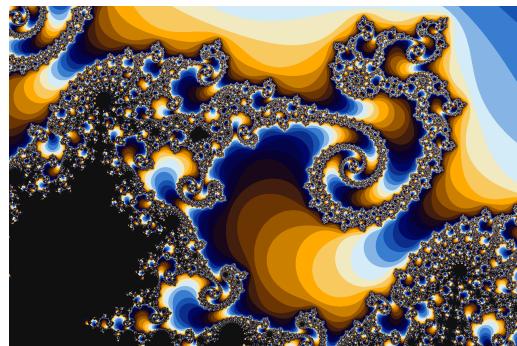


INSTITUTO DE MATEMÁTICA E ESTATÍSTICA - USP

MAC5742-0219: EP1 - Relatório dos Experimentos



Integrantes do Grupo:

Bruno Ferrero	N USP: 3690142
Marcelo Galdino	N USP: 10120500
Willy Reis	N USP: 7694112

Professor: Alfredo Goldman

Monitor: Pedro Bruel

S Ã O P A U L O

1º semestre 2017

Sumário

1	Introdução	2
2	Experimentos	2
2.1	Implementação	3
2.1.1	OpenMP	3
2.1.2	Pthreads	3
2.2	Descrição dos experimentos	4
2.2.1	Computador pessoal (PC)	4
2.2.2	Google Cloud Engine (GCE)	5
3	Resultados dos Experimentos	5
3.1	Computador pessoal (PC)	5
3.2	Google Cloud Engine (GCE)	10
3.2.1	I/O e alocação de memória	10
3.2.2	Variação do número de threads	10
3.2.3	Comparação dos algoritmos	15
4	Discussão dos Resultados	17

1. Introdução

O presente EP trata de uma família de fractal denominada de *Conjunto de Mandelbrot*¹. O objetivo deste EP é explorar a característica “*facilmente paralelizável*” do problema e com isso utilizar técnicas e recursos de paralelização (*Pthreads* e *OpenMP*) para comparar o desempenho de cada uma das tecnologias.

Abaixo ilustramos as 4 figuras indicando as regiões do fractal que serão calculadas pelos códigos implementados.

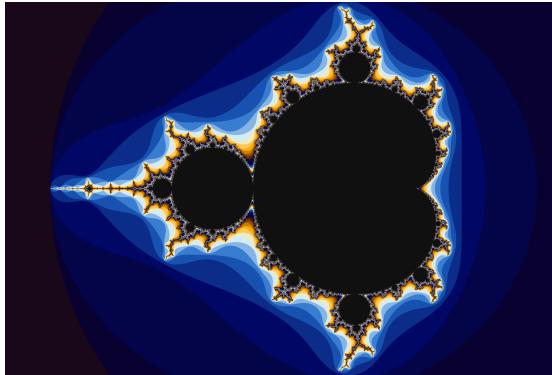


Figura 1: *Full Picture*

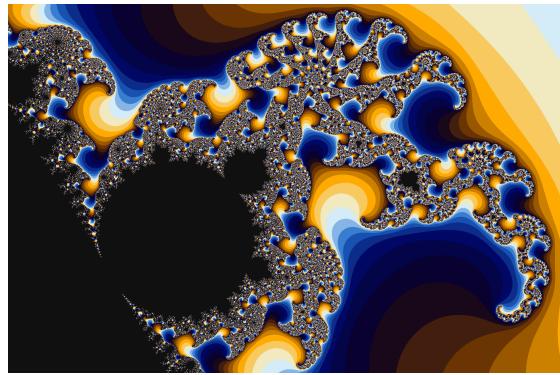


Figura 2: *Elephant Valley*

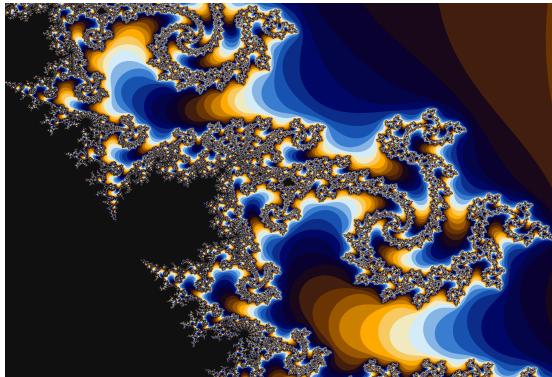


Figura 3: *Triple Spiral Valley*

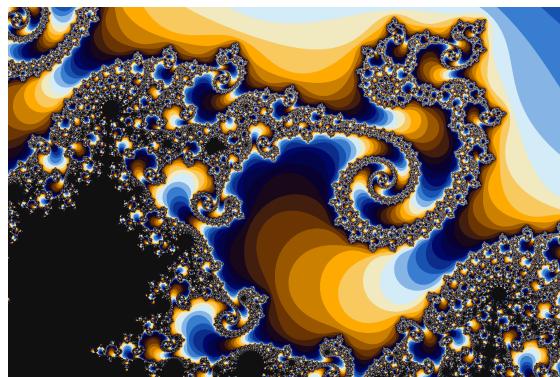


Figura 4: *Seahorse Valley*

Figura 5: Regiões do Conjunto de Mandelbrot

2. Experimentos

Foram realizados experimentos em dois computadores distintos, uma instância do tipo n1-standard-8 no Google Cloud Engine para realização dos testes solicitados, conforme

¹https://en.wikipedia.org/wiki/Mandelbrot_set

o enunciado do EP, e uma máquina alternativa, em que foram realizados testes extras. Ambos serão detalhados posteriormente na seção 2.2.

2.1. Implementação

O problema do *Conjunto de Mandelbrot*² é um problema embaranhosamente ou perfeitamente paralelizável, isto é, pode ser paralelizado de maneira trivial ou complexa, uma vez que cada ponto da imagem é calculado de maneira independente, não havendo necessidade de comunicação entre as tarefas paralelas.

Além da versão sequencial, fornecida, foram implementadas duas versões paralelizadas para a realização dos testes: uma utilizando as diretivas de compilação *OpenMP*, e uma utilizando *Pthreads*.

2.1.1. OpenMP

A API OpenMP utiliza diretrizes de pré processamento para fazer a divisão de problemas em subtarefas passíveis de execução paralela.

Foram implementadas duas versões do algoritmo utilizando OpenMP. A primeira versão paraleliza apenas o **for** externo e mantém uma cópia de cada variável da função *compute_mandelbrot* para cada *thread*, com exceção da constante de escape. A API, de forma padrão, divide o **for** para o número de *threads* definido, fazendo com que cada uma tenha hipoteticamente um número similar de iterações.

A segunda implementação paraleliza os dois **for** mais externos. Primeiramente, a variável *c_y* é mantida privada para cada *thread*, enquanto o **for** é dividido automaticamente. O segundo for mantém o restante das variáveis privadas, com exceção da constante de escape. Em ambos os casos os iteradores do **for** não são tornados privados.

2.1.2. Pthreads

Considerada uma padronização para programação com *threads*, a POSIX Threads, Pthreads, foram especificadas pela IEEE POSIX 1003.1c

Foi implementada apenas uma versão do algoritmo paralelo utilizando Pthreads. Diferentemente do algoritmo utilizando OpenMP, essa versão precisa que a divisão do for para cada *thread* seja feita explicitamente. Então, dado um número de *threads*, o *for* é dividido e executado para cada uma dessas partes paralelamene.

²B. B. Mandelbrot - The Fractal Geometry of Nature, Ed. Freeman

2.2. Descrição dos experimentos

Como o objetivo do EP era explorar o desempenho dos códigos sequencial e paralelos, a principal variável a ser analisada em cada experimento é o tempo de execução de código. Também foram analisadas outras variáveis (número de instruções, etc.), mas estas acabaram não sendo apresentadas neste relatório, uma vez que a máquina do Google Cloud Engine (GCE) não suporta para tal recurso.

Os testes iniciais e as primeiras análises dos resultados foram geradas em um computador pessoal (PC). Uma vez implementados os códigos nas versões *pthreads* e *OpenMP* esses mesmos códigos foram utilizados para gerar os resultados na GCE. A seguir são apresentados as características de cada uma das máquinas e dos experimentos realizados.

2.2.1. Computador pessoal (PC)

Os experimentos e resultados gerados no PC foram realizados em uma máquina com a seguinte configuração:

```
gcc version 4.9.2 (Debian 4.9.2-10)
bruno@iceberg:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:  0-15
Thread(s) per core:   2
Core(s) per socket:   8
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Core(TM) i7-5960X CPU @ 3.00GHz
Stepping:               2
CPU MHz:                1205.742
CPU max MHz:           3500.0000
CPU min MHz:           1200.0000
BogoMIPS:               5999.70
Virtualization:        VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:                256K
L3 cache:                20480K
NUMA node0 CPU(s):     0-15
```

Os experimentos realizados nessa máquina foram:

- **exp_pc_01:** 10 execuções (por meio do `perf`) de cada uma das versões (sequencial, *OpenMP*, *pthreads*) para cada uma das regiões apresentadas anteriormente;
- **exp_pc_02:** nesse experimento foram apenas executadas as versões *OpenMP*, *pthreads* do código. O objetivo era aumentar o número de threads de 2 até 512 e analisar o desempenho de cada um dos códigos;
- **exp_pc_03:** esse experimento consiste de 10 execuções de cada uma das versões (sequencial, *OpenMP*, *pthreads*) apenas para a região *Full*. O objetivo desse experimento é analisar a distribuição dos tempos tempo para cada uma das implementações.

2.2.2. Google Cloud Engine (GCE)

A máquina virtual criada na GCE é do tipo **n1-standard-8** e possui 8 *cores* e 30 GB de memória. Nela foram realizadas os seguintes experimentos:

- **exp_gce_01:** 10 execuções para cada região e para cada tamanho da figura por meio de variações do algoritmo sequencial, com alocação de memória e com I/O, com alocação de memória e sem I/O, e sem alocação de memória e sem I/O. Seu objetivo foi encontrar o impacto de tais operações sobre o algoritmo sequencial;
- **exp_gce_02:** 10 execuções dos algoritmos paralelos (*pthreads* e *OpenMP*) variando o número de *threads* de 1 a 32, para cada região do Conjunto de Mandelbrot para cada tamanho da imagem. Tal experimento foi realizado com a finalidade de identificar o comportamento dos algoritmos paralelos com relação ao número de *threads* levando em consideração o tamanho da imagem;
- **exp_gce_03:** 10 execuções de variações do algoritmo *OpenMP*, uma utilizando a paralelização através apenas do *for* mais externo e uma utilizando paralelização nos dois *fors* mais externos. O experimento visa verificar se alguma especificidade na implementação poderia influenciar no desempenho dos algoritmos;

3. Resultados dos Experimentos

3.1. Computador pessoal (PC)

Os resultados dos experimentos realizados com o PC são descritos e mostrados a seguir:

A figura 6 mostra tempo de execução de cada uma das implementações em função do tamanho da entrada para o `exp_pc_01`. Nota-se claramente um ganho de performance nas versões paralelas já tamanhos de entrada superior a 1024.

Com relação a variabilidade dessas medidas, a figura 7 mostra o valor médio das 10 execuções junto com sua barra de erro. Como a variabilidade do tempo entre as 10 execuções foi muito pequena em relação ao valor médio, a barra de erros na figura 7 quase não aparece, ficando quase imperceptível.

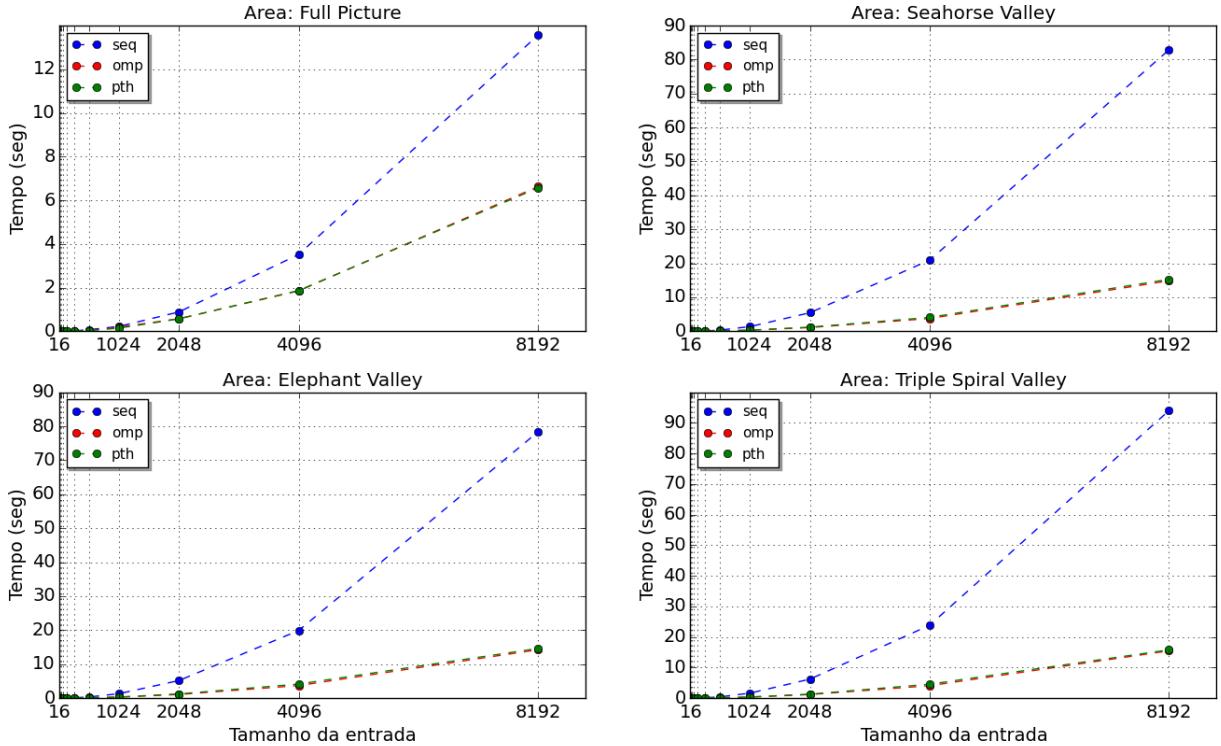


Figura 6: `exp_pc_01` Tempo de execução dos experimentos.

A figura 8 refere-se ao experimento `exp_pc_02`. Nela observa-se o desempenho dos códigos paralelos (*OpenMP* e *threads*) em função do tamanho da entrada e em função do número de *threads*. Nessa figura observa-se que após 256 *threads* já não há mais ganho de performance, inclusive ocorrendo perda da mesma.

A figura 9 mostra o uso de CPU e de memória da máquina (PC) no instante de execução do código paralelo *OpenMP* para uma entrada com tamanho 8192.

A figura 10 mostra um *boxplot* do experimento `exp_pc_03`. Nessa figura observa-se a distribuição (do tempo em segundos) de 10 experimentos para cada uma das implementações. Como a implementação sequencial leva muito mais tempo do que as implementações paralelas foi subtraído um valor arbitrário de segundos para conseguirmos visualizar os 3

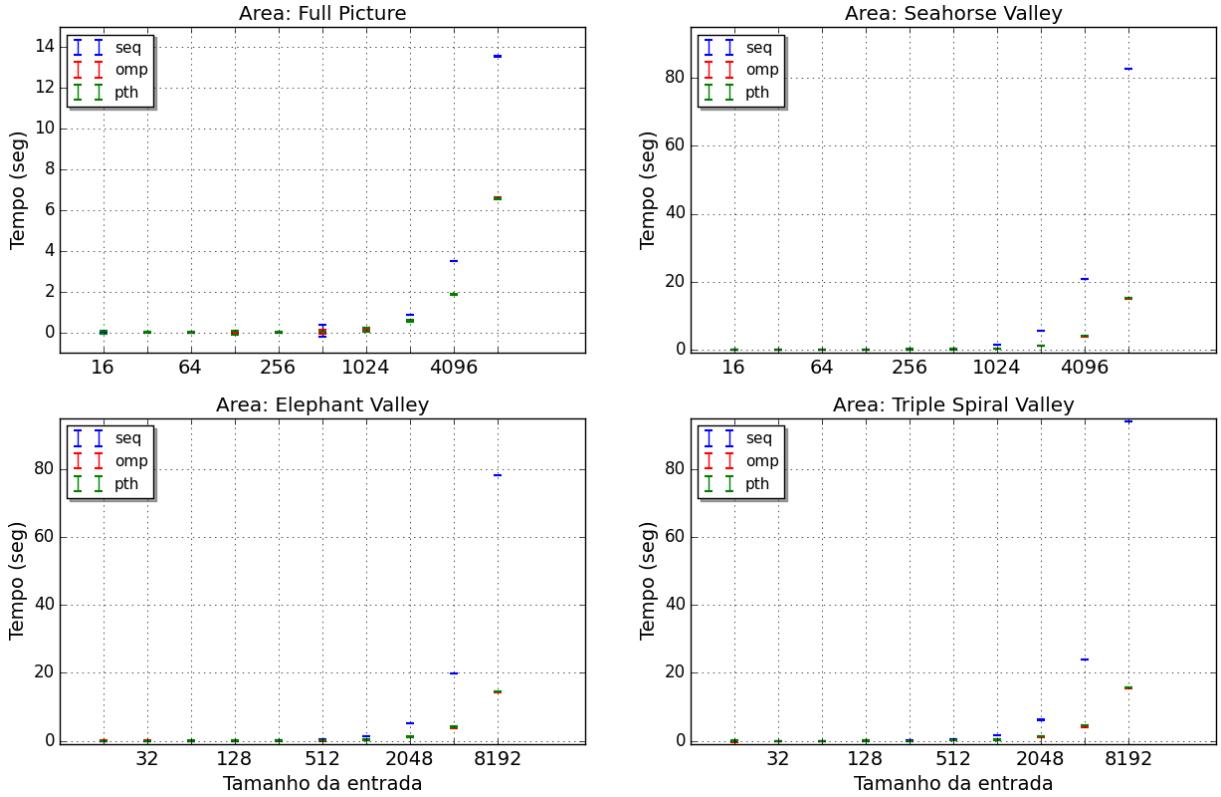


Figura 7: exp_pc_01 Tempo de execução médio e barra com 2 desvios padrões.

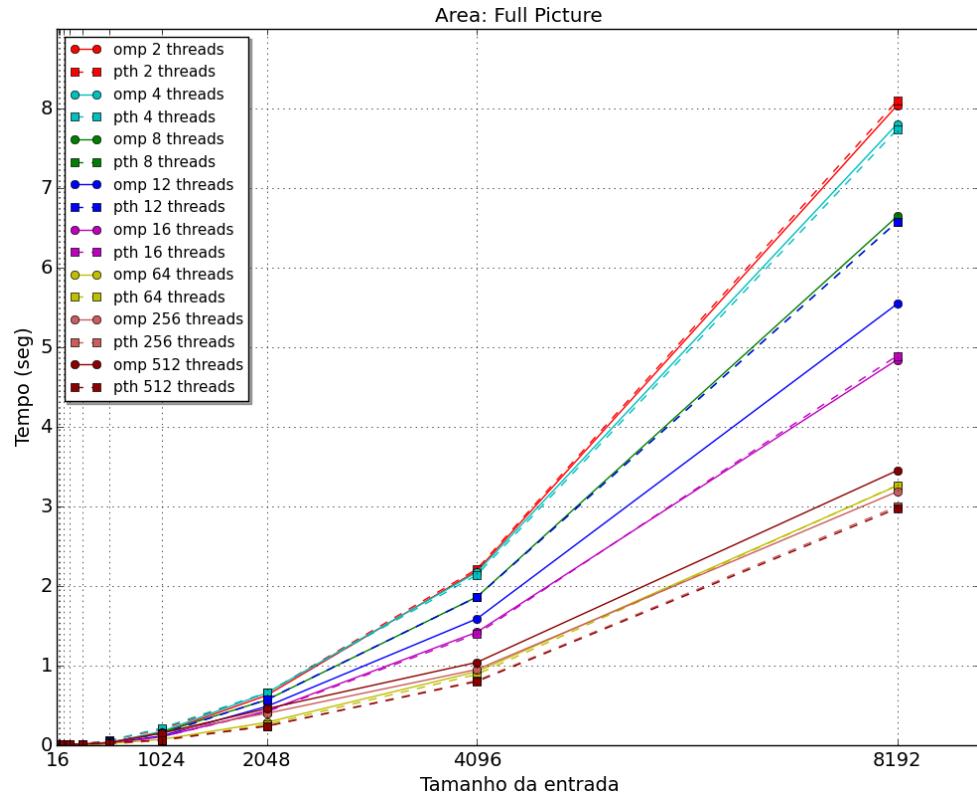


Figura 8: exp_pc_02 Tempo de execução dos experimentos em função do número de threads.

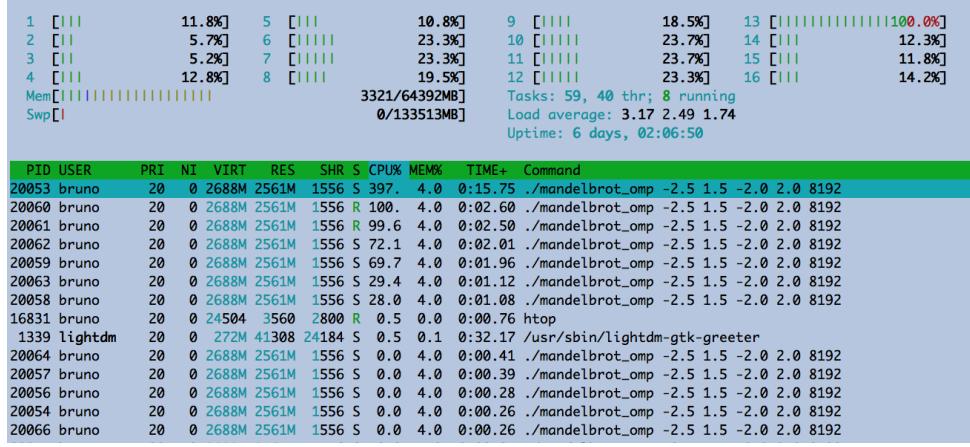


Figura 9: exemplo de saída do *htop* em uma máquina com 16 cores rodando uma instância do problema que utilizava 16 threads

boxplots em um mesmo intervalo de tempo (eixo de tempo). Apesar de haver diferenças nos tempos de execução de cada execução, esses tempos são relativamente pequenos e não encontramos nenhum padrão relevante nas distribuições.

Também foi feito um teste com uma entrada de tamanho 2^{15} para ver o consumo de memória da máquina, ilustrado pela figura 11. Esse teste foi feito meramente para ver o desempenho da máquina para uma entrada muito grande.

Em geral, os experimentos feitos com computador pessoal mostrou:

- Ganho substancial de performance nas versões paralelizadas em relação à versão sequencial;
- Não foi observado diferenças significativas nas performances da implementação em *OpenMP* e *pthreads* (*pthreads* pareceu ligeiramente mais rápido);
- Houve um ganho de performance (nas implementações paralelas) com o aumento do número de *threads*. No entanto, após 64 *threads* não observa-se mais ganhos, e no caso de 512 *threads* houve uma perda de performance em relação ao uso de 64 *threads*;
- A variabilidade do tempo consumido pelas diversas execuções para as diferentes implementações não foi significativa, apresentando um desvio muito pequeno em relação aos valores médios. Assim sendo, optou-se por não apresentar mais essas análises para os demais resultados (GCE).

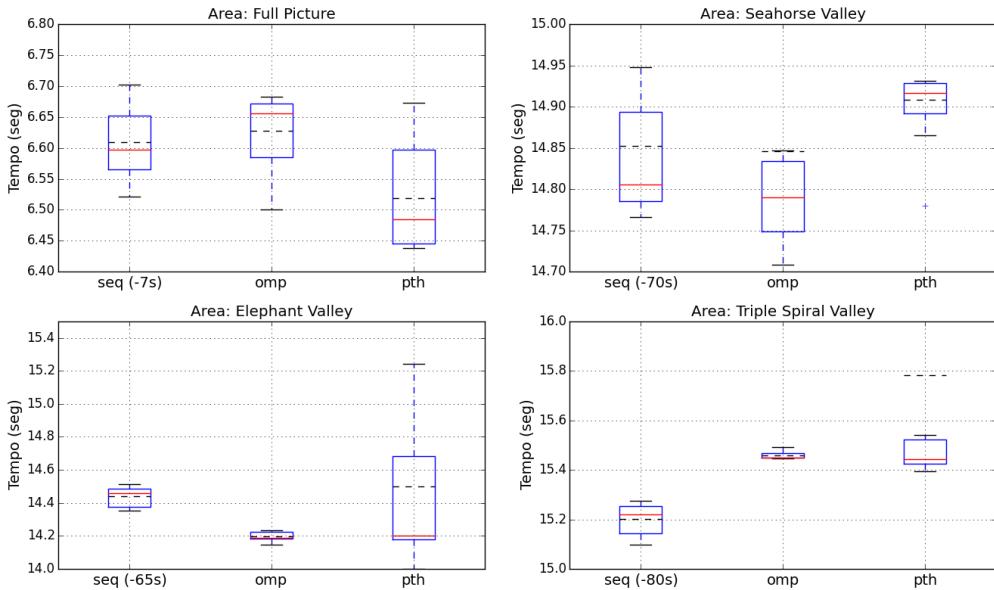


Figura 10: `exp_pc_03` Boxplot baseados em 10 experimentos com tamanho de entrada 2^{13} . A linha vermelha em cada um dos boxplots indica a média em segundos dos 10 experimentos. Para os boxplots da execução sequencial foi subtraído um valor arbitrário indicado na figura.

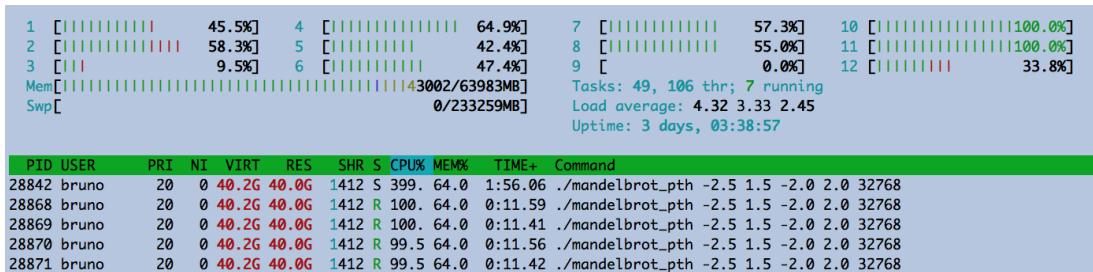


Figura 11: `htop`: teste com tamanho de entrada 2^{15} .

3.2. Google Cloud Engine (GCE)

3.2.1. I/O e alocação de memória

Neste teste foram realizados os levantamentos de tempo gasto com alocação de memória e tempo gasto para geração e impressão da imagem por meio do algoritmo sequencial. Segundo a figura 12 é possível perceber que os tempos de alocação de memória e de impressão da imagem são similares e de certa forma significativos em tempo de execução para a área *full* e pouco significativo para as demais áreas.

Levando em consideração a tabela 1, podemos dizer que o impacto da entrada e saída varia de 13% a 16% para as instâncias de tamanho 512 em diante, enquanto que o impacto com alocação de memória varia de 13% a 27% no geral, e de 13% a 16% para instâncias maiores ou iguais a 512.

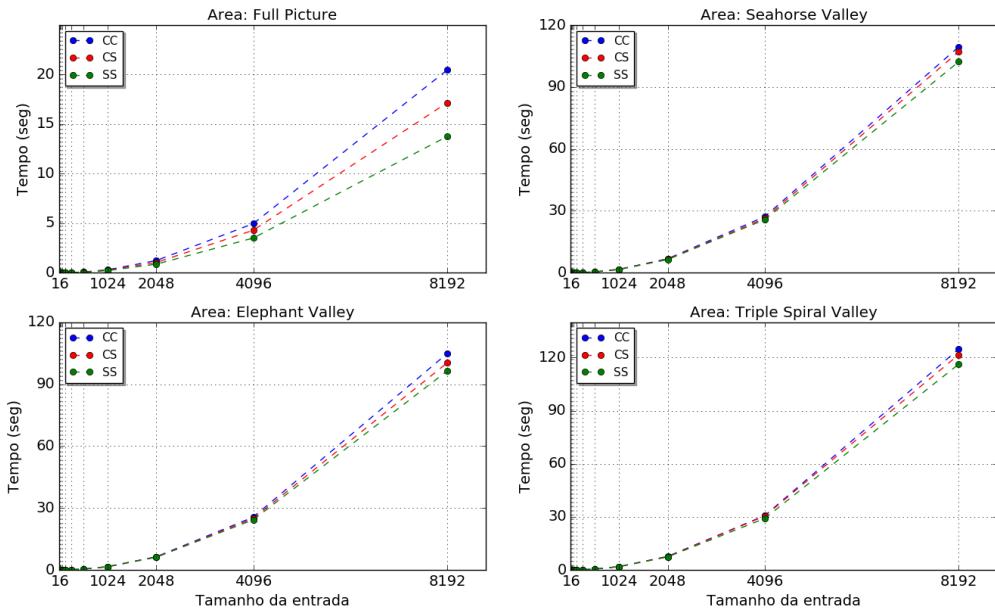


Figura 12: Tempo de execução do algoritmo sequencial com alocação de memória e com I/O (CC), com alocação de memória e sem I/O (CS) e sem alocação de memória e sem I/O (SS).

3.2.2. Variação do número de threads

Neste experimento foram realizados testes variando o número de *threads* para cada uma das quatro áreas do problema, para cada um dos tamanhos da figura com cada um dos algoritmos. O que pôde ser observado é que dependendo do tamanho do problema, o número de *threads* pode trazer benefícios ou prejuízos em questão de tempo de execução.

Tamanho da imagem	CC	CS	SS	I/O	Alocação
16	0,000761449	0,000711458	0,000594982	0,000049991	0,000116476
				6,57%	15,30%
32	0,00103361	0,001047552	0,000879873	-0,000013942	0,000167679
				-1,35%	16,22%
64	0,00194916	0,002049922	0,001505325	-0,000100762	0,000544597
				-5,17%	27,94%
128	0,005963662	0,005288959	0,004072026	0,000674703	0,001216933
				11,31%	20,41%
256	0,019875776	0,018159536	0,014418098	0,00171624	0,003741438
				8,63%	18,82%
512	0,077914966	0,067432646	0,054653457	0,01048232	0,012779189
				13,45%	16,40%
1024	0,302047558	0,262282543	0,217618714	0,039765015	0,044663829
				13,17%	14,79%
2048	1,227604506	1,030623247	0,862973884	0,196981259	0,167649363
				16,05%	13,66%
4096	4,955876546	4,287269325	3,499165869	0,668607221	0,788103456
				13,49%	15,90%
8192	20,463040039	17,169718399	13,761980548	3,29332164	3,407737851
				16,09%	16,65%

Tabela 1: Tempo de execução (em segundos) do algoritmo sequencial com alocação de memória e com I/O (CC), com alocação de memória e sem I/O (CS) e sem alocação de memória e sem I/O (SS) para a área **full**. A partir desses tempos, são calculados os tempos estimados de I/O e de alocação de memória

Para exemplificar tal afirmação mostraremos os gráficos de acordo com o tamanho da figura gerada.

Foram implementadas duas versões do algoritmo *OpenMP*, uma paralelizando apenas o **for** mais externo (*omp*), e uma paralelizando os dois **for** mais externos (*omp2*). Os testes realizados mostram que para todas as áreas do tipo *Valley* a versão *omp* é superior a *omp2*. Por este motivo a versão *omp* foi selecionada para comparações com o algoritmo utilizando *pthreads*.

O gráfico 13 mostra que partindo de uma *thread*, houve uma diminuição no tempo de execução ao aumentar o número de *threads* para 6 em ambos os algoritmos. Ao aumentar mais o número de *threads* houve um aumento do tempo médio de execução, provavelmente devido ao *overhead* gerado pelo gerenciamento das *threads*, o que tornou a execução mais lenta para uma instância pequena do problema. No geral, ambos os algoritmos apresentaram resultados similares.

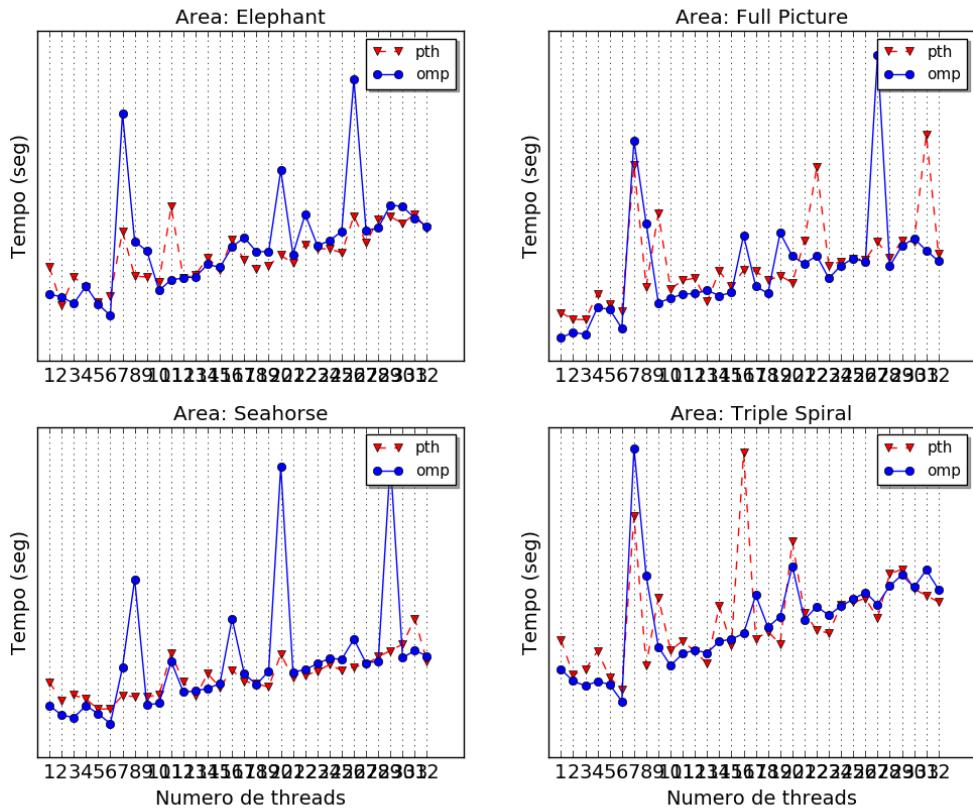


Figura 13: *Tempo de execução dos algoritmos paralelos utilizando Pthreads (pth) e usando OpenMP (omp) variando o número de threads para as quatro áreas do problema para a imagem de 8192 pixels.*

Comportamento similar é repetido com as próximas instâncias, até que na instância

de tamanho 128, o aumento do número de *threads* para maior que 10 passa a manter o tempo de execução estabilizado para todas as áreas, exceto a área *full*, como mostrado na figura 14. Nota-se o algoritmo baseado em *pthreads* apresenta melhor desempenho para um número de *threads* maior do que 8 para ambos os algoritmos.

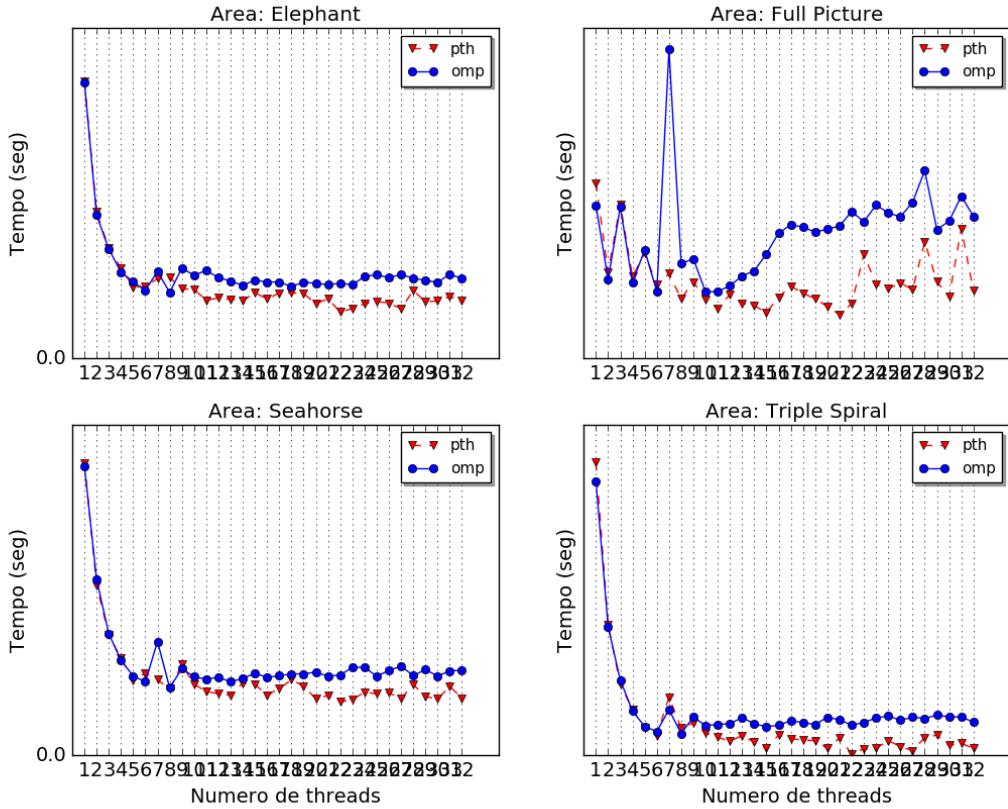


Figura 14: *Tempo de execução dos algoritmos paralelos utilizando Pthreads (pth) e usando OpenMP (omp) variando o número de threads para as quatro áreas do problema para a imagem de 8192 pixels.*

A partir da instância de tamanho 1024, a área *full* passa a ter uma melhoria de desempenho de acordo com o aumento do número de *threads*. Tal comportamento continua válido até a execução com 32 *threads*, apesar do ganho passar a ser cada vez menor. As demais áreas possuem uma melhoria de tempo de execução ao aumentar o número de *threads* de 1 até 8, e mantém esse tempo estável ao aumentar este número. A figura 15 apresenta tal comportamento.

Apesar de ambos os algoritmos apresentarem resultados muito similares, para todos as áreas, o algoritmo implementado utilizando *pthreads* possui um desempenho um pouco superior ao algoritmo utilizando *OpenMP*.

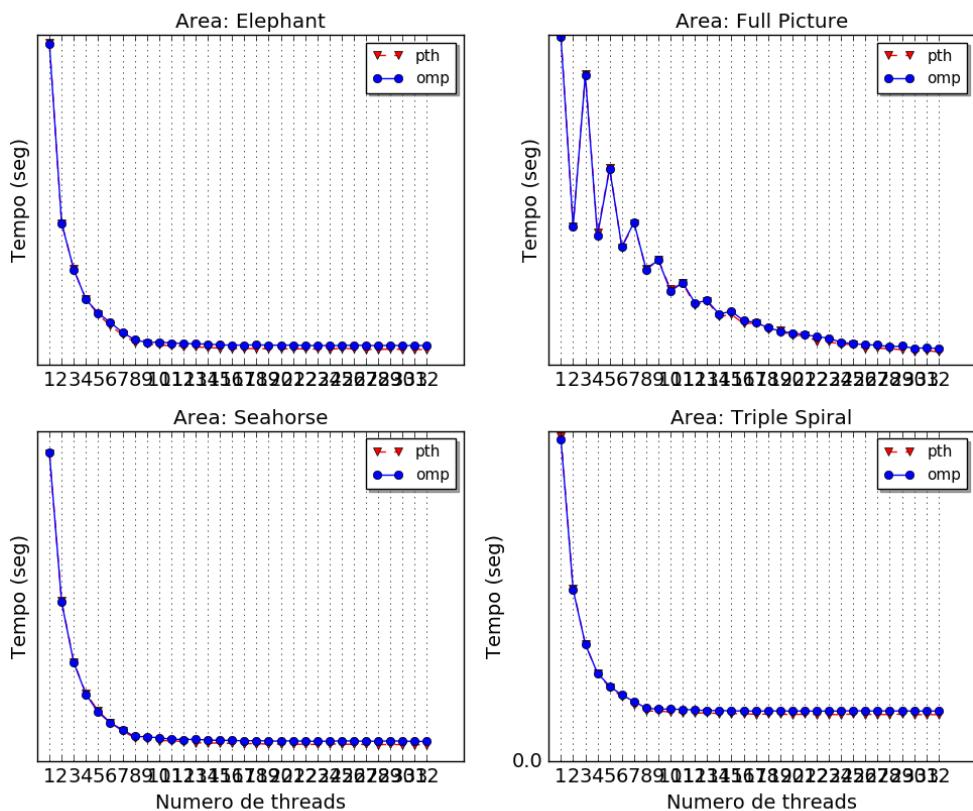


Figura 15: *Tempo de execução dos algoritmos paralelos utilizando Pthreads (pth) e usando OpenMP (omp) variando o número de threads para as quatro áreas do problema para a imagem de 8192 pixels.*

3.2.3. Comparação dos algoritmos

Uma vez que ambos os algoritmos foram apresentados, nesta seção apresentaremos a comparação entre o algoritmo sequencial sem alocação de memória e sem I/O e as duas versões paralelas a fim de mostrar qual o real ganho de se utilizar computação paralela neste problema.

Para a comparação de melhor caso dos algoritmos paralelos, o número de *threads* escolhido foi 32, devido aos melhores tempos de execução em ambas as áreas. Como pode ser visto na figura 16, na área *Elephant Valley* os algoritmos paralelos chegam a ser mais de 6 vezes superiores ao sequencial, e essa vantagem aumenta nas áreas *Seahorse Valley* e *Triple Spiral Valley*, para quase 7 vezes e pouco mais de 7 vezes, respectivamente. No caso da Área *Full Picture*, o ganho em fazer a paralelização chega a ser pouco maior do que 5 vezes em questão de tempo.

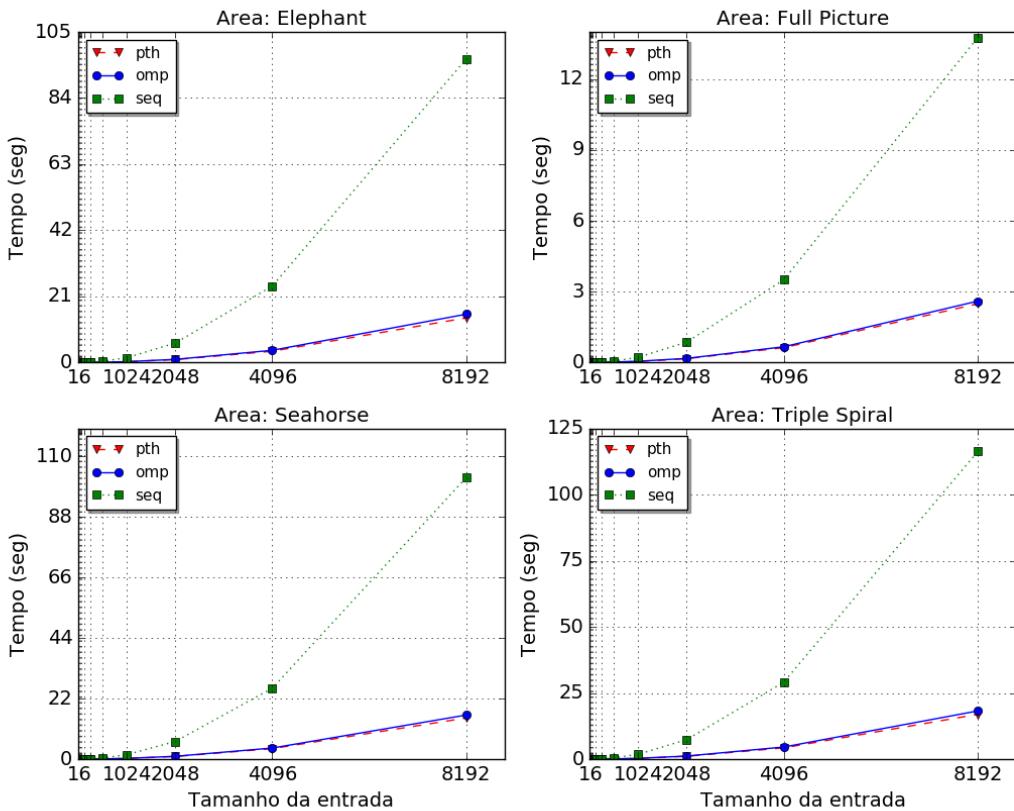


Figura 16: Tempo de execução dos algoritmos sequencial (seq), Pthreads (pth) e OpenMP (omp) para os variados tamanhos do problema e para as quatro áreas.

Para o pior caso, comparamos os algoritmos paralelos executados com apenas uma

thread, a fim de visualizar o *overhead* gasto para a criação da *thread*. A figura 17, utilizando escala logarítmica no eixo y para melhor visualização dos resultados, mostra o tempo de execução dos três algoritmos para uma *thread* (uma vez que o algoritmo sequencial possui apenas uma *thread*, por natureza). Como esperado, as versões paralelas, possuem um pequeno *overhead* para gerenciar essa *thread*, porém, apenas para os 2 ou 3 menores tamanhos do problema. No restante dos tamanhos, o tempo de *overhead* se torna desprezível.

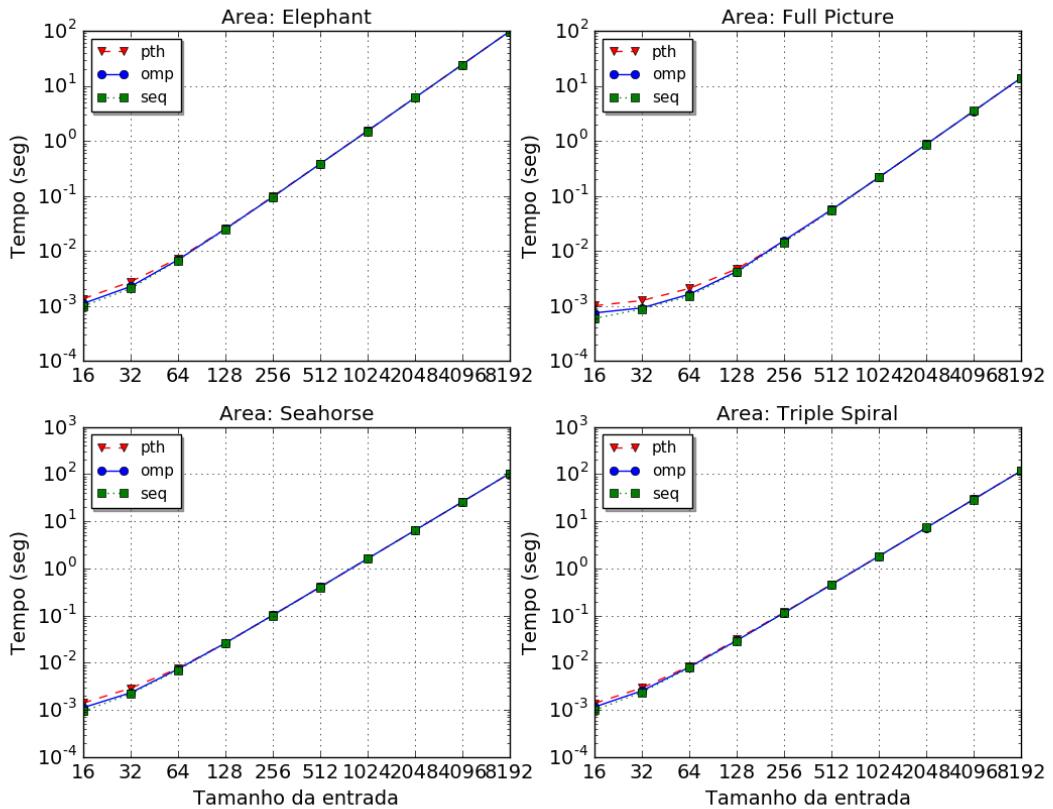


Figura 17: *Tempo de execução dos algoritmos sequencial (seq), Pthreads (pth) e OpenMP (omp) para os variados tamanhos do problema e para as quatro áreas.*

4. Discussão dos Resultados

Dada a natureza de incerteza da paralelização de *threads*, é possível, e bastante provável, que execuções distintas ocorrerão de maneira distintas no processador, no sentido de que a ordem de disponibilização de recursos computacionais para execução da *thread* será totalmente diferente a cada execução. Por esse motivo, um maior número de execuções propicia uma melhor confiabilidade nos resultados obtidos, uma vez que é possível reduzir as chances de execuções estranhas e discrepantes do normal.

O problema do Conjunto de Mandelbrot é um problema facilmente paralelizável, o que significa que não há a necessidade de troca de mensagens por *threads* distintas. Em outras palavras, cada *thread* pode executar o seu trabalho de maneira independente.

No caso dos algoritmos paralelos, a forma como o *array* de pixels é dividido vai dizer como o algoritmo deve ser executado. O algoritmo que utiliza *OpenMP* vai dividir esse *array* automaticamente, sem necessariamente o programador saber como está de fato implementada essa paralelização. Por sua vez, algoritmo utilizando *Pthreads* precisa que essa divisão seja realizada explicitamente pelo programador. No caso do algoritmo sequencial, o *array* inteiro é executado pela mesma *thread*.

O número de pixels por *thread* depende do tamanho da entrada e do número de *threads* utilizadas. Primeiro vamos aos extremos, com situações apresentadas na seção de resultados dos experimentos. Um problema com entrada pequena e muitas *threads* pode se tornar ineficiente, no sentido de gastar recursos para criar e gerenciar as *threads*, enquanto deixar de executar de fato o programa. Por outro lado, problemas com grandes entradas e poucas *threads* talvez não estejam sendo executado nas melhores condições, sendo que no pior caso, quando tiver uma única *thread*, ele é equivalente.

Podemos dizer que, se o número de pixels por *thread* for pequeno, talvez perdemos desempenho por *overhead*. Caso esse número seja grande, um processador pode demorar a fazer todo o trabalho necessário. Então um *trade off* pode ser adequado para encontrar o melhor ponto para otimização de tempo.

Das regiões do Conjunto de Mandelbrot estudadas e experimentadas, podemos dizer que as regiões do tipo *Valley* possuem características similares, uma vez que os mesmos algoritmos se comportaram de maneira similar para as mesmas entradas.

No caso do algoritmo sequencial, onde o tempo de alocação de memória e I/O foram levados em consideração, a área cujo tempo de execução foi mais influenciado foi para

a área *Full Picture*, uma vez que esta convergia mais rápido. Não foram realizados testes em que alocação de memória e I/O eram realizados nos algoritmos paralelos, mas é importante notar, que apesar de eles melhorarem várias vezes (cerca de 5 vezes no caso do Full Picture), em um suposto teste com tais algoritmos, essa características poderiam representar cerca de 50% do tempo de execução.