



INSTITUTO FEDERAL DO ESPÍRITO SANTO
IFES

André Felipe Santos Martins
Vinícius Freitas Rocha

RELATÓRIO
Introdução à Programação Multithread com PThreads

Serra – ES
2018

INSTITUTO FEDERAL DO ESPÍRITO SANTO

André Felipe Santos Martins
Vinícius Freitas Rocha

RELATÓRIO

Introdução à Programação Multithread com PThreads

Este relatório é referente ao primeiro trabalho apresentado à disciplina de Sistemas Operacionais do Instituto Federal do Espírito Santo, como requisito para avaliação.
Orientador: Flávio Giraldele Bianca
Semestre letivo: 2018/2.

SUMÁRIO

1 INTRODUÇÃO	3
2 OBJETIVOS	4
3 DESENVOLVIMENTO	5
3.1 Ambiente	5
3.2 Resumo do algoritmo	6
4 TESTES E ANÁLISES	8
4.1 Modo Serial	8
4.2 Modo Paralelo	9
3 CONCLUSÃO	16

1 INTRODUÇÃO

O uso de threads e processos no multiprocessamento de tarefas foi um assunto bastante recorrente na disciplina de Sistemas operacionais. A Thread é um fluxo de execução de uma parte de processo ou dele por completo. Um processo pode criar múltiplas threads e executar uma tarefa paralelamente e possivelmente concorrendo a recursos compartilhados, multithread.

Esse tipo de programação otimiza diversas soluções em relação a tempo e consumo energético, mas para isso é necessário repartir as atividades, equilibrar o uso dos núcleos, definir divisão dos dados, garantir a integridade na região crítica. Isso traz um aumento significativo na complexibilidade de debug e teste das aplicações.

Em nível de hardware já é possível realizar o multiprocessamento, porém, fica a cargo do programador desenvolver algoritmos paralelizados, os quais visam aproveitar o máximo da potência de multiprocessamento do computador.

A PThreads é a biblioteca de threads que é extremamente importante para o desenvolvimento deste trabalho. Ela utiliza o padrão POSIX (*Portable Operating System Interface*), que oferece ao programador um conjunto de ferramentas para criar e manipular threads e semáforos.

2 OBJETIVOS

Este trabalho tem como objetivo aplicar os conhecimentos adquiridos em sala sobre threads e multiprocessamento. Para isso foi implementado um algoritmo na linguagem C com auxílio da biblioteca PThreads, que tem como função fazer a contagem de números primos, no intervalo de 0 a 29999, em uma matriz de $N \times M$, dividindo-a em macroblocos a serem processados por diferentes threads de forma paralela.

Com o resultado deste algoritmo podemos estudar os custos, benefícios e consequências da programação multithread na solução de um problema. Para tanto foi feita a contagem da quantidade de números primos em uma matriz relativamente grande, de forma single thread e multithread, na qual as operações multithread foram efetuadas em diversos cenários diferentes de macroblocos e quantidades de threads simultâneas.

3 DESENVOLVIMENTO

3.1 Ambiente

A implementação do algoritmo foi feita na IDE Visual Studio Community 2017, recomendada pelo professor. Toda a preparação do ambiente, a instalação da biblioteca PThreads e demais detalhes para iniciar o desenvolvimento e testes foram feitos seguindo um passo a passo disponibilizados pelo professor [neste link](#).

Após a implementação do algoritmo, foram utilizados dois computadores de diferentes especificações para uma bateria de testes, com objetivo de obter resultados com maiores diferenças, o que nos possibilita fazer comparativos entre como cada sistema se comporta ao executar as mesmas tarefas.

O computador 1 (PC1) tem um processador Intel I7-8550U (4 núcleos reais e 8 lógicos), 16 GB de RAM DDR4. A seguir, uma imagem com maiores detalhes da especificação.

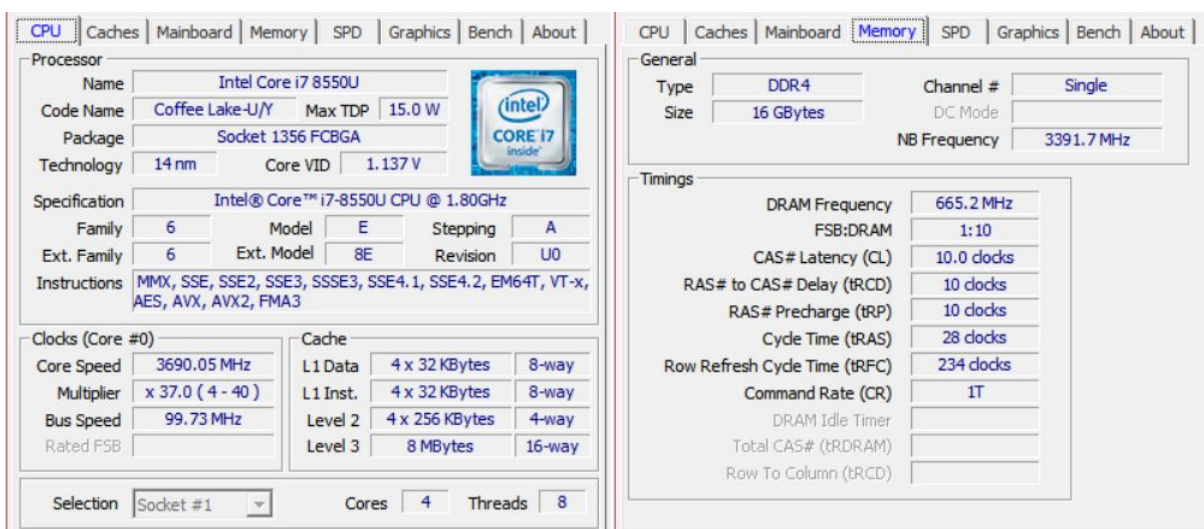


Imagem 1: Especificações do PC1.

O computador 2 (PC2) tem um processador Intel I5-4300U (2 núcleos reais e 4 lógicos) e 8 GB de RAM DDR3. Abaixo, uma imagem com maiores detalhes da especificação.

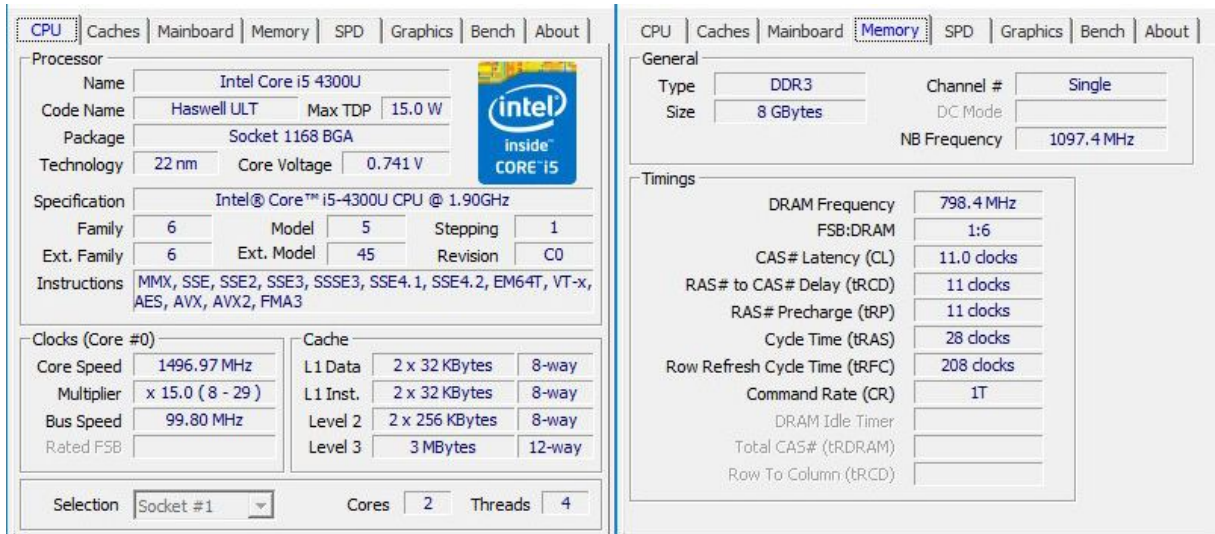


Imagem 2: Especificações do PC2.

3.2 Resumo do algoritmo

Logo de início, fizemos uma tentativa de desenvolver o algoritmo de forma modular, ao criar outros arquivos de código que seriam importados para o arquivo em que se encontra a função principal e desenvolver as funções do algoritmo de forma parametrizada. Porém, por conta de algumas dificuldades geradas por trabalharmos com variáveis globais, optamos por mudar e desenvolver todas as funções em um único arquivo, apesar de mantermos a estrutura das funções parametrizadas.

Todas as funções do código estão devidamente comentadas, além disso fizemos uma organização no código por blocos (pragmas) no intuito de manter um padrão de organização e facilitar o entendimento do mesmo.

Para facilitar os testes do algoritmo, todas as variáveis que definem o tamanho da matriz, os macroblocos e a quantidade de threads estão definidas no bloco de defines, logo no topo do código, como podemos observar abaixo.

```

#pragma region Defines
#define LINHA_MATRIZ 15000 // Número total de linhas da matriz
#define COLUNA_MATRIZ 15000 // Número total de colunas da matriz

#define LINHA_MB 100 // Quantidade de linhas de um macrobloco
#define COLUNA_MB 100 // Quantidade de colunas de um macrobloco

#define VALOR_MAX 29999 // Valor máximo a ser preenchido na matriz
#define VALOR_MIN 0 // Valor mínimo a ser preenchido na matriz

#define TRUE 1
#define FALSE 0

#define NUM_THREADS 8 // Número de threads
#define IS_SERIAL FALSE // TRUE para busca serial; FALSE para busca paralela
#define SEED 10
#pragma endregion

```

Imagem 3: defines (variáveis constantes em C).

No bloco de variáveis globais estão principalmente as variáveis que precisam ser controladas na região crítica e as que guardam os valores do tempo de processamento a ser exibido. Também criamos uma struct para controlar os macroblocos e quais pontos da matriz já foram verificados.

```

#pragma region Variáveis Globais
struct bloco {
    int linha_inicial;
    int coluna_inicial;

    int linha_final;
    int coluna_final;
};
typedef struct bloco tBloco, *pBloco;

int total_numeros_primos = 0;
double inicio_tempo_execucao, fim_tempo_execucao, tempo_total_execucao = 0;
int total_blocos, contador_blocos;
double inicio_tempo_bloco, fim_tempo_bloco;

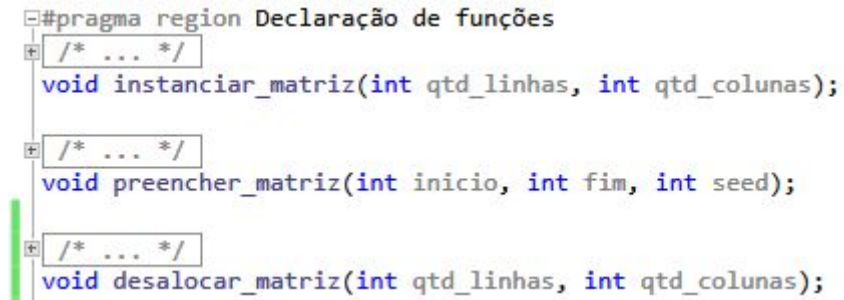
int **matriz;
pthread_mutex_t mutex_bloco;
pthread_mutex_t mutex_count_primos;
tBloco *bloco_verificado;
#pragma endregion

```

Imagem 4: bloco de variáveis globais.

Para manipular a matriz antes da contagem foram criadas as seguintes funções para preenchê-la: `instanciar_matriz()`, `preencher_matriz()`. No fim do programa a matriz é

desalocada usando `desalocar_matriz()`. As funções que fazem a contagem serão abordadas nas seções a seguir.



```
#pragma region Declaração de funções
/* ... */
void instanciar_matriz(int qtd_linhas, int qtd_colunas);

/* ... */
void preencher_matriz(int inicio, int fim, int seed);

/* ... */
void desalocar_matriz(int qtd_linhas, int qtd_colunas);
```

Imagem 5: bloco de funções declaradas previamente.

4 TESTES E ANÁLISES

Os testes foram realizados com uma matriz de dimensões diferentes para cada modo de busca, preenchida com inteiros no intervalo de 0 a 29999. Todos os testes, feitos no modo *Release x86* do Visual Studio, foram executados no PC1 e no PC2, e tiveram seus resultados colhidos para análise.

4.1 Modo Serial

```
int contagem_serial(int **matriz)
{
    int total_primos = 0;
    for (int i = 0; i < LINHA_MATRIZ; i++)
    {
        for (int j = 0; j < COLUNA_MATRIZ; j++)
        {
            if (is_primo(matriz[i][j])) total_primos++;
        }
    }
    return total_primos;
}
```

Imagem 6: função de contagem serial.

O algoritmo de contagem no modo serial percorre a matriz e verifica uma posição de cada vez. Como não há busca na matriz dividida em macroblocos e também não há o uso de threads, a mudança destes parâmetros no bloco de define não altera o resultado da busca.

Matriz	Tempo PC1 (segundos)	Tempo PC2 (segundos)
1000 x 1000	0,084	0,398
5000 x 5000	1,792	5,091
10000 x 10000	6,741	19,763
15000 x 15000	14,457	44,216
20000 x 20000	26,950	77,936

Tabela 1: Tempo de processamento no modo serial.

Na tabela acima é possível observar o resultado dos testes no modo serial, com diversas dimensões de matrizes. Percebe-se logicamente que, quanto maior o tamanho da matriz, maior o tempo necessário para fazer a contagem. A diferença de tempo entre os dois computadores está diretamente ligada à performance individual de cada um. Dada a sua velocidade de clock, cada um faz a verificação de cada número individualmente em velocidades diferentes, e nesse caso, o PC1 se sai muito melhor.

4.2 Modo Paralelo

```
int contagem_paralela(pthread_t *threads, tBloco *bloco_verificado_local)
{
    bloco_verificado = bloco_verificado_local;

    printf("Threads Iniciadas: ");

    for (int i = 0; i < NUM_THREADS; i++)
    {
        int *thread_id = (int*)malloc(sizeof(int*)); // Id da threads a ser criada
        *thread_id = i;

        /* Em caso de sucesso, pthread_create retorna 0 */
        if (pthread_create(&threads[i], NULL, contagem_thread, thread_id) != 0)
        {
            free(thread_id);
            printf("Falha na Thread %d.\n", i);
            getchar();
            exit(1);
        }
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        /* Em caso de sucesso, pthread_join retorna 0 */
        if (pthread_join(threads[i], NULL) != 0) {
            printf("Falha ao retornar a Thread %hi.\n", i);
            getchar();
            exit(1);
        }
    }

    printf("\n\n");
    return total_numeros_primos;
}
```

Imagem 7: função de contagem paralela, responsável pela criação das threads.

O algoritmo de contagem em modo paralelo divide a matriz em blocos menores, atribui cada bloco a uma thread. As threads são executadas em paralelo, portanto, cada uma faz a contagem do bloco que lhe foi atribuída. Para os testes feitos no modo paralelo, mantemos as dimensões da matriz em 15000 x 15000 e alteramos somente as dimensões do macrobloco e a quantidade de threads.

Matriz / Macrobloco	15000x15000 e 1x1	15000x15000 e 100x100	15000x1500 0 e 200x200	15000x15000 e 700x700	15000x15000 e 1500x1500	Média MB > 1x1
1	18,364	15,004	14,981	14,863	14,896	14,936
2	28,734	10,268	10,266	10,304	10,303	10,285
4	33,691	8,049	8,099	8,786	8,160	8,274
8	46,242	7,262	7,315	7,247	7,259	7,271
16	47,945	7,237	7,295	7,348	7,300	7,295
32	47,425	7,288	7,389	7,296	7,342	7,329
64	46,773	7,227	7,307	7,407	7,399	7,335
128	47,444	7,253	7,369	7,403	7,435	7,365
512	41,970	7,576	7,662	7,558	7,656	7,613
900	48,149	7,908	8,171	7,944	8,001	8,006

Tabela 2: Tempo de execução paralela PC1.

Matriz / Macrobloco	15000x15000 e 1x1	15000x15000 e 100x100	15000x1500 0 e 200x200	15000x15000 e 700x700	15000x15000 e 1500x1500	Média MB > 1x1
1	28,460	23,375	23,302	23,349	22,923	23,237
2	41,850	17,283	17,086	17,146	16,941	17,114
4	58,647	14,298	14,115	14,274	14,158	14,211
8	60,795	14,283	13,796	13,962	14,021	14,016
16	60,136	13,995	13,932	14,023	14,222	14,043
32	60,346	14,433	13,990	14,318	14,246	14,247
64	60,441	14,073	14,238	14,229	14,041	14,145
128	60,300	14,061	14,039	14,009	14,536	14,161
512	60,886	14,538	14,707	15,898	14,928	15,018
900	61,382	15,045	15,119	15,207	15,671	15,261

Tabela 3: Tempo de execução PC2.

Ao observar individualmente a coluna de testes realizados com o macrobloco de dimensões 1x1, que seria o extremo mínimo dos testes, pode-se notar que os resultados são disparadamente os piores tempo. Quanto maior é a quantidade de threads em execução, mais demorado é o processo de contagem, e, assim, pode atingir 3 vezes mais tempo do que os testes seguintes com a mesma quantidade de threads.

Esse resultado é devido sobrecarga de acesso a **região crítica** que é protegida pelos mutexes, e como, o acesso dentro a região crítica deve ser de forma serial, as atividades com macroblocos muito pequeno geram uma fila de espera grande. E no fim, o tempo de espera pelo acesso a região crítica é maior que o tempo de processamento do macrobloco.

```
// Início da região crítica de macrobloco
pthread_mutex_lock(&mutex_bloco);

/* Guarda posições onde se inicia o macrobloco */
bloco_local->linha_inicial = bloco_verificado->linha_inicial;
bloco_local->coluna_inicial = bloco_verificado->coluna_inicial;

/* Guarda onde acaba o macrobloco */
bloco_local->linha_final = bloco_local->linha_inicial + LINHA_MB;
bloco_local->coluna_final = bloco_local->coluna_inicial + COLUNA_MB;

/* Atualiza o bloco que representa a matriz para controlar o que já foi separado para verificação */
bloco_verificado->coluna_inicial += COLUNA_MB;
int tem_incremento_linha = bloco_verificado->coluna_inicial / COLUNA_MATRIZ;
if (tem_incremento_linha > 0)
{
    bloco_verificado->linha_inicial += LINHA_MB;
    bloco_verificado->coluna_inicial = 0;
}

contador_blocos++;

// Fim região crítica
pthread_mutex_unlock(&mutex_bloco);
```

Imagem 8: região crítica de controle dos macroblocos.

```
if (contador_primos_local > 0)
{
    // Região crítica do contador de primos
    pthread_mutex_lock(&mutex_count_primos);
    total_numeros_primos += contador_primos_local;
    contador_primos_local = 0;
    pthread_mutex_unlock(&mutex_count_primos);
}
```

Imagem 9: região crítica que soma a quantidade de números primos.

Ao observar os demais testes com macroblocos maiores, pode-se perceber uma grande melhora em relação ao uso de macroblocos de dimensões 1x1. Isso vale mesmo para os testes realizados com apenas 1 thread. Mas o mais interessante é perceber que o ganho é muito significativo ao trabalhar com mais 2 threads ou mais. A coluna “Média Thread > 1” é referente a média dos tempos de macroblocos de dimensões diferentes, para os testes realizados com macroblocos maiores que 1x1.

É importante pontuar que, de acordo com os testes, quanto mais próximo do número de núcleos da CPU (físicos e lógicos) do PC1 e do PC2, melhor a performance na execução. Isso ocorre por ser feita uma melhor divisão das tarefas e assim todos os núcleos estão sempre ocupados até o fim da contagem. Dada as informações da tabela, montamos gráficos comparativos entre os números de threads para uma melhor visualização do desempenho.

Tempo de execução por número de threads do PC1

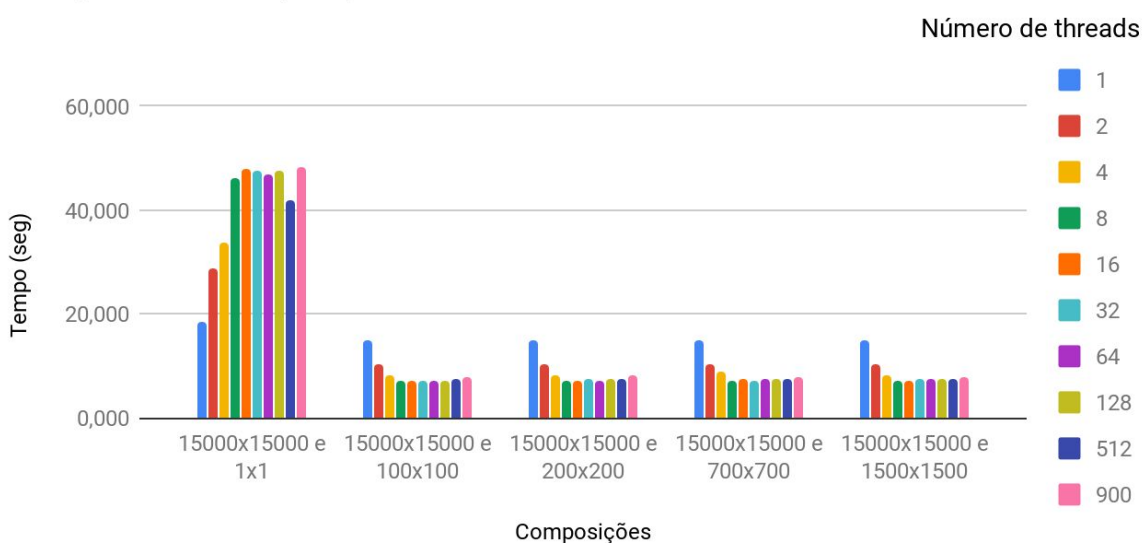


Gráfico 1: Tempo de execução por número de threads do PC1.

Tempo de execução por número de threads do PC2

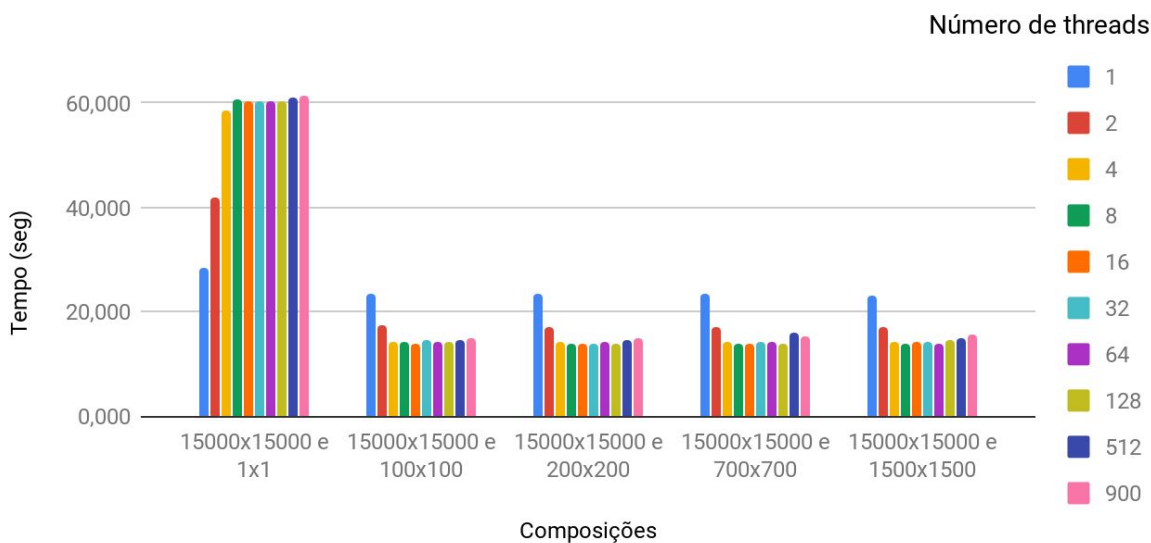


Gráfico 2: Tempo de execução por número de threads do PC2.

Apesar do leve aumento, esperávamos um tempo muito maior do que conseguimos com um maior número de threads como 512 e 900, o que não ocorreu. Além disso, por algum motivo, não conseguimos criar mais que 916 threads simultâneas e nem matrizes maiores que 20000 x 20000, pois, aparentemente, o SO limitou o número de threads e o máximo de consumo de memória pelo programa.

Com o auxílio da ferramenta *Profile Performance*, presente no Visual Studio, pudemos verificar o uso de CPU do algoritmo, além dos possíveis gargalos existentes no código.

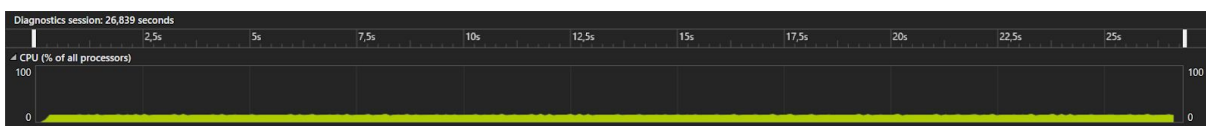


Imagem 10: uso de CPU em modo serial. [Link](#)

No gráfico da imagem 10 pode-se afirmar que o uso de CPU é quase contínuo, sem grandes alterações. Isso ocorre devido a forma da contagem. O algoritmo percorre a linha por linha da matriz, um elemento por vez, de forma padrão.

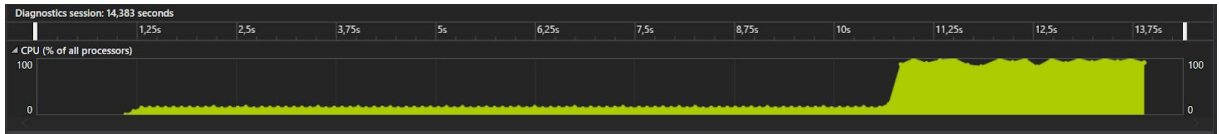


Imagem 11: uso da CPU em modo paralelo, 8 threads. [Link](#)

Já no gráfico da imagem 11 podemos perceber que aos 11 segundos, ao iniciar a criação e o processamento das threads, o consumo da CPU aumenta quase ao máximo possível. Isso ocorre devido ao processamento das várias threads em paralelo.

```

295      /* Looping pelo tamanho de elementos do macrobloco */
296      /* Para se leu todos os elementos ou se acabou a quantidade de linhas válidas da matriz */
297
143 (0,40%) 298      for (int i = 0; i < quantidade_posicoes_macrobloco && linha_index < LINHA_MATRIZ; i++)
299      {
25032 (69,62%) 300          if (is_primo(matriz[linha_index][coluna_index++])) contador_primos_local++;
301
583 (1,62%) 302          if (coluna_index > COLUNA_MATRIZ || coluna_index >= bloco_local->coluna_final)
303          {
304              linha_index++;
474 (1,32%) 305              coluna_index = bloco_local->coluna_inicial;
306          }
307      }

```

Imagem 12: gargalo do uso da CPU na função `is_primo()`.

Na imagem 12 as porcentagens exibidas à esquerda das linhas é referente a uma pontuação dada pelo Visual Studio de acordo com o uso da CPU. Como podemos observar, a função `is_primo()` é a função que mais consome durante todo o processamento. Após algumas tentativas de melhora, na tentativa de remover comparações e loopings, esse foi o melhor resultado obtido.

Ganho percentual PC1

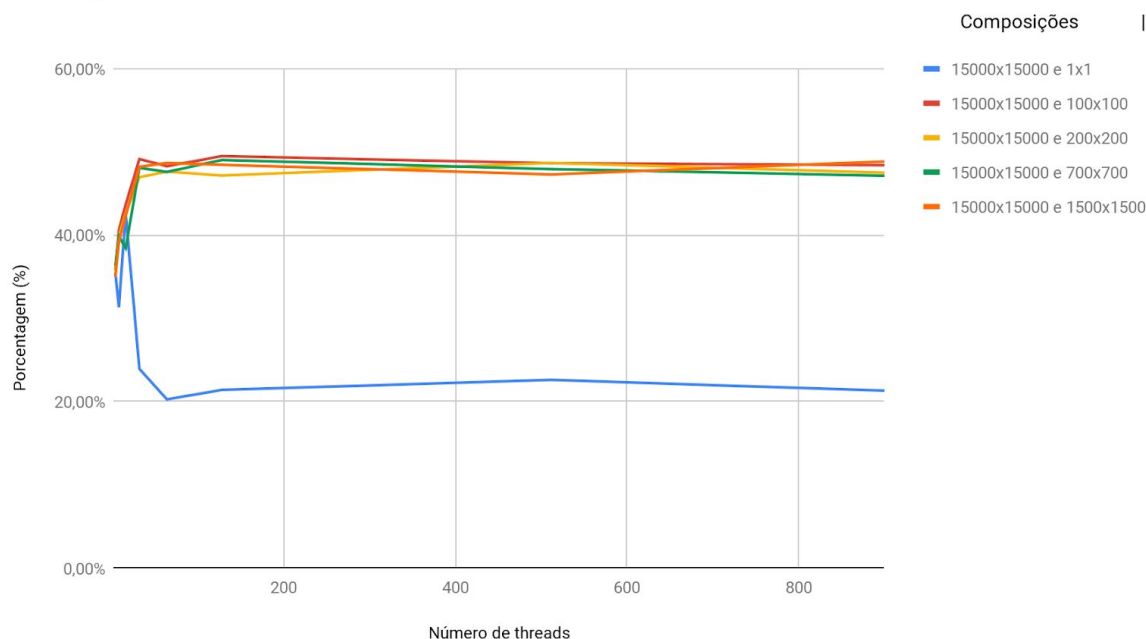


Gráfico 3: Ganho percentual PC1 em relação ao PC2.

O gráfico 3 nos mostra com mais clareza em porcentagem o quanto melhor é a performance do PC1 em relação ao PC2. O cálculo para relacionar a discrepância do tempo entre o PC1 e PC2 é $\text{Ganho} = 1 - (\text{Tempo PC1} / \text{Tempo PC2})$.

Para os testes com macrobloco de dimensões 1x1 o ganho de performance foi em torno de 20%, e para os demais os resultados beiram uma performance de quase 50% superior. Estes resultados se dão por causa da diminuição da litografia de produção da CPU de 22nm para 14nm além da maior quantidade de núcleos do PC1 que são 4 núcleos reais, contra 2 núcleos reais do PC2. Isso somado à tecnologia de hyperthread que aumenta ainda mais a quantidade PC1 e possibilita o uso de 8 núcleos lógicos contra 4 do PC2.

Apesar dos testes não tomarem proveito da capacidade máxima de RAM dos computadores, 16GB e 8GB respectivamente, o fato de um ser uma DDR4 contra DDR3 influencia no desempenho pelo aumento na frequência e a pouca mudança na latência.

Um dos requisitos era testar o código novamente, porém, desativando os mutexes que controlam as regiões críticas. Ao fazer isso, é facilmente notável que a contagem resultaria em um valor incorreto. Porém, também é importante frisar que, quando o trabalhamos com

macroblocos muito grandes, como 5000 x 5000, a chance de haver divergência é menor, pois pode ser que não haja concorrência para alterar as variáveis globais. Contudo, quanto menor o macrobloco, maior a chance de haver concorrência, e assim, maior a chance de divergências no resultado.

3 CONCLUSÃO

Com a implementação do algoritmo deste trabalhos, pudemos verificar precisamente os cinco principais desafios na programação multi-core. Tivemos que ter cuidado na divisão de atividades e no equilíbrio para que nenhuma thread ficasse sobrecarregada. Além da correta divisão dos dados a serem processados, a dependência dos dados da contagem que são compartilhados entre as threads. Por fim, o maior dos desafios que é o teste e a depuração do algoritmo, no qual se mostrou de grande complexidade, e de necessária grande repetição. Encontrar falhas fica muito mais difícil, o programa pode estar funcionando perfeitamente para determinados cenários e apresentando erros em outros por um simples descuido do programador.

A construção do algoritmo também se mostrou mais complicada no quesito da lógica. É necessário pensar em como o código compartilhador vai funcionar e como tratar das regiões críticas, o que é uma dificuldade a mais em relação ao código serial. Isso também requer um conhecimento sobre a manipulação da biblioteca Pthreads ou de outra biblioteca de threads para criar e manipular threads e semáforos.

Repartir as tarefas para várias linhas de execução paralelas se mostrou vantajoso na maioria dos casos testados porém é importante ressaltar que é necessário encontrar um ponto de equilíbrio entre tamanho da tarefa e quantidade de threads e que não é necessário criar mais threads do que a arquitetura da CPU consegue executar simultaneamente.

Este trabalho fez com que o aprendizado na disciplina de Sistemas Operacionais fosse colocado em prática, e assim pudemos testar nosso domínio sobre o que foi lecionado. E o melhor disso foi unir nosso conhecimento de programação à disciplina de Sistemas Operacionais, e assim concomitamos assuntos que são abordados em outras disciplinas do curso.

Por fim, temos que pontuar que o uso da IDE Visual Studio foi de extrema ajuda, pois conta com ótimas ferramentas de debug para auxiliar na solução dos problemas. A possibilidade de executar o algoritmo linha por linha e analisar o estado atual das variáveis é um recurso fantástico. A IDE também oferece recursos para mostrar uso da CPU e de memória durante a execução, isso molda um conhecimento mais sólido sobre o impacto das ações do programador na execução do programa.