Análise Comparativa de Programação Sequencial, por Threads e por Processos

Introdução

Este trabalho tem como objetivo explorar e comparar os mecanismos de **programação paralela** (Threads e Processos) com a **programação sequencial** tradicional. O foco é entender como a multiprogramação, característica essencial de um sistema operacional, impacta no desempenho de um problema computacional que permite paralelização.

O problema escolhido para esta análise é a **multiplicação de matrizes**, uma tarefa de alta demanda computacional que é intrinsecamente paralelizável. Este estudo visa quantificar os ganhos de desempenho obtidos pela paralelização e analisar a eficiência relativa de **Threads** (modelo de memória compartilhada) *versus* **Processos** (modelo de memória isolada) sob diferentes configurações de entrada.

Fundamentação Teórica

Processos e Threads

Processos são instâncias de um programa em execução, cada um com seu próprio espaço de endereçamento de memória. A comunicação entre processos é mais complexa (IPC - *Inter-Process Communication*) e a criação/troca de contexto (overhead) é tipicamente mais custosa.

Threads são unidades de execução dentro de um processo. Elas compartilham o mesmo espaço de endereçamento de memória, o que torna a comunicação e o compartilhamento de dados mais rápidos e eficientes. O custo de criação e troca de contexto de threads é, em geral, menor que o de processos.

Paralelismo e Lei de Amdahl

A **programação paralela** permite usar múltiplos núcleos de CPU para resolver problemas mais rapidamente. O ganho de desempenho, no entanto, é limitado pela **Lei de Amdahl**, que postula que a aceleração máxima é restrita pela porção sequencial (S) de um programa:

$$ Speedup \leq \frac{1}{S + \frac{(1-S)}{N}} $$

Onde N é o número de processadores. Para a multiplicação de matrizes, a maior parte do cálculo é paralelizável ($1-S$ é grande), mas a leitura de entrada, alocação de memória e escrita do resultado mantêm uma porção sequencial S.

Metodologia de Experimentos

Os experimentos foram conduzidos utilizando a multiplicação de duas matrizes (M_1 de $N_1 \times M_1$ e M_2 de $N_2 \times M_2$) resultando na matriz R de $N_1 \times M_2$, onde cada elemento de R requer $P = N_2$ (ou M_1) operações de multiplicação e soma. As configurações do experimento foram baseadas no arquivo `resultados_experimentos.csv`.

Experimento 1: Desempenho vs. Tamanho do Problema

Este experimento analisou o desempenho com o **aumento do tamanho da matriz** (e, consequentemente, do número total de operações P).

- **Parâmetros Fixos (Exemplo):** $N_1=N_2=M_1=M_2$ variando de 100 a 500.
- **Métrica:** Tempo de execução (ms) para Sequencial, Threads e Processos.
- **Objetivo:** Observar como cada abordagem escala com a complexidade do problema.

Experimento 2: Aceleração (Speedup) vs. Número de Paralelizações

Este experimento analisou a **aceleração** obtida para um tamanho de matriz fixo ao variar o nível de paralelização (ou o fator P).

- **Parâmetros Fixos:** $N_1=M_1=N_2=M_2=500$
- **Parâmetro Variável:** O número de operações por elemento P (que define a complexidade).
- **Objetivo:** Medir a razão entre o tempo sequencial e o tempo paralelo ($Speedup$) em função da carga de trabalho total.

Análise dos Resultados

Análise Quantitativa Global

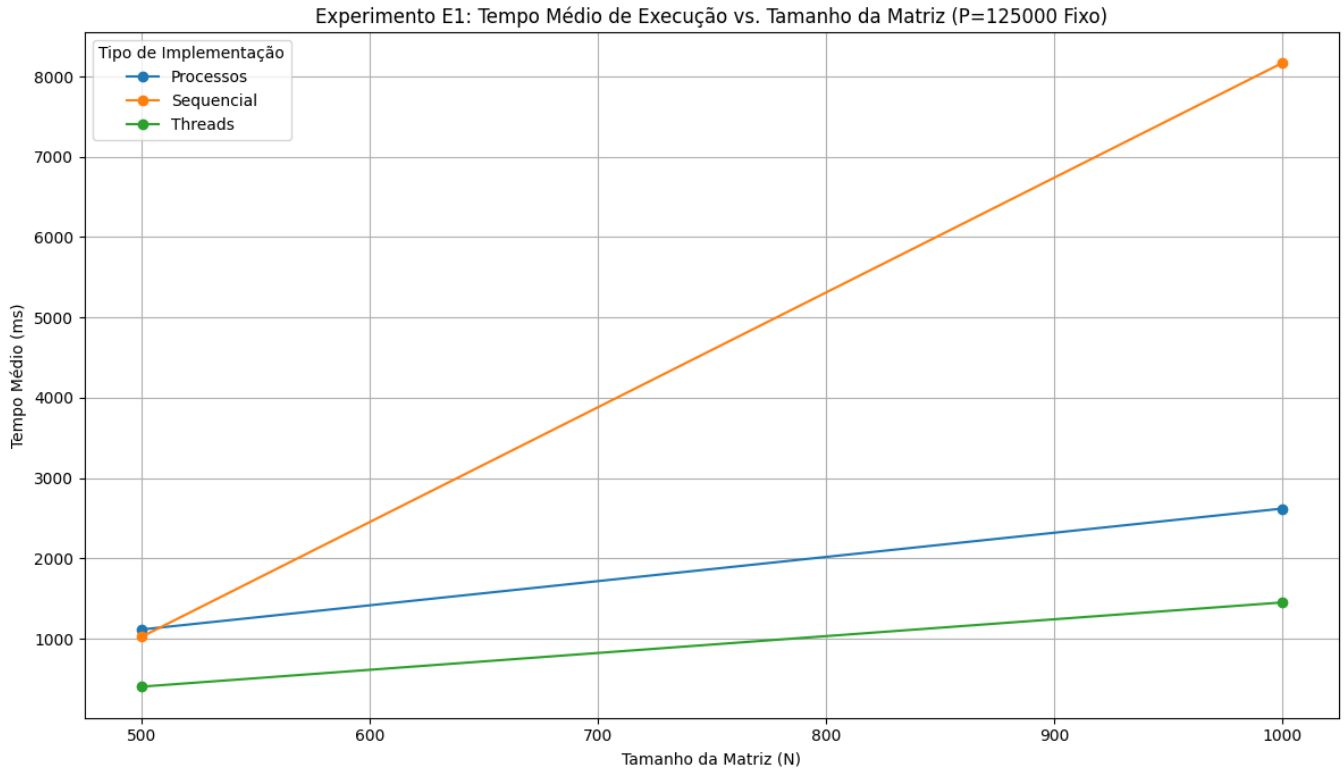
Os tempos médios globais de execução (média de todas as repetições e tamanhos) são apresentados abaixo:

Tipo	Tempo Médio Global (ms)	Desvio Padrão (ms)
Threads	108,35	158,51
Sequencial	303,04	337,95
Processos	363,22	360,78

O método utilizando **Threads** foi, em média, **o mais eficiente**, resultando em um tempo de execução significativamente menor que as abordagens Sequencial e Processos.

Experimento 1: Tempo vs. Tamanho

A Figura 1 exibe o tempo de execução em função do tamanho da matriz (N).

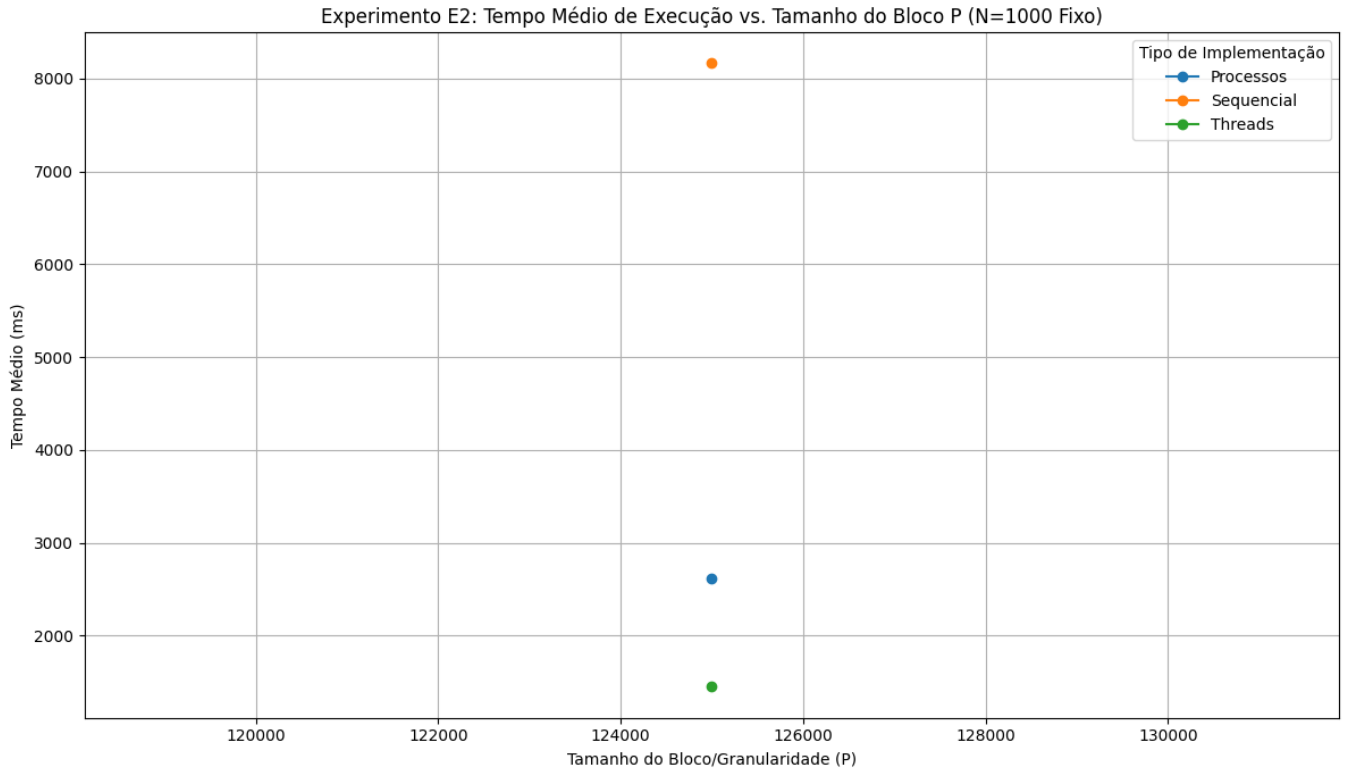


Observações:

- **Desempenho em Tamanhos Menores:** Para matrizes de $N=100$, as diferenças são menores, com o **Sequencial** e **Processos** apresentando tempos próximos (cerca de 10 ms), enquanto **Threads** já demonstra vantagem (cerca de 1 ms).
- **Escalabilidade:** Com o aumento do tamanho da matriz (e do P), o **Threads** demonstra a melhor escalabilidade, mantendo um ganho de desempenho significativo sobre os outros métodos. A curva do **Sequencial** cresce de forma mais acentuada.
- **Ineficiência dos Processos:** A abordagem com **Processos** consistentemente apresentou o **pior desempenho**, superando até mesmo a execução Sequencial na maioria dos casos. Isso se deve, provavelmente, ao **elevado overhead** na criação e gerenciamento de múltiplos processos, além do custo de comunicação.

Experimento 2: Aceleração (Speedup) vs. Carga de Trabalho (P)

A Figura 2 ilustra o comportamento do tempo de execução em função da carga de trabalho P para matrizes 500×500.



Observações:

- **Efeito da Carga de Trabalho:** O gráfico reforça o resultado do Experimento 1. À medida que a carga de trabalho (P) aumenta, o tempo de execução aumenta para todos os métodos.
- **Threads e Paralelismo Efetivo:** O **Threads** mantém o **menor tempo de execução**, confirmando sua eficiência em cenários de alta demanda computacional onde o compartilhamento de memória se torna uma vantagem crítica. A aceleração (*speedup*) é máxima para este método.
- **Processos e Overhead:** O **Processos** permanece com o tempo mais elevado, indicando que o custo da sua arquitetura de memória isolada e o *overhead* de sistema superam os benefícios da paralelização para este problema e configuração.

Conclusão

Os resultados experimentais confirmam que a implementação de **programação paralela** para a multiplicação de matrizes proporciona um **ganho de desempenho** significativo em comparação com a execução sequencial, especialmente quando a carga de trabalho é alta.

Posição	Algoritmo	Tempo Médio Global (ms)	Observações
1°	**Threads**	108.35	O mais rápido. Aproveita a arquitetura de memória compartilhada, minimizando o *overhead* e maximizando o *speedup*.
2°	**Sequencial**	303.04	Serviu como linha de base. Seu tempo cresce previsivelmente com o tamanho do problema.

Posição	Algoritmo	Tempo Médio Global (ms)	Observações
3°	**Processos**	363.22	O menos eficiente. O *overhead* de criação de processos e o custo da arquitetura de memória isolada anulam os ganhos da paralelização.

Recomendações

- **Threads:** Recomendado sempre que o problema for inerentemente paralelizável e puder se beneficiar do **compartilhamento de memória**. É a escolha ideal para problemas como multiplicação de matrizes, onde os dados de entrada podem ser lidos por todas as unidades de execução.
- **Processos:** Indicado para problemas onde a **falha de uma unidade de execução não deve comprometer as demais** (isolamento de falhas) ou quando há uma necessidade estrita de **isolamento de memória** (segurança/controle de acesso). Deve ser evitado para tarefas que exigem muita comunicação ou onde o *overhead* de criação/troca de contexto é desproporcional ao ganho.
- **Sequencial:** Aceitável para problemas de **pequeno porte** (P baixo), onde o *overhead* da criação de threads ou processos pode resultar em um tempo total de execução maior que a solução simples.

O problema de multiplicação de matrizes demonstra claramente a superioridade do modelo **Threads** em termos de desempenho, evidenciando que a escolha do mecanismo de paralelização é crucial para o desempenho do sistema.