UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

Marcelo Rabello Rossi

EXPERIMENTOS COM LINGUAGENS, PARALELISMO E ALTO CONSUMO DE CPU

SÃO PAULO
2017

MARCELO RABELLO ROSSI

EXPERIMENTOS COM LINGUAGENS, PARALELISMO E ALTO CONSUMO DE CPU

Trabalho de Conclusão de Curso submetido à disciplina "MAC0499 - Trabalho de Formatura Supervisionado", sob a orientação do Prof. Alfredo Goldman

São Paulo, dezembro de 2017

Agradecimentos

À minha namorada, Andréia, pela paciência ao longo das diversas horas em que estive distante, fisicamente ou mentalmente, enquanto escrevia este texto. Ao Prof. Alfredo Goldman, pelo direcionamento que me deu ao longo do desenvolvimento deste trabalho. Ao Pedro Bruel, pelas dicas sobre paralelismo e características das linguagens.

“Take some more tea," the March Hare said to Alice, very earnestly.
"I’ve had nothing yet," Alice replied in an offended tone, "so I can’t take more."
"You mean you can’t take less," said the Hatter: "it’s very easy to take more than nothing."
(Alice’s Adventures in Wonderland, Lewis Carroll)

Resumo

Para se obter os benefícios das mais atuais evoluções em *hardware* passa a ser necessário escrever softwares concorrentes e paralelos. Diversas linguagens, clássicas e atuais, implementam um ou mais tipos de modelos de paralelismo. O presente trabalho estudou o comportamento de quatro dessas linguagens – Go, Julia, Python e C/OpenMP – em um contexto perfeitamente paralelo de uso intenso de CPU, utilizando-se como ferramenta de comparação o algoritmo de obtenção do conjunto de Mandelbrot. Foi possível mostrar que, para as linguagens estudadas, pequenas adições ao código podem trazer um grande ganho em desempenho. Para a situação específica envolvendo C/OpenMP, executando paralelamente em 8 núcleos com escalonamento dinâmico de tarefas e com matriz de entrada de tamanho 14000×10000, verificou-se um desempenho quase 7x maior para uma adição de apenas 2 linhas de código em relação à execução em modo sequencial. Os programas implementados em Julia, utilizando paradigma funcional, mostraram alto desempenho mesmo quando executados em modo sequencial, e tiveram um *speedup* em latência considerável com a adição de 2 linhas de código. O desempenho comparável a C, a simplicidade do código e a existência de diversas bibliotecas otimizadas fazem de Julia a melhor escolha geral dentre as linguagens estudadas. No caso específico em que há necessidade de se iniciar muitos processos paralelos, Go passa a ser uma boa opção devido à leveza de suas rotinas. As linguagens mais atuais e as extensões mais recentes para linguagens clássicas estão, a cada dia, tornando o paralelismo mais acessível mesmo aos programadores mais leigos. Espera-se mostrar, com este trabalho, que todos os programadores estão convidados a experimentar o paralelismo em sua linguagem de preferência e, possivelmente, obter proveito dele em sua rotina diária.

Palavras-chave: paralelismo; desempenho; tamanho do código; uso intenso de CPU.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Por muito tempo a estratégia para se obter processadores com melhor desempenho se baseou no aumento do número de transístores que esses processadores abrigavam, visando o aumento de sua frequência (*clock rate*). Verificou-se, ao longo das décadas, que o número de transístores (componentes responsáveis pelo controle do fluxo de eletricidade e, portanto, do fluxo de dados dentro do processador) em um circuito integrado dobra a cada dois anos, fato esse conhecido como *Lei de Moore*, devido ao artigo publicado por Gordon E. Moore em 1965 [1]. Apesar disso, a partir de 2004 pode-se verificar uma mudança nesta tendência, e os processadores atingem um patamar em termos de frequência e consumo de energia [2]. Os primeiros processadores consumiam menos de 1 W e os primeiros microprocessadores de 32 bits consumiam algo próximo de 2 W, enquanto que um processador 3.3 GHz Intel Core i7 consome aproximadamente 130 W. Dado que o calor dissipado por esta energia está contido em um espaço muito pequeno, atingiu-se o limite de calor capaz de ser dissipado por resfriamento a ar [3, p. 24].

A frequência parou, então, de crescer e os processadores passaram a evoluir de outras formas, sendo uma delas o aumento do número de núcleos de processamento. Isso, de certa forma, finalizou uma era em que os softwares se beneficiavam de forma gratuita do aumento da capacidade dos processadores [4]. Para se obter os benefícios dessa evolução, passa a ser necessário escrever softwares concorrentes e paralelos.

Apesar de estas duas palavras, *concorrência* e *paralelismo*, serem utilizadas indistintamente em diversos contextos, tratam-se de conceitos diferentes: concorrência é uma característica estrutural do programa e está relacionada a como as tarefas são divididas. Aplicações concorrentes *lidam* com diversas tarefas ao mesmo tempo. Já o paralelismo é uma característica de execução do programa e está relacionado ao número de tarefas ou trechos de tarefas que podem ser *executados* ao mesmo tempo [5]. Um programa pode ser concorrente, mas não paralelo, e pode ser paralelo sem ser concorrente. Como estes dois conceitos aparecem em conjunto em muitas situações, acabam por serem confundidos.

Em termos de software, o paralelismo pode existir em diversas formas: paralelismo em nível de instrução (ILP), em nível de bit (BLP), em nível de tarefa ou *thread* (TLP), em nível de dados (DLP), entre outros [6]. No que diz respeito ao hardware e, mais especificamente, à memória do computador, existem sistemas paralelos de memória compartilhada, em que os diversos processos acessam o mesmo conjunto de endereços, e de memória distribuída, em que cada processo é responsável pelo seu próprio conjunto de endereços de memória [3, pp. 351-391].

Por fim, existem os *paradigmas* (ou modelos) de implementação do paralelismo.

Esses paradigmas estão relacionados com a forma que cada linguagem usa os recursos descritos anteriormente, e envolve o controle, a comunicação entre os processos e o acesso aos dados [7].

Atualmente existem diversas linguagens de programação que implementam nativamente um ou mais modelos de paralelismo. Linguagens mais tradicionais, como C/C++, possuem algumas formas bastante difundidas de lidar com o processamento em paralelo, embora pouco práticas do ponto de vista de desenvolvimento de código (ex. *pthreads*). No entanto, algumas delas também dispõem de APIs para facilitar o desenvolvimento de softwares paralelos. *OpenMP* é uma API multiplataforma baseada no modelo de programação paralela de memória compartilhada para arquiteturas de múltiplos processadores, cuja especificação para C/C++ surgiu em 1998. Consiste em um conjunto de diretivas para o compilador, bibliotecas e variáveis de ambiente que influenciam o comportamento em tempo de execução [8]. A implementação do paralelismo utilizando C/OpenMP permite, inclusive, realizar certos ajustes finos, tais como o tipo de escalonamento de processos que será utilizado durante a execução. O escalonamento estático subdivide, *a priori*, o número de atividades atribuídas a cada um dos processos, enquanto que o escalonamento dinâmico atribui as atividades aos processos em tempo de execução, entregando novos pacotes de atividades à medida que os processos terminam de executar as anteriores. É possível até definir o tamanho desses pacotes via diretivas do OpenMP. *C* é a segunda linguagem mais popular, de acordo com o índice TIOBE [9] – que avalia as linguagens de programação de acordo com sua popularidade nos mais diversos buscadores –, perdendo apenas para Java e, mesmo competindo com uma infinidade de novas linguagens de programação, continua sendo uma referência no que diz respeito a desempenho [10]. Muitos dos sistemas operacionais que utilizamos, sejam *desktop* ou *mobile*, dos sistemas de bancos de dados que conhecemos e dos sistemas embarcados em utensílios domésticos (IoT) são baseados em código *C* [11].

A linguagem *Go*, criada em 2007 pela Google, teve o maior ganho de popularidade entre as linguagens, em 2016, de acordo com o índice TIOBE. Uma de suas propostas, de acordo com a sua documentação, é a de prover mecanismos de concorrência que facilitam a criação de programas capazes de extrair o máximo de sistemas multiprocessados. Esses mecanismos, nomeados *go routines*, são extremamente leves ($\approx$ 2 KB) quando comparados a *threads* de outras linguagens de programação. *Go* é conhecida por ter uma concorrência eficiente, como Java e C/C++, ao mesmo tempo em que mantém o código simples e fácil de escrever [12]. A linguagem se mantém, ainda hoje, entre as 20 mais populares, de acordo com o índice TIOBE.

A linguagem *Julia* (43ª posição no índice TIOBE), cujo desenvolvimento foi iniciado em 2009 e sua primeira aparição se deu em 2012, surgiu com o objetivo de ser uma linguagem de alto nível de abstração e, ao mesmo tempo, de alto desempenho [13]. A sua otimização

e o fato de ser uma linguagem compilada faz com que seu desempenho se equipare, em muitas situações, ao de linguagens como C e Fortran, que são referências em desempenho na área científica [14]. De acordo com sua documentação [15], que mostra o resultado de diversos testes de desempenho comparados a outras linguagens, *Julia* foi desenvolvida com o paralelismo e a computação na nuvem (distribuída) em mente. Ainda em fase *beta*, a linguagem se mostra como a grande promessa da computação científica e da ciência de dados, sendo adotada por grandes empresas ao redor do mundo – tais como Amazon, Apple, Disney, Facebook, Ford, Google, Grindr, IBM, Microsoft, NASA, Oracle e Uber – devido à simplicidade do seu código, ao seu desempenho e a sua aplicação em situações que envolvem paralelismo [16]. É também tópico de diversos cursos e livros destinados a essas áreas [17–21]. O MIT possui, atualmente, um pequeno grupo de desenvolvedores focado nos aspectos numéricos e teóricos do núcleo da linguagem [22].

Python é uma linguagem extensamente utilizada para múltiplos propósitos. Sua popularidade (5ª posição no índice TIOBE) é inquestionável e, em 2014, atingiu o status de linguagem mais popular nos cursos introdutórios da área de computação, ranking que levou em conta diversas das mais renovadas universidades do mundo em Ciência da Computação [23]. A área científica, antes dominada por certas linguagens técnicas como Matlab e R, tem voltado a sua atenção para Python [24], ao mesmo tempo em que poderosas bibliotecas como *NumPy* e *SciPy* são desenvolvidas para servir essa comunidade. Apesar da dificuldade de se trabalhar com múltiplas *threads* em Python, devido ao GIL (*Global Interpreter Lock*) [25], mecanismo *thread-safe* imposto pelo interpretador Python por garantia de simplicidade, diversas estratégias foram criadas para permitir que os programas desenvolvidos em Python pudessem acessar as vantagens trazidas pelo paralelismo. Uma delas, implementada pela biblioteca *multiprocessing*, envolve a criação de diversos subprocessos, em uma situação de memória logicamente distribuída, em que cada um é responsável por executar um conjunto de tarefas do programa.

Se um dos caminhos para o ganho de desempenho reside, atualmente, na utilização de sistemas paralelos, e se as opções existentes são diversas, passa a ser interessante estudar a usabilidade e o desempenho de cada um para um determinado propósito de interesse. Wilson e colaboradores descrevem um conjunto de *toy problems*, nomeados *Problemas de Cowichan*, com a finalidade de testar a usabilidade dos sistemas paralelos em diversas situações comuns em paralelização [26]. Em sua proposta, além do *speedup* ganho ao se paralelizar o código, cujo comportamento pode ser descrito pela Lei de Amdahl [27], deve-se também levar em consideração o tempo de programação necessário para se paralelizar o código [28]. É possível encontrar na literatura diversos trabalhos que utilizam um subconjunto desses problemas para avaliar sistemas paralelos [29, 30].

Dentre os problemas propostos nesse conjunto encontra-se um algoritmo para a obtenção do *conjunto de Mandelbrot*. O conjunto de Mandelbrot é o conjunto de pontos c

do plano complexo em que a recorrência $z_n = z_{n-1}^2 + c$ *não diverge*, com z iniciando em 0 [31]. É possível provar que, uma vez que a recorrência ultrapassa o valor 2, ela certamente irá divergir [32]. Esta informação é importante do ponto de vista de implementação, pois fornece um momento claro no qual as iterações devem ser interrompidas.

Da maneira proposta nos Problemas de Cowichan, dados:

1. o tamanho da matriz de saída;
2. as coordenadas do ponto de início no plano complexo (ponto do vértice inferior esquerdo da região de interesse);
3. as distâncias em x e y a serem consideradas a partir desse ponto inicial

informações essas que delimitam uma região retangular no plano complexo, deve-se gerar uma matriz de saída de mesmo tamanho da matriz de entrada, em que cada elemento da matriz representa o número de iterações até que haja a divergência da recorrência $z_n = z_{n-1}^2 + c$, com c sendo o ponto de mesmo índice na matriz de entrada. O número máximo de iterações, neste caso, foi definido em 150.

O algoritmo para obtenção do conjunto de Mandelbrot representa uma situação usual de uso intenso da CPU, tal como buscas, ordenações, renderizações, entre outras. Além disso, trata-se de uma situação *perfeitamente paralela* (do inglês *embarrassingly parallel*) [33, p. 14]: como cada elemento da matriz de saída pode ser calculado independentemente dos outros, em qualquer ordem, é necessário pouco esforço para subdividir o problema em tarefas a serem executadas em paralelo. Dessa forma, situações perfeitamente paralelas podem se beneficiar de um grande ganho em desempenho com pouco efeito no tamanho do código.

O presente trabalho visa estudar o comportamento das quatro linguagens cuja relevância foi discutida anteriormente (Go, Julia, Python e C/OpenMP) em um contexto perfeitamente paralelo de uso intenso de CPU, utilizando-se como ferramenta de comparação o algoritmo de obtenção do conjunto de Mandelbrot. Foram tomadas como métricas de comparação o desempenho dos programas gerados, o *speedup* em latência (ganho em desempenho ao se aumentar o número de processadores) e o tamanho do código, três métricas normalmente usadas na literatura para *benchmarking* de linguagens paralelas. Algumas variáveis levadas em consideração foram o paradigma utilizado na implementação do código (imperativo ou funcional), o número de núcleos de processamento envolvidos nas execuções em paralelo e o escalonamento das atividades em relação aos núcleos de processamento disponíveis. Apesar de as linguagens consideradas serem multiplataforma, suas implementações possuem diferenças e refletem as características do sistema operacional em que são executadas, bem como as características do hardware utilizado. Dessa

forma, o sistema operacional e o tipo de hardware também foram explorados ao longo dos experimentos.

Os capítulos a seguir estão organizados da seguinte maneira: o capítulo 2 tem como objetivo descrever detalhadamente os experimentos realizados e os sistemas computacionais utilizados, bem como os códigos implementados do algoritmo de obtenção do conjunto de Mandelbrot. Os resultados obtidos são indicados na forma de gráficos e tabelas e discutidos de maneira aprofundada no capítulo 3. Uma breve conclusão geral pode ser vista no capítulo 4 e uma pequena cobertura do que ainda há para ser feito está disponível no capítulo 5. Por fim, o apêndice A traz pequenos trechos de códigos e resultados auxiliares, que serviram como ferramentas para a obtenção dos resultados apresentados nos capítulos anteriores.

2 Parte Experimental

2.1 Descrição dos sistemas utilizados

Com a finalidade de se comparar o comportamento das linguagens em diferentes sistemas operacionais, tipos e configurações de hardware, os algoritmos do presente trabalho foram executados em dois sistemas, descritos abaixo:

1. MacBook Pro Mid 2015, com processador Intel Core i7 (4 núcleos físicos, 8 virtuais) e 16 GB de RAM - Sistema Operacional MacOS Sierra 10.12.6 [mbp]

2. Google Cloud Compute Engine Virtual Machine, com 8 núcleos virtuais de processamento e 30 GB de RAM - Sistema Operacional Ubuntu 17.04 Zetsy [gce]

Ao longo do estudo serão utilizados os termos [gce] e [mbp] para designar os diferentes sistemas. É importante ressaltar que ambos apresentam características bem diferentes: enquanto [mbp] é um computador pessoal portátil, de pequeno porte, para uso geral, [gce] é um componente de *Infrastructure as a service*, que permite que máquinas virtuais personalizadas sejam iniciadas sob demanda [34]. Sistemas como [gce] apresentam alto desempenho e são, em geral, utilizados para atividades que demandam alta disponibilidade de recursos de hardware.

Em ambos os sistemas foram instalados o Z shell (zsh), e as seguintes implementações das linguagens: Python 3.6.1, Go 1.7.6 e Julia 0.6. Os programas em C foram compilados usando GCC 7.2.0, com suporte às diretivas OpenMP (*flag* `-fopenmp`). Para a linguagem Python foram também instaladas as bibliotecas `numpy` e `matplotlib`.

2.2 Igualdade das matrizes de saída

Para se estudar as diferenças nos tempos de execução nas condições descritas anteriormente, é necessário garantir que todas as implementações do algoritmo de obtenção do conjunto de Mandelbrot (`mandel`) gerem a mesma matriz de saída. Como a matriz de saída do problema representa o número de iterações necessárias até que ocorra a divergência (ou não), garantir a igualdade das matrizes de saída garante também que todos os algoritmos iteraram o mesmo número de vezes durante a sua execução. Essa isonomia é importante para que as possíveis diferenças nos tempos de execução possam ser associadas diretamente às variáveis de interesse do presente estudo.

Dessa forma, foram comparadas as saídas de todos as variações de `mandel` geradas pelas combinações das seguintes variáveis,

1. Tamanho da matriz de entrada: 3500 x 2500, 7000 x 5000 e 14000 x 10000

2. Linguagem de programação: Go 1.7.6, Julia 0.6 (implementação imperativa), Julia 0.6 (implementação funcional), Python 3.6.1 e C/OpenMP (compilado com GCC 7.2.0)

3. Tipo de código: sequencial (em que as iterações são executadas uma por vez, em sequência) e paralelo (em que mais de uma iteração ocorre ao mesmo tempo, utilizando-se mais de um núcleo de processamento)

totalizando 30 experimentos diferentes. As diferenças entre os códigos serão descritas de forma detalhada na seção 2.3. As combinações de tamanhos para a matriz de entrada foram selecionadas a partir de experimentos prévios, que mostraram que estes são bons exemplos para se verificar variações no efeito da paralelização do código e possíveis *overheads* e que, ao mesmo tempo, permitem que sejam feitas várias replicatas sem demandar tempos proibitivos de disponibilidade do hardware. Assumiu-se o fato de que a diferença no número de processadores utilizados nas execuções em paralelo não influenciam na matriz da saída, obtendo-se o mesmo resultado para qualquer quantidade de núcleos empregada na execução. Portanto, os testes de saída foram feitos apenas para execuções em 8 processadores, capacidade máxima dos sistemas utilizados. Da mesma forma, assumiu-se que o sistema utilizado para a execução, [mbp] e [gce], também não tem influência na saída do programa. Desse modo, o experimento de comparação das saídas foi realizado apenas em [mbp].

Independentemente da implementação, a saída consiste em uma matriz de números inteiros com as mesmas dimensões da entrada, indicando o número de iterações necessárias até a divergência do método (ou o número 100, caso essa divergência não ocorra dentro de 100 iterações). Este limite máximo estabelecido traz um bom balanço entre tempo de execução e a resolução dos fractais gerados em uma possível exportação do conjunto para uma imagem. Quanto maior for o limite estabelecido, menor será a chance de se encontrar um bloco de pixels sem formato bem definido ao dar *zoom* na imagem. Por outro lado, quanto maior for o limite, mais tempo será necessário para computar todas as iterações da obtenção do conjunto. Uma exportação da matriz de saída para imagem é exemplificada na seção 2.3.1.

Em todos eles foi implementado uma código auxiliar responsável por exportar a matriz de saída para um arquivo `.csv` separado por ponto-e-vírgula (ver apêndice, seção A.1).

A comparação entre os diversos arquivos `.csv` foi realizada a partir do comando `diff` (POSIX.1-2008) [35], que recebe dois arquivos de texto (`arq1` e `arq2`) e mostra quais linhas devem ser substituídas em `arq1` para que ele se torne igual ao `arq2`. Dessa forma,

saídas vazias para o comando `diff` indicam que os arquivos são exatamente iguais e, por consequência, que as saídas dos programas também foram exatamente iguais. Para essa comparação, os arquivos foram separados em grupos de acordo com os tamanhos das saídas (Grupo 1: matrizes de 3500 x 2500 geradas por todas as implementações; Grupo 2: matrizes de 7000 x 5000 geradas por todas as implementações; Grupo 3: matrizes de 14000 x 10000 geradas por todas as implementações). Essa abordagem garante a igualdade de todos os arquivos entre si, dentro dos respectivos grupos, uma vez que a igualdade de matrizes é transitiva.

Dadas as características do problema, é necessário tomar certos cuidados em relação aos tipos de variáveis utilizados. A obtenção do conjunto de Mandelbrot, como descrito na seção introdutória, consiste em calcular a equação $z = z^2 + c$ de modo iterativo até que haja divergência ou se atinja o limite máximo de iterações. Pequenas diferenças nos valores de z podem fazer com que sejam necessárias mais ou menos iterações até se verificar a divergência.

Lembremos que os números complexos, nas linguagens estudadas, são representados por uma tupla de dois números de ponto flutuante (formato `float`). Cada linguagem tem a sua própria implementação de `float`, que pode ou não se enquadrar no padrão IEEE 754 de dupla precisão (64 bits). O tipo `float` em Python, por exemplo, é implementado de forma a ter precisão infinita, o que não ocorre no padrão IEEE de 64 bits.

Dessa forma, tomou-se o cuidado de utilizar a implementação de ponto flutuante no padrão IEEE 754 oferecida por cada uma das linguagens: `double` em C/OpenMP, `Float64` em Go e Julia, implementados nativamente, e `numpy.Float64` em Python, oferecido pela biblioteca *numpy* [36].

Vale ressaltar que essa dificuldade vai além de garantir que todas as linguagens estejam implementando tipos de ponto flutuante com a mesma precisão. Qualquer mudança envolvendo o tipo de arredondamento (*round up*, *round down*, *round to zero* ou *round nearest*) pode fazer com que haja diferença no número final de iterações, dependendo dos valores de entrada. Em experimentos prévios foi verificado que a simples mudança da versão do *Go* (de 1.7.6 para 1.9) faz com que, para certos valores de entrada, ocorra uma iteração a menos ou a mais. Verificar qual é a diferença na implementação de ponto flutuante entre as duas versões está fora do escopo do presente trabalho e, por isso, utilizou-se a versão 1.7.6, cujos resultados condizem com os obtidos pelas outras linguagens.

2.3 Detalhamento dos códigos

2.3.1 Obtenção do conjunto de Mandelbrot (`mandel`)

A implementação do algoritmo `mandel` não oferece grandes desafios lógicos. O pseudocódigo abaixo representa uma rotina que devolve o número de iterações partindo do complexo c até que haja divergência ou se atinja o limite máximo de iterações:

```
z = 0 + c
para k entre 1 e 100
    se |z| > 2
        retorna k-1
    fim-se
    z = z^2 + c
fim-para
retorna 100
```

Uma versão, em linguagem Julia, pode ser encontrada diretamente na documentação da linguagem [15]. Na implementação sequencial em Julia utilizada no presente trabalho, algumas pequenas modificações foram realizadas:

```
function mandelbrotorbit(c)
    z = Complex128(0,0) + c
    for k = 1:100
        if abs(z) > 2
            return k-1
        end
        z = z^2 + c
    end
    return 100
end
```

A função `mandelbrotorbit` acima devolve, para o número complexo c fornecido como argumento, o número de iterações do tipo $z = z^2 + c$, com z iniciando em 0, executadas até que o módulo de z ultrapasse o valor 2 (este valor é explicado na seção 1). A variável `maxiter`, nesse caso, é configurada como 100. Isso significa que serão realizadas, no máximo, 100 iterações desse tipo. Uma vez atingido esse valor, assume-se que $z = z^2 + c$ converge para o valor de c fornecido.

Essa função `mandelbrotorbit` é, então, calculada em todos os pontos de uma matriz de entrada, que simboliza pontos em um plano complexo. O resultado é uma matriz de saída que traz como informação a *velocidade* da divergência para cada um dos pontos. Uma implementação sequencial desse trecho em linguagem Julia pode ser visualizada a seguir, e consiste em dois laços aninhados:

```
function mandelbrot(inputmat::Array{Complex128,2}, outputmat::Array{Integer,2})
    for j=1:size(inputmat,1)
        for k=1:size(inputmat,2)
            @inbounds outputmat[j,k] = mandelbrotorbit(inputmat[j,k])
        end
    end
end
```

A matriz de saída pode ser representada visualmente, gerando os conhecidos fractais do conjunto de Mandelbrot, tal como mostra a Figura 1 a seguir, obtida a partir da adição de duas linhas ao final do código da implementação de `mandel` em Python:

```
pyplot.imshow(N)
pyplot.savefig('mandel.png')
```

sendo N a representação numérica da matriz de saída do algoritmo. A convergência ocorre no plano complexo de acordo com a formação de fractais bem característicos.

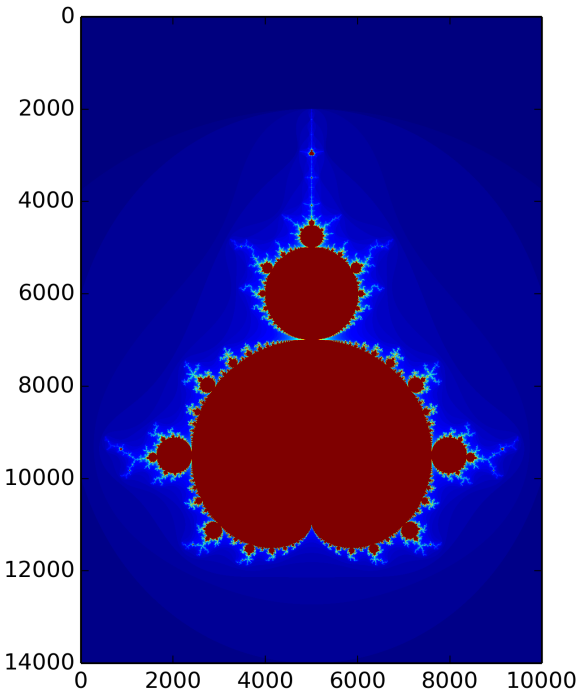

Figura 1 – Representação visual da matriz de saída gerada pela função `mandelbrot` implementada em Python, com matriz de entrada contendo $n = 14000$ linhas e $m = 10000$ colunas. O mapa de cores utilizado é padrão da classe `pyplot` e a resolução foi alterada para 300 dpi via arquivo de configuração da biblioteca `matplotlib`.

A implementação sequencial em Go possui aspecto muito semelhante, utilizando *slices*, que são estruturas de dados características da linguagem, para armazenar os dados de entrada e saída.

```
func mandelbrot(inputmat [][]complex128) [][]int {
        outputmat := make([][]int, rows)
        for i := range outputmat {
                outputmat[i] = make([]int, columns)
        }
        for i := 0; i < rows; i++ {
                for j := 0; j < columns; j++ {
                        outputmat[i][j] = mandelbrotorbit(inputmat[i][j])
                }
        }
        return outputmat
}
```

O mesmo pode ser dito para a implementação em C/OpenMP:

```
int** mandelbrot(Complex** inputmat) {
    int **outputmat = malloc(rows * sizeof(int*));
    for (int i = 0; i < rows; i++){
        outputmat[i] = malloc(columns * sizeof(int));
    }

    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < columns; j++) {
            outputmat[i][j] = mandelbrotorbit(inputmat[i][j]);
        }
    }
    return outputmat;
}
```

É importante ressaltar que a linguagem C não possui uma implementação nativa de números complexos, de modo que o tipo `Complex` precisou ser desenvolvido a partir do zero. Mais detalhes sobre essa implementação estão disponíveis na seção A.1.

Já a implementação sequencial em Python, por sua vez, tem um caráter mais funcional, utilizando listas e *maps*. A função `mandelbrotorbit` é bastante análoga às existentes nas outras duas implementações. A função `mandelbrot`, responsável por iterar no plano complexo, adquire a forma a seguir:

```
def mandelbrot(x):
    Z = [complex(x,y) for y in Y]
    return list(map(mandelbrotorbit, Z))

N = map(mandelbrot, X)
```

Note a redução em linhas de código quando comparada com a implementação anterior, característica do paradigma funcional de programação. Essa funcionalização do código também permite que o paralelismo seja implementado quase que diretamente, a partir da biblioteca *multiprocessing*. Essa biblioteca utiliza múltiplas instâncias de Python para trazer paralelismo à execução, driblando o problema do GIL [25], característico das linguagens interpretadas. A única diferença do código paralelo utilizado em relação ao sequencial é a troca da linha

```
N = map(mandelbrot, X)
```

pelas linhas

```
p = Pool(k)
N = p.map(mandelbrot, X)
```

sendo que a primeira linha é responsável pela criação de um *pool* de k processos e a segunda por executar o método `map` de forma paralela, implementada em objetos do tipo *pool*.

A implementação paralela em Go é baseada na utilização de *go routines*, que é uma forma característica do Go de se trabalhar com paralelismo e/ou concorrência. Em termos de aparência do código, as *go routines* lembram *threads*, com comandos do tipo *add*, *wait* e *done* para criar e controlar o fluxo de execução do programa, mas sem a preocupação de lidar manualmente com semáforos e *mutexes*:

```
func mandelbrot(inputmat [][]complex128) [][]int {
        runtime.GOMAXPROCS(k)
        var wg sync.WaitGroup
        wg.Add(rows)

        outputmat := make([][]int, rows)
        for i := range outputmat {
                outputmat[i] = make([]int, columns)
        }

        for i := 0; i < rows; i++ {
                go func(i int) {
                        for j := 0; j < columns; j++ {
                                outputmat[i][j] = mandelbrotorbit(inputmat[i][j])
                        }
                        wg.Done()
                }(i)
        }
        wg.Wait()
        return outputmat
}
```

A linha `runtime.GOMAXPROCS(k)` é responsável por estabelecer o número máximo de processadores liberados para executar *go routines* em paralelo, no caso k processadores. A implementação das rotinas é feita, por padrão, de forma concorrente, competindo pelos recursos de um único processador [37]. Mais detalhes a respeito da diferença entre paralelismo e concorrência podem ser vistos na seção 1.

O paralelismo em Julia [38] é feito, por padrão, com base no paradigma de *message passing*, que é mais adequado a sistemas distruibuídos. Para situações de memória compartilhada, no entanto, ela oferece como recursos os `SharedArrays`, que são equivalentes aos `Arrays`, mas foram implementados de forma a trabalhar com múltiplos acessos. Diretivas como `@everywhere`, `@sync` e `@parallel` indicam que certos trechos do código devem ser executados em paralelo:

```
function mandelbrot(inputmat::SharedArray{Complex128,2}, outputmat::SharedArray{Int8,2})
    @sync @parallel for k=1:size(inputmat,2)
        for j=1:size(inputmat,1)
            @inbounds outputmat[j,k] = mandelbrotorbit(inputmat[j,k])
        end
    end
end
```

A diretiva `@inbounds` não está diretamente relacionada ao paralelismo. Ela faz com que o código execute sem que haja verificação das bordas da matriz para garantir que os índices estão dentro do range, assumindo que isso é verdade [39]. Isso traz um pequeno aumento no desempenho do código, tanto sequencial quanto paralelo.

Os objetos `SharedArray` são estruturas de bits e, por isso, o tipo `Int8` foi utilizado em substituição ao tipo `Integer` empregado no código paralelo.

Julia, assim como Python, também oferece uma implementação paralelizada da função *map* (`pmap`), que também funciona em sistemas de memória compartilhada. Uma versão funcional do algoritmo em Julia, baseada em `pmap`, também foi estudada.

```
function mandelbrot(x::Float64)
    Z = [complex(x,y) for y in Y]
    return map(mandelbrotorbit, Z)
end

X = linspace(-2.5, 1.0, rows)
Y = linspace(-1.25, 1.25, columns)

N = pmap(mandelbrot, X)
```

Diferentemente das outras duas linguagens, o controle do número máximo de processos permitidos, em Julia, é realizado a partir da *flag* de execução `-p numero_de_threads`,

de modo que essa definição é passada no comando de execução do programa e não dentro do código, como no caso das outras linguagens estudadas.

Por fim, em C/OpenMP, o paralelismo é definido por meio de diretivas, como pode ser visto no exemplo abaixo:

```
int** mandelbrot(Complex** inputmat) {
    int **outputmat = malloc(rows * sizeof(int*));
    for (int i = 0; i < rows; i++){
        outputmat[i] = malloc(columns * sizeof(int));
    }

    #pragma omp parallel for
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < columns; j++) {
            outputmat[i][j] = mandelbrotorbit(inputmat[i][j]);
        }
    }
    return outputmat;
}
```

Esta implementação, cuja diretiva não apresenta nenhuma informação a respeito do escalonamento dos múltiplos processos, utiliza escalonamento estático. Pode-se optar por um escalonamento dinâmico, controlando o tamanho dos *chunks* de tarefas para cada processo (`chunk_size`), e até mesmo definir o número de núcleos de processamento que serão utilizados (`nthreads`). Para isso, basta substituir a diretiva

```
#pragma omp parallel for
```

por

```
#pragma omp parallel for schedule(dynamic, chunk_size) num_threads(nthreads)
```

Os programas completos podem ser vistos no apêndice, na seção A.1. Os códigos discutidos aqui foram inspirados em outros existentes na literatura [40–43], com modificações no sentido de extrair apenas as partes interessantes para a análise e adequá-las às condições de execução. Em alguns casos certas estruturas de dados foram substituídas. Os códigos em C/OpenMP foram desenvolvidos do zero. Todos os códigos utilizados partem de dados legíveis para humanos (números complexos baseados em pontos flutuantes) e chegam a dados legíveis para humanos (números inteiros), apesar de não serem impressos ao final da execução para não interferir nas medições. Procurou-se, dentro do possível, utilizar apenas tipos e funções nativas das linguagens, com exceção da implementação em Python, em que foi utilizado o tipo `float64` da biblioteca `numpy` [36] para garantir que todos os algoritmos utilizassem ponto flutuante de 64-bits padrão IEEE 754, uma

vez que o tipo `float` nativo do python possui precisão infinita, provocando diferenças no número de iterações da função `mandelbrot` (essa discussão é realizada de forma detalhada na seção 2.2). O autor procurou, dentro de seu conhecimento das linguagens, escrever códigos idiomáticos, buscando não penalizar o desempenho dos programas.

2.3.2 Geração de matrizes de entrada

As matrizes de entrada para `mandel` possuem, em cada uma de suas células, um número complexo, que por sua vez é composto por dois números de ponto flutuante: a parte real e a parte imaginária.

Para evitar que essas matrizes, que em seu tamanho máximo possuem 140 milhões de células, fossem inseridas no código dos programas, ou mesmo lidas de um arquivo externo, optou-se por gerá-las em tempo de execução. Entre as vantagens dessa abordagem está o fato de não ser necessário representar esses números, o que poderia trazer imprecisões na hora de importá-los (já que seriam exportados com um número finito de casas decimais por um outro programa).

A abordagem para a geração dessas matrizes foi a mesma em todas as linguagens estudadas: as células foram geradas a partir de espaços lineares nos eixos das abscissas e ordenadas: dado um valor de início (`start`), um valor de fim (`end`), e o número de pontos a serem gerados (`n`), gera-se um conjunto de `n` pontos linearmente espaçados entre `start` e `end`. Como o problema de geração de espaços lineares não faz parte do escopo deste trabalho, optou-se por gerá-los todos de forma sequencial.

O código em Julia para a geração das matrizes de entrada encontra-se a seguir:

```
function linspace(start::Float64, finish::Float64, n::Integer)
    linspace = [start]
    distance = (finish - start) / Float64(n-1)
    next = start
    while abs(next - finish) > 0.000000000001
        next = next + distance
        linspace = append!(linspace, next)
    end
    return linspace
end

X = linspace(-2.5, 1.0, rows)
Y = linspace(-1.25, 1.25, columns)
Z = [complex(x,y) for x in X, y in Y]
```

De modo análogo, as matrizes de entrada em Python são obtidas a partir do código a seguir:

```
def linspace(start, end, n):
    lspace = [start]
    distance = (end - start) / np.float64(n - 1)
    next = start
    while np.abs(next - end) > 0.000000000001:
        next += distance
        lspace.append(next)
    return lspace


X = linspace(xmin, xmax, nx)
Y = linspace(ymin, ymax, ny)

def compute_all_y(x):
    return [complex(x, y) for y in Y]

Z = list(map(compute_all_y, X))
```

As linguagens Go e C não oferecem a função `map` (ou qualquer outra similar) nativamente, de forma que os complexos foram construídos a partir de laços.

Em Go:

```
func linspace(start float64, end float64, n int) []float64 {
        var linspace []float64
        linspace = append(linspace, start)
        distance := (end - start) / float64(n-1)
        next := start
        for math.Abs(next-end) > 0.000000000001 {
                next = next + distance
                linspace = append(linspace, next)
        }
        return linspace
}

func clinspace(start complex128, end complex128, m int, n int) [][]complex128 {
        realParts := linspace(real(start), real(end), m)
        imagLinspace := linspace(imag(start), imag(end), n)
        var cmplxParts []float64
        for im := range imagLinspace {
                cmplxParts = append(cmplxParts, imagLinspace[im])
        }

        var cmplxLinspace [][]complex128
        for r := range realParts {
                var cmplxRow []complex128
                for c := range cmplxParts {
                        cmplxRow = append(cmplxRow, complex(realParts[r], cmplxParts[c]))
                }
                cmplxLinspace = append(cmplxLinspace, cmplxRow)
        }
```

```
        return cmplxLinspace

    func main() {
                clinspace(-2.5-1.25i, 1.0+1.25i, rows, columns)
        }
}
```

Em C:

```
double* linspace(double start, double end, int n) {
    double *lnsp = malloc(n * sizeof(double));
    lnsp[0] = start;
    double distance = (end - start) / (double)(n-1);
    double next = start;
    for (int i = 1; i < n; i++) {
        next = next + distance;
        lnsp[i] = next;
    }
    return lnsp;
}

Complex** clinspace(Complex start, Complex end, int m, int n) {
    double *reLnsp = linspace(start.re, end.re, m);
    double *imLnsp = linspace(start.im, end.im, n);

    Complex** cmplxLnsp = malloc(m * sizeof(Complex*));
    for(int i = 0; i < m; i++) {
        cmplxLnsp[i] = malloc(n * sizeof(Complex));
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            Complex c;
            c.re = reLnsp[i];
            c.im = imLnsp[j];
            cmplxLnsp[i][j] = c;
        }
    }
    free(reLnsp);
    free(imLnsp);
    return cmplxLnsp;
}

int main() {

    Complex start, end;
    start.re = -2.5; start.im = -1.25;
    end.re = 1.0; end.im = 1.25;
    Complex **inputmat = clinspace(start, end, ROWS, COLS);
    return 0;
}
```

Os códigos completos referentes à geração das matrizes de entrada podem ser consultados no apêndice, na seção A.1. Algumas das linguagens, como Julia (ou mesmo Python, a partir da biblioteca `NumPy`), possuem funções que geram espaços lineares. Optou-se, no entanto, por implementá-los do zero de formas análogas, sem grandes esforços para otimizá-los. Acredita-se que essa abordagem diminui um possível viés para uma determinada linguagem.

2.4 Tempos de execução

Uma série de experimentos foram realizados para se estudar a influência das variáveis consideradas nos tempos totais de execução (geração das matrizes de entrada + trecho referente apenas à obtenção dos conjuntos de Mandelbrot) dos programas discutidos em detalhes na seção 2.3.

2.4.1 Estudo dos sistemas, linguagens e tamanhos de entrada

Com o objetivo de comparar os tempos de execução das diferentes implementações para obtenção do Conjunto de Mandelbrot, foram realizados experimentos combinando os seguintes critérios:

1. Sistema: [mbp] ou [gce]
2. Linguagem de programação: Go, Julia (implementação imperativa), Julia (implementação funcional), Python e C/OpenMP
3. Tamanho das matrizes de entrada: 3500 x 2500, 7000 x 5000 e 14000 x 10000
4. Tipo de computação: sequencial ou paralela (8 processadores)

totalizando 60 experimentos, sendo que para cada um deles foram realizadas 102 replicatas. As duas primeiras são realizadas apenas para efeito de *warm-up* do sistema, sendo desconsideradas nas estatísticas. Isso garante que o tempo extra necessário para compilação do programa (no caso de Julia, por exemplo), não seja levado em conta. Além de lidar com o tempo de compilação, o *warm up* é responsável por levar o sistema a um estado estacionário de máximo desempenho para a execução do programa, inicializando variáveis e métodos em *cache*, por exemplo [44]. Sendo assim, apenas as 100 últimas são consideradas no cálculo das médias.

2.4.2 Estudo do número de processadores

Com o objetivo de se estudar o *speedup* em latência gerado pela paralelização dos algoritmos em função do número de processadores utilizados na execução, foram realizados experimentos combinando os seguintes critérios:

1. Sistema: [gce]
2. Linguagem de programação: Go, Julia (implementação imperativa), Julia (implementação funcional), Python e C/OpenMP
3. Tamanho da matriz de entrada: 14000 x 10000
4. Tipo de computação: sequencial (1 processador) e paralela (2, 4 e 8 processadores)

totalizando 20 experimentos, sendo que para cada um deles foram realizadas 102 replicatas.

O *speedup* em latência, neste caso específico, é dado pela razão entre o inverso do tempo de execução da versão sequencial e o inverso do tempo de execução da versão paralela em n processadores, com n variando entre 2, 4 e 8.

2.4.3 Estudo do tipo de escalonamento de processos

Com o objetivo de se estudar o tipo de escalonamento de processos e o tamanho dos pacotes de atividades selecionados a partir das diretivas de C/OpenMP (ver seção 1), foram realizados experimentos combinando os seguintes critérios:

1. Sistema: [gce]
2. Linguagem de programação: C/OpenMP
3. Tamanho da matriz de entrada: 14000 x 10000
4. Tipo de computação: paralela (8 processadores)
5. Tipo de escalonamento: estático, dinâmico
6. Tamanho dos pacotes (*chunk sizes*): 1, 100, 1000 e 2000

totalizando 8 experimentos, sendo que para cada um deles foram realizadas 102 replicatas. Os tamanhos dos pacotes foram escolhidos de forma a partir de uma situação limite inferior (uma única atividade por pacote) até uma situação próxima à gerada pelo escalonamento estático, com 2000 atividades por pacote. Vale notar que o escalonamento estático divide as 14000 atividades igualmente entre os 8 núcleos disponíveis, já em tempo de compilação, resultando em 1750 atividades por processo. Essa questão será avaliada com mais detalhes na seção 3.1.3.

2.4.4 Tempos de execução da geração de matrizes de entrada

Com a finalidade de se eliminar o viés introduzido pela geração das matrizes de entrada nas diversas implementações do código, os tempos de execução do trecho responsável pela geração das entradas foram computados também em separado, visando a sua subtração dos tempos totais dos experimentos. O código Python responsável por esse procedimento está disponível no apêndice, seção A.2.1.

2.4.5 Distribuição dos tempos de execução e geração das entradas

Com a finalidade de se obter a medida de comparação mais adequada entre os tempos de execução, foi realizado um estudo prévio da distribuição dos tempos obtidos nas 100 últimas replicatas de cada experimento.

No geral, pôde-se verificar a existência de assimetrias nas curvas obtidas (também verificadas nos gráficos quantis-quantis), exemplo na Figura 2, indicando que a mediana é mais apropriada do que a média para exprimir a ideia de centralidade dos conjuntos de dados, por ser uma medida mais robusta. Apesar disso, a diferença entre ambas, em módulo, não ultrapassou 5% para qualquer um dos experimentos realizados, devido às variabilidades razoavelmente pequenas. Considerando-se a versatilidade da média e do desvio padrão da média como forma de expressar a medida e sua incerteza, no que diz respeito a propagações, optou-se por utilizar essas duas medidas ao longo deste trabalho. Dessa forma, foram computados $(\bar{x} \pm sd(x))$ das 100 últimas replicatas para todos os experimentos realizados.

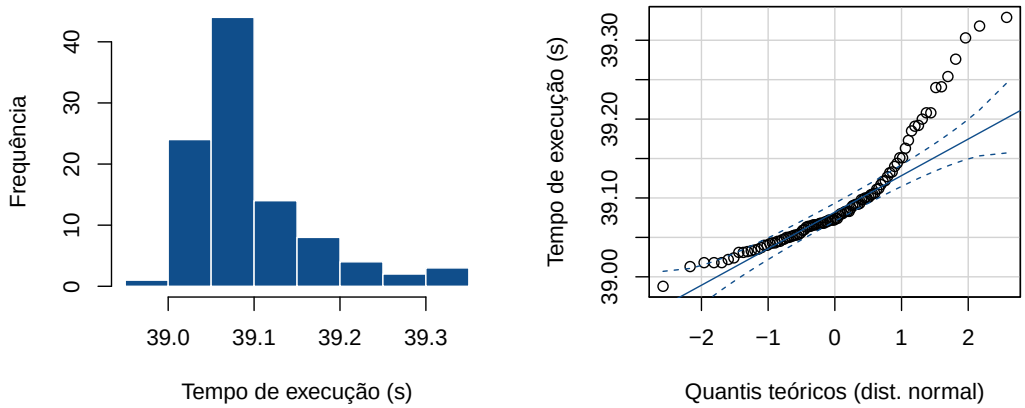

Figura 2 – Distribuição dos tempos de execução da implementação do algoritmo `mandel` em C/OpenMP, executando em modo paralelo (4 núcleos de processamento) com escalonamento dinâmico, com matriz de entrada de tamanho 14000×10000. À esquerda: histograma de frequências dos tempos de execução. À direita: gráfico quantil-quantil dos tempos de execução *vs* distribuição normal teórica

Em alguns poucos experimentos verificou-se uma distribuição bimodal, tal como pode ser visto na Figura 3 a seguir. Nestes casos específicos, verificou-se a possibilidade de adotar como tendência central a média entre os intervalos modais. Considerando-se que a diferença entre esses valores e as médias dos conjuntos de dados também não ultrapassaram 5%, em módulo, a média e o desvio padrão foram adotados também nesses casos por questão de simplicidade.

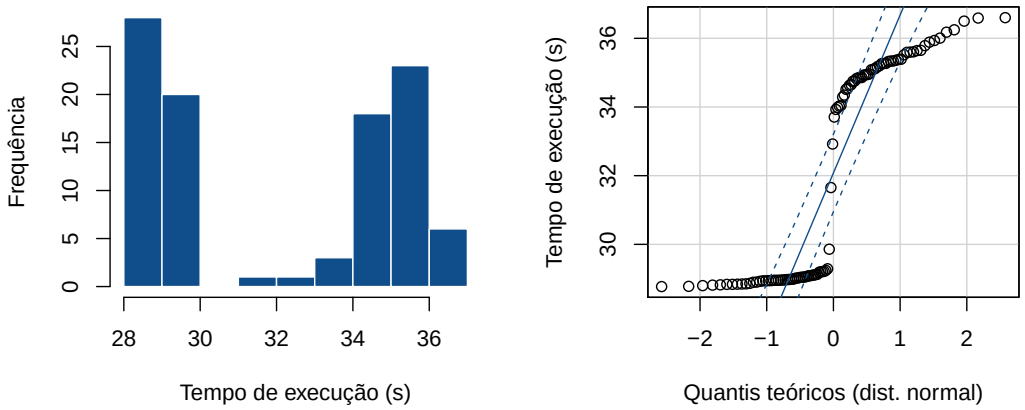

Figura 3 – Distribuição dos tempos de execução da implementação do algoritmo `mandel` em Go, executando em modo sequencial, com matriz de entrada de tamanho 3500×2500. À esquerda: histograma de frequências dos tempos de execução. À direita: gráfico quantil-quantil dos tempos de execução *vs* distribuição normal teórica

Cabe notar que, para qualquer uma das abordagens discutidas anteriormente para expressar a centralidade, a influência na comparação entre os tempos de execução das implementações seria mínima, uma vez que a diferença entre esses tempos para cada um dos experimentos, em geral, ultrapassa essa diferença de 5% entre as abordagens discutidas. Dessa forma, qualquer uma das abordagens (média, mediana ou média das modas) poderia ser utilizada.

Um *script* R para gerar histogramas e gráficos quantil-quantil de cada um dos experimentos está disponível na seção A.2.2.

2.4.6 Ferramentas

Para a medida dos tempos de execução foi utilizado o comando `time` do Z shell (*zsh*), instalado em ambos os sistemas. Um exemplo de saída do comando é explicado brevemente a seguir:

```
julia mandel-seq.jl 34.75s user 2.04s system 102% cpu 35.997 total
```

A primeira informação é o nome do processo cujo tempo foi medido. A segunda informação, seguida da palavra `user`, é o tempo de CPU gasto **dentro do processo** (o tempo gasto em outros processos e com o processo em *standby* não são computados). Para o caso de computações em paralelo, o tempo gasto por cada um dos núcleos é somado. A terceira informação, seguida da palavra `system` é o tempo de CPU gasto **dentro do processo e dentro do kernel** com chamadas de sistema. A quarta informação, seguida da palavra `cpu`, indica a porcentagem de CPU gasta durante a execução. Para execuções sequenciais com alto consumo de CPU, esse valor costuma se manter ao redor de 100%. Para execuções em paralelo, esse valor normalmente ultrapassa os 100%, e depende do número de núcleos utilizados. Por fim, a quinta informação, seguida da palavra `total`, indica o tempo cronometrado entre o início e o fim da execução. Esta informação é indicada na literatura em outros contextos pelos termos *clock time* ou *elapsed time*, e foi a medida de tempo escolhida para o presente estudo.

Quando o tempo cronometrado ultrapassa 60 segundos, o comando `time` passa a contabilizá-lo em minutos, gerando saídas com precisão de centésimos de segundo ao invés de milésimos de segundo:

```
python mandel-seq-14000.py 855.92s user 1.96s system 100% cpu 14:17.07 total
```

Para tornar as execuções mais automáticas, as implementações de `mandel` foram criadas de modo a receber os parâmetros de cada execução por meio de *variáveis de ambiente* (ex. tamanho das matrizes de entrada, número de núcleos de processamento que serão utilizados, tipo de escalonamento de processos, etc.). Os programas foram, então, disparados por um *shell script*, responsável por definir essas variáveis de ambiente e, posteriormente, encadear todas as execuções sem que fosse necessária a intervenção do autor entre cada uma delas. Este processo facilitou a realização dos experimentos e otimizou a utilização dos serviços em nuvem, cobrados pelo tempo de uso.

Para cada experimento foi gerado um arquivo `.txt` com as 102 saídas do comando `time` aplicado a cada um dos algoritmos. A partir de um *script* Python (ver apêndice, seção A.2.1), os tempos totais foram convertidos para segundos e as médias e desvios padrões foram computados. Das 102 saídas de cada experimento, as duas primeiras foram desconsideradas. Essa abordagem foi utilizada para evitar que tempos de *set-up* e compilação pudessem influenciar nos tempos estudados.

As linguagens utilizadas possuem formas de se medir o tempo de execução dos trechos de interesse no interior do programa (ex. comando `@time` em Julia, módulo `timeit` em Python, etc). Optou-se por medir o tempo externamente aos programas, utilizando o comando `time`, para garantir que o mesmo tratamento seja dado para todas as implementações, evitando possíveis comparações entre diferentes tipos de medição de

tempo (ex. *clock time*, tempo de cpu, etc.).

2.5 Número de linhas de código

Com a finalidade de comparar a diferença no tamanho do código entre as diversas implementações, sequenciais e paralelas, considerou-se, para cada uma delas, o número de linhas de código referentes aos trechos de obtenção do conjunto de Mandelbrot. Para essa finalidade, foram ignoradas as linhas responsáveis pela geração das matrizes de entrada.

3 Resultados e discussão

3.1 Tempos de execução

3.1.1 Estudo dos sistemas, linguagens e tamanhos de entrada

Calculou-se, para cada implementação, a média do tempo de execução das 100 replicatas. Destas médias foram subtraídas as médias dos tempos de geração das matrizes de entrada para as implementações consideradas. Mais detalhes sobre essa abordagem podem ser obtidos nas seções 2.4.4 e 3.3. Os resultados obtidos, com as devidas propagações dos erros experimentais, podem ser vistos na Tabela 1.

Tabela 1 – Tempos médios de execução do algoritmo `mandel`, em segundos, corrigidos pela subtração dos tempos médios de geração das matrizes de entrada. As variáveis consideradas foram: sistema utilizado, tipo de execução, tamanho da entrada e linguagem de programação utilizada na implementação do algoritmo.

	Go					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
mbp	$25{,}02 \pm 0{,}06$	$99{,}39 \pm 0{,}14$	$396{,}19 \pm 0{,}16$	$5{,}42 \pm 0{,}09$	$21{,}32 \pm 0{,}26$	$83{,}36 \pm 0{,}43$
gce	$32{,}0 \pm 3{,}1$	$15{,}6 \pm 4{,}5$	460 ± 16	$5{,}26 \pm 0{,}07$	$21{,}18 \pm 0{,}06$	$83{,}73 \pm 0{,}43$
	JuliaFuncional					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
mbp	$2{,}78 \pm 0{,}03$	$7{,}91 \pm 0{,}05$	$28{,}95 \pm 0{,}96$	$3{,}47 \pm 0{,}03$	$5{,}04 \pm 0{,}04$	$9{,}86 \pm 0{,}09$
gce	$3{,}37 \pm 0{,}32$	$9{,}47 \pm 0{,}81$	$30{,}2 \pm 3{,}6$	$3{,}99 \pm 0{,}07$	$5{,}33 \pm 0{,}10$	$8{,}41 \pm 0{,}13$
	JuliaImperativo					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
mbp	$1{,}94 \pm 0{,}09$	$7{,}50 \pm 0{,}11$	$26{,}9 \pm 1{,}3$	$5{,}68 \pm 0{,}04$	$7{,}46 \pm 0{,}05$	$14{,}79 \pm 0{,}35$
gce	$2{,}03 \pm 0{,}26$	$6{,}83 \pm 0{,}62$	$24{,}25 \pm 0{,}42$	$6{,}57 \pm 0{,}05$	$8{,}17 \pm 0{,}08$	$15{,}55 \pm 0{,}79$
	Python					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
mbp	$41{,}25 \pm 0{,}64$	$164{,}8 \pm 3{,}1$	657 ± 11	$12{,}19 \pm 0{,}52$	$47{,}24 \pm 0{,}81$	$186{,}5 \pm 3{,}1$
gce	$48{,}9 \pm 1{,}1$	$194{,}1 \pm 4{,}8$	798 ± 30	$11{,}61 \pm 0{,}33$	$44{,}90 \pm 0{,}83$	$176{,}3 \pm 3{,}5$
	C/OpenMP					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
mbp	$3{,}26 \pm 0{,}03$	$13{,}10 \pm 0{,}30$	$52{,}6 \pm 1{,}5$	$0{,}78 \pm 0{,}01$	$3{,}23 \pm 0{,}03$	$12{,}97 \pm 0{,}06$
gce	$9{,}90 \pm 0{,}19$	$40{,}54 \pm 0{,}41$	$164{,}7 \pm 3{,}4$	$1{,}580 \pm 0{,}022$	$6{,}15 \pm 0{,}06$	$24{,}28 \pm 0{,}36$

A primeira informação que o experimento nos traz é a de que os códigos implementados em Julia, tanto em sua versão funcional quando imperativa, executam razoavelmente

mais rápido do que as implementações em Go e Python para todos os tamanhos de matrizes de entrada estudados e desempenham, inclusive, um pouco melhor do que em C/OpenMP em algumas ocasiões. O melhor desempenho de execuções sequenciais de Julia diante das outras linguagens estudadas é discutido na própria documentação [15], apesar disso, a diferença entre Julia e Go se mostrou bem maior nesses resultados quando comparado com os resultados descritos na documentação. Esse fato é bastante curioso, uma vez que a leitura do código utilizado nos *benchmarks* de Julia [45] mostra que não existem grandes diferenças entre eles e os utilizados no presente trabalho, no que diz respeito à função `mandelbrotorbit`. Mais estudos seriam necessários para explicar a fonte dessas diferenças nos resultados, que podem estar no sistema utilizado e em possíveis otimizações realizadas. Python apresentou os tempos mais lentos tanto para as execuções sequenciais quanto as paralelas. Esse resultado também é esperado, dado que é a única linguagem interpretada deste estudo, enquanto todas as outras são compiladas.

Um fato interessante a ser discutido é que, apesar de a diferença entre os tempos de execução ser, em geral, pequena entre os dois sistemas testados, a diferença na variabilidade entre os 100 experimentos foi, em média, 8 vezes maior. Para a grande maioria dos experimentos, o sistema [gce] apresentou maior variabilidade em relação ao sistema [mbp], sendo que para a implementação sequencial em Go, com matriz de entrada de tamanho 14000×10000, a diferença foi de, aproximadamente, 94 vezes. A explicação para isso pode estar no fato de que o hardware do sistema [gce] é compartilhado entre diversos usuários, de modo a estar sujeito a diferentes cargas de uso dependendo do horário da execução, enquanto que o sistema [mbp] é local e não esteve sujeito a nenhuma utilização extra de hardware ao longo da execução dos experimentos.

As diferenças entre as duas execuções sequenciais de Julia são pequenas, com a versão imperativa sendo levemente mais rápida do que a versão funcional. Já as execuções em paralelo mostram ganhos em tempo de execução bem diferentes, com a versão funcional (aproximadamente 3,6 vezes mais rápida do que a versão sequencial) apresentando maior ganho em relação à sua versão imperativa (aproximadamente 1,6 vezes mais rápida do que a versão sequencial), no sistema [gce]. Observando estes resultados com atenção aos diferentes tamanhos das matrizes, o paralelismo utilizado no código imperativo parece apresentar maior *overhead* quando comparado com o utilizado na versão funcional, que pode estar relacionado à utilização das estruturas `SharedArray`. Como exemplo, pode-se citar a implementação imperativa de Julia, executada em paralelo, com matriz de entrada de tamanho 3500×2500, teve desempenho 3,2 vezes mais lento que sua versão sequencial. No caso da implementação funcional, a versão paralela foi apenas 1,2 vezes mais lenta do que a sequencial, ambos no sistema [gce]. Vale ressaltar que isso é apenas uma especulação e mais experimentos seriam necessários para comprovar esse fato. De qualquer forma, esses resultados deixam claro que a utilização do paralelismo nem sempre traz um ganho em tempo de execução, e que o tamanho do problema é algo a ser considerado ao se

implementar uma estratégia paralela. Às vezes o *overhead* do paralelismo faz com que a execução em paralelo seja mais custosa do que a sequencial, e isso pode variar bastante de linguagem para linguagem, dependendo da forma que o paralelismo é implementado.

A linguagem que apresentou o maior ganho em tempo de execução em relação à sua versão sequencial foi C, seguida por Go, Python e, em último lugar, Julia (em sua implementação funcional) – aproximadamente 6,8x; 5,5x; 4,5x; 3,6x – respectivamente, para as execuções em [gce] com matrizes de entrada de tamanho 14000×10000. A implementação imperativa de Julia teve o pior ganho entre todos eles, de aproximadamente 1,6x. Isso mostra que a estratégia de paralelismo utilizada, mesmo considerando a mesma linguagem de programação, é importante para se obter o ganho desejado.

3.1.2 Estudo do número de processadores

Calculou-se, para cada implementação, o *speedup* com base nos inversos dos tempos de execução (corrigidos pela subtração dos tempos de geração das matrizes de entrada), para 2, 4 e 8 processadores. Os resultados obtidos, com as devidas propagações dos erros experimentais, podem ser vistos na Figura 4.

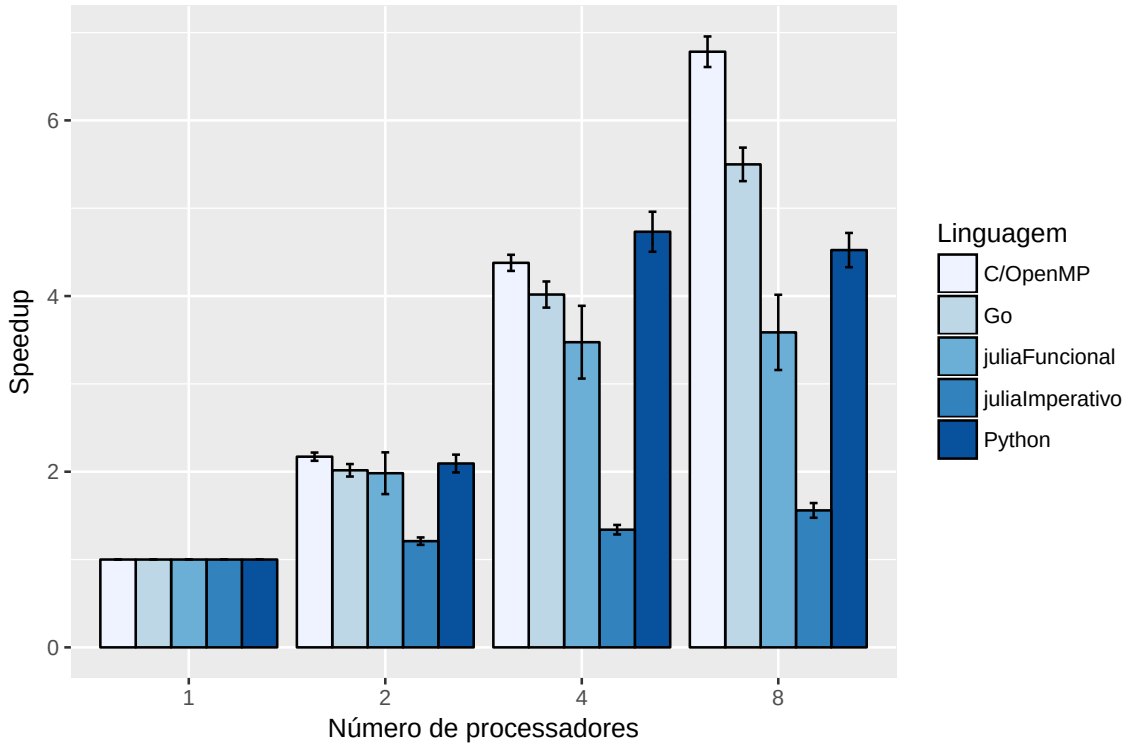

Figura 4 – *Speedup* das diversas implementações do algoritmo `mandel`, executadas no sistema gce com matrizes de entrada de tamanho 14000×10000.

Os resultados mostram que, apesar de todas as barras representarem o mesmo problema, a obtencão do Conjunto de Mandelbrot para uma matriz de entrada de tamanho

14000×10000, os valores obtidos foram bastante diferentes para cada uma das implementações. C/OpenMP apresentou o maior *speedup* máximo, com um ganho de quase 7x em relação ao tempo de execução sequencial. Go apresentou o segundo maior *speedup* máximo e Python apresentou o maior speedup com 4 processadores. As implementações em Julia apresentaram os menores *speedups*, apesar de possuírem os menores tempos de execução sequencial e paralelo (para matrizes de entrada de tamanho 14000×10000). A implementação imperativa em Julia apresentou pouco *speedup* para qualquer quantidade de núcleos de processamento estudada.

Exceto por Go e OpenMP, as outras implementações aparentam ter atingido um patamar de *speedup* com 4 processadores. A seguir estão descritas algumas das possíveis razões para esse fato.

Como previsto pela a Lei de Amdahl, apenas parte da tarefa executada pode se beneficiar do aumento do número de processadores. Dessa forma, qualquer implementação possui seu *speedup* limitado pela parte não paralelizável. É possível que o algoritmo tenha sido implementado de forma não otimizada em algumas das linguagens estudadas, no que diz respeito à minimizar os trechos não paralelizáveis, aumentando assim a percentagem do programa capaz de usufruir do ganho em processamento.

Outra possibilidade leva em conta o *overhead* causado pelo aumento do número de processos. Iniciar diversos processos durante a execução possui um custo de processamento, e esse custo pode variar de acordo com a implementação do paralelismo em cada uma das linguagens. Isso corrobora a hipótese anterior, de que os processos na implementação imperativa em Julia apresentam bastante *overhead* devido às estruturas do tipo `SharedArray`, bem como a literatura que afirma que as *go routines* custam pouco em termos de processamento [5].

3.1.3 Estudo do tipo de escalonamento de processos

Calculou-se a média do tempo de execução das 100 replicatas para as diversas formas de implementações paralelas em C/OpenMP: escalonamento estático e dinâmico com vários tamanhos de pacotes: 1, 100, 1000 e 2000. Destas médias foram subtraídas as médias dos tempos de geração das matrizes de entrada para as implementações consideradas. Os resultados obtidos, com as devidas propagações dos erros experimentais, podem ser vistos na Figura 5.

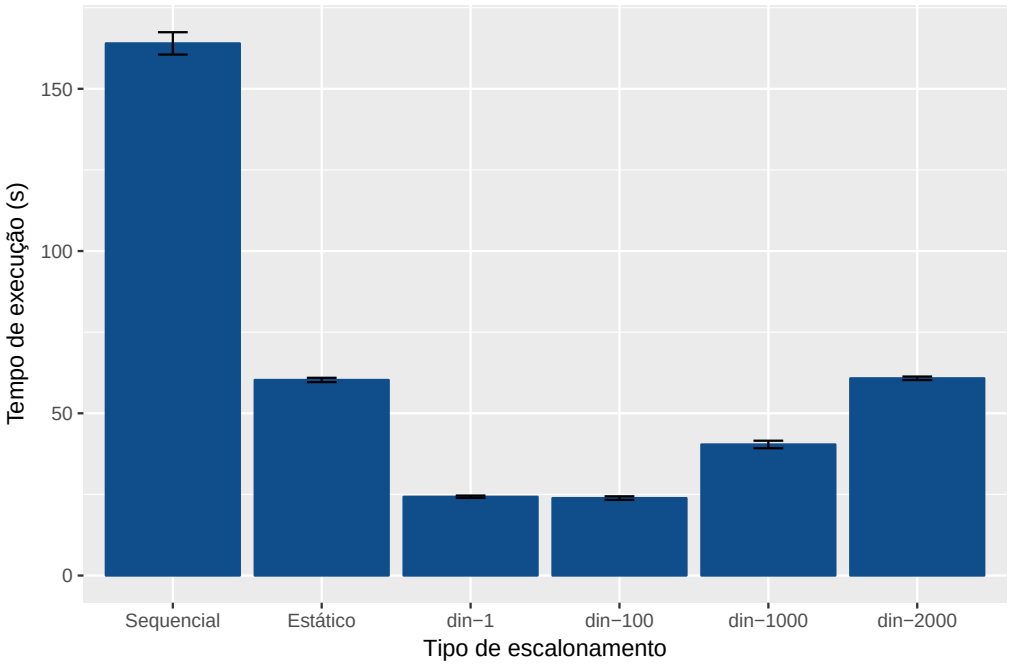

Figura 5 – Tempos médios de execução, em *s*, do algoritmo `mandel` implementado em C/OpenMP em modo paralelo para os diferentes tipos de escalonamento: estático, dinâmico com *chunk size* = 1 (din-1), dinâmico com *chunk size* = 100 (din-100), dinâmico com *chunk size* = 1000 (din-1000) e dinâmico com *chunk size* = 2000 (din-2000). O resultado obtido para a execução sequencial foi indicado para fins de comparação.

Os resultados indicam que o escalonamento estático apresentou menor desempenho em relação ao escalonamento dinâmico padrão (com *chunk size* = 1). Este comportamento é esperado para situações em que os processos devem lidar com diferentes *workloads* em cada uma das iterações. Isso ocorre porque o processo que lida com o *workload* menor finaliza a sua tarefa mais rapidamente e acaba por ficar ocioso por um tempo, enquanto os que lidam com maiores *workloads* finalizam a execução de suas tarefas. No modo dinâmico, assim que um processo termina de realizar sua tarefa, já recebe outra do escalonador, minimizando o tempo ocioso. Pode-se ressaltar que esse dinamismo depende de ações tomadas em tempo de execução, o que traz consigo um *overhead* maior do que na situação de escalonamento estático, cuja distribuição das tarefas já é realizada em tempo de compilação. Nos casos em que os *workloads* realizados por cada processo são parecidos, o escalonamento estático resolve o problema com *overhead* mínimo [46].

Analisando-se o problema em questão, a obtenção do conjunto de Mandelbrot, verificamos que ele se enquadra no primeiro caso: devido às características do problema, cada iteração do laço externo (que varre os elementos da matriz de entrada) lida com um número diferente de iterações dentro da função `mandelbrotorbit`, que é responsável pela verificação de divergência ou convergência dos valores. Valores do plano complexo em que as iterações divergem mais rápido proporcionam menor *workload* para o processo. Caso

não ocorra divergência, o número de iterações é maior (com um máximo de 100 iterações, conforme descrito na seção 1).

Nota-se que o escalonamento dinâmico com *chunk size* = 1 desempenha aproximadamente 2,5 vezes mais rápido do que o escalonamento estático. À medida que aumentamos o tamanho do *chunk*, ou seja, o tamanho do pacote de tarefas dados de uma só vez ao processo, o tempo de execução vai se aproximando àquele obtido pelo escalonamento estático. Na situação de escalonamento dinâmico com *chunk size* = 2000 o tempo de execução é igual ao obtido com o escalonamento estático (aproximadamente 60 s). Isso ocorre porque as situações são, de fato, muito parecidas. Considerando-se que a matriz de entrada apresenta 14000 linhas, o escalonamento estático divide igualmente o *workload* para os 8 processos, de modo que cada um deles lida com as tarefas geradas por 1750 linhas. O escalonamento dinâmico com *chunk size* = 2000 oferece um *workload* muito próximo para cada um dos processos.

3.2 Tamanho do código

Considerando-se os códigos das implementações do algoritmo (disponíveis na seção A.1 do apêndice), pode-se verificar que, em todos os casos, o número de linhas de código necessárias para incluir o paralelismo a partir da versão sequencial foi pequeno.

O código em Go tem a característica de ser mais extenso do que os códigos em Julia e Python, mesmo levando em conta apenas a implementação sequencial. Para se obter o paralelismo a partir da versão sequencial em Go, foram necessárias 10 linhas extras. Duas referentes a importações de bibliotecas, 1 relativa à recuperação do número de processos a partir de uma variável de ambiente e 7 relativas à introdução das *go routines* no laço mais externo e seus controles de finalização e espera.

O código implementado de forma imperativa, em Julia, ganhou 3 linhas em relação à sua versão sequencial. Duas delas com o objetivo de disponibilizar algumas das variáveis para todos os processos envolvidos, e a terceira para indicar que o laço externo deveria ser realizado em paralelo. Já o código implementado de forma funcional, também em Julia, ganhou apenas as duas linhas responsáveis por disponibilizar as variáveis para todos os processos, já que o restante do paralelismo foi obtido a partir da substituição de um `map` por um `pmap`, ambos ocupando apenas uma linha do código.

O código Python ganhou apenas duas linhas: uma relativa à importação de biblioteca `multiprocessing` e outra referente à definição do *pool* de processos. O paralelismo, como em Julia funcional, foi obtido substituindo-se um `map` por um `p.map`, ambos ocupando apenas uma linha do código.

O mesmo pode ser dito o código em C/OpenMP. Apesar de ser o maior de todos

em sua versão sequencial, ganhou apenas duas linhas com sua paralelização, uma referente à importação da biblioteca `omp` e a outra referente à diretiva de paralelismo.

A partir destes resultados e dos resultados obtidos na seção anterior a respeito dos tempos de execução, pode-se concluir que, para o tipo do problema estudado (perfeitamente paralelo, de intensa utilização de CPU), um pequeno esforço em termos de código pode resultar em ganhos bastante relevantes do ponto de vista de tempo de execução.

3.3 Ameaças à validade dos experimentos

É importante notar que antes de o algoritmo `mandel` ser executado, são geradas as matrizes de entrada. As implementações das funções geradoras dessas matrizes variam entre as linguagens e podem ser uma fonte razoável de viés nos experimentos. Para amenizar esse efeito, os tempos de execução das funções geradoras das entradas foram computados e os tempos dos experimentos foram corrigidos, com as devidas propagações dos erros. Mesmo assim, os tempos de geração das matrizes não foram computados exatamente no mesmo momento em que os experimentos completos foram executados, de modo que o estado do computador em termos de disponibilidade de recursos não era o mesmo, podendo provocar variações entre ambas as execuções. Apesar das diferenças entre os tempos de geração das matrizes para as diferentes linguagens, a ordem dos tempos de execução totais não é alterada com a correção.

Uma forma de se reduzir esse viés seria iniciar os experimentos com as matrizes já construídas, que poderiam existir *hardcoded* nos programas. Isso resolveria o problema do tempo de geração dessas matrizes, mas traria outros problemas envolvendo as representações de ponto flutuante de cada uma das células da matriz, já que muitos deles seriam apenas uma aproximação do valor real gerado pelo espaço linear, impresso com um número finito de casas, além de dificultar o processo de experimentação, já que seriam matrizes enormes a serem escritas em uma estrutura de dados específica para cada uma das linguagens. Sendo assim, optou-se pela geração dos números em tempo de execução.

Outra fonte de viés pode surgir do diferente conhecimento que o autor apresenta em relação às linguagens utilizadas. Seu *background* em programação Python pode fazer com que o código em outras linguagens seja penalizado, seja pelo desconhecimento das outras linguagens, seja pela maior otimização do código em Python. Por esse motivo, procurou-se basear seus códigos em exemplos encontrados na literatura, extensamente discutidos, ao invés de criá-lo a partir do ponto zero. Além disso, os tempos de execução obtidos para cada uma das linguagens foram bastante diferentes, sendo necessário um viés muito grande para explicar tais variações se elas não fossem reais.

4 Conclusão

As diversas linguagens estudadas mostraram potencial na obtenção dos conjuntos de Mandelbrot para diferentes tamanhos de matrizes de entrada, de forma paralela. Apesar disso, verificou-se que as estruturas de dados utilizadas na implementação do paralelismo por cada uma das linguagens, bem como o tamanho do problema, podem trazer um *overhead* indesejado, de modo a paralelização nem sempre trará o melhor resultado.

O presente trabalho foi eficiente ao mostrar que, para as linguagens estudadas, pequenas adições ao código podem trazer um grande ganho em desempenho. Para a situação específica envolvendo C/OpenMP, executando paralelamente em 8 núcleos e escalonamento dinâmico de tarefas, com matriz de entrada de tamanho 14000×10000, verificou-se um desempenho quase 7x maior para uma adição de apenas 8 linhas de código em relação à execução em modo sequencial. Outras linguagens apresentaram ganhos razoáveis em desempenho, mesmo para menores quantidades de núcleos de processamento.

Os programas implementados em Julia, utilizando paradigma funcional, mostraram alto desempenho mesmo quando executados em modo sequencial. Verificou-se também um ganho razoável em desempenho ao serem paralelizados, com a adição de apenas 3 linhas de código. O desempenho comparável a C, a simplicidade do código e a existência de diversas bibliotecas otimizadas fazem de Julia a melhor escolha geral dentre as linguagens estudadas. No caso específico em que há necessidade de se iniciar muitos processos paralelos, Go desponta como a melhor opção devido à leveza de suas *go routines*. Não é a toa que Go é uma linguagem bastante utilizada em servidores web com altos índices de requisições [47].

Não é incomum a concepção de que o paralelismo deve ser deixado na mão de especialistas. Em geral, trabalhar com situações paralelas demandam cuidado extra para lidar com os possíveis efeitos colaterais envolvendo condições de corrida, situações de *deadlocks* e até mesmo *overheads* desnecessários, mas as implementações de paralelismo das linguagens mais atuais estão, a cada dia mais, tornando o paralelismo acessível mesmo aos programadores mais leigos. Espera-se mostrar, com este trabalho, que todos os programadores estão convidados a experimentar o paralelismo em sua linguagem de preferência e, possivelmente, trazê-lo para perto da sua rotina diária.

Apesar de o presente trabalho ter se restringido a uma situação perfeitamente paralela, mais simples de se lidar do que as diversas outras, situações análogas não são incomuns nas mais diversas áreas. Em Ciência de Dados, por exemplo, são diversos os momentos em que um determinado tratamento deve ser aplicados a *dataframes* com milhões ou até bilhões de linhas, e a vivência nessa área mostra que diversos softwares construídos para essa finalidade não exploram o potencial completo do hardware que o

programador têm disponível, uma vez que, hoje em dia, diversos computadores pessoais de médio porte possuem multiprocessadores com até 8 núcleos, e os sistemas de computação na nuvem, geralmente utilizados por empresas que lidam com *Big Data*, podem disponibilizar hardware sob medida ainda melhor, a um custo acessível.

5 Trabalhos futuros

O presente trabalho não teve a intenção de ser uma abordagem extensiva do paralelismo. Apesar de as situações perfeitamente paralelas surgirem em diversos campos do conhecimento, elas não representam a totalidade das situações que podem se beneficiar do paralelismo. O mesmo pode ser dito a respeito das situações de uso intenso de CPU. Novos estudos envolvendo outras situações de paralelismo, em contextos de uso intenso de entrada e saída, por exemplo, em conjunto com os realizados no presente trabalho, comporiam um panorama mais completo dos ganhos e das limitações do multiprocessamento.

O foco deste trabalho foram os sistemas de memória compartilhada. Algumas das linguagens estudadas possuem implementações de paralelismo adaptadas também a sistemas distribuídos. Os resultados nesse tipo de sistema poderiam ser bastante diferentes em relação aos obtidos aqui. Estudos em sistemas distribuídos poderiam ser realizados com a adição de outras linguagens que implementam modelos de paralelismo pouco eficientes em situações de memória compartilhada, tais como *Message passing*.

Referências

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, p. 114–117, 1965. Citado na página 15.

[2] K. Rupp, "40 years of microprocessor trend data." https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/. Acessado: 2017-11-24. Citado na página 15.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaupmann, 5th ed., 2012. Citado na página 15.

[4] H. Sutter, "The concurrency revolution." http://www.drdobbs.com/the-concurrency-revolution/184401916, 2005. Acessado: 2017-11-24. Citado na página 15.

[5] R. Pike, "Concurrency is not parallelism @ Heroku's Waza Conference." https://vimeo.com/49718712, 2012. Acessado: 2017-11-24. Citado 2 vezes nas páginas 15 e 42.

[6] V. Nagarajan, "Lectures on CS4/Parallel Architectures: Types of Parallelism. Institute for Computing Systems Architecture, University of Edinburgh." http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture02-types.pdf, 2016. Acessado: 2017-11-24. Citado na página 15.

[7] B. Barney, "Introduction to Parallel Computing: Parallel Programming Models." https://computing.llnl.gov/tutorials/parallel_comp/#Models, 2017. Acessado: 2017-11-24. Citado na página 16.

[8] Centro Nacional de Processamento de Alto Desempenho - UNICAMP, "Apostila de treinamento: Introdução ao OpenMP." https://www.cenapad.unicamp.br/servicos/treinamentos/apostilas/apostila_openmp.pdf, 2014. Acessado: 2017-11-24. Citado na página 16.

[9] TIOBE, "TIOBE Programming Community Index Definition." https://www.tiobe.com/tiobe-index/programming-languages-definition/, 2017. Acessado: 2017-11-24. Citado na página 16.

[10] D. Lemire, "Best programming language for high performance." https://lemire.me/blog/2017/01/16/best-programming-language-for-high-performance-january-2017/, 2017. Acessado: 2017-11-24. Citado na página 16.

[11] D. A. M. Trejo, "After all these years, the world is still powered by C programming." https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming, 2015. Acessado: 2017-11-24. Citado na página 16.

[12] K. Patel, "Why should you learn Go." https://medium.com/@kevalpatel2106/why-should-you-learn-go-f607681fad65, 2017. Acessado: 2017-11-24. Citado na página 16.

[13] J. Bezanson, S. Karpinski, V. Shah, and A. Edelman, "Why whe created Julia." https://julialang.org/blog/2012/02/why-we-created-julia, 2017. Acessado: 2017-11-24. Citado na página 16.

[14] P. Krill, ""new julia language seeks to be the c for scientists"." https://www.infoworld.com/article/2616709/application-development/new-julia-language-seeks-to-be-the-c-for-scientists.html, 2012. Acessado: 2017-11-24. Citado na página 17.

[15] "Julia Programming Language." https://julialang.org/. Acessado: 2017-11-24. Citado 3 vezes nas páginas 17, 24 e 40.

[16] S. D. D'Cunha, "How a new programming language created by four scientists now used by the world's biggest companies." https://www.forbes.com/sites/suparnadutt/2017/09/20/this-startup-created-a-new-programming-language-now-used-by-the-worlds-biggest-companies/#63686e977de2, 2017. Acessado: 2017-11-24. Citado na página 17.

[17] MITOpenCourseware, "Parallel Computing." https://ocw.mit.edu/courses/mathematics/18-337j-parallel-computing-fall-2011/, 2011. Acessado: 2017-11-24. Citado na página 17.

[18] B. McLean, "Julia: a programming language for scientific computing @ University of New South Wales." http://web.maths.unsw.edu.au/~mclean/talks/Julia_talk.pdf, 2014. Acessado: 2017-11-24. Citado na página 17.

[19] B. Poulson, "Julia for Data Scientists First Look @ Lynda.com." https://www.lynda.com/Julia-tutorials/Julia-Data-Scientists-First-Look/512735-2.html, 2016. Acessado: 2017-11-24. Citado na página 17.

[20] Z. Voulgaris, *Julia for Data Science*. Technics Publications, 1st ed., 2016. Citado na página 17.

[21] A. Joshi, *Julia for Data Science*. Packt Publishing, 2016. Citado na página 17.

[22] M. J. L. Research and Development, "Julialab." http://julia.mit.edu/. Acessado: 2017-11-24. Citado na página 17.

[23] P. Guo, "Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities @ Blog-CACM." https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext, 2014. Acessado: 2017-11-24. Citado na página 17.

[24] H. Koepke, "10 Reasons Python Rocks for Research (And a Few Reasons it Doesn't) @ University of Washington." https://www.stat.washington.edu/~hoytak/blog/whypython.html, 2010. Acessado: 2017-11-24. Citado na página 17.

[25] "GlobalInterpreterLock." https://wiki.python.org/moin/GlobalInterpreterLock. Acessado: 2017-11-24. Citado 2 vezes nas páginas 17 e 27.

[26] G. V. Wilson, R. B. Irvin, and A. Sukul, "Assessing and Comparing the Usability of Parallel Programming Systems," tech. rep., University Of Toronto, Department of Computer Science, 1995. Citado na página 17.

[27] R. G. Brown, "Amdahl's Law & Parallel Speedup." http://webhome.phy.duke.edu/~rgb/brahma/Resources/als/als/node3.html, 2000. Acessado: 2017-11-24. Citado na página 17.

[28] G. Wilson, "The Cowichan Problems." https://software-carpentry.org/blog/2010/06/the-cowichan-problems.html#bib-par-lang-survey, 2010. Acessado: 2017-11-24. Citado na página 17.

[29] S. Nanz, S. West, and K. S. da Silveira, "Examining the Expert Gap in Parallel Programming @ Europar." http://se.inf.ethz.ch/people/west/expert-gap-europar-2013.pdf, 2013. Acessado: 2017-11-24. Citado na página 17.

[30] S. Nanz, S. West, K. S. da Silveira, and B. Meyer, "Benchmarking Usability and Performance of Multicore Languages @ ACM / IEEE International Symposium on Empirical Software Engineering and Measurement." http://ieeexplore.ieee.org/document/6681351/, 2013. Acessado: 2017-11-24. Citado na página 17.

[31] E. W. Weisstein, "Mandelbrot Set @ Wolfram MathWorld." http://mathworld.wolfram.com/MandelbrotSet.html. Acessado: 2017-11-24. Citado na página 18.

[32] R. P. Munafo, "Escape Radius." http://mrob.com/pub/muency/escaperadius.html, 1997. Acessado: 2017-11-24. Citado na página 18.

[33] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Elsevier, 2012. Acessado: 2017-11-24. Citado na página 18.

[34] "Google Compute Engine." https://cloud.google.com/compute/. Acessado: 2017-11-15. Citado na página 21.

[35] "`diff` utility documentation." http://pubs.opengroup.org/onlinepubs/9699919799/. Acessado: 2017-11-24. Citado na página 22.

[36] "Numpy." http://www.numpy.org/. Acessado: 2017-11-24. Citado 2 vezes nas páginas 23 e 29.

[37] W. Kennedy, "Going go programming." https://www.goinggo.net/2014/01/concurrency-goroutines-and-gomaxprocs.html. Acessado: 2017-11-24. Citado na página 28.

[38] "Julia: Programação Paralela (documentação oficial)." https://docs.julialang.org/en/latest/manual/parallel-computing. Acessado: 2017-11-24. Citado na página 28.

[39] "Julia: Verificação de limites (documentação oficial)." https://docs.julialang.org/en/latest/devdocs/boundscheck/. Acessado: 2017-11-24. Citado na página 28.

[40] C. S. Bosma, "Distrust simplicity: Parallel mandelbrot in julia, C++, and OpenCL." http://distrustsimplicity.net/articles/mandelbrot-speed-comparison/#fnref4. Acessado: 2017-11-24. Citado na página 29.

[41] "Francesc campoy's github *mandelbrot* repository." https://github.com/campoy/mandelbrot. Acessado: 2017-11-24. Citado na página 29.

[42] "The MagPi Magazine: Multiprocessing with python." https://www.raspberrypi.org/magpi/multiprocessing-with-python/. Acessado: 2017-11-24. Citado na página 29.

[43] "Rosetta code: Mandelbrot set." https://rosettacode.org/wiki/Mandelbrot_set. Acessado: 2017-11-24. Citado na página 29.

[44] "What about warm up?@Appfolio Engineering." http://engineering.appfolio.com/appfolio-engineering/2017/5/2/what-about-warmup, 2017. Acessado: 2017-11-24. Citado na página 33.

[45] "Julia official performance tests." https://github.com/JuliaLang/julia/tree/master/test/perf/micro. Acessado: 2017-11-24. Citado na página 40.

[46] V. Eijkhout, "Parallel Programming in MPI and OpenMP: Loop Parallelism." http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html, 2016. Acessado: 2017-11-24. Citado na página 43.

[47] M. Castilho, "Handling 1 million requests per minute with Go." http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang/, 2015. Acessado: 2017-11-24. Citado na página 47.

A Informações complementares

A.1 Implementações do algoritmo

A.1.1 Go

A.1.1.1 Sequencial

```
package main

import "fmt"
import "flag"
import "math"
import "math/cmplx"
import "os"
import "strconv"

var rows, _ = strconv.Atoi(os.Getenv("INPUTMAT_ROWS"))
var columns, _ = strconv.Atoi(os.Getenv("INPUTMAT_COLS"))

func mandelbrotorbit(c complex128) int {
        z := cmplx.Pow(0+0i, 2) + c
        for i := 1; i <= 100; i++ {
                if cmplx.Abs(z) > 2 {
                        return i - 1
                }
                z = cmplx.Pow(z, 2) + c
        }
        return 100
}

func mandelbrot(inputmat [][]complex128) [][]int {
        outputmat := make([][]int, rows)
        for i := range outputmat {
                outputmat[i] = make([]int, columns)
        }
        for i := 0; i < rows; i++ {
                for j := 0; j < columns; j++ {
                        outputmat[i][j] = mandelbrotorbit(inputmat[i][j])
                }
        }
        return outputmat
}

func linspace(start float64, end float64, n int) []float64 {
        var linspace []float64
        linspace = append(linspace, start)
        distance := (end - start) / float64(n-1)
```

```
        next := start
        for math.Abs(next-end) > 0.000000000001 {
                next = next + distance
                linspace = append(linspace, next)
        }
        return linspace
}

func clinspace(start complex128, end complex128, m int, n int) [][]complex128 {
        realParts := linspace(real(start), real(end), m)
        imagLinspace := linspace(imag(start), imag(end), n)
        var cmplxParts []float64
        for im := range imagLinspace {
                cmplxParts = append(cmplxParts, imagLinspace[im])
        }

        var cmplxLinspace [][]complex128
        for r := range realParts {
                var cmplxRow []complex128
                for c := range cmplxParts {
                        cmplxRow = append(cmplxRow, complex(realParts[r], cmplxParts[c]))
                }
                cmplxLinspace = append(cmplxLinspace, cmplxRow)
        }
        return cmplxLinspace
}

func matrix_to_csv(matrix [][]int) {
        filename := fmt.Sprintf("output-go-seq-%d.csv", rows)
        file, _ := os.Create(filename)
        for i := 0; i < rows; i++ {
                for j := 0; j < columns; j++ {
                        if j != columns-1 {
                                fmt.Fprintf(file, "%d;", matrix[i][j])
                        } else {
                                fmt.Fprintln(file, matrix[i][j])
                        }
                }
        }
}

func main() {
        export_flag := flag.Bool("export", false, "Prints output matrix in CSV format")
        flag.Parse()
        outputmat := mandelbrot(clinspace(-2.5-1.25i, 1.0+1.25i, rows, columns))
        if *export_flag {
                matrix_to_csv(outputmat)
        }
}
```

A.1.1.2 Paralela

```
package main

import "fmt"
import "flag"
import "math"
import "math/cmplx"
import "os"
import "strconv"
import "sync"
import "runtime"

var rows, _ = strconv.Atoi(os.Getenv("INPUTMAT_ROWS"))
var columns, _ = strconv.Atoi(os.Getenv("INPUTMAT_COLS"))
var num_procs, _ = strconv.Atoi(os.Getenv("NUM_THREADS"))

func mandelbrotorbit(c complex128) int {
        z := complex(0, 0) + c
        for i := 1; i <= 100; i++ {
                if cmplx.Abs(z) > 2 {
                        return i - 1
                }
                z = cmplx.Pow(z, 2) + c
        }
        return 100
}

func mandelbrot(inputmat [][]complex128) [][]int {
        runtime.GOMAXPROCS(num_procs)
        var wg sync.WaitGroup
        wg.Add(rows)

        outputmat := make([][]int, rows)
        for i := range outputmat {
                outputmat[i] = make([]int, columns)
        }

        for i := 0; i < rows; i++ {
                go func(i int) {
                        for j := 0; j < columns; j++ {
                                outputmat[i][j] = mandelbrotorbit(inputmat[i][j])
                        }
                        wg.Done()
                }(i)
        }
        wg.Wait()
        return outputmat
}

func linspace(start float64, end float64, n int) []float64 {
        var linspace []float64
```

```
        linspace = append(linspace, start)
        distance := (end - start) / float64(n-1)
        next := start
        for math.Abs(next-end) > 0.000000000001 {
                next = next + distance
                linspace = append(linspace, next)
        }
        return linspace
}

func clinspace(start complex128, end complex128, m int, n int) [][]complex128 {
        realParts := linspace(real(start), real(end), m)
        imagLinspace := linspace(imag(start), imag(end), n)
        var cmplxParts []float64
        for im := range imagLinspace {
                cmplxParts = append(cmplxParts, imagLinspace[im])
        }

        var cmplxLinspace [][]complex128
        for r := range realParts {
                var cmplxRow []complex128
                for c := range cmplxParts {
                        cmplxRow = append(cmplxRow, complex(realParts[r], cmplxParts[c]))
                }
                cmplxLinspace = append(cmplxLinspace, cmplxRow)
        }
        return cmplxLinspace
}

func matrix_to_csv(matrix [][]int) {
        filename := fmt.Sprintf("output-go-par-%d.csv", rows)
        file, _ := os.Create(filename)
        for i := 0; i < rows; i++ {
                for j := 0; j < columns; j++ {
                        if j != columns-1 {
                                fmt.Fprintf(file, "%d;", matrix[i][j])
                        } else {
                                fmt.Fprintln(file, matrix[i][j])
                        }
                }
        }
}

func main() {
        export_flag := flag.Bool("export", false, "Prints output matrix in CSV format")
        flag.Parse()
        outputmat := mandelbrot(clinspace(-2.5-1.25i, 1.0+1.25i, rows, columns))
        if *export_flag {
                matrix_to_csv(outputmat)
        }
}
```

A.1.2 JuliaFuncional

A.1.2.1 Sequencial

```
function getenv(var::AbstractString)
    val = ccall((:getenv, "libc"), Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    unsafe_string(val)
end

rows = parse(Int32, getenv("INPUTMAT_ROWS"))
columns = parse(Int32, getenv("INPUTMAT_COLS"))

function mandelbrotorbit(c)
    z = Complex128(0,0) + c
    for k = 1:100
        if abs(z) > 2
            return k-1
        end
        z = z^2 + c
    end
    return 100
end

function linspace(start::Float64, finish::Float64, n::Integer)
    linspace = [start]
    distance = (finish - start) / Float64(n-1)
    next = start
    while abs(next - finish) > 0.000000000001
        next = next + distance
        linspace = append!(linspace, next)
    end
    return linspace
end

X = linspace(-2.5, 1.0, rows)
Y = linspace(-1.25, 1.25, columns)

function mandelbrot(x::Float64)
    Z = [complex(x,y) for y in Y]
    return map(mandelbrotorbit, Z)
end

N = map(mandelbrot, X)
if "-export" in ARGS
    writedlm("output-jl-fun-seq-$rows.csv", N, ';')
end
```

A.1.2.2 Paralela

```
@everywhere begin

function getenv(var::AbstractString)
    val = ccall((:getenv, "libc"), Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    unsafe_string(val)
end

rows = parse(Int32, getenv("INPUTMAT_ROWS"))
columns = parse(Int32, getenv("INPUTMAT_COLS"))

function mandelbrotorbit(c)
    z = Complex128(0,0) + c
    for k = 1:100
        if abs(z) > 2
            return k-1
        end
        z = z^2 + c
    end
    return 100
end

function linspace(start::Float64, finish::Float64, n::Integer)
    linspace = [start]
    distance = (finish - start) / Float64(n-1)
    next = start
    while abs(next - finish) > 0.000000000001
        next = next + distance
        linspace = append!(linspace, next)
    end
    return linspace
end

function mandelbrot(x::Float64)
    Z = [complex(x,y) for y in Y]
    return map(mandelbrotorbit, Z)
end

X = linspace(-2.5, 1.0, rows)
Y = linspace(-1.25, 1.25, columns)

end

N = pmap(mandelbrot, X)
if "-export" in ARGS
    writedlm("output-jl-fun-par-$rows.csv", N, ';')
end
```

A.1.3 JuliaImperativo

A.1.3.1 Sequencial

```
@everywhere begin

function getenv(var::AbstractString)
    val = ccall((:getenv, "libc"), Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    unsafe_string(val)
end

rows = parse(Int32, getenv("INPUTMAT_ROWS"))
columns = parse(Int32, getenv("INPUTMAT_COLS"))

function mandelbrotorbit(c)
    z = Complex128(0,0) + c
    for k = 1:100
        if abs(z) > 2
            return k-1
        end
        z = z^2 + c
    end
    return 100
end

function linspace(start::Float64, finish::Float64, n::Integer)
    linspace = [start]
    distance = (finish - start) / Float64(n-1)
    next = start
    while abs(next - finish) > 0.000000000001
        next = next + distance
        linspace = append!(linspace, next)
    end
    return linspace
end

end

X = linspace(-2.5, 1.0, rows)
Y = linspace(-1.25, 1.25, columns)

inputmat = convert(SharedArray, [Complex128(x,y) for x in X, y in Y])
outputmat = SharedArray{Int8}(size(inputmat))

function mandelbrot(inputmat::SharedArray{Complex128,2}, outputmat::SharedArray{Int8,2})
    @sync @parallel for k=1:size(inputmat,2)
        for j=1:size(inputmat,1)
            @inbounds outputmat[j,k] = mandelbrotorbit(inputmat[j,k])
        end
```

```
    end
end

mandelbrot(inputmat, outputmat)
if "-export" in ARGS
    writedlm("output-jl-imp-par-$rows.csv", outputmat, ';')
end
```

A.1.3.2 Paralela

```
function getenv(var::AbstractString)
    val = ccall((:getenv, "libc"), Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    unsafe_string(val)
end

rows = parse(Int32, getenv("INPUTMAT_ROWS"))
columns = parse(Int32, getenv("INPUTMAT_COLS"))

function mandelbrotorbit(c)
    z = Complex128(0,0) + c
    for k = 1:100
        if abs(z) > 2
            return k-1
        end
        z = z^2 + c
    end
    return 100
end

function linspace(start::Float64, finish::Float64, n::Integer)
    linspace = [start]
    distance = (finish - start) / Float64(n-1)
    next = start
    while abs(next - finish) > 0.000000000001
        next = next + distance
        linspace = append!(linspace, next)
    end
    return linspace
end

X = linspace(-2.5, 1.0, rows)
Y = linspace(-1.25, 1.25, columns)

inputmat = [Complex128(x,y) for x in X, y in Y]
outputmat = Array{Integer}(size(inputmat))

function mandelbrot(inputmat::Array{Complex128,2}, outputmat::Array{Integer,2})
    for k=1:size(inputmat, 2)
        for j=1:size(inputmat, 1)
```

```
            @inbounds outputmat[j,k] = mandelbrotorbit(inputmat[j,k])
        end
    end
end

mandelbrot(inputmat, outputmat)
if "-export" in ARGS
    writedlm("output-jl-imp-seq-$rows.csv", outputmat, ';')
end
```

A.1.4 Python

A.1.4.1 Sequencial

```
import csv
from matplotlib import pyplot
import numpy as np
import os
import sys

xmin, xmax = np.float64(-2.5), np.float64(1.0)
ymin, ymax = np.float64(-1.25), np.float64(1.25)
nx = int(os.environ.get('INPUTMAT_ROWS'))
ny = int(os.environ.get('INPUTMAT_COLS'))
maxiter = 100


def mandelbrotorbit(c):
    z = complex(0, 0) + c
    for n in range(maxiter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return maxiter


def linspace(start, end, n):
    lspace = [start]
    distance = (end - start) / np.float64(n - 1)
    next = start
    while np.abs(next - end) > 0.000000000001:
        next += distance
        lspace.append(next)
    return lspace


X = linspace(xmin, xmax, nx)
Y = linspace(ymin, ymax, ny)


def mandelbrot(x):
```

```
    Z = [complex(x, y) for y in Y]
    return list(map(mandelbrotorbit, Z))

N = list(map(mandelbrot, X))

def matrix_to_csv(m):
    filename = 'output-py-seq-{}.csv'.format(nx)
    with open(filename, 'w') as f:
        writer = csv.writer(f, delimiter=';', lineterminator='\n')
        writer.writerows(N)

if '-export' in sys.argv:
    matrix_to_csv(N)

if '-fractal' in sys.argv:
    pyplot.imshow(N)
    pyplot.savefig('mandel.png')
```

A.1.4.2 Paralela

```
import csv
from matplotlib import pyplot
from multiprocessing import Pool
import numpy as np
import os
import sys

xmin, xmax = np.float64(-2.5), np.float64(1.0)
ymin, ymax = np.float64(-1.25), np.float64(1.25)
nx = int(os.environ.get('INPUTMAT_ROWS'))
ny = int(os.environ.get('INPUTMAT_COLS'))
procs = int(os.environ.get('NUM_THREADS'))
maxiter = 100


def mandelbrotorbit(c):
    z = complex(0, 0) + c
    for n in range(maxiter):
        if abs(z) > 2:
            return n
        z = z*z + c
    return maxiter


def linspace(start, end, n):
    lspace = [start]
    distance = (end - start) / np.float64(n - 1)
    next = start
    while np.abs(next - end) > 0.000000000001:
        next += distance
        lspace.append(next)
    return lspace
```

```
X = linspace(xmin, xmax, nx)
Y = linspace(ymin, ymax, ny)


def mandelbrot(x):
    Z = [complex(x, y) for y in Y]
    return list(map(mandelbrotorbit, Z))


p = Pool(procs)
N = p.map(mandelbrot, X)

def matrix_to_csv(m):
    filename = 'output-py-par-{}.csv'.format(nx)
    with open(filename, 'w') as f:
        writer = csv.writer(f, delimiter=';', lineterminator='\n')
        writer.writerows(N)

if '-export' in sys.argv:
    matrix_to_csv(N)

if '-fractal' in sys.argv:
    pyplot.imshow(N)
    pyplot.savefig('mandel.png')
```

A.1.5 C/OpenMP

A.1.5.1 Módulo complex

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "complex.h"

Complex addComplex(Complex a, Complex b) {
    struct Complex sum;
    sum.re = a.re + b.re;
    sum.im = a.im + b.im;
    return sum;
}

Complex multComplex(Complex a, Complex b) {
    struct Complex mult;
    mult.re = a.re * b.re - a.im * b.im;
    mult.im = a.re * b.im + a.im * b.re;
    return mult;
}

double absComplex(Complex c) {
```

```
    return sqrt(c.re * c.re + c.im * c.im);
}

void printComplex(Complex a) {
    if (a.im < 0) {
        printf("%f-%fi\n", a.re, fabs(a.im));
    } else {
        printf("%f+%fi\n", a.re, a.im);
    }
}
```

A.1.5.2 Módulo linspace

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "complex.h"
#include "linspace.h"

double* linspace(double start, double end, int n) {
    double *lnsp = malloc(n * sizeof(double));
    lnsp[0] = start;
    double distance = (end - start) / (double)(n-1);
    double next = start;
    for (int i = 1; i < n; i++) {
        next = next + distance;
        lnsp[i] = next;
    }
    return lnsp;
}

Complex** clinspace(Complex start, Complex end, int m, int n) {
    double *reLnsp = linspace(start.re, end.re, m);
    double *imLnsp = linspace(start.im, end.im, n);

    Complex** cmplxLnsp = malloc(m * sizeof(Complex*));
    for(int i = 0; i < m; i++) {
        cmplxLnsp[i] = malloc(n * sizeof(Complex));
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            Complex c;
            c.re = reLnsp[i];
            c.im = imLnsp[j];
            cmplxLnsp[i][j] = c;
        }
    }
    free(reLnsp);
    free(imLnsp);
    return cmplxLnsp;
}
```

```
void freeClinspace(Complex **clsp, int m) {
    for(int i = 0; i < m; i++) {
        free(clsp[i]);
    }
    free(clsp);
}
```

A.1.5.3 Sequencial

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "complex.h"
#include "linspace.h"

int rows;
int columns;

void setGlobalVariables() {
    rows = atoi(getenv("INPUTMAT_ROWS"));
    columns = atoi(getenv("INPUTMAT_COLS"));
}

void printMatrix(int **m) {
    for(int i = 0; i < rows; i++){
        for(int j = 0; j < columns; j++){
            printf("%d ", m[i][j]);
        }
        printf("\n");
    }
}

void matrixToCsv(int **m) {
    FILE *fp;
    char filename[25];
    sprintf(filename, "output-omp-seq-%d.csv", rows);
    fp = fopen(filename, "w");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            if (j != columns-1) {
                fprintf(fp, "%d;", m[i][j]);
            } else {
                fprintf(fp, "%d\n", m[i][j]);
            }
        }
    }
    fclose(fp);
}

int mandelbrotorbit(Complex c) {
    Complex z;
```

```
    z.re = 0.0; z.im = 0.0;
    z = addComplex(multComplex(z, z), c);
    for (int i = 1; i <= 100; i++) {
        if (absComplex(z) > 2) {
            return i - 1;
        }
        z = addComplex(multComplex(z, z), c);
    }
    return 100;
}

int** mandelbrot(Complex** inputmat) {
    int **outputmat = malloc(rows * sizeof(int*));
    for (int i = 0; i < rows; i++){
        outputmat[i] = malloc(columns * sizeof(int));
    }

    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < columns; j++) {
            outputmat[i][j] = mandelbrotorbit(inputmat[i][j]);
        }
    }
    return outputmat;
}

int main(int argc, char *argv[]) {

    setGlobalVariables();
    Complex start, end;
    start.re = -2.5; start.im = -1.25;
    end.re = 1.0; end.im = 1.25;
    Complex **inputmat = clinspace(start, end, rows, columns);

    int **outputmat = mandelbrot(inputmat);
    if(argc > 1){
        if(!strcmp(argv[1], "-export")){
                matrixToCsv(outputmat);
        }
    }
    return 0;
}
```

A.1.5.4 Paralela

Esta versão implementa o escalonamento estático das tarefas entre os processos:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <omp.h>
#include "complex.h"
```

```c
#include "linspace.h"

int rows;
int columns;
int nthreads;

void setGlobalVariables() {
    rows = atoi(getenv("INPUTMAT_ROWS"));
    columns = atoi(getenv("INPUTMAT_COLS"));
    nthreads = atoi(getenv("NUM_THREADS"));
}

void printMatrix(int **m) {
    for(int i = 0; i < rows; i++){
        for(int j = 0; j < columns; j++){
            printf("%d ", m[i][j]);
        }
        printf("\n");
    }
}

void matrixToCsv(int **m) {
    FILE *fp;
    char filename[25];
    sprintf(filename, "output-omp-par-%d.csv", rows);
    fp = fopen(filename, "w");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            if (j != columns-1) {
                fprintf(fp, "%d;", m[i][j]);
            } else {
                fprintf(fp, "%d\n", m[i][j]);
            }
        }
    }
    fclose(fp);
}

int mandelbrotorbit(Complex c) {
    Complex z;
    z.re = 0.0; z.im = 0.0;
    z = addComplex(multComplex(z, z), c);
    for (int i = 1; i <= 100; i++) {
        if (absComplex(z) > 2) {
            return i - 1;
        }
        z = addComplex(multComplex(z, z), c);
    }
    return 100;
}

int** mandelbrot(Complex** inputmat) {
```

```
    int **outputmat = malloc(rows * sizeof(int*));
    for (int i = 0; i < rows; i++){
        outputmat[i] = malloc(columns * sizeof(int));
    }

    #pragma omp parallel for num_threads(nthreads)
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < columns; j++) {
            outputmat[i][j] = mandelbrotorbit(inputmat[i][j]);
        }
    }
    return outputmat;
}

int main(int argc, char *argv[]) {

    setGlobalVariables();
    Complex start, end;
    start.re = -2.5; start.im = -1.25;
    end.re = 1.0; end.im = 1.25;
    Complex **inputmat = clinspace(start, end, rows, columns);

    int **outputmat = mandelbrot(inputmat);
    if(argc > 1){
        if(!strcmp(argv[1], "-export")){
                matrixToCsv(outputmat);
        }
    }
    return 0;
}
```

Para utilizar o escalonamento dinâmico, basta substituir a linha

```
#pragma omp parallel for num_threads(nthreads)
```

por

```
#pragma omp parallel for schedule(dynamic) num_threads(nthreads)
```

Pode-se, também, definir o tamanho dos pacotes de tarefas (*chunks*) atribuídos a cada um dos processos:

```
#pragma omp parallel for schedule(dynamic, chunk_size) num_threads(nthreads)
```

O tamanho do *chunk* deve ser definido no topo do arquivo, diretamente no código (ex. `int chunk_size = 100`), ou importado de uma variável de ambiente. Caso não seja definido nenhum valor de $chunk\_size$ para o escalonamento dinâmico, ele é executado com seu valor padrão: $chunk\_size = 1$.

A.2 Códigos auxiliares

Foram utilizados códigos auxiliares em Shell Script, Python e R ao longo de todo o estudo. A seguir estão disponíveis alguns deles na íntegra. Todos eles podem ser acessados a partir do repositório deste estudo no GitHub: https://github.com/holypriest/undergrad-thesis/

A.2.1 Tempos de execução

Com base nos arquivos texto dos experimentos realizados, o *script* Python a seguir gera os resultados dos experimentos, já com a correção pelo tempo de geração das matrizes de entrada e com a devida propagação dos erros. A saída é formatada para uso em textos LaTeX. Estes valores são utilizados na geração da Tabela 1 e das Figuras 4 e 5.

```
from collections import namedtuple
import numpy as np
import os
import re


def get_experiment_file_names():
    filenames = []
    for f in os.listdir("."):
        if ('mandel' in f) and (f != 'dist_mandel.r'):
            filenames.append(f)
    return filenames


def parse_experiment_file_name(filename):
    Experiment = namedtuple(
        'Experiment',
        'label system lang paradigm execution sched chunk num_procs size'
    )
    label = filename.split('.')[0]
    info = filename.split('-')
    system = re.search(r'\[(.*)\]', info[0]).group(1)
    lang = info[1]
    paradigm = info[2]
    execution = info[3]
    sched = info[4]
    chunk = info[5]
    num_procs = info[6]
    size = info[7].split('.')[0]
    return Experiment(
        label=label,
        system=system,
        lang=lang,
        paradigm=paradigm,
        execution=execution,
```

```
        sched=sched,
        chunk=chunk,
        num_procs=num_procs,
        size=size
    )


def get_gen_file_name(exp):
    return '[{}]gen-{}-{}.txt'.format(
        exp.system,
        exp.lang,
        exp.size
    )


def convert_to_seconds(timestring):
    if len(timestring) <= 6:
        return float(timestring)
    else:
        minutes, seconds = timestring.split(':')
        return float(minutes) * 60 + float(seconds)


def latex_result(x):
    return ('${} \pm {}$'.format(x.val, x.err)).replace('.', ',')


def extract_times(filename):
    text = open(filename, 'r').read()
    times = re.findall(r'cpu (.*) total', text)
    stimes = list(map(convert_to_seconds, times))
    return stimes


def get_mean_with_error(list_of_times):
    mean = round(np.mean(np.array(list_of_times)), 2)
    stdev = round(np.std(np.array(list_of_times)), 2)
    Expnum = namedtuple('Expnum', 'val err')
    return Expnum(val=mean, err=stdev)


def diff(expnum1, expnum2):
    Expnum = namedtuple('Expnum', 'val err')
    return Expnum(
            val=(expnum1.val - expnum2.val),
            err=np.sqrt(np.power(expnum1.err, 2) + np.power(expnum2.err, 2))
    )


def main():
    filenames = get_experiment_file_names()
    for f in filenames:
```

```
        exp = parse_experiment_file_name(f)
        gen = get_gen_file_name(exp)
        exp_times = extract_times(f)
        gen_times = extract_times(gen)
        mean_exp = get_mean_with_error(exp_times)
        mean_gen = get_mean_with_error(gen_times)
        result = diff(mean_exp, mean_gen)
        print('{}: {}'.format(exp.label, latex_result(result)))

if __name__ == '__main__':
    main()
```

O funcionamento correto desse *script* demanda a sistematização dos nomes dos arquivos de experimentos. Segue um exemplo:

```
[gce]mandel-omp-imp-par-dynamic-c2000-8x-14000.txt
```

As informações contidas nele são:

- `gce`: sistema (mbp ou gce);
- `omp`: linguagem (go, jl, py, omp);
- `imp`: paradigma (fun ou imp);
- `par`: execução (seq ou par);
- `dynamic`: tipo de escalonamento (dynamic ou static);
- `c2000`: *chunk size* (c1, c100, c1000 ou c2000);
- `8x`: número de núcleos de processamento (2x, 4x, 8x);
- `14000`: tamanho da matriz de entrada (3500, 7000, 14000)

Para implementações em que nem todas as informações acima fazem sentido, elas foram representadas por `0` no nome do arquivo. Segue um exemplo:

```
[gce]mandel-jl-fun-seq-0-0-0-14000.txt
```

Para gerar a Figura 4, foi utilizado o seguinte *script* em linguagem R:

```
#!/usr/local/bin/Rscript

pdf('speedup.pdf', width = 8.84, height = 4.42)
```

```
df = read.csv("speedup.csv", sep=";")
library(ggplot2)
ggplot(df, aes(x=as.factor(procs), y=mean, fill=lang)) +
    geom_bar(position=position_dodge(), stat="identity", colour='black') +
    geom_errorbar(aes(ymin=mean-sd, ymax=mean+sd), width=.2,position=position_dodge(.9)) +
    scale_fill_brewer(palette = "Blues") +
    labs(x = "Número de processadores", y = "Speedup", fill="Linguagem")

invisible(dev.off())
```

O arquivo `speedup.csv` traz os dados extraídos do *script* anterior, no formato indicado a seguir:

```
lang;mean;sd;procs
Go;1.0;0;1
Go;2.016113495052106;0.07102654170053539;2
Go;4.017099982551038;0.1489641307261648;4
Go;5.499104263704765;0.19084536862648308;8
juliaFuncional;1.0;0;1
juliaFuncional;1.9829059829059827;0.23755351962337626;2
juliaFuncional;3.4746543778801846;0.41416984516775884;4
juliaFuncional;3.586206896551724;0.4281905147828979;8
juliaImperativo;1.0;0;1
juliaImperativo;1.2094763092269327;0.04237069245845915;2
juliaImperativo;1.3397790055248617;0.05411611075858205;4
juliaImperativo;1.5594855305466238;0.08373921227983037;8
Python;1.0;0;1
Python;2.093349779550703;0.1010957959741525;2
Python;4.732142857142857;0.22762847200224567;4
Python;4.523363955994103;0.19500668527648699;8
C/OpenMP;1.0;0;1
C/OpenMP;2.1711497890295357;0.046934766436090604;2
C/OpenMP;4.37809093326243;0.09189713446329201;4
C/OpenMP;6.78171334431631;0.17394222556969716;8
```

Por fim, para a geração da Figura 5 foi utilizado o *script* a seguir, em linguagem R.

```
#!/usr/local/bin/Rscript

pdf('sched.pdf', width = 6.66, height = 4.42)

df = read.csv("scheduler.csv", sep=";")
library(ggplot2)
```

```
sched = ggplot(df, aes(x=as.factor(sched), y=mean)) +
  geom_bar(position=position_dodge(), stat="identity", colour='black') +
  geom_errorbar(aes(ymin=mean-sd, ymax=mean+sd), width=.2,position=position_dodge(.9)) +
  scale_fill_brewer(palette = "Blues") +
  labs(x = "Tipo de escalonamento", y = "Tempo de execução (s)")
print(sched)

invisible(dev.off())
```

O arquivo `scheduler.csv` traz os dados gerados pelo primeiro script desta seção, no formato indicado a seguir:

```
sched;mean;sd
Sequencial;164.66;3.4401308114663314
Estático;60.269999999999996;0.6606814663663573
din-1;24.28;0.36124783736376886
din-100;23.860000000000003;0.5508175741568165
din-1000;40.39;1.160387866189577
din-2000;60.78;0.5408326913195984
```

A.2.2 Distribuição dos tempos de execução

Com a finalidade de se entender melhor a distribuição dos tempos de execução, em ambos os sistemas, foram gerados histogramas e gráficos do tipo quantil-quantil. Para isso, o primeiro passo consistiu em extrair os tempos de execução dos arquivos `.txt` dos experimentos e convertê-los para segundos. Esses tempos foram, então, exportados para arquivos .csv. Esse conjunto de tarefas pode ser realizado pelo *script* Python a seguir:

```
import csv
import sys
import re
import os

def convert_to_seconds(timestring):
    if len(timestring) <= 6:
        return timestring
    else:
        minutes, seconds = timestring.split(':')
        return str(float(minutes) * 60 + float(seconds))

for root, _, files in os.walk('.'):
    for f in files:
        if f[-3:] == 'txt':
            text = open(os.path.join(root, f), 'r').read()
            times = re.findall(r'cpu (.*) total', text)
            stimes = list(map(convert_to_seconds, times))
```

```
            with open(os.path.join(root, f) + '.csv', 'w') as csvfile:
                writer = csv.writer(csvfile)
                for stime in stimes[2:]:
                    writer.writerow([stime])
```

A partir dos arquivos `.csv` obtidos, os histogramas e gráficos quantil-quantil podem ser gerados a partir do *script* R a seguir:

```
library(stringr)

for (f in list.files(pattern='csv')) {

    filename = str_sub(f, 1, -9)
    times <- read.csv(
        paste(filename, '.txt.csv', sep=''),
        header=FALSE
    )

    pdf(
        paste(filename, '.pdf', sep=''),
        width = 8,
        height = 4
    )

    par(mfrow=c(1,2))
    hist(
        times$V1,
        breaks=10,
        border="white",
        col="dodgerblue4",
        main='',
        xlab="Tempo de execução (s)",
        ylab="Frequência"
    )
    library(car)
    qqPlot(
        times$V1,
        col.lines="dodgerblue4",
        lwd=1,
        xlab="Quantis teóricos (dist. normal)",
        ylab="Tempo de execução (s)"
    )

    invisible(dev.off())
```

Esses dois *scripts*, em conjunto, são capazes de gerar gráficos da distribuição dos tempos de execução de todos os experimentos, inclusive dos estudos do tempo de geração das matrizes de entrada. Essas figuras auxiliaram na decisão pela média como medida de tendência central a ser utilizada no presente trabalho.