

miniEP3

April 25, 2020

1 MiniEP3 - Aproximação de Integrais usando Médias e Amostragem Uniforme, com a biblioteca *pthread*s

1.1 Entrega do miniEP3

Preencha o nome dos 5 membros do seu grupo na tabela abaixo:

| Nome | NUSP |
|-------------------------------|----------|
| Carlos Eduardo Leal de Castro | 11921804 |
| Felipe de Lima Peressim | 11823558 |
| Luis Ricardo Manrique | 5779294 |
| Rafael Fernandes Alencar | 9344730 |
| José Luiz Maciel Pimenta | 11896720 |

Apenas um dos membros deverá entregar um **arquivo .tar**, até o dia **29 de Abril**, com:

1. Este **arquivo .ipynb**, com as soluções do miniEP3 feitas pelo grupo
 - Os gráficos e análises devem poder ser produzidos
2. O **código C escrito** (arquivo `monte_carlo.c`)
 - Deve compilar e executar **sem erros**
3. Os arquivos `Makefile`, e `Project.toml`
4. Um **arquivo .csv** com os resultados das medições feitas neste miniEP
 - Entregue os dados de cada repetição, sem processamento (não calcule a média e CI)

1.2 Configuração do Ambiente

Como no miniEP1&2, a primeira tarefa é instalar e configurar o ambiente.

1.2.1 Compilador C

Neste miniEP também vamos usar a linguagem C e a biblioteca *pthread*s. Vocês vão precisar de acesso a um sistema Linux com o compilador GCC e a biblioteca *pthread*s. Caso não consiga

instalar o GCC ou tenha dificuldades para acessar um sistema Linux, entre em contato pelo fórum do *Edisciplinas*.

1.2.2 Julia, Jupyter, IJulia

Pule essa etapa se já configurou o ambiente Julia no miniEP1&2. Para fazer o miniEP, vocês vão precisar:

- [Instalar o Jupyter Notebook](#)
- Instalar Julia 1.3:
 - [Baixando o binário](#)
 - [ou usando seu gerenciador de pacotes](#)
- Instalar o pacote *IJulia*:
 - Inicie o interpretador Julia
 - Digite `] add IJulia` e pressione `<ENTER>`

Depois disso, vocês vão conseguir iniciar o arquivo `.ipynb` do miniEP.

1.2.3 Pacotes Julia para o miniEP

Os pacotes necessários para o miniEP estão listados no arquivo `Project.toml`, mas vocês podem instalar e atualizar os pacotes rodando a célula abaixo:

```
[1]: ] up

Updating registry at `~/.julia/registries/General`

Updating git-repo
`https://github.com/JuliaRegistries/General.git`

Updating `~/Desktop/tmp_montecarlo/Project.toml`
[no changes]
Updating `~/Desktop/tmp_montecarlo/Manifest.toml`
[no changes]
```

Verifique o status dos pacotes, e se há algum problema, com o comando:

```
[2]: ] st

Status `~/Desktop/tmp_montecarlo/Project.toml`
[336ed68f] CSV v0.6.1
[a93c6f00] DataFrames v0.20.2
[31c24e10] Distributions v0.23.2
[7073ff75] IJulia v1.21.2
[8314cec4] PGFPlotsX v1.2.6
```

[1a8c2f83] Query v0.12.2
[f3b207a7] StatsPlots v0.14.5

1.3 Integração por Método de Monte Carlo

O objetivo deste miniEP é promover o estudo de programação paralela usando a biblioteca *pthread*s. Vamos utilizar uma versão sequencial em C do código em Julia do miniEP1&2. A tarefa deste miniEP será completar a implementação em C, escrever um programa paralelo usando a biblioteca *pthread*s, e analisar o desempenho do programa escrito com diferentes números de threads.

Lembrando o que vimos no miniEP1&2, a intuição por trás do método de Monte Carlo é que a integral de uma função f pode ser estimada pela média do valor de f num conjunto suficientemente grande de pontos obtidos a partir de uma distribuição uniforme. Mais formalmente, para um conjunto de pontos x_1, \dots, x_N uniformemente amostrados num intervalo $[a, b]$, a integral de f no intervalo $[a, b]$ pode ser aproximada por:

$$\int_a^b f(x)dx \approx \mathbb{E} \left[(b-a) \frac{1}{N} \sum_{i=1}^N f(x_i) \right]$$

Para uma representação gráfica da intuição, e para a prova dessa aproximação, veja [esta página](#).

A Integração por Método de Monte Carlo é um problema [embarçosamente paralelo](#), isto é, podemos executar todas as iterações envolvidas **em paralelo**, sem nos preocupar com conflitos de acesso à memória. Apesar disso, vamos ver neste miniEP que não basta apenas aumentar a quantidade de recursos computacionais para ganhar desempenho.

1.3.1 Função Alvo

Vamos estimar a integral da seguinte função:

$$f_1(x) = \frac{2}{\sqrt{1-x^2}}$$

A integral da função f_1 , é [dada por](#):

$$\int_0^1 f_1(x)dx = \int_0^1 \frac{2}{\sqrt{1-x^2}}dx = \pi$$

1.4 Exercício 1: Implementação Sequencial em C

Lembre-se do código em Julia:

```
[3]: using Distributions

function monte_carlo_integrate(f, interval, samples = 100)
    xs = rand(Uniform(interval[1], interval[2]), samples)
```

```

# Using for loops:
# accumulator = 0
#
# for x in xs
#     accumulator += f(x)
# end
#
# return accumulator / samples

# Using vectorized function application:
return sum(f.(xs)) / samples
end

```

[3]: `monte_carlo_integrate` (generic function with 2 methods)

O primeiro exercício consiste em completar a implementação sequencial em C, fornecida no arquivo `monte_carlo.c`. Baseiem-se na versão Julia, e usem seus editores de código preferidos para modificar e escrever código em C. Se vocês ainda não têm preferência de editor de código, procurem experimentar editores como o Emacs e o Vim. São programas antigos e com uma curva de aprendizado um pouco íngreme, mas são muito poderosos e os esforços se pagam com juros. Esta pode ser uma chance de aprender a usá-los.

Vocês devem escrever a função de assinatura:

```
long double monte_carlo_integrate(long double (*f)(long double), long double *samples, int size)
```

Na assinatura acima, o parâmetro `f` é um ponteiro para uma função que recebe um `long double` e devolve um `long double`, `samples` é um ponteiro para um vetor de `long double` onde vamos guardar os valores de `f` que queremos avaliar, e `size` é o tamanho de `samples`.

A primeira parte do exercício é ler e compreender o funcionamento do código em C fornecido no enunciado. É um exercício interessante para ver como uma linguagem de alto nível, como Julia, abstrai camadas conceituais e facilita a prototipagem de código complexo ao custo de esconder detalhes de implementação.

1.4.1 Exercício 1a)

Vocês podem compilar e executar o programa `monte_carlo.c` através deste notebook acessando o *modo shell* do interpretador Julia com a tecla `;`:

[4]: `; make debug`

```
gcc monte_carlo.c -o monte_carlo -Wall -lpthread -lm -DDEBUG=1
```

Depois, podemos rodar o programa compilado com:

[5]: `; ./monte_carlo`

```
usage: ./monte_carlo SAMPLES FUNCTION_ID N_THREADS
```

Seguindo as instruções de uso, temos:

```
[6]: ; ./monte_carlo 10000000 0 1
```

```
Running on: [debug mode]
Samples: [10000000]
Function id: [0]
Threads: [1]
Array size on memory: [0.16GB]
Running sequential version
3.1389556000020498, [0.326149, clock], [0.326157, clock_gettime], [0.326158,
gettimeofday]
```

Como compilamos no modo *debug*, veremos muitas mensagens extras, que coloquei no programa pra ajudar na implementação. Para compilar no modo normal, use o comando **make** sem argumentos. Note que o programa ainda não faz nada além de alocar memória, e que isso já leva algum tempo.

Note que escolhi deixar a alocação de memória e o cálculo das amostra a avaliar fora da função `monte_carlo_integrate` na versão C. A geração das amostras aleatórias foi feita com a função `rand()` e com uma função que mapeia um intervalo a outro.

Descreva abaixo como o tempo de execução é calculado e impresso pelo programa `monte_carlo`. Por que vocês acham que eu escolhi colocar as medições de tempo onde coloquei?

Resposta O tempo de execução é calculado com auxílio da função ‘clock’ declarada no arquivo cabeçalho ‘time.h’. Essa função mede o número de tiques do clock do processador por segundo de modo a fornecer o tempo do processador. Para determinar o tempo em segundos decorrido da execução de instruções computadas em uma determinada rotina, a quantidade de tiques antes e depois são armazenadas, a diferença entre elas é computada para fornecer o tempo do processador gasto para realizar tal computação, por último, para se obter o tempo em segundos o tempo de processador é dividido pela constante ‘CLOCKS_PER_SEC’, também declarada no arquivo cabeçalho supracitado.

Similarmente, algumas rotinas do arquivo cabeçalho ‘sys/time.h’ foram adicionadas posteriormente, sendo que as tais fornecem medições do tempo e fornecem a quantidade de segundos decorridos da execução de uma determinada rotina sem que seja necessário realizar calculos extras, como é necessário ao se usar a função ‘clock’.

As medições de tempo foram colocadas de modo a se mensurar apenas o tempo gasto pela função ‘monte_carlo_integrate’. Pois, se fossem colocadas em outra parte do programa, acabariam por mensurar o tempo gasto por outras instruções. Como o objetivo se concentra em estudar as vantagens de paralelização do método de monte carlo para estimar integrais, não faria sentido mensurar o tempo de outras rotinas que não fazem parte deste processo.

1.4.2 Exercício 1b)

Agora, a sua tarefa é implementar a função `monte_carlo_integrate`.

Quando terminar, rode as células abaixo. A saída deve conter a estimativa para o valor de π e o tempo de execução, produzidos pelo programa `monte_carlo`. Escolha um número de amostras adequado à quantidade de memória disponível em seu computador.

Lembre-se de **entregar também o arquivo .c** com suas modificações.

```
[7]: ; make debug
```

```
gcc monte_carlo.c -o monte_carlo -Wall -lpthread -lm -DDEBUG=1
```

```
[8]: ; ./monte_carlo 100000000 0 1
```

```
Running on: [debug mode]
Samples: [100000000]
Function id: [0]
Threads: [1]
Array size on memory: [1.60GB]
Running sequential version
3.1398076493876083, [3.298796, clock], [3.300645, clock_gettime], [3.300646,
gettimeofday]
```

1.5 Exercício 2: Implementação Paralela com *pthread*s

Este exercício é mais complexo. Vocês devem escrever a função de assinatura:

```
void *monte_carlo_integrate_thread(void *args)
```

Na assinatura acima, o parâmetro `args` é um ponteiro para uma estrutura de dados contendo dados para cada thread. Para implementar essa estrutura de dados, vocês devem decidir quais informações cada thread deve receber. Algumas ideias:

- Um ponteiro para a função a ser avaliada
- Um ponteiro para o vetor de amostras gerado
- Um inteiro com o id da thread

De acordo com a sua estratégia de implementação paralela, vocês vão precisar incluir informações diferentes. Algo de importante está faltando na lista acima. Como as threads poderiam armazenar seus resultados?

Precisamos escolher uma dentre as várias formas de implementar esse algoritmo paralelo. Pensei em algumas possibilidades:

1. Método “Criando threads dinamicamente”:

- Lançar n threads, cada uma com 1 (ou m ?) unidades de trabalho a fazer
- Usar variáveis de condição ou joins para sinalizar fim de trabalho
- Lançar novas threads conforme threads terminarem
- Usar join para finalizar

2. Método “Divisão Dinâmica do Trabalho”:

- Lançar n threads, cada uma com 1 (ou m ?) unidades de trabalho a fazer

- Cada thread busca por trabalho disponível a fazer quando acabar o trabalho dado
- Usar join para finalizar

3. Método “Divisão Estática do Trabalho”:

- Lançar n threads, cada uma com $1/n$ unidades de trabalho a fazer
- Usar join para finalizar

Usando seus conhecimentos sobre *pthread*s e sobre a execução de programas em geral, responda e explique:

1. Qual desses métodos é o mais difícil de implementar? E o mais fácil?
2. Qual método atingiria o menor tempo de execução?

1.5.1 Resposta

1. O método mais difícil de implementar é o método indicado pelo item 2, pelo fato de cada thread ter que fazer uma verificação do trabalho disponível. Neste caso, a verificação deverá ser atômica. O método mais fácil é o indicado pelo item 3 porque não há nenhum controle a se fazer entre as threads.
2. O método que aparentemente pode atingir o menor tempo de execução é o indicado pelo item 2, pois este método é o que melhor faz o balanceamento de carga e os threads estarão sempre trabalhando.

Escolha um desses, ou um quarto método que preferir, para fazer a implementação paralela do miniEP3. Vocês vão precisar implementar toda a estrutura de suporte à execução das threads.

Quando terminar, rode as células abaixo. A saída deve conter a estimativa para o valor de π e o tempo de execução, produzidos pelo programa `monte_carlo`. Escolha um número de amostras e de threads adequado à quantidade de memória e aos núcleos de processamento disponíveis em seu computador.

Lembre-se de **entregar também o arquivo .c** com suas modificações.

```
[9]: ; make debug
```

```
gcc monte_carlo.c -o monte_carlo -Wall -lpthread -lm -DDEBUG=1
```

```
[10]: ; ./monte_carlo 100000000 0 32
```

```
Running on: [debug mode]
Samples: [100000000]
Function id: [0]
Threads: [32]
Array size on memory: [1.60GB]
Running parallel version
3.1414768942916269, [5.584519, clock], [2.705744, clock_gettime], [2.705744,
gettimeofday]
```

1.6 Exercício 3: Análise de Desempenho

Agora, vamos medir o desempenho do programa que vocês implementaram neste trabalho. Vamos usar funções em Julia, adaptadas do miniEP1&2, e gerar gráficos do tempo de execução e da estimativa da integral para diferentes números de threads.

1.6.1 Funções Úteis

A função abaixo recebe parâmetros `size`, com tamanho da amostra, `f`, com a id da função a estimar, e `threads`, com o número de threads do programa paralelo. A função executa o programa `monte_carlo` com os parâmetros dados e devolve um `DataFrame` com os resultados.

```
[11]: using DataFrames, Query, StatsPlots, Statistics

function measure_monte_carlo(size, f, threads)
    results = parse.(Float64,
        split(chomp(read(`./monte_carlo $size $f $threads`, String)), ", "))

    return DataFrame(size = size,
        f = f,
        threads = threads,
        estimate = results[1],
        duration = results[2])
end
```

```
[11]: measure_monte_carlo (generic function with 1 method)
```

A função `run_experiments` recebe os mesmos parâmetros `size`, `f`, e `threads`, e um parâmetro adicional `repetitions`, com o número de repetições de cada experimento com um dado número de threads. A função devolve um `DataFrame` com todos os experimentos.

```
[12]: function run_experiments(size, f, threads, repetitions)
    run(`make`)

    results = DataFrame(size = Int[],
        f = Int[],
        threads = Int[],
        estimate = Float64[],
        duration = Float64[])

    for t in threads
        for r in 1:repetitions
            append!(results,
                measure_monte_carlo(size, f, t))
        end
    end
end
```



```

    return results
end

```

[12]: run_experiments (generic function with 1 method)

A função `parse_results` recebe um `DataFrame` de resultados, produzido pela função `run_experiments`, e um parâmetro `target_integral`, com o valor da integral a estimar. A função devolve um `DataFrame` com a média e o intervalo de confiança da média a 95% das estimativas e dos tempos de execução, agrupados por número de threads.

```

[13]: function parse_results(results, target_integral)
    parsed_results = results |>
        @groupby(_.threads) |>
        @map({threads = key(_),
              mean_estimate = mean(_.estimate),
              ci_estimate = 1.96 * std(_.estimate),
              mean_duration = mean(_.duration),
              ci_duration = 1.96 * std(_.duration),
              target = target_integral}) |>
        DataFrame

    return parsed_results
end

```

[13]: parse_results (generic function with 1 method)

1.6.2 Exercício 3a)

Realize os experimentos em sua máquina rodando a célula abaixo. Ajuste os valores para a sua máquina. **Não faça menos de 5 repetições.**

```

[14]: size = 200000000
f = 0
threads = [2 ^ x for x in 0:11]
repetitions = 30

results = run_experiments(size, f, threads, repetitions)
parsed_results = parse_results(results, pi)

```

```
gcc monte_carlo.c -o monte_carlo -Wall -lpthread -lm
```

[14]:

| | threads | mean_estimate | ci_estimate | mean_duration | ci_duration | target |
|----|---------|---------------|-------------|---------------|-------------|-------------|
| | Int64 | Float64 | Float64 | Float64 | Float64 | Irration... |
| 1 | 1 | 3.14176 | 0.000896404 | 6.25341 | 0.0331995 | |
| 2 | 2 | 3.14158 | 0.000870465 | 7.24222 | 0.416042 | |
| 3 | 4 | 3.14156 | 0.000949496 | 6.56509 | 0.265377 | |
| 4 | 8 | 3.14139 | 0.000959438 | 6.01221 | 0.0946995 | |
| 5 | 16 | 3.14153 | 0.000836918 | 5.63254 | 0.19371 | |
| 6 | 32 | 3.14158 | 0.000693871 | 5.34058 | 0.136407 | |
| 7 | 64 | 3.14161 | 0.000875557 | 5.15034 | 0.0656987 | |
| 8 | 128 | 3.14144 | 0.000834068 | 5.09652 | 0.09541 | |
| 9 | 256 | 3.14157 | 0.00070867 | 5.0769 | 0.0404673 | |
| 10 | 512 | 3.14173 | 0.000710703 | 5.07763 | 0.0618524 | |
| 11 | 1024 | 3.14152 | 0.000744385 | 5.10198 | 0.0868164 | |
| 12 | 2048 | 3.1416 | 0.000584525 | 5.13375 | 0.0956064 | |

Agora, escreva uma função em Julia chamada `save_csv_results`, que recebe um `DataFrame` e um nome de arquivo, e escreve o `DataFrame` em disco, no formato `.csv`, com o nome passado no argumento.

Utilize a biblioteca `CSV`, já instalada no ambiente deste notebook.

```
[15]: using CSV
function save_csv_results(Dataframe, target_name)
    CSV.write(target_name, Dataframe)
end

save_csv_results(results, "results.csv")
save_csv_results(parsed_results, "parsed_results.csv")
```

```
[15]: "parsed_results.csv"
```

Escreva uma função em Julia chamada `read_csv_results`, que recebe um nome de arquivo e lê o arquivo correspondente, devolvendo um `DataFrame`.

```
[16]: using CSV
function read_csv_results(target_name)
    return CSV.read(target_name)
end
```

```
[16]: read_csv_results (generic function with 1 method)
```

Salve o `DataFrame` `results` em disco. Vocês devem entregar o `.csv` também.

1.6.3 Exercício 3b)

Como vocês já se familiarizaram um pouco com funções para geração de gráficos no `miniEP1&2`, explique o que faz a função `plot_results` abaixo. Ela é uma generalização das funções usadas no `miniEP` anterior. Para ajudar, você pode modificar e usar as chamadas de função no Exercício 3c).

1.6.4 Resposta

A função `plot_results` irá construir os gráficos 2D, utilizando os dados fornecidos nos atributos `x` e `y`, onde `x` representará o eixo horizontal do gráfico e o `y` representará o eixo vertical do gráfico. A função tem como atributos, além dos dados `x` e `y`:

- `target_label`: este atributo indicará o rótulo do objetivo que, neste caso, o algoritmo pretende alcançar (``pi'`)
- `series_label`: este atributo indicará o rótulo da legenda do gráfico.
- `hline`: este atributo indicará a linha tracejada do objetivo pretendido (``pi'`). Neste caso, se este atributo não for indicado na função, ele indicará outros atributos para o plot, de acordo com o `if` da célula abaixo, como atribuir uma linha horizontal, definir a escala do gráfico, definir o label e os pontos do eixo horizontal, o formato da linha e sua espessura.
- `yerror`: este atributo indicará o intervalo de confiança nos gráficos. Neste caso, se este atributo não for indicado, ele indicará diferentes atributos para o plot parecidos com os indicados para o atributo `hline`, além de indicar a transparência dos pontos utilizando o atributo `alpha`.
- `max_thread_power`: indica a quantidade máxima de threads a serem indicados nos plots.

Por fim, a função retorna o plot na variável `p`.

Além desses atributos, outros podem ser incluídos na função `plot`, como `ambient`, `lightposition`, entre outros.

<https://makie.juliaplots.org/stable/plot-attributes.html>

```
[17]: pgfplotsx()

function plot_results(x, y, target_label, series_label; hline = [], yerror = []
↳ [], max_thread_power = 10)
    if yerror != []
        p = scatter(x,
            y,
            xaxis = :log2,
            xlabel = "Threads",
            xticks = [2 ^ x for x in 0:max_thread_power],
            yerror = yerror,
            alpha = 0.6,
            labels = series_label,
            legend = :topright)
    else
        p = scatter(x,
            y,
            xaxis = :log2,
            xlabel = "Threads",
            xticks = [2 ^ x for x in 0:max_thread_power],
```

```

        alpha = 0.6,
        labels = series_label,
        legend = :topright)
    end

    if hline != []
        plot!(x,
            hline,
            xaxis = :log2,
            xlabel = "Threads",
            xticks = [2 ^ x for x in 0:max_thread_power],
            labels = target_label,
            line = :dash,
            width = 2.0)
    end

    return p
end

```

[17]: plot_results (generic function with 1 method)

1.6.5 Exercício 3c)

1. Rode as células na seção abaixo e gere os gráficos. Entregue o notebook com os gráficos gerados.
2. Descreva o comportamento do tempo de execução conforme aumentamos o número de threads, em termos da média e do intervalo de confiança. Nos próximos EPs, vamos aprender a fazer uma *regressão linear* que explique os dados observados usando *coeficientes*.
3. Por que você acha que o tempo de execução aumenta conforme aumentamos as threads? Era isso que você esperava?

Responda na célula abaixo:

1.6.6 Resposta

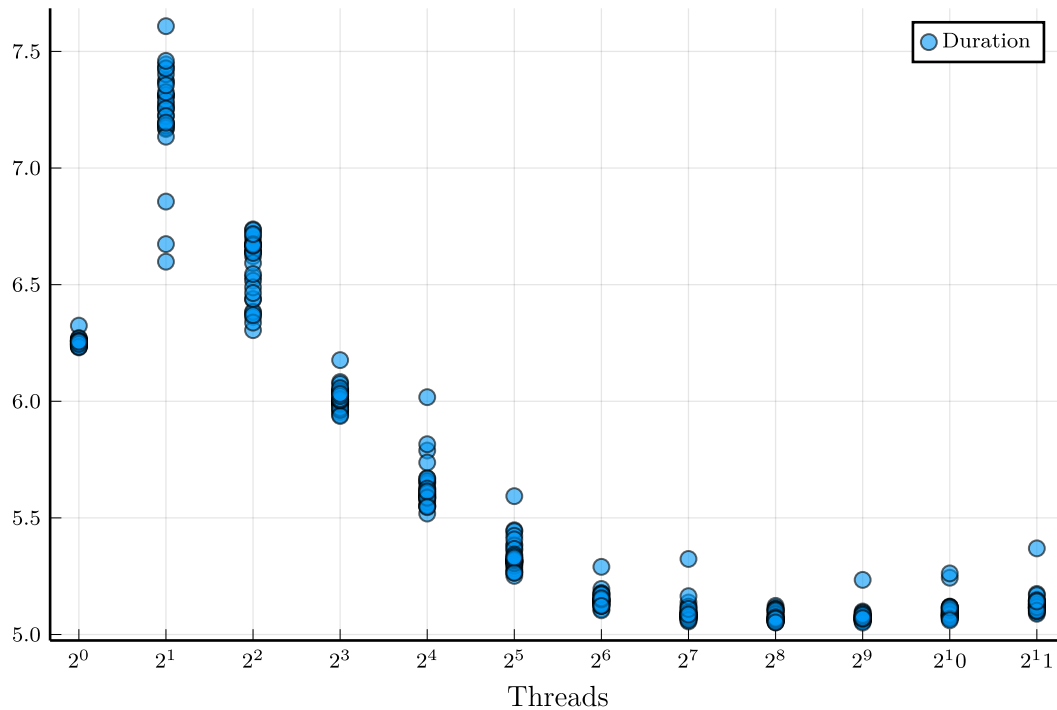
1. Ok!
2. Quanto maior o número de threads, o escalonador de curto prazo escalonará mais threads do programa. Além disso, como esse problema pode ser resolvido de forma independente, uma thread não depende das outras. Assim, mais threads estarão na CPU, resolvendo o problema proposto, obtendo, assim, um menor intervalo de confiança entre os tempos medidos. Por exemplo: se estivermos apenas uma thread, essa thread pode ser escalonada assim que ela for criada ou pode demorar a entrar, o que resultará um tempo muito diferente. Quando tem-se muitas threads, isso fica mais difícil de acontecer.

3. Inicialmente há uma melhora, entretanto, a partir de um determinado momento vai gerar um overhead, pois haverá muitas threads disputando pelos núcleos do processador, perdendo assim, a performance da divisão de trabalho.

Gerando Gráficos

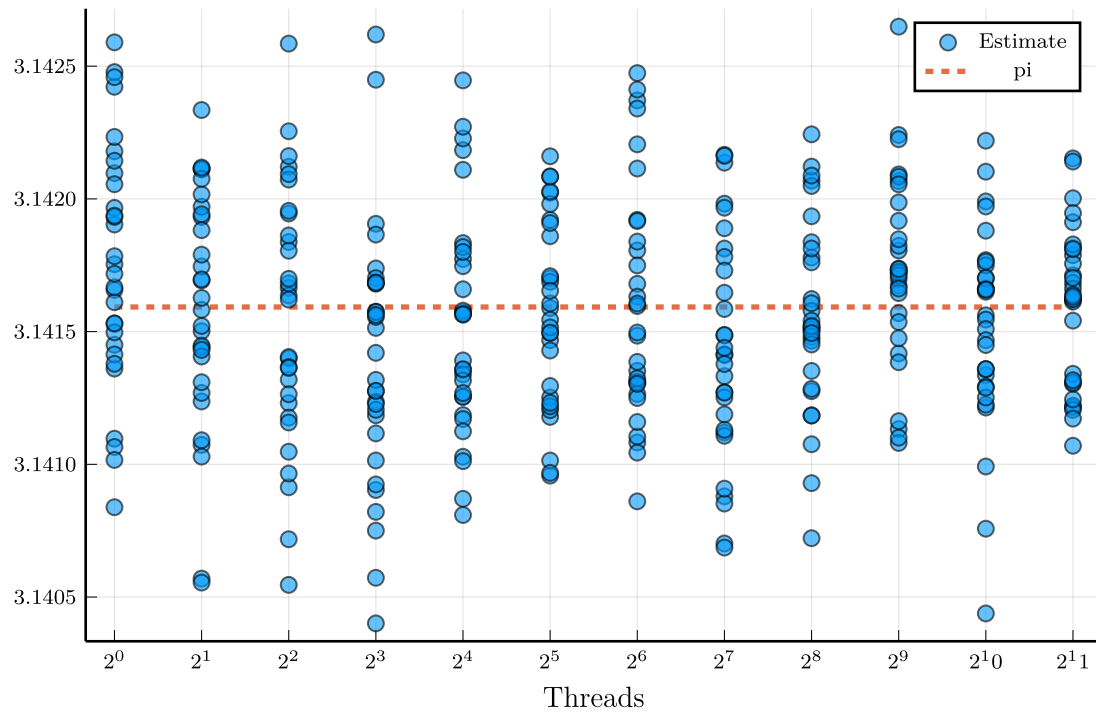
```
[18]: plot_results(results.threads,
                 results.duration,
                 "pi",
                 "Duration",
                 max_thread_power = 11)
```

[18]:



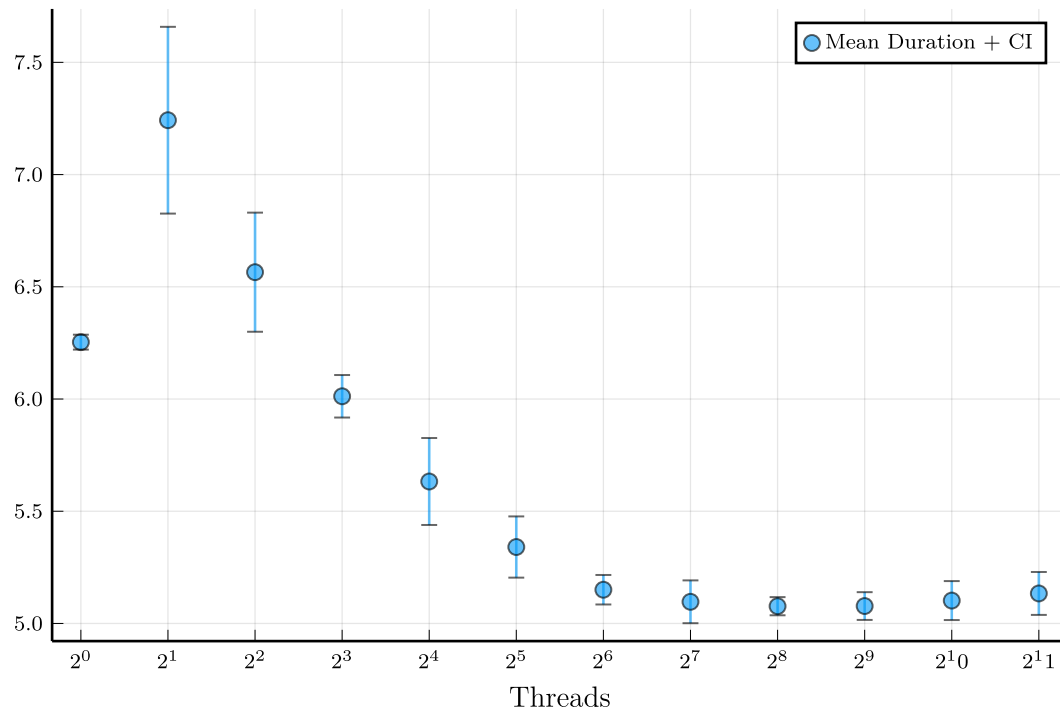
```
[19]: plot_results(results.threads,
                 results.estimate,
                 "pi",
                 "Estimate",
                 hline = [pi for i in 1:nrow(results)],
                 max_thread_power = 11)
```

[19]:



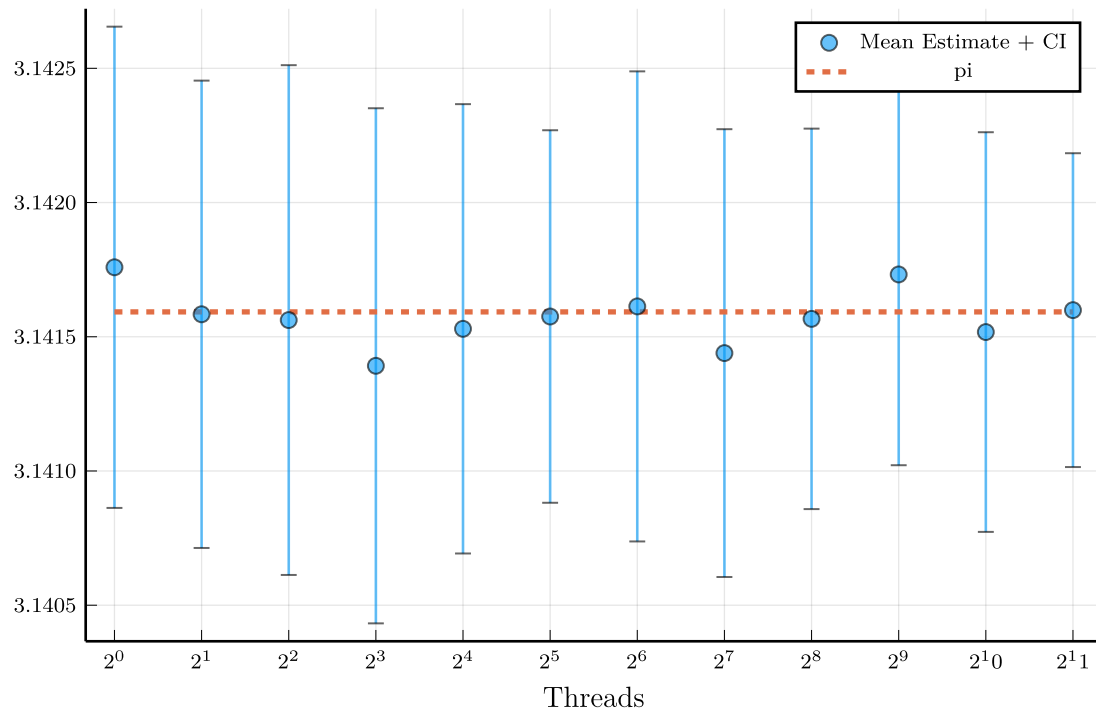
```
[20]: plot_results(parsed_results.threads,
    parsed_results.mean_duration,
    "pi",
    "Mean Duration + CI",
    yerror = parsed_results.ci_duration,
    max_thread_power = 11)
```

[20]:



```
[21]: plot_results(parsed_results.threads,
    parsed_results.mean_estimate,
    "pi",
    "Mean Estimate + CI",
    hline = [pi for i in 1:nrow(parsed_results)],
    yerror = parsed_results.ci_estimate,
    max_thread_power = 11)
```

[21]:



[]: