MAC0219 - Relatório EP01

Nathália Yukimi Uchiyama Tsuno (14600541)
Renata Meyer Hobold (14605790)

19 de outubro de 2024

1 Introdução

Computação paralela é uma abordagem de processamento em que várias operações são realizadas simultaneamente, sob a hipótese de que um subproblema computacional pode ser resolvido de maneira independente e concorrente de outros subproblemas. O principal objetivo dessa técnica é acelerar a execução e tornar possível a resolução de problemas complexos que seriam inviáveis ou demorariam muito para serem processados de maneira sequencial (ou seja, utilizando apenas um núcleo ou uma unidade de processamento por vez).

Ao mesmo tempo, Benoît Mandelbrot foi um matemático francês de origem judaico-polonesa que ficou conhecido principalmente pelos seus estudos e suas pesquisas sobre aplicações de fractais para descrição de elementos naturais cujos padrões vão se repetindo em escalas cada vez menores. O conjunto de Mandelbrot exprime bem esse ideal. Essencialmente, sequências que seguem a regra: "Faça o quadrado do número anterior e some a constante C, considerando o primeiro termo igual a zero".

Assim, para o cálculo dessas sequências que tendem ao infinito, a termos matemáticos, utilizamos o poderio computacional que cada dia mais recebem melhorias e potência. Isso pode ser expresso pelo programa `mandelbrot_seq.c`. De corretude inquestionável, esse programa retorna parte desses fractais da definição de dimensões do plano complexo que o usuário deseja.

Contudo, em razão da natureza complexa e pesadamente computacional, a essência puramente sequencial pode não ser muito satisfatória em termos do interesse em se computar tais imagens, graças à demora no processamento da imagem final requisitada. Portanto, uma opção viável é o uso da computação paralela para tentar maximizar o poder computacional que as máquinas modernas de múltiplos núcleos de processamento podem oferecer.

Aplicamos então a paralelização por `pthreads` no arquivo `mandelbrot_pth.c` e pela diretiva `OpenMP` em `mandelbrot_omp.c`. Posteriormente, fizemos a execução sobre o arquivo `run_measurements.sh` para se ter uma intuição inicial sobre a variáveis que influenciavam o resultado final, mas, construímos um segundo script para obter os tempos individuais de cada execução para podermos construir os intervalos de confiança a partir do cálculo da média e do desvio padrão. Além disso, para o cálculo das execuções sem as operações de I/O, medimos com auxílio da biblioteca `<time.h>` sobre a parte paralelizável do programa.

Esse trabalho visa medir os impactos de variáveis secundárias sobre o desempenho das paralelização de um programa, como as partes não paralelizáveis do programa (operações de I/O, alocação de memória e impressão/coloração final), tamanho da entrada (que variou entre 2^4 a 2^{13}, número de Threads (entre 1 e 2^5), tipo da região do conjunto de Mandelbrot (Full Picture, Elephant Valley, Seahorse Valley, Triple Spiral Valley) e o tipo do Hardware em que os experimentos foram executados.

Por padrão, defina os valores de referência como $N = 2048$, 8 Threads, na região `Full Picture` e com o Hardware de especificações: processador i3, 16 cores, 12 GB de RAM e versão do SO Ubuntu 22.04.

Utilizamos também um script estatístico que verificava a distribuição dos dados obtidos pela média, mediana, mínimo, máximo, desvio padrão, primeiro e terceiro quartis e calculava o intervalo de confiança, assumindo-se uma distribuição Normal.

2 Qual o impacto das operações de I/O e alocação de memória no tempo de execução?

Conforme a Lei de Amdahl, o desempenho obtido pela otimização de uma única parte de um sistema é limitado pela fração de tempo que a parte melhorada realmente é usada. Em outras palavras, isso significa que, mesmo a mais eficiente paralelização de um programa é limitada pelas suas partes de natureza sequencial. Podemos ver isso de forma evidente no programa `mandelbrot_seq.c`, que é inteiramente sequencial, mas que possui algumas partições de natureza mais sequencial que outras, como as operações de input e output (I/O), alocação de memória e impressão/coloração da imagem final. Note que elas não possuem independência, podendo gerar conflitos entre as operações de leitura, escrita e acesso incorreto à memória se fossem paralelizadas.

As versões paralelizadas do programa `mandelbrot_seq.c`, `mandelbrot_pth.c` e `mandelbrot_omp.c`, focam em paralelizar apenas a função `compute_mandelbrot`, e logo todas essas ainda executam essas operações essencialmente sequenciais de I/O e alocação de memória. Realizemos alguns experimentos para explorar qual o impacto dessas operações tanto na versão puramente sequencial do programa como nas versões paralelizadas. Fixemos 8 threads e vejamos os tempos para computar a região `full` do conjunto para com e sem essas operações:

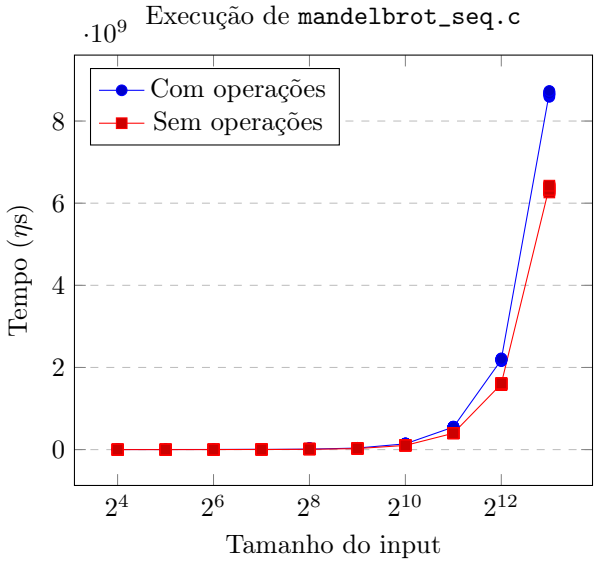


Figure 1: Tempo de execução

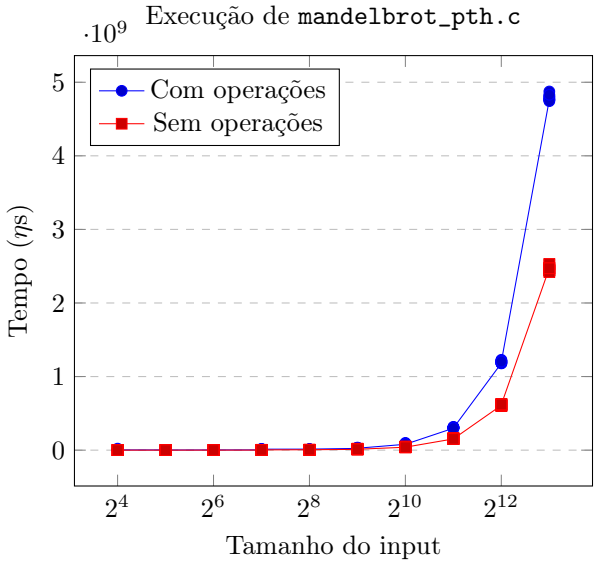


Figure 2: Tempo de execução

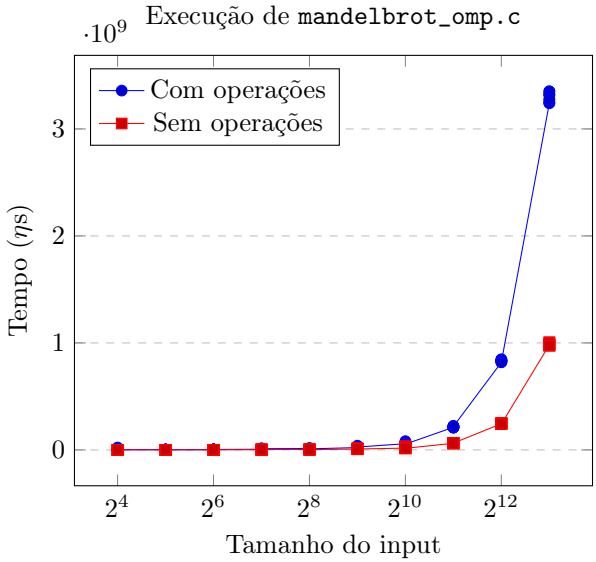
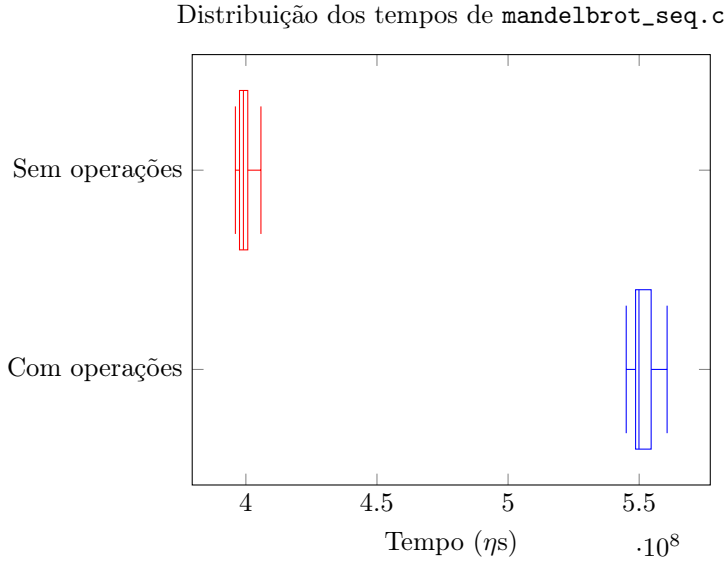


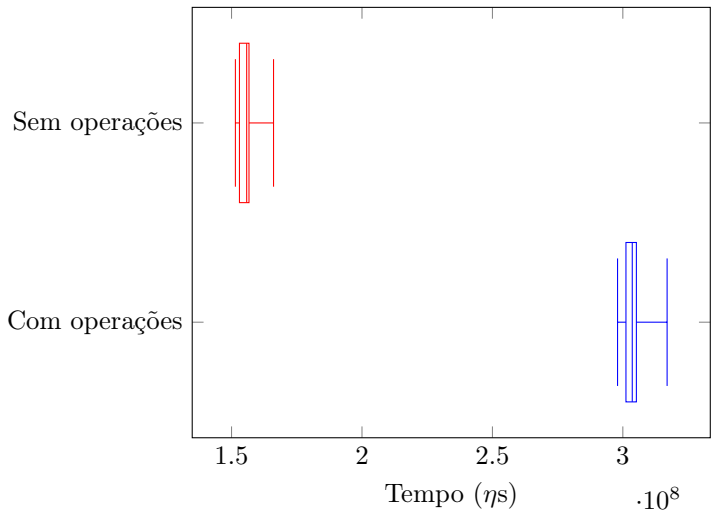
Figure 3: Tempo de execução

Na execução de experimentos, realizamos dois tipos deles para investigar o impacto das operações de I/O e alocação de memória no tempo de execução. O primeiro consistia em fazer a medição unicamente da parte paralelizável do programa, a função `compute_mandelbrot`. E, posteriormente, em sua íntegra.

Note visualmente como é claro que as operações de I/O afetam o tempo total de execução significativamente — nas operações maiores de tamanho de entrada 8192, essa proporção parece ser quase 50% do tempo de execução. É conveniente observar as estatísticas com um boxplot. Vamos colocar apenas para uma entrada razoável de $N = 2048$ e novamente 8 theads sobre a região `full`:

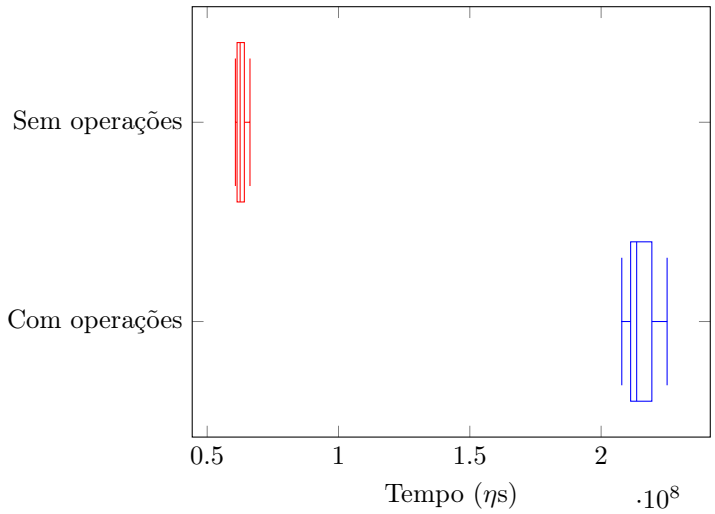


Distribuição dos tempos de `mandelbrot_pth.c`



Fazendo outro boxplot:

Distribuição dos tempos de `mandelbrot_omp.c`



Como uma forma de melhor visualização sobre os dados obtidos, note a variância sobre eles representados pelos box-plots. Nos três casos analisados, os experimentos apresentaram complexidade semelhante, o que se reflete em uma variância relativamente baixa. A maioria dos tempos de execução para cada um dos 6 tipos de experimento se concentra em torno da sua mediana, indicando que as execuções foram consistentes e sem desvios significativos em relação aos tempos médios esperados. Embora existam algumas variações naturais entre execuções, não houve flutuações críticas que pudessem impactar a interpretação dos resultados.

Na versão sequencial, que não utilizou paralelismo, foi possível observar que as operações de entrada e saída (I/O) e a alocação de memória acrescentaram, ao que aparenta, 1.5 segundos ao tempo total de execução. A computação do conjunto de Mandelbrot em si sem considerar o custo das operações auxiliares levou em torno de 4 segundos. Dessa forma, o tempo total para a execução completa do programa foi de aproximadamente 5.5 segundos, e logo a maior parte do tempo foi gasta na computação do conjunto.

Na versão paralelizada com `pthreads`, houve um impacto semelhante nas operações de I/O e alocação de memória, com um custo também próximo de 1,5 segundos. No entanto, ao focarmos apenas na parte paralelizável do código, observamos uma redução significativa no tempo de processamento em relação à versão sequencial. A paralelização com Pthreads conseguiu reduzir o tempo de execução em cerca de 2.5 segundos, que é uma melhoria de aproximadamente 37.5%. A paralelização pode ter acelerado a execução, mas não eliminou a influência das operações sequenciais de I/O e alocação de memória.

Por fim, a versão paralelizada com `OpenMP` também teve impactos expressivos. Tal como nos casos anteriores, em média, as operações de I/O e alocação de memória tiveram um custo aproximado de $1.5s$, quando comparadas

à computação de Mandelbrot por si. A diferença é visível também quando comparada aos outros casos. Em relação à versão sequencial, tivemos um ganho de quase $3.5s$ e à paralelização por `pthreads`, de aproximadamente, $1s$.

Para explorar ainda mais isso, fixemos execuções com 8 threads e tamanho de entrada $N = 2048$, na região `full` do conjunto. Segue uma tabela do resumo estatístico:

Table 1: Resumo estatístico para o programa `mandelbrot_seq.c` (tempo em ηs). IC para a média sem operações de I/O: [0.397823s, 0.401494s]; IC para a média com operações de I/O: [0.548501s, 0.554452s].

Versão	Com operações de I/O e alocações	Sem operações de I/O e alocações
Média	551476900	399658800
Desvio Padrão	4800715	2961625
Mínimo	545064052	395999829
Quartil 1	548643700	397567200
Mediana	549925200	399057200
Quartil 3	554575300	400721000
Máximo	560659534	405740837

A média do tempo se conteve no entorno dos $0.54s$ para a medição com operações de I/O, e $0.40s$ para o sem, com baixo desvio padrão. Cerca de $0.14s$ só de operação sequencial.

Table 2: Resumo estatístico para o programa `mandelbrot_pth.c` (tempo em ηs). IC para a média sem operações de I/O: [0.153256s, 0.158359s]; IC para a média com operações de I/O: [0.301033s, 0.308357s]

Versão	Com operações de I/O e alocações	Sem operações de I/O e alocações
Média	304695200	155807300
Desvio Padrão	5908858	4116270
Mínimo	297976060	151467936
Quartil 1	301192600	153023000
Mediana	303610300	155799200
Quartil 3	305187600	156680300
Máximo	316968962	166096240

A média do tempo se conteve no entorno dos $0.30s$ para a medição com operações de I/O, e $0.15s$ para o sem, com baixo desvio padrão. Cerca de 0.15s só de operação sequencial.

Table 3: Resumo estatístico para o programa `mandelbrot_omp.c` (tempo em ηs). IC para a média sem operações de I/O: [0.061731s, 0.064095s]; IC para a média com operações de I/O: [0.211661s, 0.218459s].

Versão	Com operações de I/O e alocações	Sem operações de I/O e alocações
Média	215060300	62913233.3
Desvio Padrão	5484098	1907258
Mínimo	207866000	60738253
Quartil 1	211237000	61379260
Mediana	213532300	62546116.5
Quartil 3	219315800	64149490
Máximo	225122834	66293500

A média do tempo se conteve no entorno dos $0.21s$ para a medição com operações de I/O, e $0.06s$ para o sem, com baixo desvio padrão. Cerca de $0.15s$ só de operação sequencial.

Assim, ainda que se possa depender do tipo do hardware em que os testes foram executados, podemos observar a tendência desse custo não paralelizável oriundo das operações de input e output, alocação de memória e impressão/coloração da imagem final como um valor constante para qualquer situação, inerente ao problema, e independente de paralelização ou não. No exemplo para $N = 2048$, 8 threads na região `full` do conjunto de Mandelbrot, tivemos um custo constante próximo de $0.15s$ para todas as ocasiões, tanto para as paralelizadas, quanto para a puramente sequencial.

Esse custo fixo reforça que existe uma parcela do programa que não pode ser acelerada pelo aumento de threads, o que se alinha aos princípios da Lei de Amdahl. Assim, mesmo com a paralelização das outras operações, a eficiência global do sistema é limitada inferiormente por essas operações não paralelizáveis. A melhor maneira de otimizar ainda mais o desempenho seria reestruturar essas partes sequenciais. Após certo ponto, aumentar a quantidade de threads talvez tenha retornos cada vez menores, especialmente quando o restante do cálculo já está paralelizado. Veremos isso em outra seção.

3 Qual o impacto do tamanho da entrada no tempo de execução?

A proporção do tamanho do arquivo define a quantidade de processamento e de memória que o programa precisará para garantir a corretude da imagem final. Logo, supõe-se que quanto maior a imagem, mais recursos serão movidos para processar toda a imagem. Podemos verificar a veracidade dessa hipótese posteriormente. para isso, realizemos alguns experimentos com entradas de tamanho entre 2^4 a 2^{13}, inclusos, que crescem exponencialmente. Fixe 8 threads e calculemos a região `full` do conjunto de Mandelbrot:

Por algum motivo os gráficos não ficam nessa página, só na seguinte...

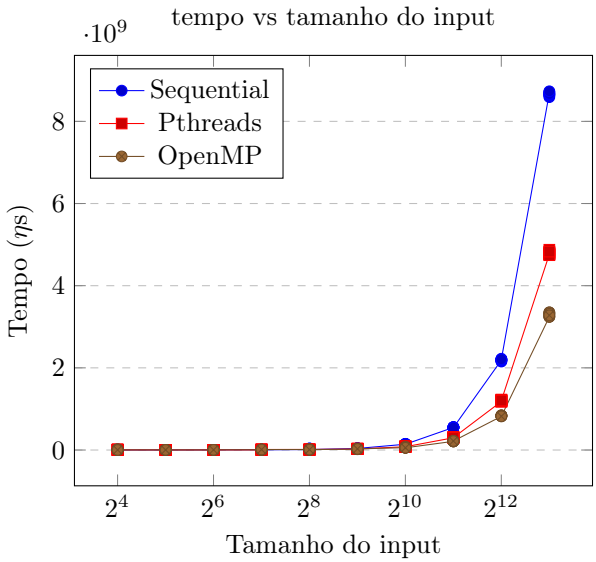


Figure 4: Tempo de execução

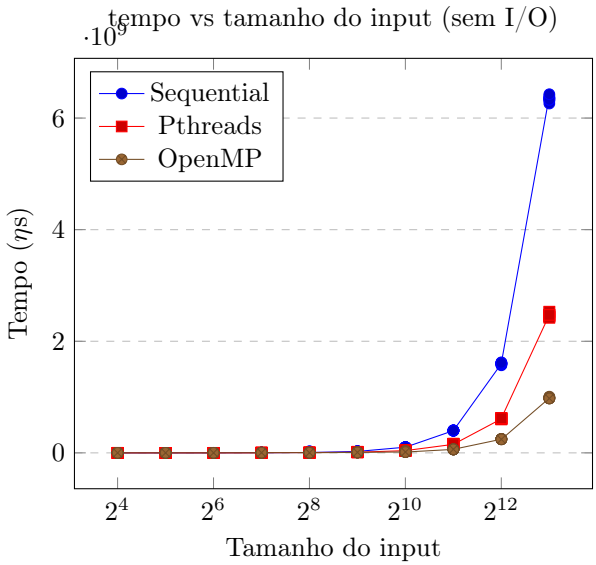


Figure 5: Tempo de execução

Visualmente falando, para valores menores, anteriores a 2^{10}, o processamento é rápido e os três programos têm tempos de execução próximos nessa região. Por outro lado, a partir de 2^{11}, os tempos de execução dos programas começam a se distanciar e as diferenças tomam dimensões mais expressivas. Destaque especial à execução sequencial, que é a mais lenta de todas, sucedida pela versão paralelizada por `pthreads` e, finalmente, pela paralelizada pela diretiva `OpenMP`. Notas-se que todas as implementações apresentam alguma tendência de cresimento análogo ao quadrático (nos gráficos, aparenta exponencial porque o eixo da abscissa está em escala logarítmica), contudo, para a versão sequencial isso é mais evidente, pois é mais discrepante.

Também separamos no caso de contagem de tempo na íntegra e no caso de apenas focado na função de computação de Mandelbrot. É notável que, para todos os tamanhos de entrada, a versão sem as operações de I/O foi a mais rápida, mas a tendendência de crescimento do tempo de execução conforme aumenta-se o tamanho da entrada permanece o mesmo comparada à da versão com as operações de I/O.

Vejamos os resumos estatísticos para a imagem da região `full` do conjunto de Mandelbrot, com 8 threads:

Table 4: Resumo estatístico para cada tamanho de entrada, não paralelizado (tempo em ηs)

T. Entrada	16	32	64	128	256	512	1024	2048	4096	8192	Geral
Média	3748565	2220935	3086089	8193020	20307120	40227870	143943000	551476900	2190725000	8667555000	1163148000
Desvio Padrão	4267803	307592	352230	2671507	10769790	3137942	4045968	4800715	20578885	43554130	2596149000
Mínimo	1890052	1904763	2392704	4174274	11218031	36960669	140171294	545064052	2152917996	8589147049	1890052
Quartil 1	2157654	1997287	2925640	5958908	11965660	37210240	141154100	548643700	2181909000	8642630000	3194473
Mediana	2446870	2158438	3093532	8425197	20307120	40227870	142735700	549925200	2191056000	8663921000	37017930
Quartil 3	2696230	2308729	3231320	9681825	26042060	43296670	145015700	554575300	2205699000	8698565000	549761500
Máximo	15857029	2883437	3724427	11791121	39879568	44013630	151481260	560659534	2222462349	8732536770	8732537000

Table 5: Intervalos de confiança para cada tamanho de entrada, não paralelizado (tempo em s)

T. Entrada	IC
16	$[0.0011034s, 0.00639373s]$
32	$[0.00203029s, 0.00241158s]$
64	$[0.00286778s, 0.003304399s]$
128	$[0.00653723s, 0.00984881s]$
256	$[0.01363206s, 0.02698218s]$
512	$[0.03828299s, 0.04217275s]$
1024	$[0.14143530s, 0.14645064s]$
2048	$[0.54850146s, 0.55445237s]$
4096	$[2.17797078s, 2.20348011s]$
8192	$[8.64055998s, 8.69454925s]$
Geral	$[0.654s, 1.672s]$

Para valores maiores, a expressividade temporal de execução é cada vez mais gritante. Note que houve um salto praticamente exponencial entre $N = 2048$ a $N = 4096$ e a $N = 8196$.

Table 6: Resumo estatístico para cada tamanho de entrada, paralelizado com `pthreads` (tempo em ηs)

T. Entrada	16	32	64	128	256	512	1024	2048	4096	8192	Geral
Média	5344066	2380960	2522616	8112897	9834288	27347950	82825560	304695200	1200678000	4791325000	643445200
Desvio Padrão	5471946	321897	171683	5422713	2841479	4424235	6574594	5908858	16586410	4015938	1434046000
Mínimo	2553165	1862763	2304457	3021718	7392902	22912132	77936909	297976060	1178975456	4742718729	1862763
Quartil 1	3129864	2135610	2403924	3510571	7564551	23233670	78867060	297976060	1178975456	4742718729	3212578
Mediana	3396378	2380960	2500582	8112897	20307120	40227870	142735700	303610300	1200678000	4784638000	218130700
Quartil 3	4576324	2600614	2652750	12648910	12904640	31034730	82690520	303610300	1200678000	4791325000	303426900
Máximo	20714009	2833505	2819155	16560785	13932616	34879215	98998683	316968962	1227974239	4874512280	4874512280

Para valores maiores, novamente, a expressividade temporal de execução é cada vez mais gritante. Novamente houve um salto praticamente exponencial entre $N = 2048$ a $N = 4096$ e a $N = 8196$. Apesar disso, os valores ainda são menores do que a versão sequencial.

Table 7: Intervalos de confiança para cada tamanho de entrada, paralelizado com `pthreads` (tempo em s)

T. Entrada	IC
16	$[0.00195258s, 0.00873555s]$
32	$[0.00218145s, 0.00258047s]$
64	$[0.00241621s, 0.00262903s]$
128	$[0.00475193s, 0.01147387s]$
256	$[0.00807315s, 0.01159542s]$
512	$[0.02460583s, 0.03009007s]$
1024	$[0.07875066s, 0.08690046s]$
2048	$[0.30103292s, 0.30835749s]$
4096	$[1.18978308s, 1.21034343s]$
8192	$[4.76643449s, 4.81621566s]$
Geral	$[0.362s, 0.925s]$

Table 8: Resumo estatístico para cada tamanho de entrada, paralelizado com a diretiva `OpenMP` (tempo em ηs)

T. Entrada	16	32	64	128	256	512	1024	2048	4096	8192	Geral
Média	4726754	2550544	2952499	8411455	10836970	26612870	61731630	215060300	832274600	3294185000	445934300
Desvio Padrão	6244310	189340	561676	3671873	3142508	5908208	8239403	5484098	8929192	4368821	985179300
Mínimo	2280305	2268908	2322737	4809328	6698960	17794071	54425582	207866000	818980538	3240413650	2268908
Quartil 1	2395373	2454120	2671696	5499719	9439182	22876780	56136560	211237000	818980538	3352304371	3205143
Mediana	2751860	2541905	2829941	6384403	10000820	26481040	54425580	213532300	831482700	3298068000	197355100
Quartil 3	3253589	2617276	3078768	11855090	12286160	29040610	63730590	219315800	837252700	3323117000	212881300
Máximo	22455384	2948046	4339750	13975649	17068501	39610494	81709849	225122834	847363764	3352304371	3352304371

Table 9: Intervalos de confiança para cada tamanho de entrada, a diretiva `OpenMP` (tempo em s)

T. Entrada	IC
16	$[0.00085656s, 0.00859695s]$
32	$[0.00243319s, 0.00266790s]$
64	$[0.00260437s, 0.00330062s]$
128	$[0.00613565s, 0.01068726s]$
256	$[0.00888926s, 0.01278468s]$
512	$[0.02295099s, 0.03027475s]$
1024	$[0.05662489s, 0.06683837s]$
2048	$[0.21166130s, 0.21845933s]$
4096	$[0.82674035s, 0.83780888s]$
8192	$[3.26710751s, 3.32126298s]$
Geral	$[0.253s, 0.639s]$

Sendo repetitivo, mas verdadeiro: Para valores maiores, novamente, a expressividade temporal de execução é cada vez mais gritante. Houve um salto exponencial entre $N = 2048$ a $N = 4096$ e a $N = 8196$. Mas valores ainda são menores do que a versão sequencial e paralelizado por `pthreads`.

O que podemos tirar disso? Parece que, com aumento do tamanho da entrada N, ocorre uma alteração direta no tempo de execução dos programas porque ele amplia a quantidade de trabalho necessário. Cada ponto na imagem representa um cálculo independente que precisa ser realizado para determinar se ele pertence ao conjunto de Mandelbrot. Portanto, à medida que aumentamos N, aumentamos o número total de pontos que o programa precisa processar de forma quadrática, resultando em mais operações de iteração e comparação para cada ponto — e consequentemente, maior tempo de execução.

Para cada valor de N, o número de cálculos cresce quadraticamente. Ao dobrar a resolução de uma imagem, passamos de uma matriz $N \times N (= N^2$ pontos) para $2N \times 2N (= 4N^2$ pontos), o que quadruplica o número total de pontos a serem calculados; dobrando de novo, temos $4N \times 4N (= 16N^2$ pontos). Esse crescimento exponencial (de 2^0N^2 para 2^2N^2 para 2^4N^2 pontos) conforme dobramos a entrada faz com que o tempo de execução aumente rapidamente, especialmente na versão sequencial, onde cada ponto é processado de maneira linear e não pode ser distribuído entre várias threads. Já nas versões paralelizadas, o impacto do aumento de N é atenuado, pois os cálculos são distribuídos, mas o tempo de execução ainda aumenta com N, já que há um limite para a quantidade de paralelismo que pode ser efetivamente aproveitado. Veremos isso em outra seção.

Assim, podemos concluir que existe um crescimento do tempo de execução exponencial conforme N dobra. Ele é inevitável, pois quanto maior for a entrada, maior será o número de operações de cálculo e maior será o tempo necessário para que o programa conclua a execução, e a máquina não tem recursos infinitos.

4 Como as soluções paralelas escalam com o número de threads?

Da maneira que paralelizamos com `pthreads` e com a diretiva `OpenMP`, podemos arbitrariamente escolher a quantidade de threads. Vejamos o desempenho de `mandelbrot_pth.c` e `mandelbrot_omp.c` conforme variamos a quantidade de threads. Considere 10 execuções para cada tamanho de entrada:

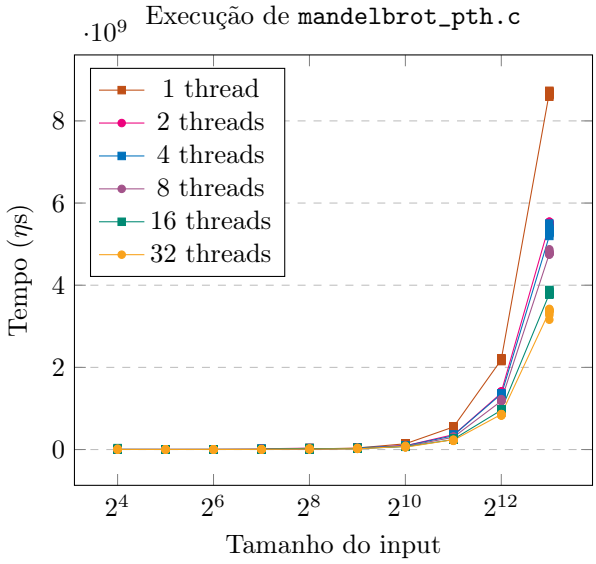


Figure 6: Tempo de execução

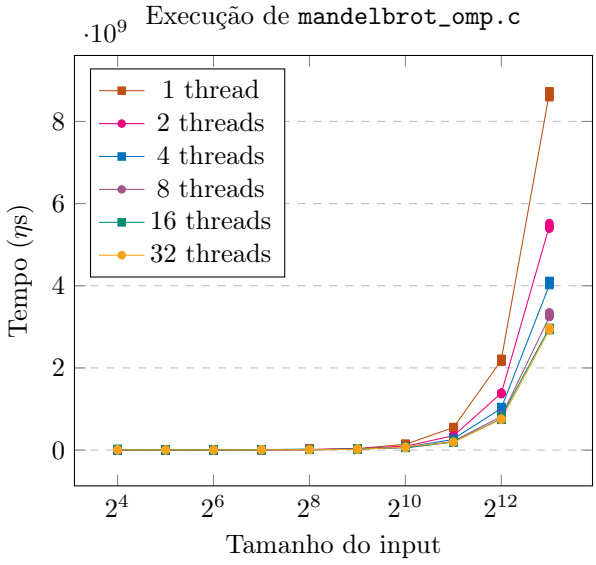


Figure 7: Tempo de execução

Claramente existe uma relação entre o número de threads e o desempenho do programa, uma vez que é visível que a execução com 1 thread (sequencial) é mais lenta que todas as outras; e que a de 2 threads é a segunda mais lenta, e assim por diante, até que cheguemos à de 32 threads, a mais rápida. Talvez seja conveniente ver esses gráficos em escala logarítmica:

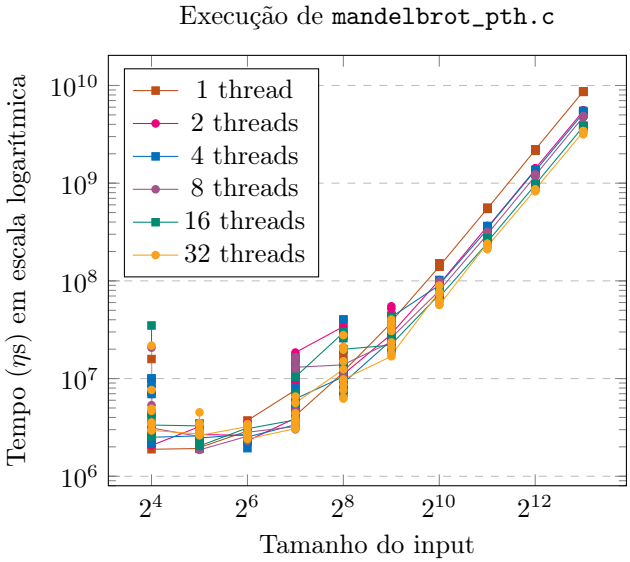


Figure 8: Tempo de execução

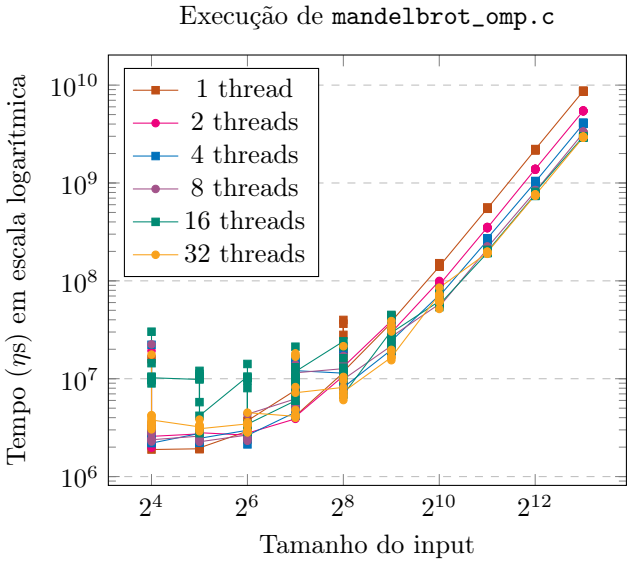


Figure 9: Tempo de execução

Apenas observando os gráficos, o aumento simultâneo do número de threads e do tamanho da entrada parece ter um comportamento semelhante ao logarítmico. Embora o desempenho melhore com mais threads – a linha de 32 threads, por exemplo, se encontra bem das demais – essa melhoria não segue um padrão linear. Quando dobramos o número de threads, aumentamos a capacidade de processamento paralelo, mas ao dobrar também o tamanho da entrada, a carga de processamento cresce significativamente, o que impacta a eficiência geral. Essa relação mais ou menos logarítmica pode ser observada nos gráficos de escala logarítmica, onde as linhas para diferentes números de threads se apresentam como retas mais ou menos na diagonal principal e têm padrões ascendentes semelhantes.

Note como para tamanhos muito pequenos o gargalo dos $1.5s$ das operações de I/O e alocação de memória se mostra muito mais dominante no tempo de execução que o cálculo do conjunto de Mandelbrot em si.

Podemos explorar melhor o impacto do número de threads observando apenas um tamanho de entrada. Fixemos $N = 2048$ e realizemos 10 execuções para cada quantidade de núcleos, e observemos o gráfico:

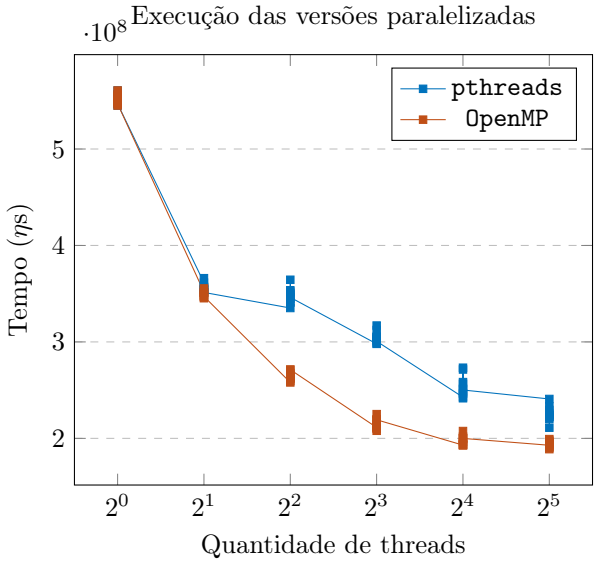


Figure 10: Tempo de execução

Visualmente, isso confirma que aumentar o número de threads diminui o tempo de execução, tanto de `mandelbrot_pth.c` como de `mandelbrot_omp.c`. O resumo estatístico pode nos dar uma análise mais profunda:

de novo, formatação que não funciona...

Table 10: Resumo estatístico para cada quantidade de threads, paralelizado com `pthreads` (tempo em ηs). Intervalo de confiança para estatísticas gerais: $[0.313s, 0.367s]$

Nº Threads	1	2	4	8	16	32	Geral
Média	551476915	356345587	347768735	304695204	253946636	225318679	339925300
Desvio Padrão	4800715	4565871	8177224	5908858	11065990	7966711	106684900
Mínimo	545064052	351301536	335158756	297976060	241576341	211060882	211060882
Quartil 1	548643700	354019200	342191800	301192600	246692500	222126200	251068700
Mediana	549925175	355367505	347613585	303610342	250762586	224220485	326063900
Quartil 3	554575303	357365470	351750043	305187599	257909194	228477202	356592600
Máximo	560659534	366064740	364443202	316968962	273264555	240842504	560659534

Table 11: Resumo estatístico para cada quantidade de threads, paralelizado com a diretiva `OpenMP` (tempo em ηs). Intervalo de confiança para média das estatísticas gerais: $[0.263s, 0.328s]$

Nº Threads	1	2	4	8	16	32	Geral
Média	551476915	349867494	264493015	215060315	197819663	194256527	295495700
Desvio Padrão	4800715	3487414	4552037	5484098	4874430	3344288	127433900
Mínimo	545064052	345299918	257981653	207866000	192782969	188813629	188813629
Quartil 1 (Q1)	548643700	347267300	261886000	211237000	193350000	191638700	199424200
Mediana (Q2)	549925175	349601433	263526066	213532262	197909760	195050377	241552200
Quartil 3 (Q3)	554575300	351573800	267306600	219315800	200217500	196305700	349053400
Máximo (Q4)	560659534	355362005	271282154	225122834	207595748	199255876	560659534

Numericamente, é mais fácil de ver aumentar o número de threads resulta em uma redução no tempo de execução do programa, o que era esperado dado o comportamento paralelo dos programas, mas que essa redução não segue um padrão linear. À medida que aumentamos as threads de 1 para 32, a média dos tempos cai significativamente, saindo de aproximadamente 0.551 segundos com uma thread para 0.225 segundos com 32 threads (`pthreads`) ou de 0.551 segundos para 0.194 (`OpenMP`); mas dobrar a quantidade de threads não parece diminuir o tempo pela metade. Pela figura 7 e pelo resumo da tabela, os tempos talvez pareçam seguir uma escala analoga à logarítmica de redução, provavelmente porque as regiões criticamente sequenciais representam um gargalo no tempo de execução e acabam criando uma assíntota horizontal no gráfico. O primeiro quartil para 1 thread está em torno de 0.548 segundos, ou seja, no mínimo, 75% das execuções com uma thread levaram mais do que esse tempo. Em contrapartida, com 32 threads, o Q1 cai para 0.222, mostrando como a maioria das execuções é significativamente mais rápida com maior paralelismo, mas o fato do tempo mínimo não cair drasticamente após 16 threads sugere que a partir de certo ponto o ganho marginal se reduz.

Enquanto com 1 thread o desvio padrão é baixo, de cerca de 0.00480072 segundos, conforme a quantidade de threads aumenta, o desvio padrão também aumenta. Provavelmente a distribuição dos recursos começa a afetar a estabilidade dos tempos à medida que mais threads são usadas. Afinal, com uma única thread, todo o cálculo é sequencial e por isso, o desvio padrão é relativamente baixo, já que não há interferências externas relevantes além das pequenas variações naturais do sistema; a variabilidade depende apenas de fatores externos do sistema, como o agendamento do processo pelo sistema operacional ou acesso à memória. Conforme aumentamos o número de threads, o desvio padrão começa a aumentar porque, embora o paralelismo reduza o tempo médio de execução, ocorre o desbalanceamento de carga entre as threads e a competição por recursos (como o uso simultâneo de núcleos ou memória compartilhada). Como no caso estamos trabalhando com o conjunto de Mandelbrot, cada thread pode ser atribuída a diferentes blocos do conjunto a calcular, mas nem todos os blocos têm a mesma complexidade, o que cria desigualdade nos tempos individuais de cada thread. A divisão do trabalho pode não ser tão eficiente; algumas threads podem terminar antes e ficar ociosas, enquanto outras continuam em execução. Também existe o fator hardware: essas execuções foram feitas em uma máquina de 16 núcleos, e como nem todos os núcleos estão disponíveis a todo momento, nem todas as threads tiveram um núclo exclusivo, e logo tiveram que compartilhar recursos. Isso pode ter sido um outro gargalo. E nota-se que nunca poderemos chegar a um tempo de execução virtualmente zero, porque existe o tempo constante das operações sequenciais de I/O e alocação de memória, que é algo na casa dos $1.5s$ para um tamanho de entrada $N = 2048$ na região `full`.

Então talvez uso de threads melhora o desempenho de forma significativa, mas com limites práticos, onde mais threads não significam sempre maior eficiência em igual proporção.

5 Qual o impacto da região escolhida no tempo de execução?

O conjunto de Mandelbrot é conhecido pela complexidade infinita em determinadas regiões do seu plano complexo. Dentro desse conjunto, existem regiões chamam a atenção por sua estrutura e relevância visual: O fractal em si na íntegra (`full`), que é a imagem completa do conjunto de Mandelbrot, abrangendo todas as áreas (densa e dispersa); o vale do cavalo-marinho (`seahorse valley`), que é conhecida pela aparência de "cavalos-marinhos" e rica em detalhes e com padrões complexos, mas também é menos exigente que outras regiões mais densas; o vale do elefante (`elephant valley`) que é caracterizada por formas que lembram elefantes, é também uma região complexa, mas com menos densidade que a espiral tripla; e a espiral tripla (`triple spiral`), que é uma das regiões mais complexas e densas, repleta de padrões intricados que precisam de muitas iterações para verificar se os pontos pertencem ao conjunto.

Alguns experimentos simples podem ajudar a responder a questão de como a escolha da região impacta o tempo de execução. Fixemos um número de threads — oito, por exemplo — e tentemos executar `mandelbrot_seq.c`, `mandelbrot_pth.c` e `mandelbrot_omp.c` 10 vezes cada um. Podemos ver como cada região afeta o tempo de execução:

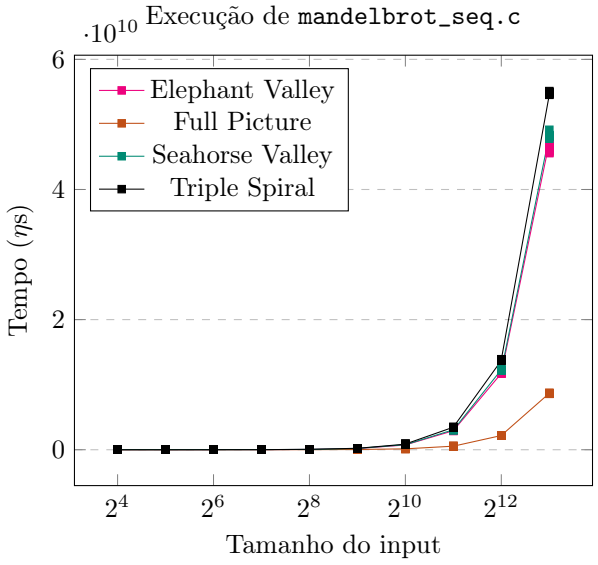


Figure 11: Tempo de execução para o programa sequencial

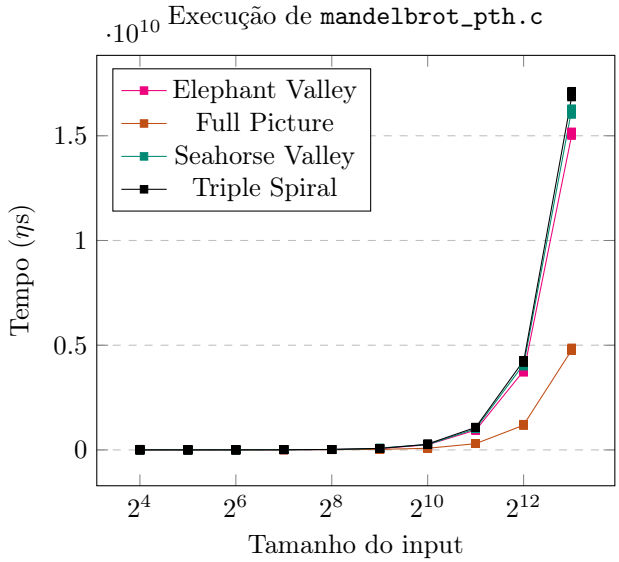


Figure 12: Tempo de execução para o programa paralelizado com `pthreads`

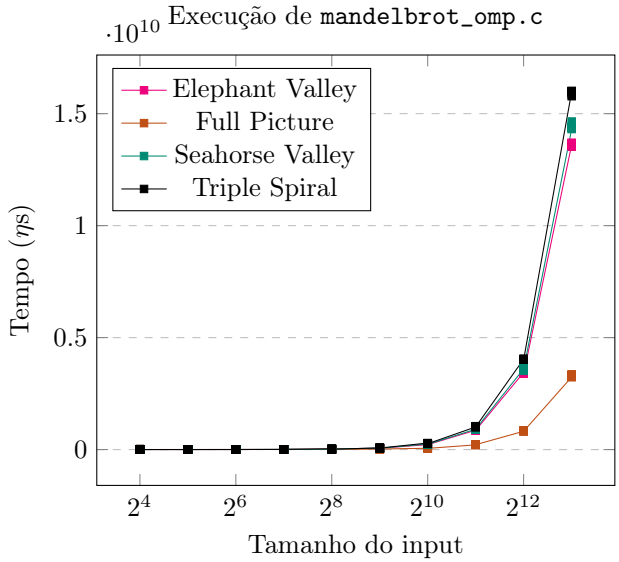


Figure 13: Tempo de execução para o programa paralelizado com a diretiva `OpenMP`

Visualmente falando, é bem óbvio que, para todos os tamanhos de entrada, a execução dos programas para a imagem inteira (`full`) é bastante veloz quando comparada a qualquer outra região, em particular à região da espiral tripla (`triple spliral`). A diferença nos tempos de execução entre dessa quatro regiões do Mandelbrot está relacionada principalmente à complexidade computacional de cada uma dessas áreas e como o paralelismo é aproveitado durante o processamento. Podemos verificar que essas diferenças numericamente também; considere essas tabelas, para o tamanho de entrada $N = 2048$ e 8 threads:

Table 12: Resumo estatístico para cada região, não paralelizado (tempo em ηs)

Região	`full`	`seahorse`	`elephant`	`triple spiral`
Média	551476900	3056148000	2929768000	3456474000
Desvio Padrão	480071500	32663440	20936730	27986340
Mínimo	545064052	3002865241	2893436763	3390331524
Quartil 1	548643700	3038704000	2924915000	3447728000
Mediana	549925200	3058932000	2931199000	3461631000
Quartil 3	554575300	3076377000	2938690000	3473330000
Máximo	560659534	3101855525	2962828737	3493082809

Table 13: Resumo estatístico para cada região, paralelizado com `pthreads` (tempo em ηs)

Região	`full`	`seahorse`	`elephant`	`triple spiral`
Média	304695200	1018960000	952572700	1081437000
Desvio Padrão	5908858	8448723	9685151	3862253
Mínimo	297976060	1007077025	938122492	1077677849
Quartil 1	301192600	1014634000	947807500	1078959000
Mediana	303610300	1018249000	951451600	1080857000
Quartil 3	305187600	1021358000	957715800	1081877000
Máximo	316968962	1036894833	971039447	1090327628

Table 14: Resumo estatístico para cada região, paralelizado com a diretiva `OpenMP` (tempo em ηs)

Região	`full`	`seahorse`	`elephant`	`triple spiral`
Média	215060300	913704900	868547800	1018069000
Desvio Padrão	5484098	5079572	5938393	10241580
Mínimo	207866000	908907286	861101293	1009959557
Quartil 1	211237000	909700300	864117900	1011808000
Mediana	213532300	911609400	868115400	1015312000
Quartil 3	219315800	916849100	870521300	1019318000
Máximo	225122834	922568101	881571740	1044653932

Pelas tabelas, podemos ver que a região `full` têm uma média de tempo $0.551477s$, $0.304695s$ e $0.21506s$ para os programas sequencial, paralelizado com `pthreads` e paralelizado com a diretiva `OpenMP` respectivamente, com respectivos intervalos de confiança das médias (95%) de $[0.654s, 1.672s]$, $[0.362s, 0.925s]$ e $[0.253s, 0.639s]$ caso assumíssemos uma distribuição normal.

Ela tem tempos de execução mais rápidos e consistentes em todas as implementações, o que faz sentido dado o tipo de carga que ela representa. Como essa área abrange a imagem completa do conjunto, a distribuição dos pontos tende a ser mais homogênea. Ou seja, embora alguns pontos exijam mais iterações para determinar se pertencem ao conjunto, a maioria não é tão custosa em termos de cálculo. Além disso, essa regularidade na distribuição facilita o paralelismo. Cada thread recebe uma carga de trabalho relativamente equilibrada e termina suas tarefas em tempos próximos, evitando que algumas fiquem ociosas enquanto esperam outras terminarem. Assim temos um uso mais eficiente dos recursos e em tempos de execução mais rápidos, especialmente nas versões paralelizadas.

Os intervalos de confiança para essa região também reforçam essa ideia de consistência no desempenho. Mesmo na implementação sequencial, onde o tempo de execução pode variar um pouco devido à carga de trabalho geral do sistema, o IC é suficientemente estreito para mostrar que essa variação não é muito significativa. Nas versões paralelizadas, a boa divisão da carga faz com que os tempos fiquem bem concentrados e previsíveis. O fato de o IC da versão com `OpenMP` ser ligeiramente mais estreito que o de `threads` pode ser porque o gerenciamento automático de threads pelo `OpenMP` consegue balancear a carga de forma ainda mais eficiente, mas não necessariamente; além disso o fato dessas IC se sobreporem parcialmente pode indicar que pode não haver uma grande vantagem clara entre as duas implementações.

Chamamos a atenção também para as semelhanças das regiões de `seahorse` e `elephant`, que visualmente são bem próximas em tempo de execução e nas estatísticas associadas. Afinal:

- A região de `seahorse` tem médias $3.0561s$, $1.019s$ e $0.914s$, com respectivos intervalos de confiança (%95) de $[3.629s, 9.301s]$, $[1.215s, 3.118s]$ e $[1.079s, 2.770s]$ para os programas sequencial, paralelizado com `pthreads` e paralelizado com a diretiva `OpenMP` respectivamente;
- A região de `elephant` tem médias $2.9298s$, $0.953s$ e $0.869s$, com respectivos intervalos de confiança (%95) de $[3.484s, 8.929s]$, $[1.133s, 2.906s]$ e $[1.025s, 2.626s]$ para os programas sequencial, paralelizado com `pthreads` e paralelizado com a diretiva `OpenMP` respectivamente.

Ambas as áreas têm padrões bastante detalhados, precisando de uma quantidade maior de iterações para determinar se cada ponto pertence ao conjunto, mas a natureza desses padrões é uma mistura de áreas densas e espaçadas, o que faz da carga de trabalho para processar tanto a `seahorse` como a `elephant` bem próxima. Nenhuma apresenta tantas zonas de carga excessivamente pesada, o que evita desequilíbrios extremos entre as threads e logo a carga total é distribuída de forma parecida entre as threads de `seahorse` e entre as de `elephant`, e portanto os tempos de execução são parecidos. Por isso, suas médias de tempo de execução são muito próximas em todas as implementações.

Note que a região `elephant` tem médias ligeiramente menores que as da `seahorse`. A pequena diferença nos tempos pode sugerir que, embora essas regiões compartilhem padrões de complexidade semelhantes, a `elephant` é ligeiramente menos custosa, possivelmente porque tem uma distribuição mais uniforme dos pontos em comparação à `seahorse`.

Os intervalos de confiança dessas duas regiões também reforçam essa similaridade. Ambos os ICs para o programa sequencial são um tanto amplos, indicando que o tempo pode ser afetado por pequenos desequilíbrios na complexidade dos pontos processados. No entanto, nas versões paralelizadas, especialmente com `OpenMP`, os intervalos são mais estreitos, mostrando que, com o paralelismo, a variabilidade no tempo de execução é menor. Isso sugere que tanto `pthreads` quanto `OpenMP` conseguem dividir a carga de forma eficiente, mas `OpenMP` parece ter um leve benefício em garantir tempos mais previsíveis e consistentes. Note a grande sobreposição dos intervalos do `seahorse` e do `elephant`. Como os intervalos indicam a faixa na qual esperamos que a média real dos tempos se situe com 95% de confiança, essa o tamanho das interseções sugere que, para diferentes execuções, o desempenho de ambas as regiões pode ser praticamente indistinguível. Em outras palavras, embora existam pequenas variações entre os tempos médios, essas diferenças em geral são tão sutis que caem dentro do mesmo intervalo esperado para variações aleatórias. Então talvez os padrões computacionais dessas duas áreas são muito próximos em termos de demanda de recursos, o que faz sentido considerando que as duas imagens têm mais ou menos a densidade de complexidade visual.

Por fim, a região de `triple spiral` tem médias de $3.4565s$, $1.0814s$ e $1.0181s$ e de longe os maiores desviões padrões de todos, as médias tendo respectivos intervalos de confiança (%95) de $[4.108s, 10.549s]$, $[1.274s, 3.270s]$ e $[1.201s, 3.074s]$ para os programas sequencial, paralelizado com `pthreads` e paralelizado com a diretiva `OpenMP` respectivamente. Essa área parece ser particularmente complexa e imprevisível. Em comparação com outras regiões, a `triple spiral` tem uma quantidade significativamente maior de pontos que exigem muitas iterações para determinar se pertencem ao conjunto de Mandelbrot. Então, a carga de trabalho flutua bastante, o que afeta tanto a consistência e o desempenho geral de tempo.

Os ICs são amplos, indicando que o tempo de execução pode variar bastante de uma rodada para outra, especialmente no programa sequencial. Mesmo nas abordagens paralelas com `pthreads` e `OpenMP`, os intervalos continuam largos, e logo o paralelismo não resolve completamente o problema de variabilidade. Nota-se a grande sobreposição dos ICs referentes às implementações paralelizadas; não importa qual método seja escolhido, a demanda de recursos para computar a região permanece mais ou menos a mesma. O grande desafio no `triple spiral` está justamente no desequilíbrio de carga: algumas threads ficam sobrecarregadas com blocos densos de cálculos, enquanto outras acabam ociosas, o que eleva o tempo total. Então isso faz com que o `triple spiral` seja a região mais lenta de todas.

6 Qual o impacto do Hardware no tempo de execução?

Finalmente, temos algumas observações quantos ao tipo de Hardware executado. Fizemos uma comparação com um segundo computador de especificações: Processador i5, 8 cores, 8 GB de RAM e versão do SO Ubuntu 20.04.6 LTS. Os programas foram mantidos, setados para $N = 8$ threads, sob a região `Full Picture`.

Note que com o uso de $N = 8$ threads, a segunda máquina não poderia usar todas as CPU's requisitadas para rodar os programas com 8 threads. Isso porque ela tem exatamente 8 núcleos e precisa manejar recursos para o próprio funcionamento, como o SO e outros processos, o que exige parte do uso dessas CPU's para atividades não relacionadas diretamente aos programas. Portanto, por limitações físicas, não se pode usar toda a capacidade do número de threads simultâneas escolhida e as melhorias que elas poderiam dar. Isso acaba sendo um ponto de interessante discussão.

Abaixo seguem os gráficos para as execuções de cada programa, diferindo no tipo de Hardware executado e se a medição foi considerando ou não as operações de input e output:

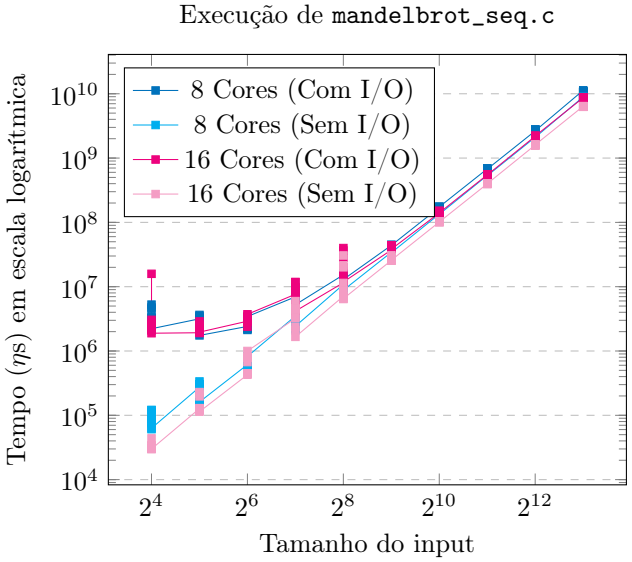


Figure 14: Tempo de execução para o programa sequencial

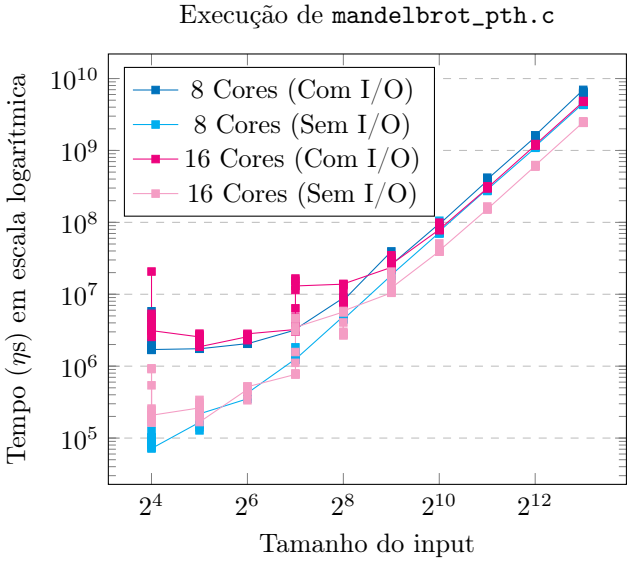


Figure 15: Tempo de execução para o programa paralelizado com `pthreds`

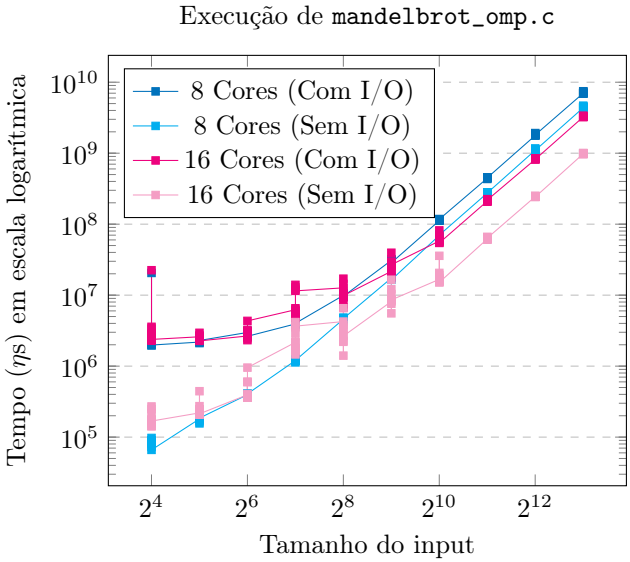


Figure 16: Tempo de execução para o programa paralelizado com a diretiva `OpenMP`

Com base na observação dos gráficos, em geral a maior quantidade de núcleos tem um impacto na capacidade e complexidade de execução dos programas envolvidos. Também porque a máquina mais forte pode usar 8 CPU's (além de outros auxiliares), ao passo que o outro não. Isso acaba tendo impactos na velocidade de execução para uma mesma saída requisitada.

Para a execução sequencial do `mandelbrot_seq.c`, os gráficos seguem o que já foi previsto, dado seu respectivo par na mesma máquina, a versão sem operações de I/O é mais rápida do que com elas. Por outro lado, as diferenças de tempo entre os Hardwares é gritante. Inicialmente, porque a máquina com 8 cores teve sempre um desempenho mais lento do que o de 16. Em segundo lugar, note que a medição sem I/O da máquina mais fraca teve um desempenho equivalente ao que considera essas operações extraídas da máquina mais potente. Ainda assim, para tamanhos de entrada menores, anteriores à 2^{10}, o desempenho é semelhante entre as máquinas.

Para a execução do programa paralelizado com Pthreads, `mandelbrot_pth.c`, os gráficos também seguem um esquete semelhante quanto aos pares entre as versões com e sem I/O medidos nas próprias máquinas. Por outro lado, a cena se repetiu, em que o computador com 8 cores sempre demonstrou um desempenho mais lento, sobretudo, com a medição sem I/O extraído dela tenha sido comparável à com I/O obtido da outra máquina.

Finalmente, para a execução do programa paralelizado com OpenMP, `mandelbrot_omp.c`, temos uma distribuição mais disjunta sobre os desempenhos. Além dos observados acima, a medição sem I/O do computador com 8 cores teve um desempenho pior do que a medição com I/O da máquina mais potente. Isso, sobretudo, pode evidenciar que, até as melhorias e técnicas mais eficientes sobre o algoritmo podem ser influenciados pelas limitações físicas de onde se executa o programa.

7 Conclusão

Para cada experimento, fizemos 10 execuções cada, tanto para as medições considerando com ou sem operações de I/O e alocação de memória.

Em resumo, no trabalho, pudemos observar que as execuções com operações de I/O carregam um custo constante, que não pode ser melhorado com computação paralela, uma vez que sejam partições puramente sequenciais. Assim, mesmos para os programas com as maiores e mais eficientes otimizações com esses recursos, eles sempre serão limitados pela sua parte não paralelizável.

A posteriori, o tamanho da entrada no tempo de execução também tem sua influência na complexidade final. Quanto maior a saída requisitada, mais computação, maior uso da RAM e maior alocação de armazenamento para os processamentos. Logo, certamente, programas com saída menores exigem menos tempo de computação do que com as maiores. Portanto, tamanho de entrada é um gargalo para a otimização de tempo na execução de programas.

Com efeito, o número de Tthreads tem um impacto significativo na execução dos programas, uma vez que podem repartir o trabalho de forma simultânea e encerrá-lo de forma muito mais efetiva e mais rápida, reduzindo a carga de computação de um processo único. Isso pode ser expressivo para melhorar a questão anterior sobre entradas cujos tamanhos sejam expressivamente grandes. Contudo, não se pode imaginar que o desempenho seja proporcional à quantidade de threads, uma vez que limitações físicas de Hardware como número de núcleos, ou problemas de natureza não tão uniformes podem tornar muitos delas ociosas, fazendo uma distribuição de maior densidade computacional para umas do que para outras. Pudemos observar isso pela tendência assintótica dos gráficos. Portanto, computação paralela precisa encontrar um ponto de equilíbrio quanto ao número de threads requisitadas para resolver um problema.

Ademais, a complexidade inerente ao problema é outra variável de interesse. No conjunto de Mandelbrot, pudemos notar que regiões menos densas e menos detalhadas, como a região `full`, tinham um desempenho mais rápido do que outras, como o `elephant` e o `seahorse`. Note que estas regiões, por sua natureza mais detalhistas, não permitiam uma distribuição tão uniforme e justa de trabalho entre as Threads e, portanto, resultavam na ociosidade de muito delas e resultando num desempenho não tão satisfatório final. Assim, a natureza do problema também é um gargalo para a programação paralela.

Por fim, o tipo de Hardware tem impacto no desempenho final do programa. Mesmo as otimizações mais eficientes algorítmicas podem ser limitados por fatores físicos, como número de núcleos, tipo de processador, sistema operacional, quantidade de processos simultâneos abertos e quantidade de RAM da máquina. Como pudemos observar, para os mesmos programas, um computador com 16 cores teve um desempenho melhor do que para um de 8.