FACULDADE DE ENGENHARIA MECÂNICA

FACULDADE DE COMPUTAÇÃO

GRADUAÇÃO EM ENGENHARIA MECATRÔNICA

SISTEMAS OPERACIONAIS

ALEXANDRE MENDES LANHOSO
LUÍS FELIPE COSTA FERNANDES DE MENEZES
PAULO RICARDO BUENO BRANQUINHO

Multithreading C Programming for Multicore Computer MTP - MC

TRABALHO DE CONCLUSÃO DE DISCIPLINA

Uberlândia

2023

Sumário

Resumo

Neste relatório será apresentado um estudo de caso que investigou as diferenças de desempenho na execução de um processo de multiplicação de matrizes quando implementada executando em single, double e quad-thread, com e sem afinidade de processador.

O objetivo dessa análise é verificar o impacto do uso de múltiplas threads no desempenho geral do cálculo da multiplicação de matrizes e com isso observar as vantagens da paralelização dessa tarefa.

Com base nisso, a multiplicação de matrizes foi realizada de elemento a elemento no cenário single thread, enquanto no cenário dual thread e quad thread, o cálculo foi dividido em seções independentes calculadas por múltiplas threads e reagrupados na matriz resultante após o término.

Essa abordagem resultou em redução significativa no tempo de execução, especialmente em matrizes grandes, devido ao aproveitamento dos múltiplos cores disponíveis no processador.

1. Introdução

Um dos conceitos fundamentais em sistemas operacionais é denominado **processo**, de modo geral, ele é uma representação abstrata de um programa em execução. Geralmente, um processo consiste em um programa com um único thread de execução, acompanhado de seu próprio espaço de endereçamento e informações relevantes, como o contador de programa, ponteiro de pilha e registradores. De uma forma mais simples para a compreensão, um processo se assemelha à um programa com sua função principal, denominada main, sendo executado sequencialmente, instrução por instrução. No entanto, em certos sistemas, é possível criar vários threads dentro do mesmo processo, permitindo que múltiplos fluxos de execução ocorram simultaneamente no mesmo espaço de endereçamento.

É importante ressaltar também que threads e processos são conceitos diferentes. Conforme mencionado anteriormente, um processo atua como um “contêiner” para recursos (código e dados) e tem uma identidade distinta, enquanto as threads são criadas no contexto de um processo e compartilham o mesmo espaço de endereço. É importante notar que as threads não são independentes como os processos, uma vez que, embora compartilhem o espaço de endereço dentro de um processo, cada thread possui mecanismos para gerenciar seu próprio contexto de execução.

Em resumo, processos e threads coexistem e trabalham em conjunto, com os processos fornecendo o ambiente global e as threads realizando tarefas específicas de forma concorrente e cooperativa. Essa combinação de processos e threads permite a criação de aplicativos multitarefa e multithreaded mais eficientes e responsivos .

Portanto, a partir da definição desses conceitos e entendendo um pouco mais sobre como funcionam processos e threads, foi desenvolvido o estudo de caso de uma multiplicação entre matrizes para avaliar as vantagens de uma programação paralelizada e como isso influencia no desempenho e na eficiência da resolução de um problema computacional bastante custoso para a CPU como este.

2. Aplicações utilizando Multithreading

O multithreading é um conceito poderoso que permite que programas executem múltiplas threads simultaneamente, aproveitando o poder de processamento de computadores multicore (que possuem mais de um núcleo de processamento). Com isso, é possível realizar várias tarefas ao mesmo tempo, como navegar na internet enquanto executa uma verificação de vírus ou baixar arquivos em segundo plano enquanto joga um jogo, permitindo que diferentes partes de um programa sejam executadas melhorando o desempenho e a eficiência.

Entretanto, apesar de citarmos acima uma execução “simultânea” de threads é importante separar dois conceitos na execução multithreading, que são os de execução paralela e execução simultânea. A execução paralela quer dizer que o sistema está cuidando de mais de uma thread em um determinado período de tempo, mas isso ocorre em diferentes núcleos de processamento do processador.

Já a execução simultânea, mesmo em sistemas com vários núcleos, não é necessariamente simultânea, pois o sistema operacional é responsável por alternar rapidamente entre as threads em execução, atribuindo pequenos intervalos de tempo (time slices) para cada uma delas, dando a ilusão de simultaneidade, ou seja, isso significa que o sistema tratará de uma thread por vez, e por isso, terá eficiência para mover-se rapidamente entre duas ou mais threads.

A seguir, temos alguns exemplos de aplicações do nosso cotidiano que utilizam esse conceito de programação multithread e como elas se beneficiam com este modelo de execução, que possibilita que as threads sejam criadas e executadas de forma independente nos processos, mas compartilhando recursos de forma simultânea o que acaba garantindo mais performance no sistema.

2.1. Editor de Texto (Microsoft Word)

O Microsoft Word é um dos principais editores de texto utilizados em todo o mundo. Ele faz parte do pacote de aplicativos de produtividade da Microsoft conhecido como Pacote Office e é amplamente reconhecido por sua versatilidade, recursos avançados e facilidade de uso.

Assim como outros editores de texto, o Word aplica a técnica de programação multithreading de várias maneiras para melhorar a experiência do usuário. Abaixo, é possível ver alguns desses exemplos:

- **Interface do usuário responsiva:** Um dos principais objetivos do multithreading em um editor de texto é garantir que a interface do usuário permaneça responsiva durante operações demoradas, como abrir ou salvar arquivos grandes, realizar verificações ortográficas ou executar

formatações complexas. Ao executar essas tarefas em threads separadas, o programa pode continuar a responder aos comandos do usuário em uma thread separada, garantindo que a interface não fique bloqueada;

- **Renderização de documentos:** O Word e outros editores de texto precisam exibir o conteúdo do documento de maneira rápida e precisa. Isso envolve a renderização de texto, formatação, exibição de imagens e muito mais. O uso do multithreading permite que diferentes partes da renderização sejam executadas em paralelo, acelerando o processo de exibição e tornando a rolagem e a navegação pelo documento mais suaves;
- **Verificação ortográfica e gramatical:** Os editores de texto geralmente possuem recursos de verificação ortográfica e gramatical que analisam o texto inserido pelo usuário em busca de erros. Essas verificações podem ser demoradas, especialmente para documentos extensos. Com o multithreading, a verificação ortográfica e gramatical pode ser realizada em threads separadas, permitindo que o usuário continue digitando ou editando o texto sem interrupções.

O objetivo geral é melhorar a capacidade de resposta, o desempenho e a eficiência do programa, permitindo que diferentes partes dos aplicativos sejam executadas em paralelo para aproveitar ao máximo os recursos do hardware e proporcionar uma experiência mais fluida ao usuário.

A imagem abaixo ilustra de maneira simplificada como diferentes instruções e tarefas são distribuídas entre as threads de um dos vários processos de um editor de texto:

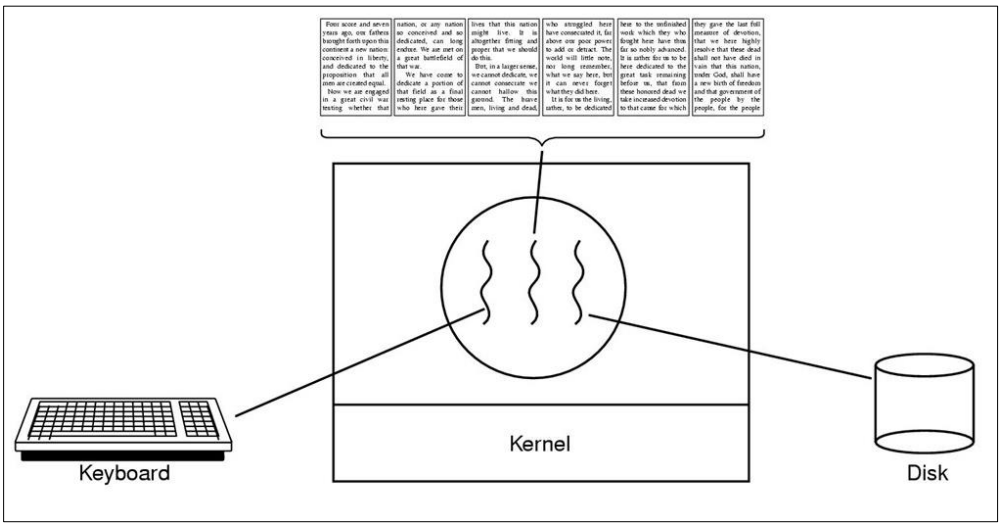

Figura 1 – Diagrama demonstrativo de como as threads funcionam em um editor de texto.
Fonte: TANENBAUM, 2000.

2.2. Navegador (Browser)

Atualmente, existem diversos navegadores disponíveis que oferecem uma experiência de navegação na web mais ágil e eficiente. Esses navegadores, como o Google Chrome, Mozilla Firefox, Microsoft Edge, Safari e Opera, utilizam uma variedade de técnicas, incluindo a programação multithreading, para aprimorar o desempenho e a capacidade de resposta durante a navegação.

Uma das principais áreas em que os navegadores utilizam a programação multithreading é o carregamento de páginas da web. Quando visitamos um site, o navegador precisa buscar vários recursos, como imagens, arquivos CSS e scripts JavaScript, para exibir corretamente a página. Ao dividir essas tarefas em threads separadas, o navegador pode buscar e processar esses recursos em paralelo, acelerando significativamente o tempo de carregamento.

A implementação específica do multithreading pode variar entre os diferentes navegadores disponíveis hoje em dia, mas entre as principais aplicações de threads em navegadores temos:

- **Thread principal (main thread):** O navegador inicia com um thread principal que é responsável por processar eventos de entrada do usuário, como cliques, digitação e movimentos do mouse. Essas interações do usuário são convertidas em ações, como carregar uma página, fazer solicitações de rede ou atualizar a interface do usuário;
- **Thread de renderização (rendering thread):** Cada aba ou guia aberta em um navegador geralmente possui sua própria thread de renderização. Essa thread é responsável por processar e renderizar o conteúdo HTML, CSS e JavaScript de uma página da web. Ele realiza tarefas como analisar o código HTML, executar scripts JavaScript e aplicar estilos CSS para exibir o conteúdo na tela**;**
- **Thread de rede (network thread):** Uma thread separada é usada para realizar operações de rede, como fazer solicitações HTTP para carregar recursos da web, como imagens, arquivos CSS, JavaScript, etc. Essa thread lida com a comunicação de rede, gerenciamento de conexões e transferência de dados;
- **Thread de gerenciamento de eventos (event handling thread):** Essa thread é responsável por manipular eventos de sistema, como eventos de temporizador, eventos de mouse e teclado, eventos de redimensionamento de janelas, etc. Ela garante que os eventos sejam processados e distribuídos para as threads apropriadas para execução.

- **Threads adicionais:** Além das threads mencionadas acima, um navegador pode usar threads adicionais para realizar tarefas específicas, como processamento de imagens, decodificação de vídeo, execução de tarefas em segundo plano, entre outros.

Vale ressaltar também que a configuração de threads em um navegador pode variar conforme o projeto e as otimizações implementadas por cada equipe de desenvolvimento.

Além disso, os navegadores modernos empregam diversas técnicas, como o isolamento de processos (utilizando processos separados para cada guia) e avançados métodos de agendamento de threads, com o intuito de aprimorar a segurança, estabilidade e desempenho global durante a navegação na web.

Com essa informação, torna-se natural quando o monitor de recursos ou o gerenciador de tarefas do sistema operacional é aberto e nele temos diversos processos para um navegador que está sendo executado. Como é possível ver na figura abaixo:

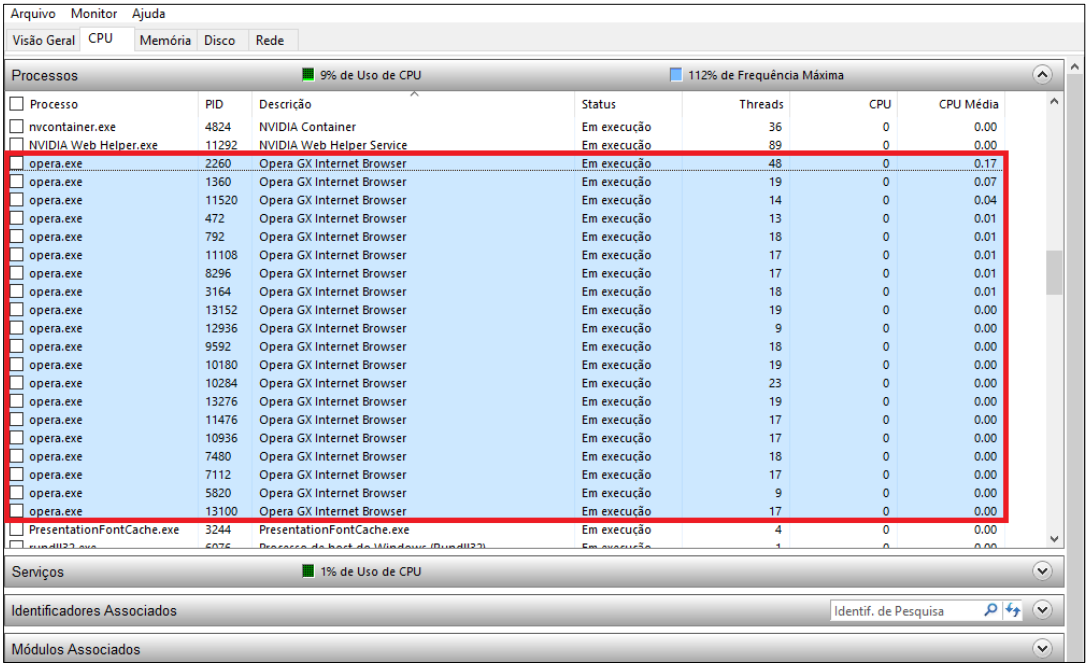

Figura 2 – Imagem do Monitor de Recursos do Windows 10, mostrando os processos de um navegador em execução com algumas abas criadas.

Fonte: De autoria própria, 2023.

Tudo isso é justificado para garantir, por exemplo, estabilidade para a aplicação, pois se um único site ou aba em um navegador causar um travamento ou problema de desempenho, a separação de processos garante que apenas o processo específico associado a essa aba seja afetado. Os outros processos do navegador continuarão funcionando normalmente, permitindo que você feche ou atualize a aba problemática sem prejudicar a experiência de navegação como um todo.

3. Desenvolvimento

Com a finalidade de obter dados para analisar quantitativamente o caso estudado, um plano experimental foi elaborado: executar cada uma das situações cinco vezes, sendo elas single, dual e quad-thread com e sem afinidade de processador, para obter estatísticas mais significativas, mas mantendo um tempo de experimentação plausível. Também foi adicionado um código de carga de fundo, responsável por emular outros possíveis processos executando na máquina multiprogramada.

Este planejamento tornou necessário não só a construção dos códigos em C responsável por cada situação, como também uma automação do teste, realizada através de shell scripting.

Inicialmente, foi desenvolvido o código base, como o da Figura 3, o qual realiza uma multiplicação de duas de matrizes quadradas (M1 e M2) de 49.000.000 elementos inicializados com valor 1, de forma single thread, contando o tempo da multiplicação e o salvamento em um arquivo. Este código também contou com argumentos de linha de comando para diferenciar entre as execuções dos diferentes casos.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 7000

int main(int argc, char *argv[]) {
    static int M1[SIZE][SIZE], M2[SIZE][SIZE], M3[SIZE][SIZE];
    struct timespec start_time, end_time;
    double execution_time;

    // Inicializar M1 e M2 com valor 1
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            M1[i][j] = 1;
            M2[i][j] = 1;
        }
    }

    // Inicializar M3 com valor 0
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            M3[i][j] = 0;
        }
    }

    // Multiplicação
    clock_gettime(CLOCK_REALTIME, &start_time);

    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            for (int k = 0; k < SIZE; k++) {
                M3[i][j] += M1[i][k] * M2[k][j];
            }
        }
    }

    clock_gettime(CLOCK_REALTIME, &end_time);
    execution_time = (end_time.tv_sec - start_time.tv_sec) +
                     (end_time.tv_nsec - start_time.tv_nsec) / 1e9;

    // Salvar o tempo levado
    FILE* file = fopen("/home/kirlin/multithreading-C-multicore-computer/results/resultado_single_thread.txt", "a");
    if (file == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    printf("Tempo levado: %.5f segundos\n", execution_time);
    if (atoi(argv[2]) == 0){
        fprintf(file, "Execução sem afinidade %s: %.5f segundos\n", argv[1], execution_time);
    }
    else{
        fprintf(file, "Execução com afinidade %s: %.5f segundos\n", argv[1], execution_time);
    }
    fclose(file);

    return 0;
}
```

Figura 3 – Código base, multiplicação single thread salvando tempo de execução.
Fonte: De autoria própria, 2023.

Sobre o código acima, vale ressaltar a necessidade da inicialização das matrizes como static, para um grande número de elementos. Isso se dá devido à um segmentation fault por um overflow na pilha, pois $3 \cdot 7000 \cdot 7000 \cdot \text{sizeof}$ excede o tamanho da pilha de memória alocada para o processo. Ao usar static na declaração das matrizes, altera-se o local de armazenamento dessas variáveis, para área de armazenamento estático, ou seja, a área de dados, que possui um tamanho superior à pilha.

Posteriormente, foi realizada a implementação de múltiplas threads no código, através da biblioteca pthread, dividindo o trabalho da multiplicação em seções de colunas para cada thread. As principais diferenças deste código de múltiplas threads para o código base está evidenciado abaixo, na Figura 4.

```c
#include <pthread.h>
#define NUM_THREADS 4
.
.
.
// Estrutura para armazenar os argumentos das threads
typedef struct {
    int start_row;
    int end_row;
} ThreadArgs;

// Matrizes globais
static int M1[SIZE][SIZE], M2[SIZE][SIZE], M3[SIZE][SIZE];

// Função executada por cada thread
void* multiply(void* arg) {
    ThreadArgs* args = (ThreadArgs*)arg;
    int start_row = args->start_row;
    int end_row = args->end_row;

    for (int i = start_row; i < end_row; i++) {
        for (int j = 0; j < SIZE; j++) {
            for (int k = 0; k < SIZE; k++) {
                // Cada thread atualiza sua parte correspondente de M3
                M3[i][j] += M1[i][k] * M2[k][j];
            }
        }
    }

    pthread_exit(NULL);
}

.
.
.

int main(int argc, char* argv[]) {
    pthread_t threads[NUM_THREADS];
    ThreadArgs thread_args[NUM_THREADS];
    .
    .
    .

    clock_gettime(CLOCK_REALTIME, &start_time);

    // Criar as threads e atribuir a cada uma a responsabilidade de calcular uma parte de M3
    for (int i = 0; i < NUM_THREADS; i++) {
        int start_row = i * (SIZE / NUM_THREADS);
        int end_row = (i + 1) * (SIZE / NUM_THREADS);

        thread_args[i].start_row = start_row;
        thread_args[i].end_row = end_row;

        pthread_create(&threads[i], NULL, multiply, (void*)&thread_args[i]);
    }

    // Aguardar a finalização das threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    clock_gettime(CLOCK_REALTIME, &end_time);
    .
    .
    .

    return 0;
}
```

Figura 4 – Implementação de múltiplas threads na multiplicação de matrizes.
Fonte: De autoria própria, 2023.

Por fim, um código de carga de fundo foi construído, com um for interminável, o qual executaria em conjunto com cada teste da multiplicação, como na Figura 5.

```
int main(){
    for (;;)
    {
    }

    return 0;
}
```

Figura 5 – Código de carga de fundo.
Fonte: De autoria própria, 2023.

Com todos os códigos do experimento desenvolvidos, os testes foram automatizados através de scripts do shell, que, no geral, compilam o código em C da situação, limpam o arquivo .txt que salva os resultados, realizam a execução cinco vezes para cada caso, passando como argumento de linha de comando o número da execução e um indicador de situação (com e sem afinidade), como na Figura 6.

```
#!/bin/bash

gcc codes/quad_thread.c -o execs/quad_thread.o -pthread
echo "" > results/resultado_quad_thread.txt

for i in 1 2 3 4 5; do
  ./execs/quad_thread.o $i 0
done

sync; sync; sync;

for i in 1 2 3 4 5; do
  taskset -c 0,1,2,3 ./execs/quad_thread.o $i 1
done
```

Figura 6 – Exemplo da estrutura base dos scripts de shell utilizados para automação.
Fonte: De autoria própria, 2023.

Dois comandos específicos são utilizados nessa parte: sync e taskset. O primeiro é utilizado para forçar a gravação de todos os dados armazenados em memória cache para os dispositivos de armazenamento e, assim, eliminar quaisquer influências de testes anteriores no próximo teste. Já o segundo permite definir afinidades de CPU para processos específicos, ou seja, controlar quais núcleos de CPU um determinado processo pode utilizar.

Por fim, a automação de cada cenário de threads foi executada, em conjunto com o código de carga de fundo, em uma máquina com as seguintes especificações:

- Ubuntu 20.04 Nativo;
- Processador Intel® Core™ i5-9400F 6 núcleos;
- Frequência baseada em processador de 2.90 GHz;
- Cache de 9 MB Intel® Smart Cache;
- 16GB de RAM.

4. Análise e Resultados

O experimento descrito na seção anterior gerou uma sequência de dados, descritos na Tabela 1 abaixo, com suas estatísticas de média e desvio padrão amostral.

Tabela 1 – Dados experimentais e suas médias e desvio padrão.

	Tempo da situação (s)					
	Single Thread		**Dual Thread**		**Quad Thread**	
Exec.	Sem afinidade	Com afinidade	Sem afinidade	Com afinidade	Sem afinidade	Com afinidade
1	2771,69	2852,06	1332,64	1345,19	686,27	973,98
2	2806,82	2833,99	1332,48	1344,91	680,68	969,66
3	2673,04	2825,06	1333,56	1343,94	686,87	974,93
4	2827,87	2820,27	1362,24	1344,11	720,85	979,42
5	2785,74	2829,28	1344,81	1343,41	681,74	970,14
$\bar{t}$	2773,03	2832,13	1341,15	1344,31	691,28	973,63
σ_t	59,82	12,24	12,88	0,73	16,75	3,98

A fim de melhorar a visualização desses dados, um gráfico relacionando o número de threads com o tempo levado foi construído, apresentando as médias e desvio padrão com e sem afinidade de processador.

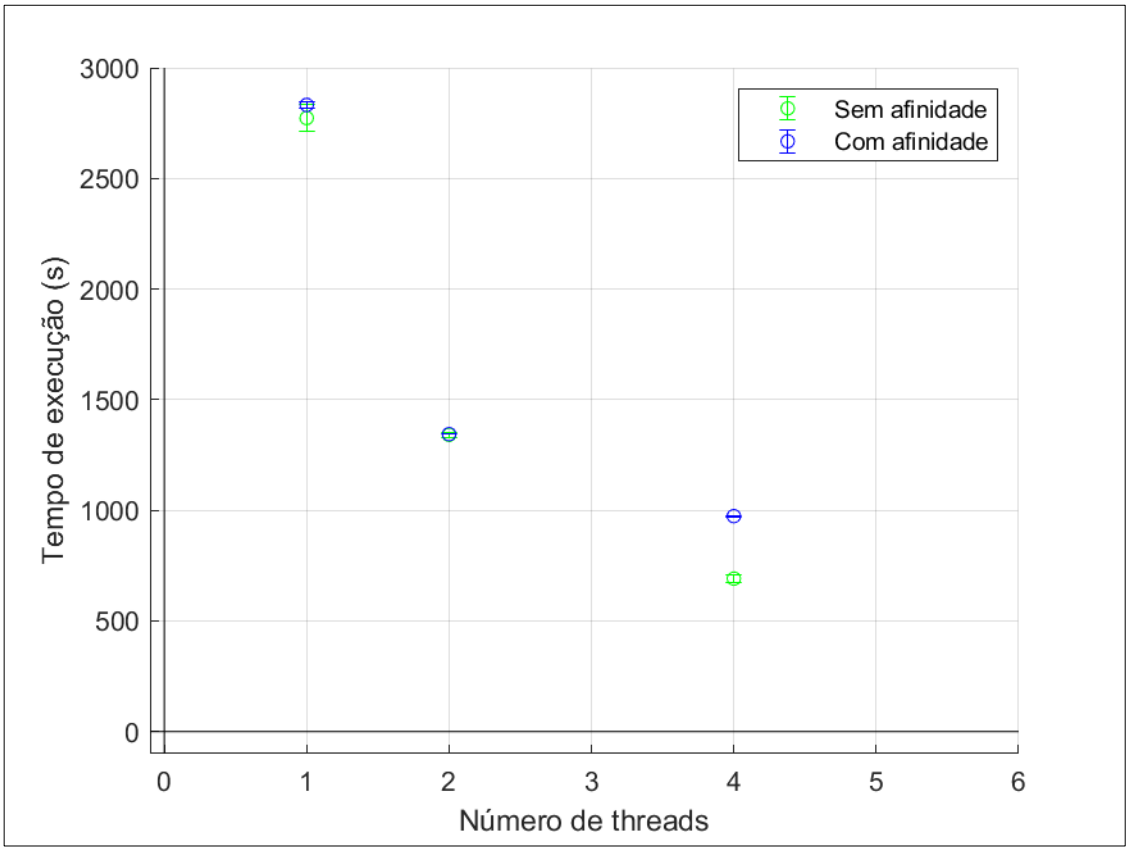

Figura 7 – Relação entre tempo de execução e número de threads.
Fonte: De autoria própria, 2023.

Ao analisar tanto os dados quanto o gráfico é possível observar uma relação na qual ao dobrar o número de threads executando a tarefa, reduz o tempo de execução à metade, especialmente para a situação sem afinidade de processador.

Também é possível notar, para a máquina utilizada no teste, atribuir afinidade de processador trouxe um impacto negativo na performance, especialmente para a computação com quatro threads. Isso é um possível indicador da melhor capacidade de gestão dos recursos de processamento do escalonador de processos do Sistema Operacional.

5. Conclusão

A melhora de performance para aplicações de alta carga computacional é uma problemática complexa. A segmentação da carga de trabalho e execução em um ambiente computacional multiprocessado abre possibilidades para a melhoria do desempenho da aplicação. Nesse contexto, uma simulação realizando uma multiplicação de matrizes de grandes dimensões com carga de fundo se demonstrou um exercício apto para a simulação das condições a serem estudadas.

Os resultados evidenciam o comportamento esperado para as condições propostas. Com o tempo de execução da aplicação diminuindo proporcionalmente à quantidade threads criadas, com a execução em single thread como referência, a execução em dual thread executou em metade o tempo, e em quad thread um quarto do tempo original, para o cenário sem afinidade de processador.

Para o cenário com afinidade de processador, observa-se que apesar de causar a impressão de seguir a mesma tendência que o cenário anterior, na execução em quad thread observa-se uma grande discrepância entre os tempos de execução com e sem afinidade de processador.

Tal comportamento anômalo pode ser atribuído a certas características intrínsecas dos sistemas operacionais de uso geral modernos. Especificamente, as otimizações de balanceamento de carga no escalonador de processos desempenham um papel significativo. Quando há a execução concorrente de uma carga de fundo e outros processos do sistema operacional, as threads da aplicação em questão provavelmente competem pelo tempo de processador se a afinidade for definida.

Por outro lado, quando a afinidade não é definida, é provável que a aplicação seja realocada várias vezes entre processadores durante sua execução, com o objetivo de equilibrar a carga em cada processador e minimizar a utilização de cada núcleo. Essa abordagem visa reduzir o tempo de execução, especialmente para aplicações com baixa prioridade de execução e característica de uso intensivo de CPU.

Em vista disso, a subdivisão de macrotarefas em subtarefas executadas por diferentes threads é uma boa solução para problemas análogos ao proposto neste trabalho, visto que melhor utiliza os recursos computacionais disponíveis e resulta em melhor desempenho da aplicação, sem causar ônus exagerado em contrapartida.

Bibliografia

1. TANENBAUM, A.S. and WOODHULL, A. S. Sistema Operacionais – Projeto e Implementação. Bookman, 2000.

2. LEWIS, B. and BERG, D.J. Threads primer: a guide to multithreaded programming. New Jersey, Prentice-Hall, 1996.

3. STALLINGS, W. Operating Systems – Internals and Design Principles, 3a ed. Englewood Cliffs, NJ, Prentice Hall, 1998.

4. SILBERSCHATZ, A.; GALVIN, P.B.; GAGNE, G. Fundamentos de Sistemas Operacionais, 6a ed. Editora Campus, 2004.

Apêndice

Link do GitHub com os códigos implementados no relatório:

- https://github.com/luis-cmenezes/multithreading-C-multicore-computer