EP3: Cálculo Distribuído do Conjunto de Mandelbrot

Victor Mayrink, André Lopes e Bernardo Amorim

MAC 5742-0219 Introdução à Programação Concorrente, Paralela e Distribuída

1. Introdução

Message Passing Interface (MPI), ou interface de troca de mensagens, é um padrão de comunicação de dados para o desenvolvimento de programas com troca de mensagens.

O MPI tem rotinas definidas e com uma variedade de implementações disponíveis. Além disso, permite colocar a aplicação feita em plataformas diferentes sem precisar alterar o código. Os processos ou *threads*, no padrão MPI, se comunicam através de envio e recebimento de mensagens, sendo que esta comunicação pode ser ponto-a-ponto, por exemplo, e de forma síncrona ou assíncrona. Atualmente há diversas bibliotecas que implementam o MPI, entre elas:

1. *MVAPICH2* é uma implementação de código aberto do MPI. Permite o uso de clusters com GPUs NVIDIA fazendo chamadas via MPI.
2. *IBM™ Spectrum MPI* implementa o padrão MPI e é fornecido pela IBM. Promete uma qualidade superior em comparação aos competidores.
3. *The Open MPI Project*, também conhecida apenas como *OpenMPI*, é a implementação do padrão *MPI-2* e é mantido por um consórcio acadêmico, de pesquisa e parceiros na industria. Permite uso de GPU's.

Neste trabalho, fazemos a implementação do cálculo do conjunto de *Mandelbrot* de forma distribuída no padrão MPI, usando a biblioteca *OpenMPI*. Além disso, foi utilizado o em português, *OpenMP*, a qual permite paralelização do código de forma rápida e simples.

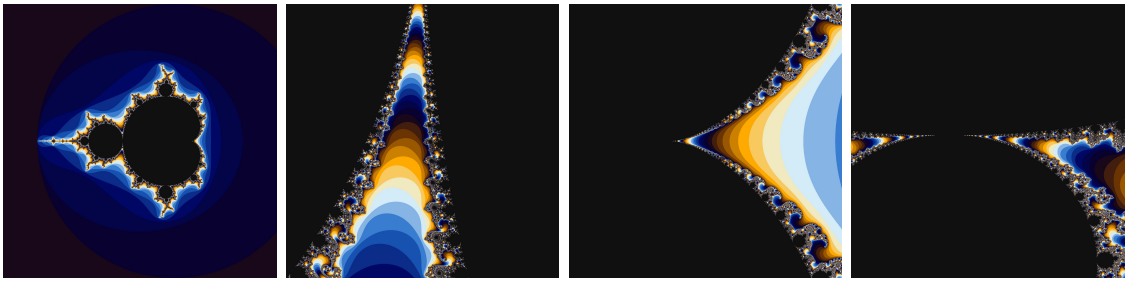
1.1. Mandelbrot

O conjunto de *Mandelbrot* compreende um fractal no plano dos números complexos, definido pela primeira vez em 1905 pelo matemático Francês *Piere Fatou*, que estudava processos recursivos do tipo:

$$z \to z^2 + c \tag{1}$$

Onde z é um ponto qualquer no plano complexo. Assim, é possível aplicar a transformação acima a partir de um ponto inicial z_0 para gerar uma sequência de pontos, denominada *órbita* de z_0.

Mais tarde, o professor *Benoît Mandebrot*, que trabalhava na IBM durante a década de 1960, foi o primeiro a utilizar um computador para gerar imagens de geometria fractal do conjunto que hoje recebe o seu nome. O conjunto de *Mandelbrot* pode ser informalmente definido como o conjunto dos números complexos c para os quais a função $f_c(z) = z^2 + c$ não diverge quando é iterada começando em $z = 0$. Isto é, os pontos para os quais a órbita de $z_0 = 0$, definida pela sequência $f_c(0), f_c(f_c(0)), f_c(f_c(f_c())), \ldots$ é limitada. A figura 1 ilustra algumas regiões do conjunto de *Mandelbrot*.



(a) Full image (b) Seahorse (c) Elephant (d) Triple Spiral

Figura 1: Regiões do conjunto de Mandelbrot

As imagens do conjunto de *Mandelbrot* podem ser geradas através da discretização de uma região do plano complexo em uma imagem com um número finito de *pixels*. Assim, cada *pixel* está associado à um ponto no c no plano complexo, sendo que a cor do pixel é atribuída de acordo com iteração em que transformação expressa na equação 1 convergiu.

Tabela 1: Limites das regiões representadas na figura 1

$c = x + yi$	x_{min}	x_{max}	y_{min}	y_{max}
Full image	-2.500	1.500	-2.000	2.000
Seahorse	-0.800	-0.700	0.050	0.150
Elephant	0.175	0.375	-0.100	0.100
Triple Spiral	-0.188	-0.012	0.554	0.754

2. Objetivos

Os objetivos deste trabalho são:

1. Analisar as implementações do código de *Mandelbrot* e propor uma estratégia de paralelização para ser executada, utilizando o padrão MPI;
2. Combinar as bibliotecas OpenMPI e OpenMP para explorar ao máximo os recursos computacionais disponíveis, através de uma estratégia que seja distribuída e paralela.
3. Comparar o tempo de execução dos algoritmos sequenciais com a estratégia de paralelização proposta;
4. Discutir os resultados obtidos e os casos em que foi possível obter os melhores ganhos de desempenho.

3. Estratégias de Paralelização

A geração das imagens com os fractais de *Mandelbrot* é uma tarefa que possui um grande potencial paralelização, uma vez que a determinação da cor de cada pixel da imagem depende apenas do valor correspondente às suas coordenadas no plano complexo. Assim, não é necessário respeitar qualquer ordem de precedência,

pois a cor de um pixel não depende da cor dos pixels vizinhos nem do resultado de nenhuma outra operação.

Diante desta flexibilidade, é possível elaborar diversas estratégias para paralelizar a geração das imagens. No entanto, a elaboração de uma estratégia de paralelização é muito importante, pois poderá impactar de forma bastante expressiva no ganho de desempenho que pode ser obtido com a exploração de recursos de computação paralela.

3.1. Implementação apenas com OpenMPI

Já discutimos algumas estratégias de paralelização no relatório do EP1. Como vimos, o ideal é evitar pré-definir a carga de processamento para cada uma das *threads*; pois o tempo que cada *thread* leva para executar sua tarefa pode variar, e portanto, as *threads* que finalizam suas tarefas primeiro, acabam tendo que esperar as demais. Ao invés disso, uma ideia mais inteligente consiste em criar uma pilha de subtarefas comum entre todas as *threads*, e deixar que elas trabalhem em conjunto para processar toda a pilha de subtarefas.

Esta estratégia funciona particularmente bem em processadores multi-core, no qual existe uma memória compartilhada entre todas as *threads*, onde é possível construir uma pilha de subtarefas acessível por todas as *threads*. No entanto, o padrão MPI foi desenvolvido para arquiteturas distribuídas, e, portanto, não pressupõe a existência de uma memória compartilhada entre os processos. Ao invés disso, os processos comunicam-se através de mensagens que seguem o padrão MPI.

Apesar de ser possível, implementar uma pilha de subtarefas acessível por todos os processos utilizando uma comunicação por mensagem não seria uma tarefa trivial. Além disso, em circunstâncias controladas e, considerando que os experimentos realizados (ver seção 4) foram executados em um grupo de instâncias onde todas as máquinas possuem as mesmas especificações, é razoável distribuir as carga de processamento igualmente entre todos os processos.

Para distribuir a carga entre os processos de forma equilibrada, optamos por atribuir as linhas da imagem à cada processo, de forma intercalada (conforme discutimos no relatório do EP1). Assim cada processo ficou encarregado de processar as linhas cujo resto da divisão inteira pelo total de processos existentes fosse igual ao próprio ID do processo, isto é:

$$\underset{k=0..K}{processo\ k-1} \rightarrow \underset{i_y=1..N-1}{\text{linhas}} \Big|\ i_y \mod K = k \tag{2}$$

Onde K é o número total de processos e N é o número de linhas da imagem.

Outro critério importante que deve ser definido envolve a estratégia de comunicação entre os processos, isto é quando os processos devem trocar mensagens entre si. No caso do problema de geração das imagens com os fractais de Mandelbrot,as mensagens podem ser enviadas, basicamente, em três pontos:

1. **Após processar cada pixel:** cada processo envia uma mensagem imediatamente após determinar a iteração de convergência de cada pixel. Nesse caso, as mensagens têm um tamanho bem curto, cada uma tem apenas três inteiros: o índice da linha, o índice da coluna e número da iteração em que o ponto correspondente ao pixel convergiu. Em contrapartida, o número total de mensagens trocadas entre os processos é muito alto.

2. **Após processar uma linha inteira:** nesse caso, cada processo possui um vetor de inteiros com comprimento igual ao número de *pixels* de uma linha da imagem. Esse vetor funciona como um *buffer* local; assim, ao invés de enviar uma mensagem após processar cada pixel, os dados são armazenados no *buffer* e a mensagem só é enviada quando todos os *pixels* de uma linha forem processados. Com isso, é possível diminuir substancialmente o número de troca de mensagens entre os processos, cada mensagem leva o índice da linha e um vetor de inteiros com o número da iteração correspondente aos *pixels* da linha.

3. **Após o processo concluir todo o processamento:** neste caso, o tamanho do *buffer* deve ser suficiente para armazenar os dados de todos os *pixels* atribuídos à um determinado processo. Apesar de consumir mais memória (devido ao tamanho do *buffer*), com essa estratégia, cada processo precisa enviar apenas uma mensagem para o *master*, ao final de todo o processamento.

Nesse trabalho, procuramos reduzir o volume total de mensagens trocadas entre os processos e, portanto, optamos pela estratégia 3. Além disso, na estratégia 1, por exemplo, são necessários enviar 3 inteiros em cada mensagem, sendo que dois servem apenas para referenciar a posição do pixel; na estratégia 3, por outro lado, não é necessário passar índices da imagem dentro da mensagem, uma vez que esses índices podem ser inferidos a partir das posições do vetor *buffer*. Em uma situação prática, contudo, é preciso levar em consideração não somente o volume total de mensagens trocadas, como também o consumo de memória e CPU.

3.2. Implementação com OpenMPI + OpenMP

A estratégia de paralelização com a biblioteca OpenMP foi exatamente a mesma descrita no relatório do EP1. Isto é, criamos uma pilha compartilhada com o número das linhas que foram atribuídas à um determinado processo. As *threads* trabalham em conjunto para processar todas as linhas atribuídas ao processo, sem que seja necessário definir de antemão quais linhas serão atribuídas a cada *thread*. Para mais detalhes, consulte [MMB17].

4. Descrição dos experimentos

A geração das imagens com os fractais de Mandelbrot foi implementada utilizando três algoritmos diferentes: *i*) método sequencial, *ii*) método distribuído com OpenMPI, e *iii*) método distribuído e paralelo (OpenMPI + OpenMP).

Os experimentos propostos consistem em gerar as imagens das quatro regiões do conjunto de Mandelbrot representadas na figura 1. O tamanho da imagem foi fixado em 8192×8192 *pixels*.

Ao todo, foram realizados 4 experimentos distintos, variando o número de instâncias e número de cores por instância. O número total de cores, no entanto, permaneceu constante. Cada experimento foi executado 10 vezes, com o objetivo de levar em avaliar o tempo médio de processamento e o desvio padrão. Todos os experimentos foram realizados na plataforma *Google Compute Engine* [GCE17]. A tabela 2 resume os experimentos realizados.

Note que optamos por utilizar a instância *n1-standard-8* para fazer os testes com o algoritmo sequencial. Apesar de o algoritmo sequencial não explorar todos

Tabela 2: Descrição dos experimentos

	Instância	№ de inst.	Cores/inst.	Rep.	Algoritmos
Experimento 1	n1-standard-1	8	1	10	MPI
Experimento 2	n1-standard-2	4	2	10	MPI, MPI+OMP
Experimento 3	n1-standard-4	2	4	10	MPI, MPI+OMP
Experimento 4	n1-standard-8	1	8	10	SEQ, MPI+OMP

os cores disponíveis na máquina, esta é a instância que possui a maior memória (30GB). Além disso, não utilizamos o algoritmo MPI+OMP no experimento com 8 instâncias de 1 CPU, uma vez que não faz sentido usar OpenMP em máquinas com apenas uma CPU. De forma análoga, não utilizamos a implementação puramente em MPI para o experimento 4, que possui apenas uma instância; apesar de ser possível emular vários processos e executá-los de forma paralela em uma máquina multi-core.

5. Resultados

A tabela 3 apresenta os resultados obtidos pelo algoritmo sequencial quando executados na máquina n1-standard-8. Já a tabela 4 apresenta os resultados dos experimentos paralelos. Os resultados estão também expostos nos gráficos da figura 2.

A presença de diversas dimensões de *overhead* nos experimentos faz com que os dados tornem-se razoavelmente caóticos. Nota-se que o experimento sequencial teve tempo de execução significativamente maior do que suas contrapartes paralelas, o que poderia sugerir que a adição de recursos computacionais seria sempre benéfica à geração destes conjuntos de mandelbrot. Esta hipótese, porém, é refutada pelo contraexemplo do experimento 3, no qual a implementação em MPI + OpenMP, que utiliza 8 cores, mostra-se menos eficiente do que a em MPI, com apenas 2 cores.

Tabela 3: Tempo de execução sequencial das imagens
na instância n1-standard-8 (média e desvio padrão em segundos)

Região da imagem	Tempo de execução sequencial
Full Image	$18,49 \pm 0,48\%$
Sea Horse	$99,37 \pm 0,12\%$
Elephant	$94,04 \pm 0,13\%$
Triple Spiral	$112,36 \pm 0,09\%$

Podemos detectar outras tendências, sobre as quais devem ser evitadas generalizações, haja visto o exemplo do último parágrafo. São elas:

- Para todos os fractais gerados, a implementação em MPI + OpenMP que utilizou 8 CPUs (1 computador com 8 cores cada, utilizando efetivamente apenas o OpenMP) teve pior desempenho do que aquela em MPI com 8 CPUs (8 computadores com 1 core)

- As implementações em MPI tiveram seu tempo de execução quase monotonamente decrescente com o aumento de instâncias utilizadas.

- As implementações em MPI + OpenMP tiveram seu melhor tempo de execução quando utilizaram apenas uma instância.
- A variância na execução dos códigos paralelos foi maior do que aquela verificada no sequencial, sendo ela própria notavelmente baixa, graças à exclusividade tinha sobre os recursos computacionais do ambiente em que foi executado.

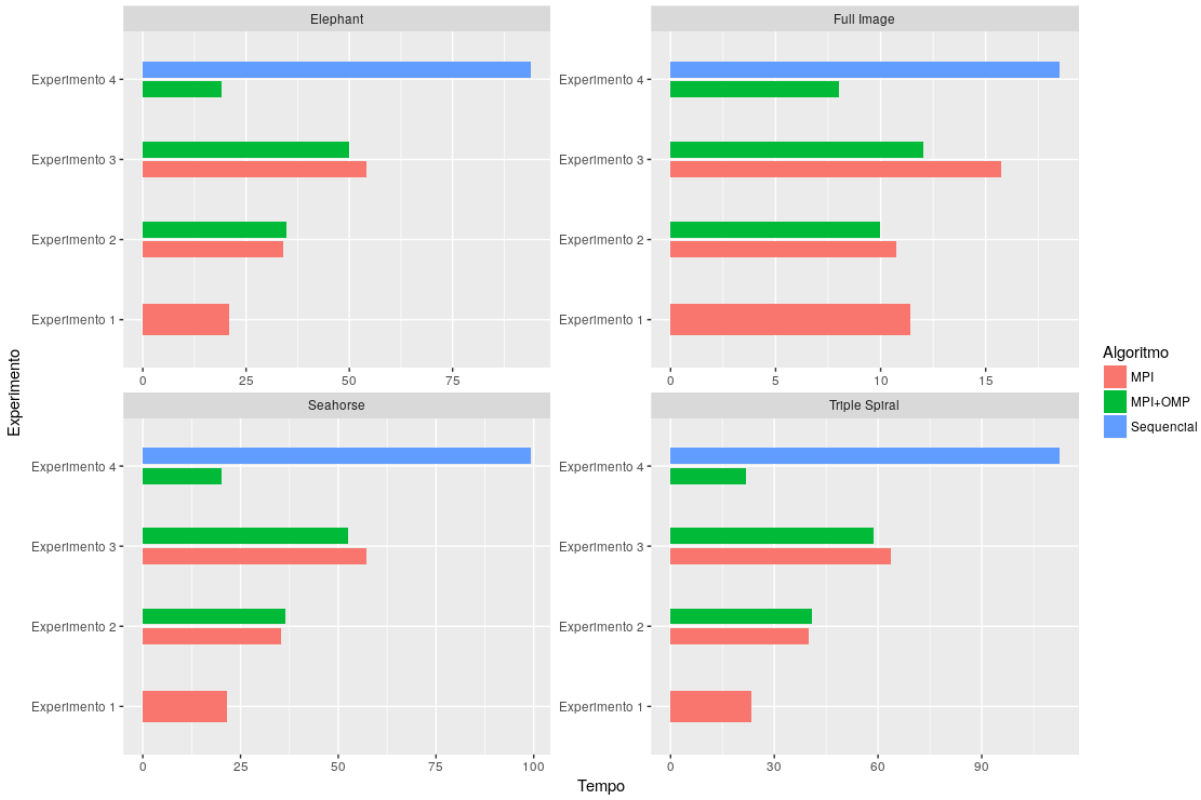


Figura 2: Regiões do conjunto de Mandelbrot

Tabela 4: Tempo de execução paralela das imagens (média e desvio padrão em segundos)

a		Experimento 1	Experimento 2	Experimento 3	Experimento 4
Full Image	MPI	$11,43 \pm 1,05\%$	$10,75 \pm 1,53\%$	$15,74 \pm 1,35\%$	–
	MPI + OMP	–	$9,94 \pm 1,40\%$	$12,03 \pm 1,24\%$	$8,01 \pm 0,78\%$
Sea Horse	MPI	$21,60 \pm 1,41\%$	$35,26 \pm 0,82\%$	$57,14 \pm 0,25\%$	–
	MPI + OMP	–	$36,42 \pm 0,56\%$	$52,50 \pm 0,21\%$	$20,02 \pm 0,78\%$
Elephant	MPI	$20,99 \pm 0,72\%$	$34,03 \pm 0,60\%$	$54,21 \pm 0,51\%$	–
	MPI + OMP	–	$34,67 \pm 0,56\%$	$49,83 \pm 0,26\%$	$19,11 \pm 0,87\%$
Triple Spiral	MPI	$23,42 \pm 1,01\%$	$39,81 \pm 0,52\%$	$63,64 \pm 0,37\%$	–
	MPI + OMP	–	$40,81 \pm 0,45\%$	$58,66 \pm 0,26\%$	$21,93 \pm 0,80\%$

6. Conclusão

Neste trabalho abordamos como executar diferentes processos utilizando o padrão de comunicação MPI. O MPI não pressupõe nenhum tipo de memória compartilhada entre os processos, ao invés disso, eles se comunicam por meio de mensagens,

que em geral são transmitidas por uma rede de dados. Assim, é possível que esses processos sejam executados em várias máquinas, com características distintas e em lugares diferentes.

Além disso, se as máquinas possuem um processador multi-core, é possível combinar o MPI com a bibliotecas de computação paralela, como OpenMP, Pthreads, ou até mesmo CUDA.

Neste trabalho descrevemos algumas estratégias para gerar imagens do conjunto de Mandelbrot em cluster, utilizando o MPI. Utilizamos também a biblioteca OpenMP para paralelizar a execução das subtarefas dentro de cada processo.

Os resultados obtidos demonstraram que é possível obter um ganho de desempenho bastante significativo e bastante escalável, uma vez que é possível máquinas com características distintas para trabalhar em conjunto em um mesmo grupo de instâncias. Os resultados obtidos utilizando 8 máquinas de 1 CPU com o padrão MPI, foram bem semelhantes aos resultados obtidos com uma uma máquina de 8 CPUs utilizando a biblioteca MPI, o que demonstra um compromisso muito interessante em desempenho e escalabilidade.

Os melhores resultados foram obtidos com a combinação das bibliotecas OpenMPI e OpenMP. Tal fato, ressalta que as duas bibliotecas podem ser usadas de forma complementar, com o objetivo de explorar ao máximo toda a capacidade computacional disponível.

Optamos por utilizar um *buffer* local em cada processo e enviar uma única mensagem para o processo *master* após concluir todo o processamento. No entanto, como perspectiva de trabalhos futuros, poderíamos avaliar o desempenho das diferentes estratégias de envio de mensagens via MPI, conforme discutimos na seção 3. Assim, seria possível comparar as soluções propostas em relação ao tempo de processamento, uso de CPU e consumo de memória.

Outra otimização que ainda pode ser implementada é a divisão das subtarefas entre os processos de forma análoga à que foi utilizada para as *threads* de cada processo (conforme discutido no relatório do EP1 [MMB17]); isto é, criando um conjunto de subtarefas compartilhado entre todos processos, ao invés atribuir previamente a quantidade de subtarefas de cada processo. Por exemplo, se tivermos um *cluster* com máquinas de características distintas, dividir as tarefas igualmente entre as máquinas pode levar a um desempenho sub-ótimo, pois as máquinas de melhor desempenho teriam que esperar as máquinas mais lentas. Da mesma forma, se uma máquina se sobrecarregar ou ficar *offline* por algum motivo, todas as demais ficam aguardando sua resposta para prosseguir. Portanto, para que a paralelização seja robusta, o ideal é que as tarefas sejam distribuídas entre os processos durante a execução do programa.

7. Códigos

Os códigos utilizados neste trabalho encontram-se disponíveis em:
https://github.com/VictorMayrink/MAC5742-EP3

8. Referências

[GCE17] Google compute engine documentation, 2017.

[MMB17] Victor Mayrink, Italo Moraes e Fernando Buzato. Ep1: Cálculo do conjunto de mandelbrot em paralelo com pthreads e openmp. *MAC5742*, 2017.