Relatório do Exercício Programa 1 de MAC0219

Giovanna Uchôa Maia Gomes
14574998

Isabella Caselli
13686685

Muriel Toneli Latorre
14599710

20 de outubro de 2024

Resumo

Este é um relatório do primeiro Exercício Programa da disciplina Programação Concorrente e Paralela (MAC0219), ministrada pelo Prof. Alfredo Goldman. O exercício consiste em paralelizar o código para o cálculo do Conjunto de Mandelbrot usando a biblioteca Pthreads e as diretivas de compilador fornecidas pelo OpenMP. O documento está dividido em introdução, objetivos, experimentos, apresentação e discussão de resultados, e conclusão. Em anexo, é possível visualizar as três versões do código-fonte usadas em formato .c, desenvolvidas de maneira colaborativa através da plataforma Github. Além disso, há uma planilha de dados .csv com as medições feitas.

Conteúdo

1 INTRODUÇÃO

Este é o primeiro Exercício Programa da disciplina MAC0219 - Programação Concorrente e Paralela, oferecida pelo Instituto de Matemática e Estatística da Universidade de São Paulo. A proposta deste exercício é paralelizar o cálculo do Conjunto de Mandelbrot utilizando a biblioteca Pthreads e as diretivas de compilador fornecidas pelo OpenMP, comparando a eficiência das versões paralelizadas com a implementação sequencial.

Os fractais, introduzidos por Benoît Mandelbrot em 1975, são objetos geométricos que exibem padrões complexos que se repetem em diferentes escalas. O Conjunto de Mandelbrot, em particular, é um dos fractais mais estudados, sendo conhecido por sua beleza visual e por representar como a complexidade pode emergir de regras matemáticas simples. A visualização desse conjunto, no entanto, demanda grande capacidade de processamento, uma vez que envolve a iteração de funções complexas sobre uma vasta gama de números.

O código base fornecido no arquivo `ep1.zip` para o cálculo do Conjunto de Mandelbrot foi distribuído em três arquivos de extensão `.c`:

- `mandelbrot_seq.c`: contém a implementação sequencial do cálculo do Conjunto de Mandelbrot. Esta versão não foi modificada para paralelização e serve como referência para as demais implementações.
- `mandelbrot_pth.c`: a versão paralelizada utilizando Pthreads. Esta implementação distribui a carga de trabalho entre múltiplas threads, permitindo a execução em paralelo de diferentes partes do cálculo.
- `mandelbrot_omp.c`: a versão paralelizada utilizando OpenMP. Com as diretivas de compilador do OpenMP, o código é modificado para realizar o cálculo de forma paralela, aproveitando a facilidade de uso dessa abordagem.

Cada versão tem como objetivo manter a corretude da saída, gerando as mesmas imagens do Conjunto de Mandelbrot, para possibilitar uma comparação justa do desempenho entre as abordagens.

2 OBJETIVOS

O principal objetivo deste trabalho é aplicar os conceitos estudados em sala para implementar versões otimizadas do cálculo do Conjunto de Mandelbrot, utilizando Pthreads e OpenMP, com as versões paralelizadas devendo ser capazes de produzir a mesma saída correta que a versão sequencial.

Para tanto, é necessário realizar medições do tempo de execução das três versões (sequencial, Pthreads e OpenMP) e analisar os resultados obtidos. As análises envolverão a construção de gráficos que apresentam a média e o intervalo de confiança das execuções, permitindo identificar como cada implementação se comporta em relação a variações do tamanho de entrada, das regiões do Conjunto de Mandelbrot e do número de threads.

Outro objetivo importante é investigar o impacto das operações de entrada/saída (I/O) e alocação de memória no tempo de execução na versão sequencial. Isso permitirá entender melhor quais fatores influenciam no desempenho e permitirá identificar oportunidades para otimizar ainda mais as implementações.

3 EXPERIMENTOS

Na realização dos experimentos, buscamos comparar o desempenho das versões sequencial, paralelizada com Pthreads e paralelizada com OpenMP do cálculo do Conjunto de Mandelbrot. Para isso, foram realizadas medições de tempo de execução variando os tamanhos de entrada e o número de threads nas versões paralelizadas. A Tabela 1 apresenta os parâmetros utilizados em cada experimento:

Versão	**Regiões**	**I/O e Alocação de Memória**	**Número de Threads**	**Tamanho da Entrada**	**Número de Execuções**
Sequencial	Triple Spiral, Elephant, Seahorse, Full	Sem e Com	-	$2^4 \dots 2^{11}$	10
Pthreads	Triple Spiral, Elephant, Seahorse, Full	Sem	$2^0 \dots 2^5$	$2^4 \dots 2^{11}$	10
OpenMP	Triple Spiral, Elephant, Seahorse, Full	Sem	$2^0 \dots 2^5$	$2^4 \dots 2^{11}$	10

Tabela 1: Parâmetros dos experimentos para cálculo do Conjunto de Mandelbrot

É importante ressaltar que as medições foram feitas para entradas até 2^{11} por conta de limitações de memória do computador usado para os experimentos.

4 APRESENTAÇÃO E DISCUSSÃO DE RESULTADOS

Para realizarmos a obtenção dos dados, utilizamos o comando `perf stat` e criamos um script em python que realizasse o processamento dos dados, a criação do arquivo `.csv` e que gerasse as visualições em gráficos.

Na nossa implementação, em cada experimento, a ferramenta `perf stat` foi executada com cada código um total de 10 vezes e, em seguida, calculamos a média dos valores e desvios-padrões para o conjunto de parâmetros apresentados anteriormente.

Inicialmente, considere o seguinte glossário com as métricas devolvidas pelo comando `perf stat`, que serão utilizadas em nossa análise.

- **Context-switches**: uma mudança de contexto ocorre quando o SO suspende a execução de um processo/thread e transfere o controle da CPU para outro processo/thread.
- **CPU-migrations**: uma migração de CPU ocorre quando uma thread é transferida de um núcleo da CPU para outro durante sua execução.
- **Page-faults**: uma falha de página ocorre quando um programa tenta acessar uma página da memória que não está presente na RAM, levando o SO a carregar a página da memória secundária para a RAM.
- **CPU Cycles**: é o número de ciclos de clock que a CPU realizou durante a execução do programa.
- **Instructions per Cycle (IPC)**: representa a relação entre a quantidade de instruções executadas e a de ciclos de CPU. O cálculo se dá pela divisão entre número total de instruções/número total de ciclos.
- **Branches e Branch Misses**: branches se refere à quantidade total de instruções que controlam fluxos, como if/else, que o processador executa. Já branch misses, são as previsões de branches que falharam, isto é, a quantidade de vezes que a CPU previu que um branch seria seguido, mas não foi de fato.
- **Tempo total de execução**: nos informa o tempo total que o programa necessitou para ser concluído, do início ao fim.

Agora, vamos abordar cada análise individualmente de como o programa se comporta com a variação do tamanho de entrada, das regiões do Conjunto de Mandelbrot e do número de threads, além da influência das operações de I/O e alocação de memória.

4.1 Tamanho de entrada

Vamos analisar o comportamento do programa sob a ótica da variação do tamanho de entrada, a partir de determinadas métricas do `perf stat`. É importante ressaltar que para essa análise, fixamos o número de threads em 16, visando minimizar problemas de overhead e de ineficiência do paralelismo. Além disso, executamos o `mandelbrot_seq` sem operações de I/O e alocação de memória.

Inicialmente, considere a métrica branch misses. O gráfico a seguir compara a quantidade de falhas de branch em função do tamanho da entrada, para cada uma das versões do código:

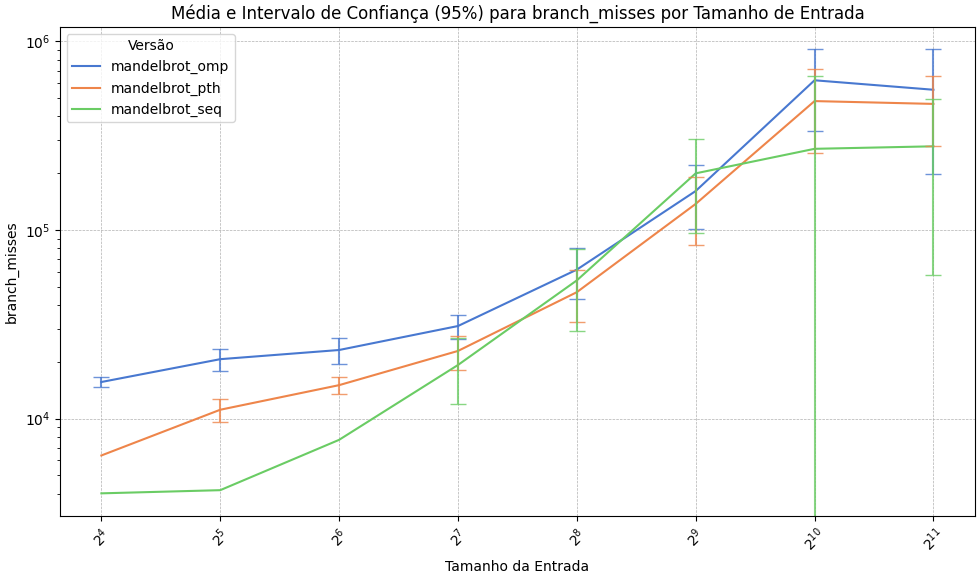

Figura 1: Falhas de branch x Tamanho da entrada

De forma geral, podemos observar que o número de falhas de branch tende a ser proporcional ao tamanho de entrada, o que era esperado, uma vez que com uma maior quantidade de dados, a quantidade de predições de desvios torna-se maior e, consequentemente, o mesmo ocorre com as falhas dessas predições. Além disso, é interessante notar que a versão sequencial apresenta, relativamente, uma menor quantidade de branch misses, o que pode ser explicado pela menor variabilidade de controle de fluxo na versão sequencial, possibilitando um maior ajuste do preditor de branch a padrões, além da existência de conflitos de sincronização entre threads, que também podem aumentar o número de falhas de branch.

Agora, vamos considerar o número de page faults. Observe a tabela abaixo simplificada com dados obtidos do `.csv` em anexo, considerando cada versão do programa e a região "full".

Versão	Tamanho da entrada	Page faults
`mandelbrot_seq`	32	71
`mandelbrot_seq`	128	221
`mandelbrot_seq`	512	2622
`mandelbrot_seq`	2048	41022
`mandelbrot_pth`	32	108
`mandelbrot_pth`	128	109
`mandelbrot_pth`	512	109
`mandelbrot_pth`	2048	108
`mandelbrot_omp`	32	116
`mandelbrot_omp`	128	117
`mandelbrot_omp`	512	117
`mandelbrot_omp`	2048	117

Tabela 2: Falhas de página x tamanho de entrada

É possível notar, para a versão sequencial, que um maior tamanho da entrada ocasiona um maior número de page faults. Isso era esperado, uma vez que há uma maior quantidade de dados e o SO precisa efetuar uma maior quantidade de trocas de páginas entre a RAM e o disco. Além disso, para ambas versões paralelizadas, é possível notar, neste caso, que o tamanho da entrada não apresenta influência significativa no número de page faults. Isso pode ser um indicativo de que a paralelização com pthreads e com OpenMP apresenta um gerenciamento de memória mais eficiente. Dessa maneira, é possível que as threads estejam distribuídas de maneira a manter uma melhor localidade de referência aos dados, o que minimiza as falhas de páginas.

Por fim, vamos verificar o tempo de execução total do programa. Veja o gráfico abaixo.

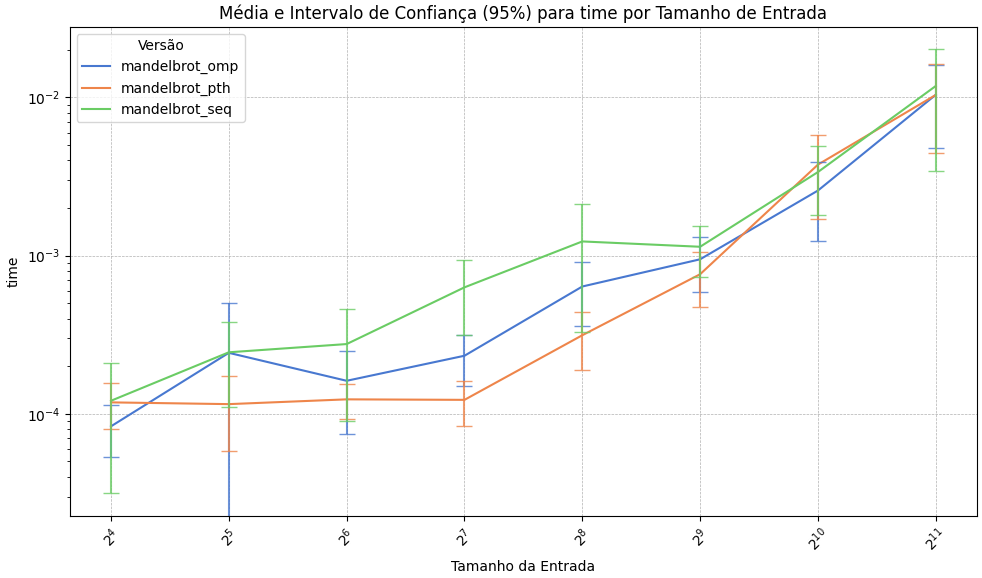

Figura 2: Tempo de execução x Tamanho da entrada

É possível verificar que o tempo é afetado pelo tamanho da entrada, já que mais cálculos são efetuados.

Para entradas até 2^9, podemos ver que as versões paralelizadas têm melhor desempenho do que a sequencial. Porém, para tamanho maiores, o tempo de execução das três versões é muito similar. Algumas hipótesess levantadas que podem explicar esse fenômeno incluem overhead, pois o tamanho maior de entrada significa que cada thread lida com mais dados, o que pode diminuir os ganhos adquiridos pelo paralelismo. Além disso, entradas grandes consomem mais recursos dos núcleos de processamento e também podem ocasionar gargalos de acesso à memória, de forma que a busca por dados na memória deixe o código mais lento.

4.2 Regiões do Conjunto de Mandelbrot

Para analisarmos o comportamento do programa sob a ótica das regiões do conjunto de Mandelbrot, podemos comparar a execução a partir de métricas do perf. É importante ressaltar que para essa análise, fixamos o número de threads em 16, visando minimizar problemas de overhead e de ineficiência do paralelismo. Além disso, fixamos o tamanho da entrada em 512. Ainda, executamos o `mandelbrot_seq` sem operações de I/O e alocação de memória.

Inicialmente, vamos analisar o número de branch misses. Observe o gráfico abaixo.

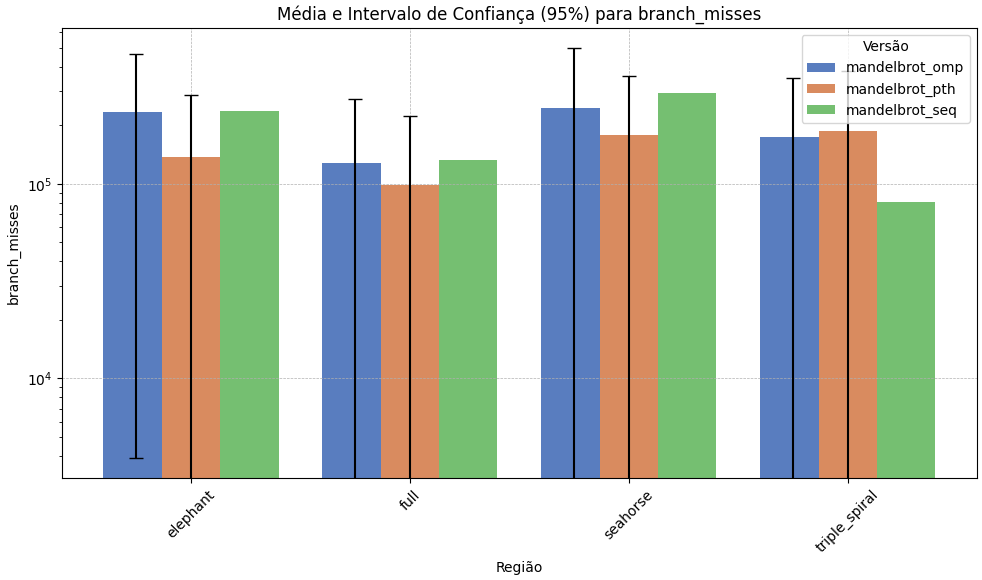

Figura 3: Falhas de branch x Região

É possível notar, para as quatro regiões, que há certo equilíbrio entre o número de branch misses. Todavia, há uma tendência de o código sequencial apresentar uma maior quantidade de falhas, o que era esperado, uma vez que, no código sequencial, o fluxo de controle é mais linear e menos previsível, dificultando a predição de desvios. Entretanto, na região "triple spiral", é possível observar que a versão sequencial apresenta melhor desempenho. Isso pode ocorrer considerando uma possível melhor localidade do acesso a dados na memória, nesta região. Outra hipótese é o overhead de paralelização na região, que pode ocorrer por conta da elevada proximidade entre os pontos e da forte dependência entre iterações.

Agora, vamos analisar a troca de contexto. Considere o gráfico abaixo.

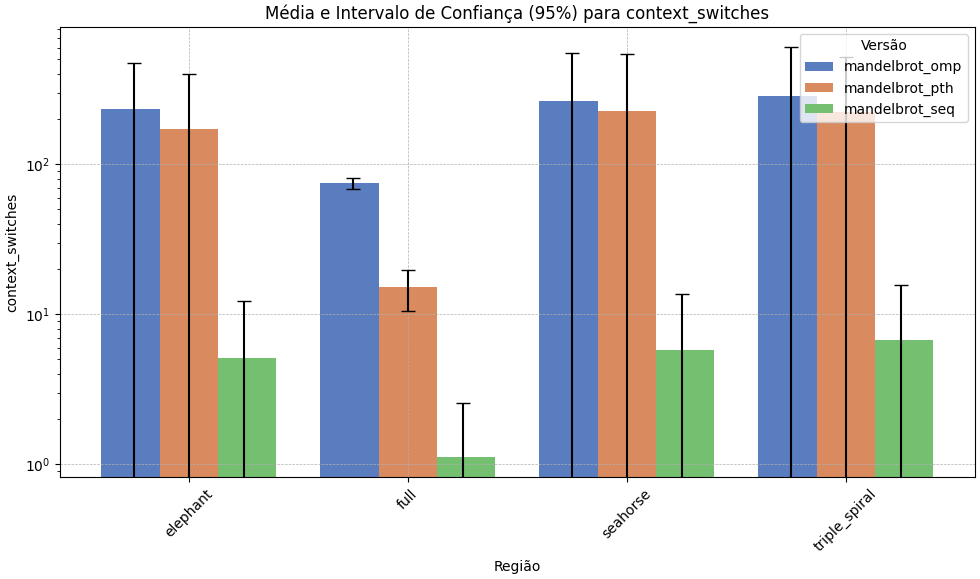

Figura 4: Troca de contexto x Região

Note que para as regiões "elephant", "seahorse"e "triple spiral", os programas seguem o mesmo padrão: um equilíbrio no valor das versões paralelizadas, com a versão OpenMP sutilmente maior, e um pequeno valor para a versão sequencial. Isso era esperado, uma vez que, com programas paralelizados, há uma maior troca de controle de processos pela CPU. Além disso, na região "full", é interessante notar que os programas realizaram uma relativa menor quantidade de trocas de contexto, o que pode ser causado por um melhor gerenciamento dos processos e uma maior sincronização das threads na região.

Ainda, vamos analisar o número de ciclos de CPU. Veja o gráfico abaixo.

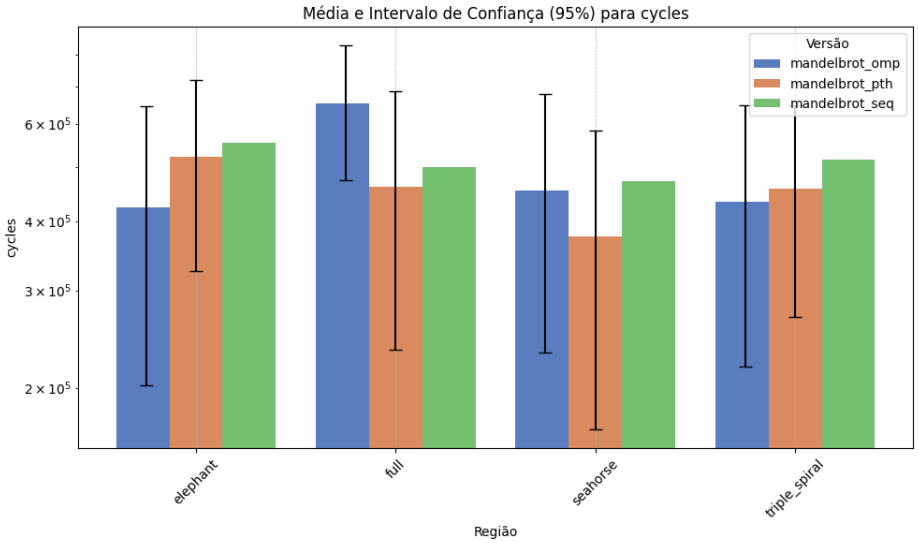

Figura 5: Ciclos de CPU x Região

De forma geral, é perceptível que o código sequencial apresenta um maior número de ciclos de CPU, o que pode indicar um acesso mais lento à memória e falta de otimizações na versão. Além disso, em particular, a região "full"apresenta um maior valor para a versão paralelizada com o OpenMP. Isso pode ser um indicativo de que, nessa região, o overhead de paralelização em razão da criação e gerenciamento das threads é mais presente. Ainda, pode estar ocorrendo uma maior espera por dados compartilhados e uma maior quantidade de conflitos entre as threads.

Por fim, vamos analisar o consumo de tempo para cada região.

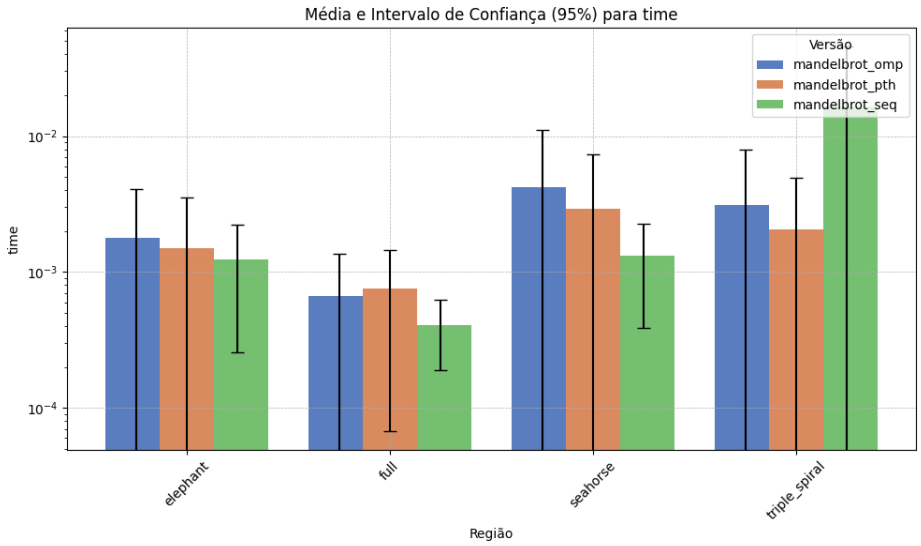

Figura 6: Ciclos de CPU x Região

É interessante notar que, para as regiões "elephant", "full"e "seahorse", há certo equilíbrio entre as 3 versões, com a sequencial apresentando melhor desempenho. Como mencionado anteriormente, isso pode ser um indicativo de overhead de paralelização e da falta de um gerenciamento efetivo das threads. Todavia, na região "triple spiral", a situação é invertida, com as versões paralelizadas apresentando melhor desempenho. Portanto, é curioso notar que, a depender da complexidade e das particularidades das regiões, a vantagem do programa paralelizado é relativa, e diversos fatores precisam ser considerados para concluir se a paralelização de fato trará os ganhos esperados.

4.3 Operações de I/O e Alocação de memória

Agora, vamos verificar o comportamento do código sequencial sob uma ótica de presença ou ausência de operações de entrada e saída de dados. É importante ressaltar que para essa análise, fixamos o número de threads em 16, visando minimizar problemas de overhead e de ineficiência do paralelismo.

Inicialmente, vamos considerar o número de page faults. Veja a tabela abaixo, com a média dos valores em função do tamanho de entrada, para o código sequencial, considerando a região "full".

Versão	Tamanho da entrada	Page faults
com I/O	16	63
sem I/O	16	58
com I/O	32	71
sem I/O	32	58
com I/O	64	101
sem I/O	64	58
com I/O	128	221
sem I/O	128	58
com I/O	256	702
sem I/O	256	58
com I/O	512	2622
sem I/O	512	58

Tabela 3: Falhas de página x Tamanho de entrada

É interessante notar o comportamento da tabela. Veja que, para a versão sem operações de I/O, o número de page faults é constante, o que indica a ausência significativa de problemas de alocação de memória. Todavia, para a versão com I/O, observa-se que a quantidade de page faults é proporcional ao tamanho de entrada, o que era esperado, uma vez que estamos realizando procedimentos de entrada e saída de dados, assim como de alocação de memória. É interessante ressaltar que este é um dos motivos da versão com I/O consumir uma maior quantidade de tempo de execução, como veremos a seguir.

Agora, vamos considerar a quantidade de branch misses. Observe o gráfico abaixo.

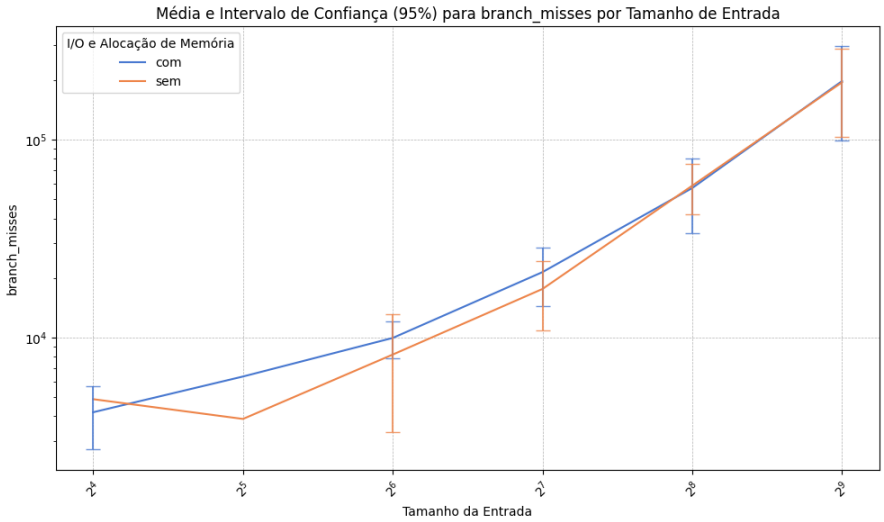

Figura 7: Falhas de branch x Tamanho de entrada

Curiosamente, o número de branch misses é similar para ambas as medições. Tal resultado pode ser um indício de que as operações de I/O e a alocação de memória não apresentam influência significativa no número de branch misses, uma vez que ambas versões possuem padrões similares de controle de fluxo. Todavia, é válido ressaltar que a quantidade de branch misses é proporcional ao tamanho de entrada.

Por fim, vamos analisar o tempo geral de execução das versões. Veja o gráfico abaixo.

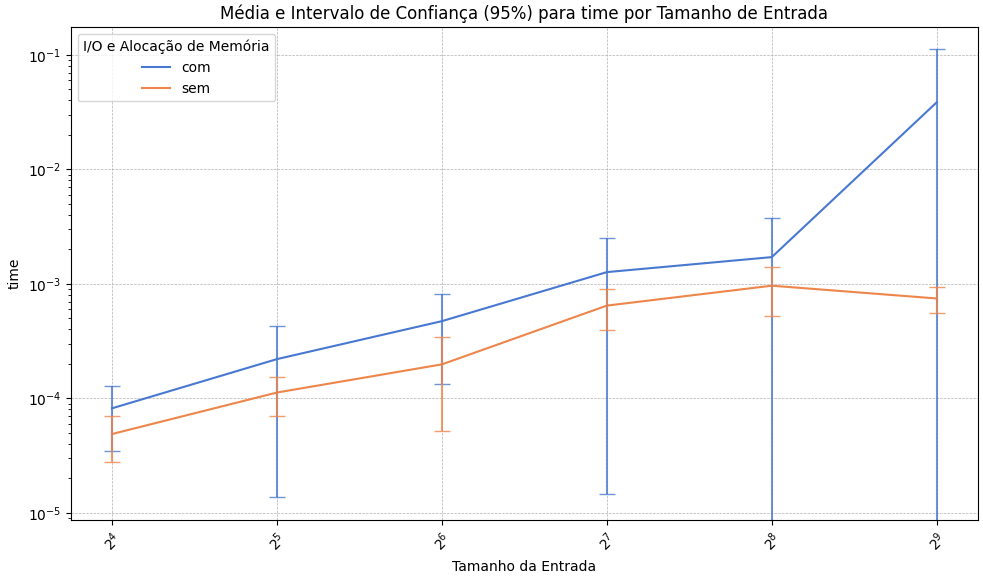

Figura 8: Tempo de execução x Tamanho de entrada

É possível observar que a versão com operações de I/O apresenta tempo de execução maior do que a versão sem, o que era um comportamento esperado. De maneira geral, operações de I/O apresentam uma latência natural, por conta da leitura/escrita em disco, por exemplo. Além disso, a alocação dinâmica de memória envolve interações com o SO e mecanismos de gerenciamento de memória, como ficou evidente pela análise do número de page faults, o que também requer um tempo de computação. Portanto, é natural que tal procedimento apresente um maior valor no tempo. De qualquer maneira, é válido ressaltar que o tempo é proporcional ao tamanho de entrada.

Todavia, é importante observar que o impacto das operações de I/O e alocação de memória é pequeno quando considerado o tempo total de execução do programa. Logo, os possíveis "gargalos"da execução não concentram-se nessa parte do programa. Tal análise permite concluir que, portanto, a paralelização deve estar focada nos cálculos da geração dos pontos do conjunto de Mandelbrot, para uma maior otimização e ganho de eficiência.

4.4 Número de threads

Agora, vamos analisar o comportamento do programa sob a ótica do número de threads. É importante ressaltar que para essa análise, fixamos o tamanho de entrada em 512. Além disso, executamos o `mandelbrot_seq` sem operações de I/O e alocação de memória.

Inicialmente, considere o gráfico abaixo, que apresenta o tempo total de execução em função do número de threads.

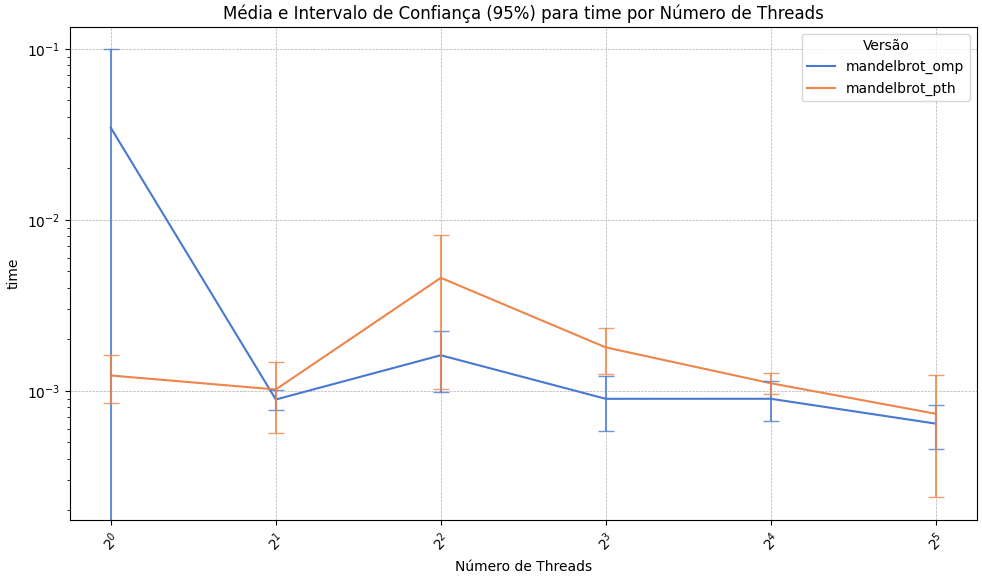

Figura 9: Tempo de execução x Número de threads

Veja que, de forma geral, ambas versões apresentam tempo similar de execução, com a versão paralelizada com o OpenMP possuindo melhor desempenho. Isso é um indicativo de que o gerenciamento de threads, as diretivas de compilação e as otimizações internas do OpenMP estão mais efetivas do que o Pthreads. Entretanto, é interessante notar que, com apenas 1 thread, isto é, sequencialmente, a versão com OpenMP apresenta desempenho pior, uma vez que todos os procedimentos de otimização realizados são ineficientes e relativamente descartados, visto que não há múltiplas threads. Ainda, é interessante notar que o gráfico apresenta um pico quando há 4 threads nos programas, o que pode ser causado por uma maior quantidade de falhas de sincronização e de desbalanceamento de tarefas para esse valor.

Agora, vamos verificar o número de branch misses a partir do gráfico a seguir.

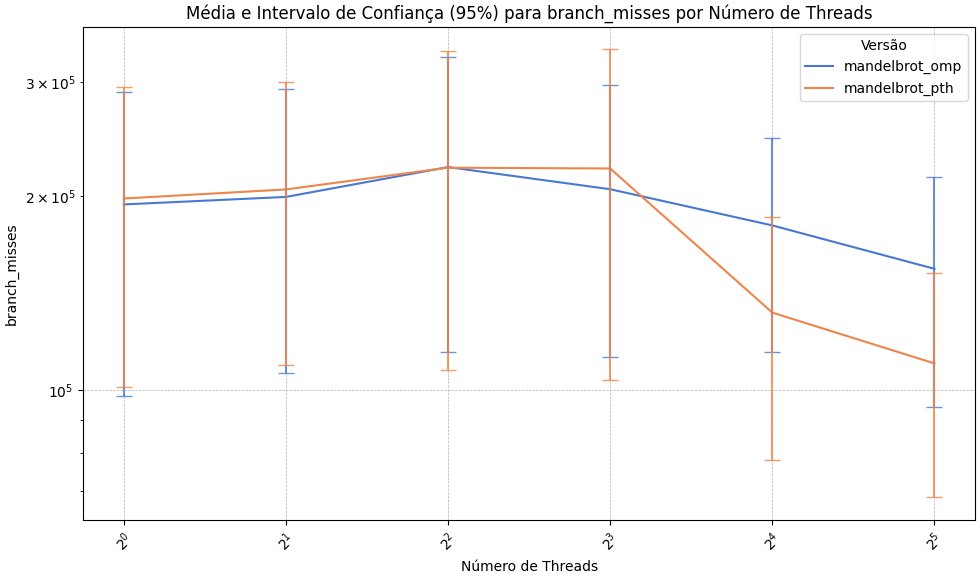

Figura 10: Falhas de branch x Número de threads

É válido notar o comportamento do gráfico. De 2^0 até 2^3, o número de branch misses mantém-se relativamente constante em torno de 2×10^5, indicando que tal variação da quantidade de threads não altera significativamente o valor de desvios de predição falhos. Todavia, a partir de 2^3 threads é possível visualizar uma notável queda no número de branch misses. Isso pode ser explicado por um possível uso mais eficiente do pipeline da CPU, além de um melhor aproveitamento da localidade de referência de dados na memória cache, uma vez que, com mais threads, uma maior quantidade de instruções é carregada no cache.

Por fim, vamos analisar as trocas de contextos. Veja o gráfico abaixo.

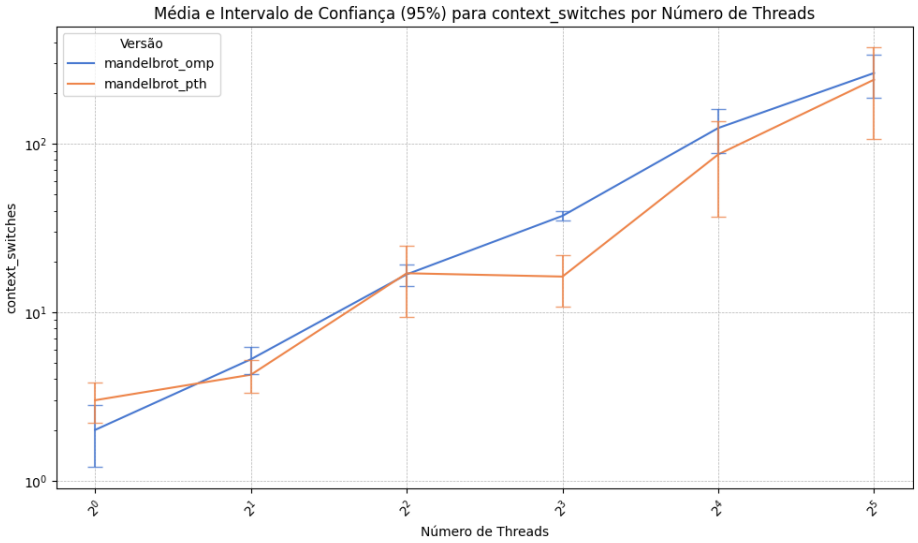

Figura 11: Troca de contextos x Número de threads

Observe que, para ambas versões, a quantidade de trocas de contexto é proporcional ao número de threads. Esse é um comportamento esperado, considerando que, com mais threads, as alterações do controle da CPU tornam-se mais frequentes. Além disso, é notável que os valores são similares, o que indica um desempenho equivalente entre as versões, neste quesito.

5 CONCLUSÃO

Com base nas análises realizadas, podemos elaborar conclusões a respeito da execução do programa para os diferentes parâmetros.

Inicialmente, o tamanho de entrada apresenta impacto significativo no tempo de execução de programa, com maiores entradas necessitando de um maior período de computação dos dados, para todas as versões. Além disso, um maior input gera uma maior quantidade de branches e de branches misses, como visto anteriormente.

Em seguida, verificamos que, apesar das particularidades e das diferentes complexidades entre as regiões do conjunto de Mandelbrot, há certa constância entre as regiões, com a versão sequencial apresentando, em geral, um desempenho levemente superior. Isso é um indicativo de que o programa pode naturalmente apresentar gargalos de paralelização. Além disso, problemas como overhead e de sincronização de threads podem estar desbalanceando o ganho em tempo da paralelização.

Após, verificamos que a presença de I/O e de alocação de memória no programa gera novas preocupações na execução do programa. De forma geral, a presença ocasiona uma maior quantidade de page faults, de branch misses e de tempo de computação. Todavia, é interessante notar que o impacto dessa presença, quando comparado com o tempo total de execução do programa, é relativamente pequeno, sugerindo que há trechos no código que podem ser otimizados para uma melhor eficiência do programa.

Por fim, analisamos o impacto do número de threads. De maneira geral, uma maior quantidade de threads, para os limites dos experimentos, melhoram a eficiência do programa, com um menor tempo de computação e um menor número de branch misses. Entretanto, outros problemas podem surgir, como visto pela elevada quantidade de troca de contextos.

Portanto, em termos gerais, a paralelização mostrou-se uma boa solução para reduzir o tempo de execução em problemas com muitos cálculos, como no Conjunto de Mandelbrot. Porém, essa abordagem traz limitações e desafios, como a gestão eficiente das threads, o overhead de sincronização, a competição por recursos e o gerenciamento de memória. Além disso, é comum que o código apresente trechos que não são totalmente paralelizáveis, como as operações de I/O e, consequentemente, o programa deve apresentar suporte e uma boa execução ainda nesses casos. Ademais, é interessante observar que há maneiras diferentes de realizarmos a paralelização, como exemplificado com o OpenMP e com o Pthreads, sendo que cada maneira possui particularidades, ganho e perda de desempenho, como visto anteriormente. Ainda, é válido ressaltar que o sucesso da paralelização depende de encontrarmos um ponto de equilíbrio entre o número de threads, os recursos de hardware disponíveis e a própria natureza do problema.