizá-las também interfere no resultado final. A partir desse trabalho, também pudemos visualizar como diferentes processadores se comportam executando um processo de forma paralela, com ou sem condições de corrida. Além disso, aprendemos como funciona a atribuição de tarefas a cada thread e para que serve a posix threads (pthreads). Também aprendemos e vimos na prática o quão importante é o tratamento das regiões críticas utilizando os mutexes, garantindo a integridade dos processos e a garantia de segurança e um compartilhamento de recursos organizado e como é interessante analisar o ganho de speedup e desempenho e saber como isso funciona na prática - estudado previamente em AOC e agora visto de maneira “aplicada” em SO - como foi explicado em sala de aula pelo professor Flávio Giraldele. Ademais, podemos dizer que, apesar das dificuldades na curva de aprendizagem e também na realização de testes e configuração do programa utilizado para fazer os testes (Visual Studio 2022) - quase perdemos todos os cabelos nessa parte - conseguimos realizar nossas análises da melhor forma que conseguimos, no aprendizado valeu a pena.