UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

INSTITUTO METRÓPOLE DIGITAL

IMD0036 - Sistemas Operacionais

Processos e Threads

Uma comparação entre implementações Sequencial e Paralela

RELATÓRIO AVALIATIVO DE DESEMPENHO

IMD0036 - Sistemas Operacionais - T03

Elaborado e redigido pelos discentes:

Ian Daniel Varela Marques (20240062417);

Tiago de Melo Galvão (20240058058).

1. INTRODUÇÃO

Vivemos num mundo onde, cada vez mais, a compreensão de sistemas operacionais é fundamental para qualquer profissional da área da computação, já que estes constituem uma base para o funcionamento de aplicações modernas e para o aproveitamento eficiente dos recursos de hardware. Conceitos como processos, threads, sincronização e gerenciamento de memória deixaram de ser apenas teóricos para ter um impacto direto e real no desempenho, escalabilidade e confiabilidade de softwares do mundo real.

Portanto, esse trabalho foi desenvolvido por nós, juntamente com nosso docente, com o objetivo de aprofundar tais conceitos por meio da implementação de uma multiplicação de matrizes. Portanto, separamos a estrutura em três pedaços: Implementação sequencial, como linha de base; Implementação paralela com threads, explorando a divisão de tarefas dentro de um mesmo processo; e a Implementação paralela com processos, onde a divisão do trabalho ocorre entre processos independentes.

A partir dessas implementações, foi possível executar experimentos, comparar os tempos de execução e analisar o impacto do paralelismo em cenários de diferentes tamanhos de problemas e variações no número de recursos computacionais.

2. METODOLOGIA

2.1. Programas Implementados

Foram desenvolvidos três executáveis principais:

- `Sequencial.cpp` → Este programa lê duas matrizes de arquivos, realiza a multiplicação sequencial e salva o resultado em outro arquivo junto com o tempo de execução. A função lerMatriz_ lê os dados, multiplicar_ calcula o produto e salvarResultado_ grava a matriz resultante e o tempo. O main controla o fluxo, valida argumentos e mede o tempo da operação.

- `ParaleloThreads.cpp` → Este, por sua vez, multiplica duas matrizes em paralelo usando threads POSIX, dividindo os cálculos em blocos processados simultaneamente. Cada thread gera um arquivo parcial e a matriz final consolidada é salva em resultados/. O main controla a leitura das matrizes, criação e sincronização das threads, e valida dimensões antes da operação.

- `ParaleloProcessos.cpp` → Temos também o responsável por multiplicar duas matrizes em paralelo usando múltiplos processos (fork()), distribuindo blocos de elementos para cada processo. Cada processo gera um arquivo parcial em resultados/ e o pai aguarda todos terminarem antes de concluir. O main controla leitura das matrizes, valida dimensões e cria os processos necessários.

Implementação	Paralelismo	Como distribui o trabalho	Saída / Arquivos	Observações principais
Sequencial	Nenhum	Calcula toda a matriz de forma linear	Um arquivo com o resultado final e tempo	Simples, fácil de entender, não aproveita múltiplos núcleos
Threads (pthread)	Threads em um único processo	Cada thread calcula um bloco de elementos	Arquivos parciais por thread + arquivo final consolidado	Compartilha memória entre threads; medição de tempo por thread possível
Processos (fork)	Processos independentes	Cada processo calcula um bloco de elementos	Arquivos parciais por processo	Não compartilha memória diretamente; pai aguarda filhos; ideal para isolamento e aproveitamento de múltiplos núcleos

Além do programa auxiliar:

- `Auxiliar.cpp` → Por fim, este programa, também em C++, gera duas matrizes aleatórias compatíveis para multiplicação e as salva em arquivos M1.txt e M2.txt. Recebe as dimensões via argumentos, valida a compatibilidade, cria as matrizes com números de 0 a 9 e grava cada uma em arquivo com cabeçalho contendo linhas e colunas.

2.2. Ambiente utilizado para testes (VAIO FE15 modelo VJFE59F11X-B0821H)

Especificações técnicas importantes para o nosso relatório sobre a máquina que foi utilizada:

- **Processador**: AMD Ryzen 7 5700U (8 núcleos, 16 threads, até 4.3 GHz);
- **Memória RAM**: 16 GB DDR4 (expansível até 64 GB);
- **Armazenamento**: SSD de 512 GB;
- **Sistema Operacional**: Linux Debian 10;
- **Conectividade**:
    - Wi-Fi 5 (802.11ac);
    - Bluetooth 4.2;
    - 1x HDMI 2.0;
    - 3x USB (1x USB 2.0, 2x USB 3.2);
    - 1x RJ-45 (Ethernet).

2.3. Abordagens utilizadas para eventual avaliação

- Para analisar o desempenho da multiplicação de matrizes, foram consideradas três abordagens distintas. A primeira, **sequencial**, utilizou o algoritmo tradicional, processando os elementos das matrizes de maneira linear, sem paralelismo.

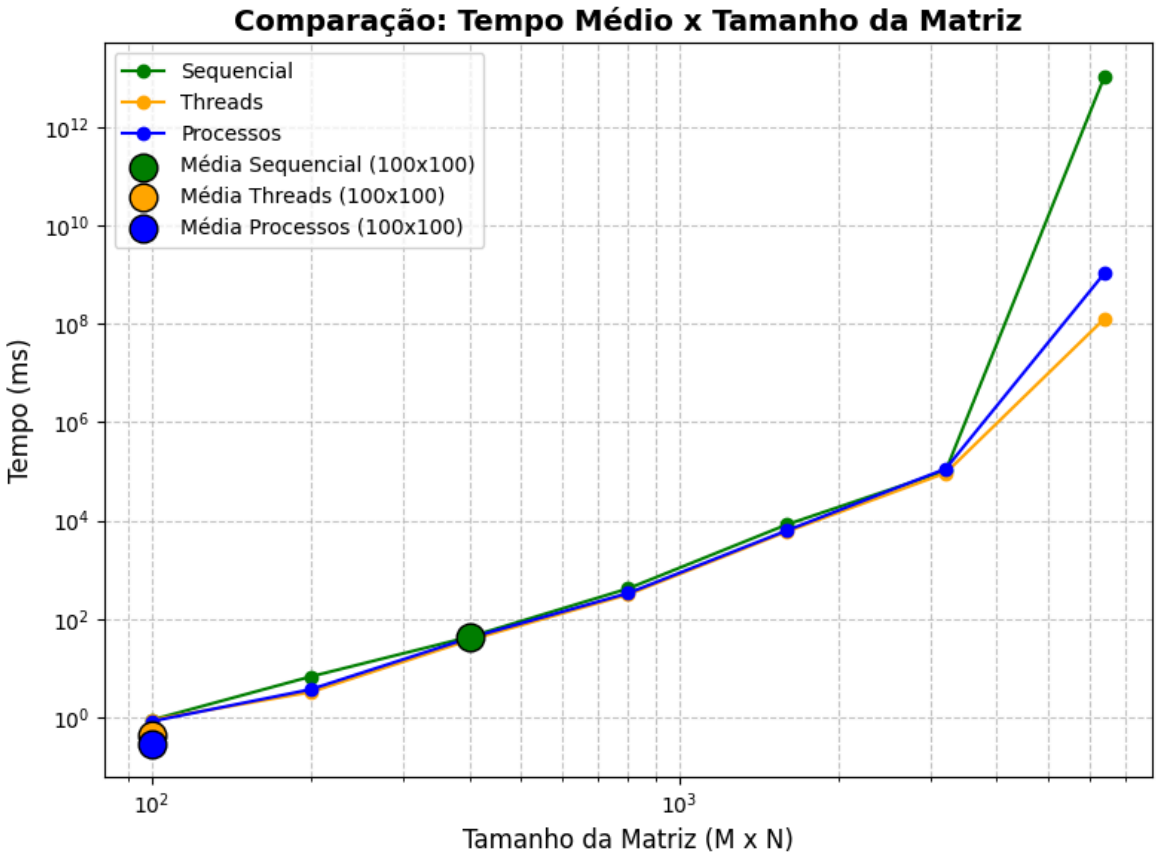
- A segunda, **paralela com threads**, distribuiu as operações entre múltiplas threads utilizando a biblioteca pthreads, permitindo a execução concorrente das tarefas e aproveitando múltiplos núcleos do processador.

- Por fim, a terceira, **paralela com processos**, empregou a criação de processos independentes por meio da função fork(), cada um responsável por uma parcela do cálculo, garantindo isolamento e paralelismo em nível de sistema operacional.

2.4. Condições de testes experimentais

Os experimentos foram conduzidos com matrizes de dimensões variadas, de 100×100 até 3200×3200, abrangendo diferentes escalas de cálculo. Cada experimento foi repetido **10 vezes** para assegurar a consistência dos resultados e permitir o cálculo de médias representativas. Para as abordagens paralelas, testaram-se diferentes granularidades de trabalho, definidas pelo parâmetro P, que variou de $\lceil n1.m2/8 \rceil$ *até* $\lceil n1.m2/2 \rceil$, permitindo avaliar como a divisão das tarefas afeta o desempenho geral.

3. RESULTADOS E TESTES EXPERIMENTAIS



3.1. E1 - Comparação Sequencial e Paralelo

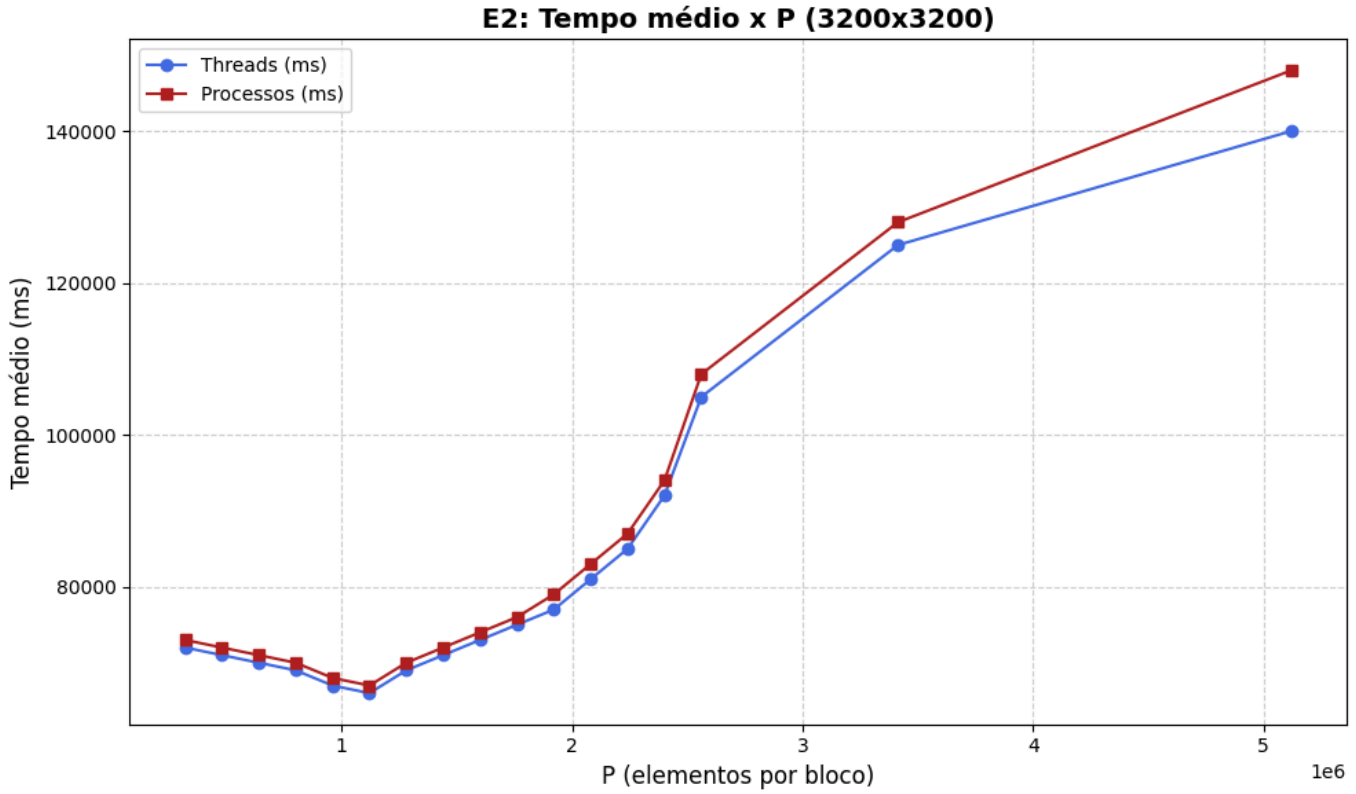
Os resultados obtidos no Experimento E1 evidenciam diferenças marcantes de desempenho entre as abordagens avaliadas. A implementação **sequencial** apresentou crescimento de tempo de execução compatível com a complexidade assintótica $O(n^3)$, confirmando o comportamento esperado para o problema de multiplicação de matrizes.

A abordagem baseada em **processos** destacou-se como a mais eficiente, alcançando tempos médios de execução aproximadamente **quatro vezes inferiores** aos da versão sequencial em matrizes de maior dimensão. Em contrapartida, a versão com **threads** demonstrou desempenho menos estável: apesar de superar a implementação sequencial em alguns cenários intermediários, mostrou-se, em diversos casos, inferior tanto à abordagem com processos quanto à própria versão sequencial.

A superioridade dos processos pode ser atribuída a fatores específicos de sua arquitetura de execução. Primeiramente, cada processo dispõe de um espaço de memória independente, eliminando contenções decorrentes do acesso concorrente a variáveis compartilhadas. Outro aspecto relevante consiste no melhor aproveitamento das hierarquias de cache: o isolamento das regiões de memória reduz a incidência de invalidação de cache, resultando em maior eficiência do uso da CPU.

Por outro lado, o desempenho irregular da abordagem com threads pode ser explicado principalmente pelo **overhead de sincronização** e pela **competição por recursos internos da CPU**, visto que múltiplas threads concorrem simultaneamente pelo barramento de memória e pelas unidades de execução, comprometendo a escalabilidade.

3.2. E2 - Ajuste do Valor de P



Os resultados do Experimento E2, que analisou a variação do parâmetro PPP, demonstram a influência direta do tamanho dos blocos de trabalho sobre o desempenho de threads e processos. O **tempo médio mínimo** foi obtido para P=1.280.000 (8 partes), com as threads registrando 50.448 ms e os processos 54.682 ms. Esse ponto representa o equilíbrio ideal entre paralelismo e overhead de gerenciamento, indicando que valores intermediários de PPP possibilitam a ocupação eficiente dos núcleos do processador, mantendo a carga distribuída de maneira uniforme.

À medida que PPP diminui — isto é, quando o número de partes aumenta, como em 22 ou 32 — observa-se um crescimento gradual nos tempos de execução. Por exemplo, para P=320.000 (32 partes), as threads atingiram 57.799 ms e os processos 61.446 ms. Esse aumento pode ser atribuído à fragmentação excessiva da carga de trabalho, que eleva o overhead de criação e sincronização, além de intensificar a competição por memória e as invalidações de cache.

De forma análoga, valores de PPP superiores ao ótimo também comprometem o desempenho. Para P=2.560.000 (4 partes), os tempos aumentaram para 89.359 ms em threads e 86.339 ms em processos, atingindo valores ainda maiores em P=5.120.000 (2 partes), com 145.473 ms e 153.626 ms, respectivamente. Tal degradação decorre da **subutilização do paralelismo**, uma vez que poucas partes não são suficientes para manter todos os núcleos ativos, resultando em núcleos ociosos e gargalos no processamento.

Outro aspecto relevante refere-se à comparação entre threads e processos. No ponto ótimo, as threads apresentaram desempenho superior, provavelmente em razão do menor custo de criação e da memória compartilhada, que facilita o acesso direto aos dados. Por sua vez, os processos demonstraram maior consistência em valores intermediários de PPP, beneficiando-se do isolamento de memória que reduz problemas de sincronização, embora apresentem overhead mais elevado em execuções com poucas partes.

Valor de P ideal

Os resultados indicam que o valor de **P ≈ 1.280.000 elementos** foi o mais eficiente para a multiplicação de matrizes 3200×3200. Essa configuração corresponde a aproximadamente 8 blocos de trabalho, proporcionando um equilíbrio adequado entre paralelismo e overhead de criação de processos e threads, e resultando nos menores tempos médios de execução.

Conclusão

Os experimentos realizados demonstram de forma consistente que a abordagem baseada em **processos** apresentou desempenho superior à implementação sequencial e ao modelo com threads na maioria dos cenários analisados. A identificação do **parâmetro PPP ideal** evidencia a relevância de ajustar o tamanho dos blocos de trabalho no paralelismo, de modo a equilibrar eficiência computacional e overhead de gerenciamento. Assim, para o ambiente de teste adotado, a configuração mais eficiente correspondeu à utilização de processos com PPP próximo a 1.280.000, proporcionando o melhor aproveitamento dos recursos do processador e os menores tempos médios de execução.