EP1 - Cálculo do Conjunto de Mandelbrot em Paralelo com Pthreads e OpenMP

Gabriel Baptista `8941300`

Hélio Assakura `8941064`

Maio de 2017

1 Introdução

O conteúdo deste relatório consiste na análise do impacto da programação paralela na construção de regiões do Conjunto de Mandelbrot[1] e na resposta das perguntas feitas no enunciado do EP[2]. Também será analisado o impacto das operações de I/O e alocação de memória. Os resultados serão mostrados em forma de gráfico de barras gerados usando a biblioteca `matplotlib` da linguagem `Python`. Para a paralelização, era obrigatório usar a biblioteca `pthreads.h` e a ferramenta `OpenMP`, e para os cálculos, variar a quantidade de *threads* e o tamanho da entrada como na Tabela 1:

	Pthreads	OpenMP	Sequencial
Regiões	*Triple Spiral*, *Elephant*, *Seahorse* & *Full*		
I/O e Aloc. Mem.	Sem		Com e sem
№ de *Threads*	$2^0 \dots 2^5$		-
Tamanho da Entrada	$2^4 \dots 2^{13}$		
№ de Execuções	10		

Tabela 1: Experimentos

Para cada valor de entrada, foram realizadas 10 medições. Os testes foram realizados na Google Compute Engine (GCE), usando a instância `n1-standard-8` e a imagem da VM disponibilizada pelo monitor (Pedro Bruel). Os resultados

[1] `https://en.wikipedia.org/wiki/Mandelbrot_set` [Acessado em 21/04/2017]

[2] `https://github.com/phrb/MAC5742-0219-EP1/blob/master/doc/enunciado_ep1.pdf` [Acessado em 21/04/2017]

obtidos na VM, o código-fonte para manipular esses resultados, os gráficos obtidos e até mesmo o código-fonte dos programas estão disponíveis em `https://github.com/hassakura/MAC0219` .

2 Resultados

Os tempos obtidos na execução do programa sequencial e nos programas paralelizados (pthreads e OpenMP) variou bastante, e conforme esperado, o tempo ficou menor quanto maior era a quantidade de threads. Quando o tamanho da entrada era pequeno (2^4 por exemplo), os resultados não foram consistentes, pois o tempo necessário para o processamento era muito pequeno (da ordem de 10^{-4}) e qualquer variação causada por elementos externos era muito significativa (figuras 3 e 8). Porém, nas entradas maiores, podemos ver claramente o ganho de rapidez usando programação paralela (figuras 4, 5, 6, 7 para Pthreads e figuras 9, 10, 11, 12 para OpenMP). Também podemos ver que operações de I/O e alocação de memória são bastante custosas (figura 1 e figura 2), diminuindo consideravelmente o tempo obtido ao não construir a imagem em si. O desvio padrão das medições não foi alto (apesar de alguns testes pequenos darem cerca de 90% da média, também devido a operações alheias ao processamento do programa). Grande parte dos testes mostrou um desvio abaixo de 3% da média.

3 Impacto de operações I/O e Aloc. Memória

Para verificar quanto tempo essas operações podem adicionar, comparamos a execução de duas versões do código `seq_mandelbrot.c`: uma que cria a imagem final, realizando todas as alocações e colocando as cores na representação, e outra que apenas calcula as regiões. Como podemos ver nas figuras 1 e 2, houve uma diferença de tempo de até 11 segundos entre as duas execuções (região *Triple Spiral Valley*). Isso se dá pois, para cada um dos n^2 pontos da imagem, escolhemos uma cor e colorimos esse pixel, e depois de colorir a imagem inteira, gravamos no arquivo. Para imagens de tamanho 11500 (como no enunciado), temos um arquivo de saída de 396,8 MB.

4 Pthreads

O código desenvolvido foi uma modificação do código base disponibilizado pelo monitor[3]. As mudanças foram:

- Adaptação do código para a criação de threads usando a biblioteca `pthread.h`;
- Retirada do pedaço de código responsável pela criação e atualização da imagem (I/O e alocação de memória);

[3] `https://github.com/phrb/MAC5742-0219-EP1/blob/master/src/mandelbrot_pth.c` [Acessado em 21/04/2017]

- Divisão das partes do cálculo de acordo com o número de threads a ser considerada;
- Mudança da função `compute_mandelbrot` para calcular apenas a região destinada a thread (agora com nome `compute_mandelbrot_partition`).

5 OpenMP

O código desenvolvido foi uma modificação do código base disponibilizado pelo monitor[4]. As mudanças foram:

- Adaptação do código para o uso da ferramenta OpenMP usando a biblioteca `omp.h`;
- Retirada do pedaço de código responsável pela criação e atualização da imagem (I/O e alocação de memória);
- Mudança da função `compute_mandelbrot` para paralelizar os *for* responsáveis pelo cálculo.

6 Respostas às perguntas

As perguntas a serem respondidas são:

1) Por que você acha que fizemos a recomendação de se realizar mais de uma medição?

2) Por que você acha que existe variabilidade entre execuções do mesmo programa?

3) Como e por que as três versões do programa se comportam com a variação:

(a) Do tamanho da entrada?

(b) Das regiões do Conjunto de Mandelbrot?

(c) Do número de *threads*?

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

5) Qual versão é mais eficiente? Comente um pouco.

[4] `https://github.com/phrb/MAC5742-0219-EP1/blob/master/src/mandelbrot_omp.c` [Acessado em 21/04/2017]

1. Ao realizar os testes, vimos que os resultados podem variar bastante, com diferença entre os tempos de mais de 90%. Portanto, usamos ferramentas estatísticas que garantem que os dados obtidos possam ser confiáveis, verificando a média e desvio padrão do mesmo teste, executado diversas vezes. No caso deste EP, o número de execuções deve ser de pelo menos 10.

2. Muitos fatores podem influenciar no tempo do programa. A principal, na nossa análise, é a do SO, tanto em execuções sequenciais quanto em paralelas. Ele implementa diversos tipos de otimização, e por trás da execução, ainda considera o uso de recursos paralelos para acelerar os cálculos. Outro fator que influencia é a troca de contexto enquanto o processo está rodando. Por isso, há diferença nos resultados do programa sequencial e em execuções com apenas 1 thread. Com mais threads, o escalonamento tem um peso maior na variabilidade de tempos. A ordem de criação das threads e cada trabalho atribuído a elas também pode acarretar em resultados diferentes.

   Execuções sequenciais possuem desvios padrões menores que paralelas[5], devido a menor quantidade de troca de contextos. Enquanto grande parte dos testes de `seq_mandelbrot.c` apresentaram desvios de 0.03% a 0.08% da média, os testes de `pth_mandelbrot.c` mostraram desvios de 0.5% até 3%, em sua maioria e os testes de `opm_mandelbrot.c` mostram desvios de até 5%.

3. (a) Como temos que preencher toda a imagem, iteramos para todo pixel. Com o tamanho menor da entrada, temos menos cálculos, fazendo com que o programa se encerre mais rapidamente. No programa sequencial, temos apenas uma função que preencherá todos os pixeis. Usando *pthreads* e *OpenMP*, dividimos a imagem em regiões de acordo com o número de threads a serem criadas. Cada uma irá processar um pedaço da imagem, para no final termos a imagem completa. Assim, podemos construir as regiões simultaneamente, diminuindo o tempo gasto. A diferença de tempo entre execuções com número de threads diferentes (até atingir o número de cores do processador) é grande. Ao ultrapassar o a quantidade de cores (8 na máquina usada para os testes), os tempos começam a ficar bem semelhantes.

   (b) No programa sequencial, regiões menores tem comportamento semelhante aos programas utilizando *pthreads* e *OpenMP*, agora nas regiões maiores, como por exemplo no caso da imagem completa, utilizando o paralelismo, as versões de *pthreads* e *OpenMP* se comportam melhores, pois conseguem fazer vários cálculos simultaneamente.

   (c) No programa sequencial , o número de *threads* não importa para o cálculo do programa em si, no caso dos programas com *pthreads* e

[5] `https://github.com/hassakura/MAC0219/tree/master/doc/results` [Acessado em 21/04/2017]

OpenMP, observa-se que quanto maior o número de threads, mais rápido é o cálculo do Conjunto de Mandelbrot, por conta do cálculo simultâneo envolvendo as threads. Nos três itens, cabe ressaltar que apesar do comportamento final de *pthreads* e *OpenMP* serem parecidos, paralelizar o problema utilizando *OpenMP* acabou sendo mais fácil por conta da ferramenta acabar "mantendo" a forma sequencial proposta, apenas acrescentando uma linha para paralelizar os *for* do programa.

4. Nas três versões do programa e para todas as regiões do Conjunto de Mandelbrot, as operações de I/O e alocação de memória acrescentam um tempo em relação ao tempo do programa sem as mesmas. Percebe-se que em regiões com zoom, o tempo de execução nos casos com e sem as operações eram de certa forma similares(havia um pequeno acréscimo no programa com as operações), enquanto em regiões com menor zoom, o tempo de execução era mais discrepante, como não caso da imagem completa, onde o tempo de execução para as operações em média levavam 1/3 do tempo de execução total.

5. Dentre as três versões, a versão não-paralelizada é a que apresenta pior desempenho, o que já era esperado. As versões paralelizadas para um número de threads relativamente alto, têm um desempenho muito superior ao sequencial. O comportamento das versões que utilizam *pthreads* e *openMP* são parecidos, como podemos ver nas figuras 3 a 12(tempos de execução para um número fixo de threads semelhantes), entretanto, o código da versão utilizando *openMP* é mais simples, pois se assemelha ao código sequencial.

   Porém, observando essas mesmas figuras, pode-se ver uma ligeira vantagem (quase que imperceptível) pelo uso de *pthreads*, contudo, programar utilizando essa ferramenta não é tão simples como a outra, é necessário preparar todo um ambiente para tal desenvolvimento. Sendo assim, para certas aplicações que não é requerido uma execução ótima, o uso de *openMP* torna-se vantajoso, por essa facilidade para ser escrito um código e pelo desempenho ser tão semelhante ao ótimo.

7 Conclusão

Depois de executarmos os programas para as diversas regiões do Conjunto de Mandelbrot, conseguimos perceber que no geral, independente da ferramenta utilizada para paralelizar um problema, o programa fica mais rápido do que um sequencial. Como proposto no enunciado, também é perceptível quão custoso para um programa é executar operações de I/O e alocação de memória. Dado estes fatos, usando as ferramentas *pthreads* e *OpenMP*, o comportamento do problema em ambos os casos são bem semelhantes, quanto maior o número de threads, menor o tempo de execução de uma mesma instância do problema.

Apesar do comportamento de ambos serem parecidos, a forma como foram implementados difere. Para *OpenMP*, usamos uma schedule dinâmica, que decide internamente quando e o que uma thread irá executar[6]. Para *pthreads*, a alocação foi estática, dividindo as regiões em pedaços de tamanhos fixos. Notamos que não há uma diferença de tempo tão grande entre as implementações, cabendo ao programador decidir se usará *pthreads* ou *OpenMP*.

Podemos concluir que, independente da ferramenta escolhida, a paralelização de programas traz grandes vantagens, resolvendo problemas de forma mais eficiente e utilizando recursos do computador de uma melhor forma.

[6] `https://software.intel.com/en-us/articles/openmp-loop-scheduling` [Acessado em 29/04/2017]

8 Gráficos

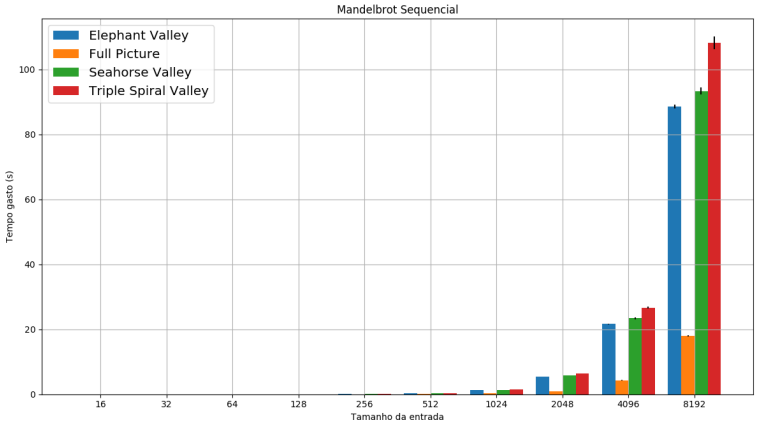

Figura 1: Tempos da execução sequencial com I/O e Aloc. de memória

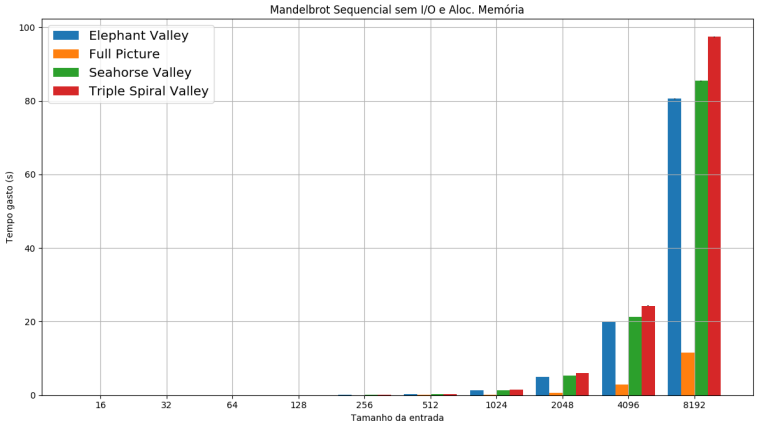

Figura 2: Tempos da execução sequencial sem I/O e Aloc. de memória

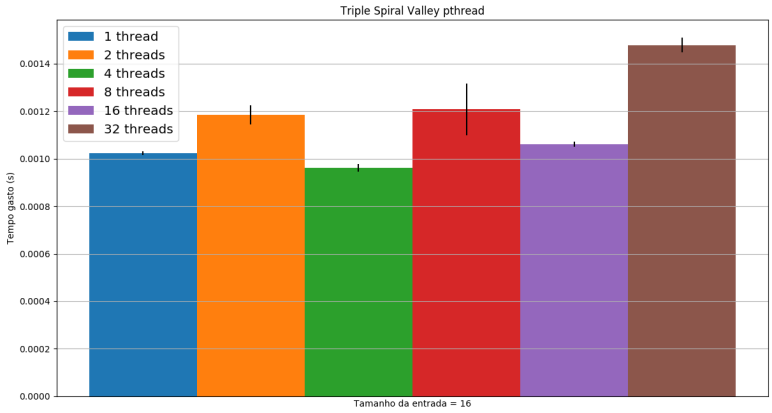

Figura 3: Tempos da região *Triple Spiral Valley* usando pthreads com entrada de tamanho 16

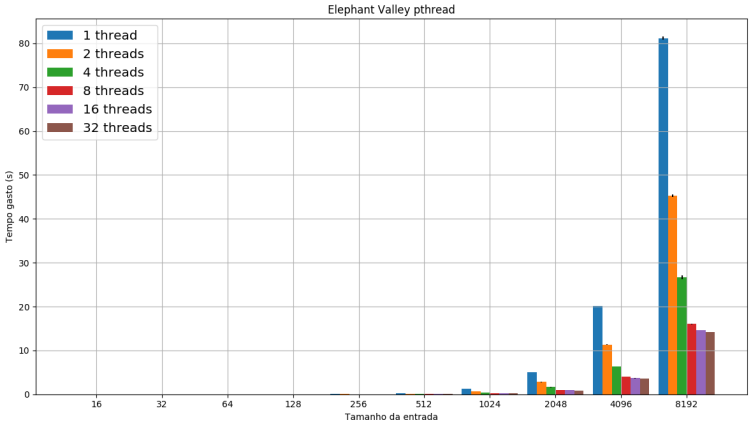

Figura 4: Tempos da região *Elephant Valley* usando pthreads

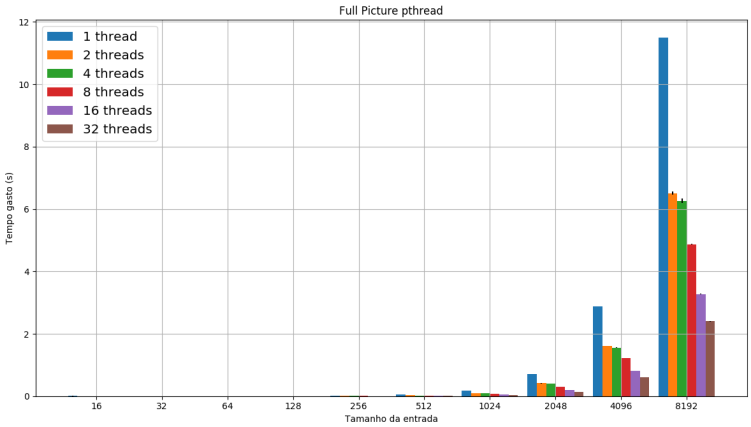

Figura 5: Tempos da imagem inteira usando pthreads

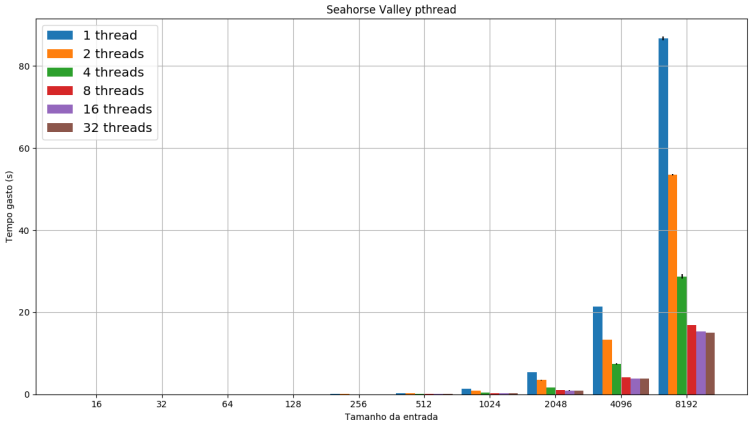

Figura 6: Tempos da região *Seahorse Valley* usando pthreads

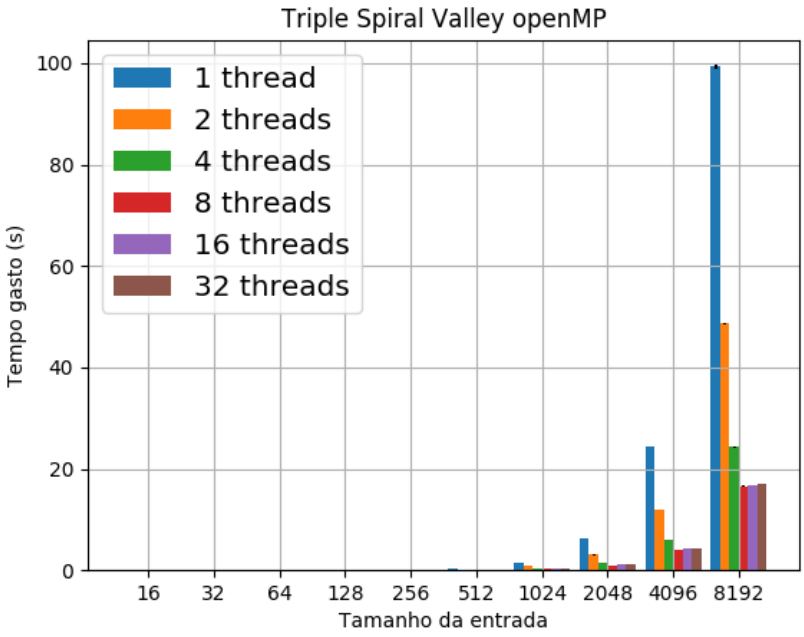

Figura 7: Tempos da região *Triple Spiral Valley* usando pthreads

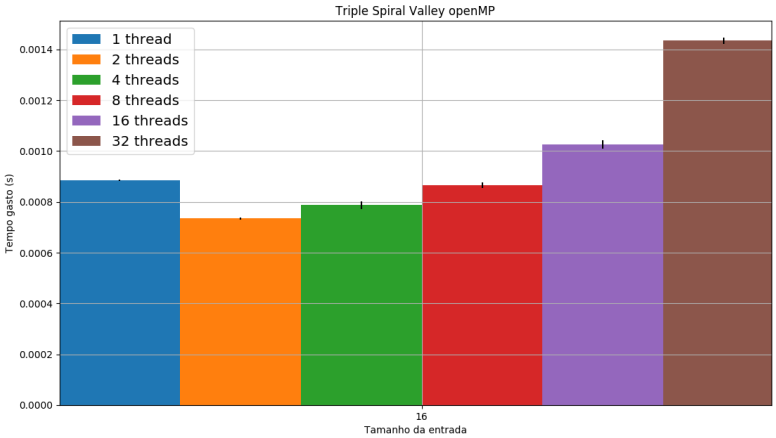

Figura 8: Tempos da região *Triple Spiral Valley* usando openmp com entrada de tamanho 16

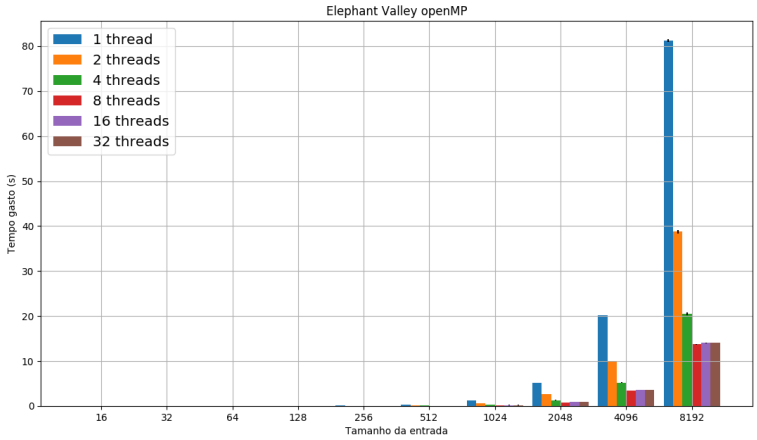

Figura 9: Tempos da região *Elephant Valley* usando openmp

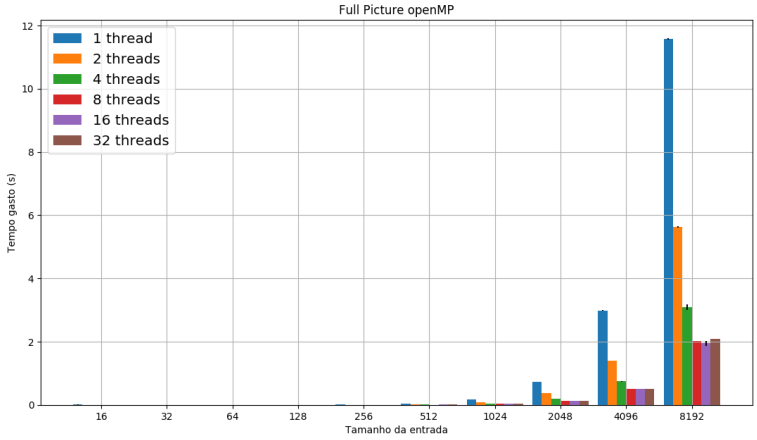

Figura 10: Tempos da imagem inteira usando openmp

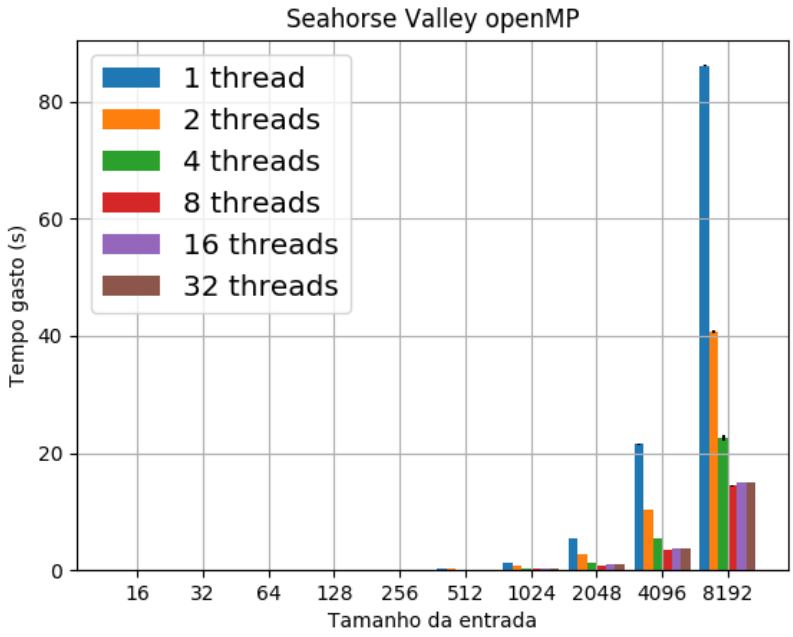

Figura 11: Tempos da região *Seahorse Valley* usando openmp

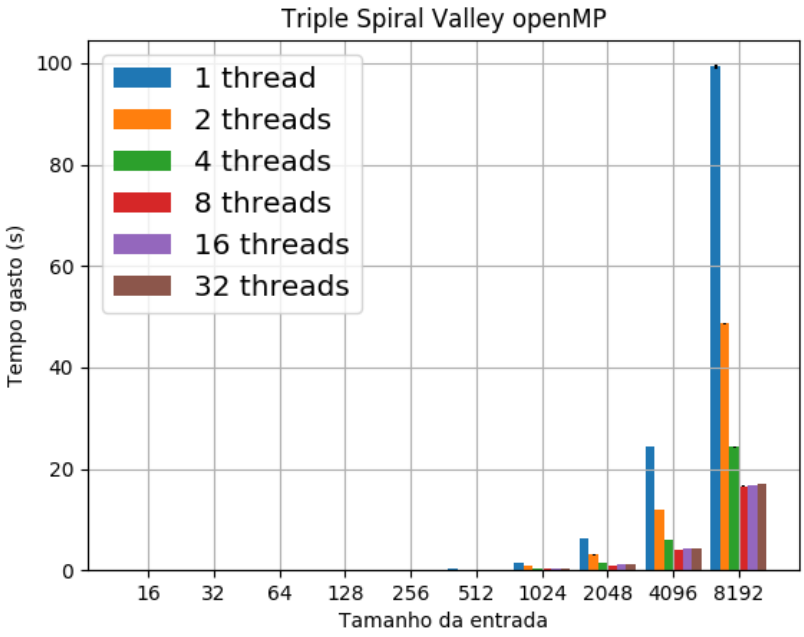

Figura 12: Tempos da região *Triple Spiral Valley* usando openmp