RelatorioEP2

July 12, 2020

1 MAC0219 - Programação Concorrente, Paralela e Distribuída

2 EP2: CUDA & OpenMPI

Nome	NUSP
Caio Andrade	9797232
Caio Fontes	10692061
Eduardo Laurentino	8988212
Thiago Teixeira	10736987
Washington Meireles	10737157

Neste relatório, temos como objetivo explicar como foram feitas as nossas implementações do cálculo do Conjunto de Mandelbrot para as diferentes metodologias de paralelização, explicar como foram realizados os experimentos com essas implementações e analisarmos os resultados obtidos.

Os arquivos-fonte de todas as implementações foram fornecidos junto a este relatório, bem como o arquivo makefile que possibilita todas as compilações. Além disso, fornecemos ainda um script em Pyhton (`run_experiments.py`) que, quando executado, realiza uma rodada de experimentos para todas as versões por nós implementadas seguindo as configurações indicadas no enunciado: 15 **repetições** da geração de uma imagem de **tamanho** 4096 da região **Triple Spiral Valley**. Tal script gera, ao final, arquivos .csv com os resultados dos experimentos.

Durante este relatório, utilizamos a função abaixo para importar as informações dos arquivoc csv que obtimos com os nossos experimentos, executados na Rede Linux. Todos estes arquivos foram também fornecidos.

```
In [ ]: using DataFrames, Query, StatsPlots, Statistics, CSV, Plots
```

```
In [2]: function read_csv_results(filename)
            return CSV.read(filename)
        end
```

```
Out[2]: read_csv_results (generic function with 1 method)
```

3 1. Implementação em CUDA

A implementação em CUDA foi feita tendo como base a versão sequencial, a qual adaptamos para as necessidades da paralelização com computação heterogênea, como é o caso aqui. Nesse sentido, transformamos o método `void compute_mandelbrot()` num método global, pois é este o **Kernel** do programa - isto é, a função que conecta o *Host* (CPU) com o *Device* (GPU). Além disso, deixamos os métodos que são utilizados somente durante o processamento dos pixels (à saber, `int mandelbrot()` e `void update_rgb_buffer()`) como exclusivas para o *device*. O restante das conexões entre as duas camadas de memória, *Host* e *Device* , foi feito manipulando adequadamente a alocação e tráfego de memória quando e onde necessário.

A divisão de tarefas é feita de maneira estática à partir das dimensões dos *Blocos* e do *Grid* no qual estes estão inseridos. Para que pudessemos determinar experimentalmente os melhores valores para essas dimensões, o programa solicita como argumento (além das coordenadas da região do conjunto e o tamanho da imagem) as dimensões (x, y) dos blocos. À partir desses valores, determina-se também as dimensões do *Grid* de forma a garantir que a divisão das tarefas contemple toda a imagem a ser gerada. No caso da nossa implementação, isso significa atribuir ao *Grid* dimensões $(\frac{image\_size}{x}, \frac{image\_size}{y})$.

Dessa forma, fica estabelecido que cada pixel (i_x, i_y) da imagem será tratado por uma *thread* de identidade determinada pela sua posição num determinado bloco do grid de acordo com as especificações abaixo:

$$i_x = blockIdx.x * blockDim.x + threadIdx.xi_y = blockIdx.y * blockDim.y + threadIdx.y$$

É importante observar uma condição de execução crucial relativa a essa implementação: devemos ter, necessariamente, $x * y \leq 1024$. Isso pois ao determinarmos as dimensões dos blocos como sendo (x, y), isso significa que a quantidade de *threads* em cada bloco é igual a $x * y$. O limite de 1024 *threads* por bloco é uma condição própria do desenvolvimento em CUDA.

3.1 1.1 Experimentos com CUDA

O objetivo aqui é realizar experimentos para diferentes valores dos parâmetros x e y, que determinam a dimensão dos blocos e, por conseguinte, do *grid*. Para tanto, determinamos uma região de interesse para esses parâmetros cujo principal critério de escolha foi garantir que mantivéssemos $x * y \leq 1024$, conforme explicado anteriormente. Nesse sentido, decidimos começar com blocos de dimensao $(2, 2)$ --- isto é, quatro threads por bloco, implicando num grid de dimensão $(\frac{image\_size}{2}, \frac{image\_size}{2}) = (\frac{4096}{2}, \frac{4096}{2})$ --- e irmos dobrando até $(32, 32)$, onde têm-se em cada bloco o limite de 1024 threads, com *grid* de dimensão $(128, 128)$.

A **região de interesse** final, com a qual realizamos os experimentos, é:

Dimensões dos blocos: $(2, 2)$, $(4, 4)$, $(8, 8)$, $(16, 16)$ e $(32, 32)$

Respectivas dimensoes do *grid* : $(2048, 2048)$, $(1024, 1024)$, $(512, 512)$, $(256, 256)$ e $(128, 128)$

Os resultados dos experimentos estão no arquivo *cuda_experiments.csv* , que importamos abaixo:

```
In [3]: experiments_cuda = read_csv_results("cuda_experiments.csv")
```

```
Out[3]:
```

	dimensions	duration
	String	Float64
1	2, 2	18.4867
2	2, 2	17.6385
3	2, 2	17.6688
4	2, 2	17.6069
5	2, 2	17.6232
6	2, 2	17.6025
7	2, 2	17.606
8	2, 2	17.7131
9	2, 2	17.7515
10	2, 2	17.6099
11	2, 2	17.6108
12	2, 2	17.6054
13	2, 2	17.6115
14	2, 2	17.6086
15	2, 2	17.605
16	4, 4	4.47781
17	4, 4	4.4789
18	4, 4	5.14805
19	4, 4	4.67234
20	4, 4	4.46192
21	4, 4	4.45937
22	4, 4	4.45696
23	4, 4	4.45597
24	4, 4	4.46125
25	4, 4	4.45959
26	4, 4	4.45692
27	4, 4	4.4575
28	4, 4	4.45686
29	4, 4	4.4604
30	4, 4	4.45654
...	...	...

Com estes resultados, podemos obter o tempo médio de execução e os respectivos intervalos de confiança para cada parâmetro da nossa região de interesse, conforme abaixo:

```
In [4]: final_results_cuda = experiments_cuda |>
                            @groupby({_.dimensions,}) |>
                            @map({dimensions = key(_).dimensions,
                                  mean_duration = mean(_.duration),
                                  ci_duration = 1.96 * std(_.duration)}) |>
                            DataFrame
```

```
Out[4]:
```

	dimensions	mean_duration	ci_duration
	String	Float64	Float64
1	2, 2	17.6899	0.440795
2	4, 4	4.52136	0.356347
3	8, 8	2.27246	0.0727005
4	16, 16	2.28517	0.144487
5	32, 32	2.27029	0.0068397

4 2. Implementação em OMPI

A implementação em OMPI também teve como base a implementação sequencial do programa, que foi adequada para a paralelização através da troca de mensagens. Para este fim, a função `compute_mandelbrot` foi adaptada para receber parâmetros adicionais e uma função responsável por gerenciar a troca de mensagens - `compute_mandelbrot_ompi`- foi criada. Iremos explicar a implementação descrevendo o que é feito pelo processo principal (de rank 0, também chamado de `MASTER` no código) e pelos processos auxiliares.

Processo Principal:

No processo principal, primeiramente são inicializadas várias variáveis globais e os tipos de dados a serem utilizados pelas mensagens são definidos. Logo a seguir, o processo realiza a inicialização e os cálculos dos parametros a serem passados a cada outro processo para definir em que região da imagem ele irá trabalhar.

A imagem é dividida em retângulos de largura `image_size` - que nos nossos experimentos é `4096` - e altura $\frac{image\_size}{num\_processes}$. Ou seja, a divisão de trabalho entre processes é igualitária em tamanho de entrada e cada um dos processos envolvidos - inclusive o processo master - processa um retângulo.

O processo principal então manda mensagens para todos os processos auxiliares com os valores que definem as coordenadas de começo e término da região que cada um deverá processar. Em seguida ele espera os processos terminarem a computação. Ao receber o resultado dos processos auxiliares, o processo master realiza a sua porção da computação de mandelbrot.

O processo principal então recebe um vetor 1D com os valores de x,y e $iterations$ para cada pixel, e uma variável `count` indicando o tamanho desse vetor. Depois de receber os resultados de todos processos o processo principal transfere os valores para a imagem determinado as cores correspondentes.

Processos Auxiliares

Os processos auxiliares realizam o mesmo processo de inicialização das variáveis globais e dos tipos de dados. Depois disso esperam, através da chamada `MPI_Recv` uma mensagem do processo principal definindo a sua região da imagem.

Assim, definimos que todos processos devem processar a região de mandelbrot igualmente, no entando, existe um processo `MASTER` que além de processar a região também define a quantidade de trabalho que cada processo deverá ficar responsável e realiza o envio dessa informação para cada processo.Além disso, ele também recebe o resultado dos processos auxiliares e, junto dos valores calculados pelo próprio processo master, monta a imagem resultante.

4.1 2.1 Experimentos com OMPI

O objetivo é determinar um número ideal n de processos para a implementação através desses experimentos. Definimos a região de interesse conforme sugestão do enunciado, mas sempre

tendo pelo menos um processo principal que coletasse os resultados, sendo assim a **região de interesse** com que realizamos os experimentos foi:

Número de Processos: 2, 3, 5, 9, 17, 33, 65

Os resultados dos experimentos estão no arquivo *ompi_experiments.csv* , que importamos abaixo:

```
In [5]: experiments_ompi = read_csv_results("ompi_experiments.csv")
```

Out[5]:

	processes	duration
	Int64	Float64
1	1	28.0747
2	1	27.9121
3	1	28.0582
4	1	27.9146
5	1	27.9695
6	1	28.1327
7	1	28.1941
8	1	28.6664
9	1	28.1974
10	1	28.0392
11	1	28.0735
12	1	27.9099
13	1	27.9673
14	1	28.0356
15	1	27.9063
16	2	27.9836
17	2	28.0246
18	2	28.0369
19	2	28.075
20	2	28.5776
21	2	27.9497
22	2	28.1654
23	2	28.0353
24	2	28.1507
25	2	28.0538
26	2	28.1861
27	2	27.9217
28	2	28.0822
29	2	28.0657
30	2	28.0113
...	...	...

Com estes resultados, podemos obter o tempo médio de execução e os respectivos intervalos de confiança para cada parâmetro da nossa região de interesse, conforme abaixo:

```
In [6]: final_results_ompi =  experiments_ompi |>
                                @groupby({_.processes,}) |>
                                @map({processes = string((key(_).processes)),
                                      mean_duration = mean(_.duration),
```

```
                                        ci_duration = 1.96 * std(_.duration)}) |>
                                DataFrame
```

Out[6]:

	processes	mean_duration	ci_duration
	String	Float64	Float64
1	1	28.0701	0.376175
2	2	28.088	0.3028
3	4	16.1945	0.247247
4	8	9.8104	0.45482
5	16	6.3391	0.219758
6	32	5.38814	0.405501
7	64	4.78659	0.268464

5 3. Implementação em OMPI + OMP

Nessa implementação, mantivemos a estrutura de comunicação definida na implementação com OMP e alteramos como cada processo processa sua região da imagem: para calcular o número de iterações até a convergência de cada pixel, paralelizamos a computação por OpenMP. Para tanto, as mudanças ocorreram apenas na função `compute_mandelbrot`.

5.1 3.1 Experimentos com OMPI + OMP

Seguindo as instruções do enunciado, os experimentos foram feitos usando os seguintes parâmetros fixos: 15 **repetições** da geração de uma imagem de **tamanho** 4096 da região **Triple Spiral Valley**.

Os números de processos foram determinados da mesma maneira que na implementação em OMPI, para fins de equivalência dos experimentos. O número t de threads varia entre 1 e 64 seguindo as potências de 2, afim de cobrir uma região grande, permitindo analisar o impacto desse parâmetro. Consideramos que valores maiores gerariam um overhead muito grande, tornando o experimento pouco informativo.

Número de Processos: 2,3,5,9,17,33,65

Número de Threads: 1,2,4,8,16,32,64

Os resultados dos experimentos estão no arquivo *ompi_omp_experiments.csv* , que importamos abaixo:

```
In [7]: experiments_ompi_omp = read_csv_results("ompi_omp_experiments.csv")
```

Out[7]:

	processes	threads	duration
	Int64	Int64	Float64
1	2	1	27.295
2	2	1	27.309
3	2	1	27.2576
4	2	1	26.9724
5	2	1	26.9656
6	2	1	27.0364
7	2	1	27.0953
8	2	1	27.1699
9	2	1	27.0046
10	2	1	27.1624
11	2	1	27.8146
12	2	1	27.763
13	2	1	27.3697
14	2	1	26.9314
15	2	1	26.9981
16	2	2	15.845
17	2	2	15.963
18	2	2	15.8725
19	2	2	17.098
20	2	2	16.5151
21	2	2	15.9097
22	2	2	16.1336
23	2	2	15.9903
24	2	2	15.8841
25	2	2	16.0674
26	2	2	16.9746
27	2	2	16.0554
28	2	2	15.9894
29	2	2	16.3131
30	2	2	16.0309
...	...	...	...

Com estes resultados, podemos obter o tempo médio de execução e os respectivos intervalos de confiança para cada parâmetro da nossa região de interesse. Dessa vez...

```
In [8]: final_results_ompi_omp = experiments_ompi_omp |>
                                  @groupby({_.threads,_.processes}) |>
                                  @map({threads = key(_).threads,
                                        processes = string(key(_).processes),
                                        mean_duration = mean(_.duration),
                                        ci_duration = 1.96 * std(_.duration)}) |>
                                  DataFrame
```

Out[8]:

	threads	processes	mean_duration	ci_duration
	Int64	String	Float64	Float64
1	1	2	27.2097	0.535302
2	2	2	16.1761	0.76688
3	4	2	16.5573	0.492432
4	8	2	16.9607	0.400679
5	16	2	17.2051	0.325129
6	32	2	17.2293	0.306383
7	64	2	17.2701	0.0918309
8	1	3	27.4547	0.452396
9	2	3	15.1189	0.315654
10	4	3	9.06702	0.386148
11	8	3	6.18638	0.298335
12	16	3	4.6963	0.224809
13	32	3	4.71005	0.338852
14	64	3	4.62261	0.351451
15	1	5	9.56214	0.464149
16	2	5	6.27879	0.450985
17	4	5	4.62318	0.225453
18	8	5	3.78137	0.396802
19	16	5	3.89263	0.218808
20	32	5	3.82131	0.298995
21	64	5	3.95306	0.488549
22	1	9	9.60592	0.208683
23	2	9	6.30602	0.32178
24	4	9	4.53306	0.467989
25	8	9	3.91911	0.304496
26	16	9	3.96972	0.261594
27	32	9	3.79557	0.237693
28	64	9	3.85202	0.455137
29	1	17	4.7371	0.410375
30	2	17	3.86911	0.322062
...	...	...	...	...

6 4. Implementação em OMPI + CUDA

Nessa implementação, mantivemos a estrutura de comunicação definida na implementação com OMPI e alteramos como cada processo auxiliar processa sua região da imagem: realizamos o cálculo do número de iterações até a convergência de cada pixel na GPU através do `cuda`.

6.1 4.1 Experimentos com OMPI + CUDA

A região de interesse foi determinada de maneira a manter a consistência experimental entre as diferentes implementações. Sendo assim:

Número de Processos: 2,3,5,9,17,33,65

Dimensões dos blocos: $(2, 2)$, $(4, 4)$, $(8, 8)$, $(16, 16)$ e $(32, 32)$

Os resultados dos experimentos estão no arquivo *ompi_omp_experiments.csv* , que importamos abaixo:

```
In [9]: experiments_ompi_cuda = read_csv_results("ompi_cuda_experiments.csv")
```

Out[9]:

	processes	dimensions	duration
	Int64	String	Float64
1	2	2, 2	4.74935
2	2	2, 2	4.0107
3	2	2, 2	4.41688
4	2	2, 2	3.98594
5	2	2, 2	4.69576
6	2	2, 2	3.50715
7	2	2, 2	4.98862
8	2	2, 2	4.12465
9	2	2, 2	3.94018
10	2	2, 2	4.63031
11	2	2, 2	3.94324
12	2	2, 2	4.27782
13	2	2, 2	3.54489
14	2	2, 2	4.34835
15	2	2, 2	3.42501
16	2	4, 4	4.55023
17	2	4, 4	4.43831
18	2	4, 4	4.71328
19	2	4, 4	5.87383
20	2	4, 4	3.08533
21	2	4, 4	2.76966
22	2	4, 4	2.86335
23	2	4, 4	3.24694
24	2	4, 4	2.8068
25	2	4, 4	3.05637
26	2	4, 4	2.87339
27	2	4, 4	3.10552
28	2	4, 4	2.81966
29	2	4, 4	2.85723
30	2	4, 4	3.45904
...	...	...	...

```
In [10]: final_results_ompi_cuda = experiments_ompi_cuda|>
                                  @groupby({_.dimensions,_.processes}) |>
                                      @map({dimensions = key(_).dimensions,
                                            processes = string(key(_).processes),
                                            mean_duration = mean(_.duration),
                                            ci_duration = 1.96 * std(_.duration)}) |>
                                      DataFrame
```

Out[10]:

	dimensions	processes	mean_duration	ci_duration
	String	String	Float64	Float64
1	2, 2	2	4.17259	0.929573
2	4, 4	2	3.50126	1.84226
3	8, 8	2	3.01322	0.565195
4	16, 16	2	3.56778	0.794565
5	32, 32	2	4.32816	1.32265
6	2, 2	3	5.46269	1.55897
7	4, 4	3	5.87547	3.06311
8	8, 8	3	2.95505	0.68763
9	16, 16	3	2.59008	0.232956
10	32, 32	3	2.7403	0.497301
11	2, 2	5	2.96337	0.722896
12	4, 4	5	2.89533	0.760279
13	8, 8	5	2.8316	0.499022
14	16, 16	5	2.73196	0.387759
15	32, 32	5	2.81389	0.483785
16	2, 2	9	2.93924	0.309103
17	4, 4	9	3.04089	0.534596
18	8, 8	9	3.15598	0.898644
19	16, 16	9	3.03545	0.505406
20	32, 32	9	3.08343	0.714556
21	2, 2	17	3.44351	0.456788
22	4, 4	17	3.55086	0.645048
23	8, 8	17	3.40701	0.292924
24	16, 16	17	3.68272	0.6648
25	32, 32	17	3.40581	0.461648
26	2, 2	33	4.54314	0.414195
27	4, 4	33	4.55174	0.28758
28	8, 8	33	4.58453	0.58128
29	16, 16	33	4.59397	0.452394
30	32, 32	33	4.57655	0.38451
...	...	...	...	...

7 5. Análise dos resultados dos experimentos

Faremos uso das duas funções abaixo para gerar os gráficos sobre os quais analisaremos cada caso:

```
In [11]: #Labels para os gráficos
         threads = [1 2 4 8 16 32 64]
         processes =[2 3 5 9 17 33 65]
         dimensions = ["(2,2)" "(4,4)" "(8,8)" "(16,16)" "(32,32)"]

Out[11]: 1Œ5 Array{String,2}:
          "(2,2)"  "(4,4)"  "(8,8)"  "(16,16)"  "(32,32)"

In [12]: function plot_results(x, y1, series_label, xlabel_, ylabel_, title_, yerror)
             if yerror == []
```

```
            p = scatter(x, y1,
            alpha = 0.6,
            labels = series_label,
            xlabel = xlabel_,
            fmt = :png,
            ylabel = ylabel_,
            title = title_,
            legend = :topright)

            return p
        end

        p = plot(x,
                y1,
                yerror = yerror,
                alpha = 0.9,
                labels = series_label,
                fmt = :png,
                xlabel = xlabel_,
                ylabel = ylabel_,
                title = title_,
                color = "red",
                seriestype = :scatter,
                lw = 1,
                legend = :topright)

        return p
    end
```

```
Out[12]: plot_results (generic function with 1 method)
```

7.1 5.1 Resultados da versão CUDA

```
In [13]: plot_results(experiments_cuda.dimensions, experiments_cuda.duration, "", "Dimensão", "
```

```
Out[13]:
```


```
In [14]: plot_results(final_results_cuda.dimensions,
             final_results_cuda.mean_duration,
             "", "Dimensão", "Tempo(s)", "Experimentos com CUDA",
             final_results_cuda.ci_duration)
```

Out[14]:

Análise e escolha de melhores parâmetros:

No primeiro gráfico, vemos que para experimentos com blocos de dimensão $(2,2)$ e $(4,4)$ há uma variação maior dos tempos de execução individual dentre as 15 feitas em cada caso, o que, portanto, se reflete na maior amplitude dos respectivos intervalos de confiança em comparação com as dimensões de ordem superior, como fica evidente no segundo gráfico. Além disso, nota-se que conforme aumenta-se a dimensão dos blocos menor é o impacto em tempo de execução ao dobrar a dimensão do bloco, inclusive, é praticamente nula a partir de blocos de tamanho (8,8).

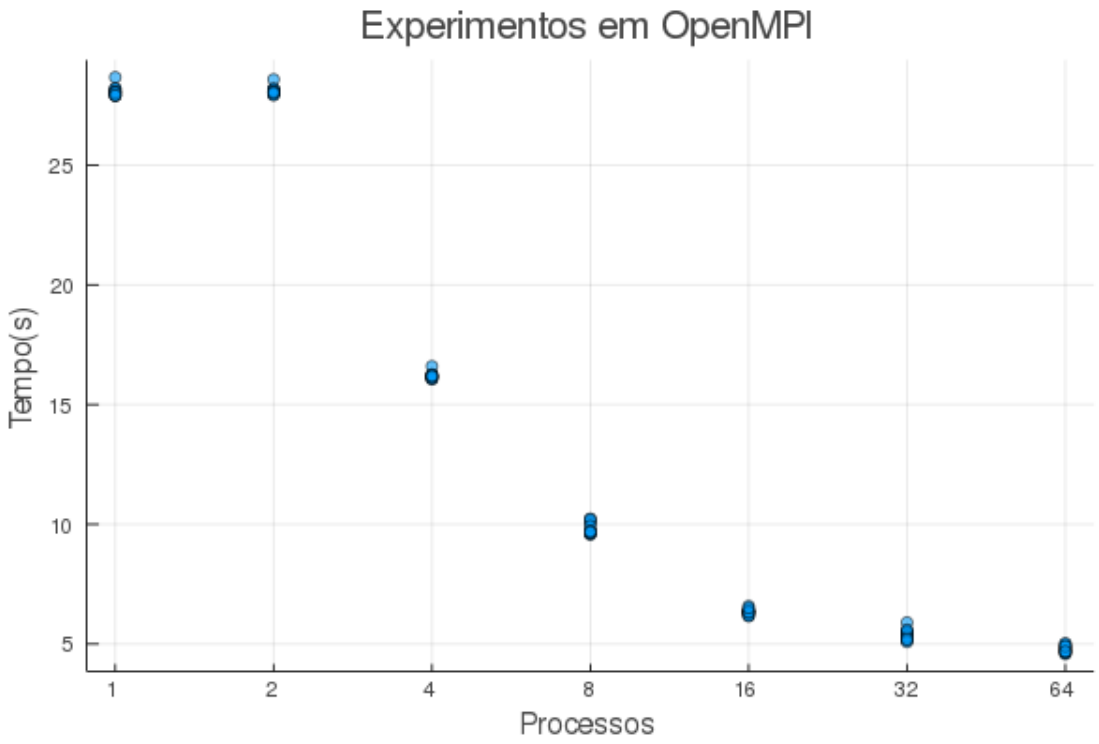
Observamos, por exemplo, que para blocos de dimensão $(2,2)$ e $(4,4)$, a diferença de tempo médio de execução é notável ao dobrar o tamanho do bloco no experimento: aproximadamente 17.5 segundos se dobrarmos de $(2,2)$ para $(4,4)$ e menos de 5s se dobrarmos de $(4,4)$ para $(8,8)$. Para acréscimos acima disso, a diferença de tempo é imperceptível pelo gráfico.

Posto isso, os dados dos experimentos nos permitem concluir com segurança que, para as condições de hardware em questão, execuções com blocos de dimensão $(8,8)$, e portanto em *grid* de dimensão $(\frac{image\_size}{8}, \frac{image\_size}{8}) = (512,512)$, são a melhor escolha dentro da região de interesse na qual os experimentos foram feitos. Isto pois o resultado dos experimentos mostram que o desempenho com tal dimensão é tão bom quanto com dimensões maiores e, sendo assim, faz sentido optar por permanecer com a necessidade de gerenciamente de um menor número de threads por bloco.

7.2 5.2 Resultados da versão OMPI

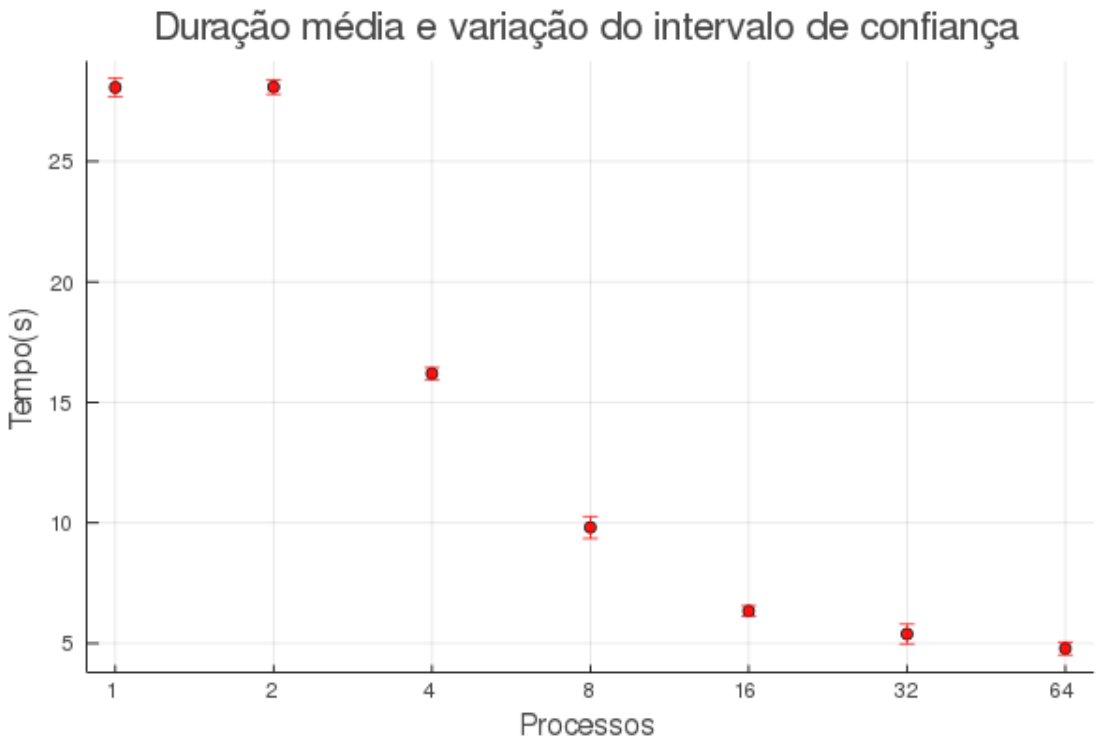
```
In [15]: experiments_ompi[! , :processes_string] = [string(x) for x in experiments_ompi[!, :pro
         plot_results(experiments_ompi.processes_string, experiments_ompi.duration,
             "", "Processos", "Tempo(s)", "Experimentos em OpenMPI", [])
```

Out[15]:



In [16]:
```
plot_results(final_results_ompi.processes,
             final_results_ompi.mean_duration,
             "", "Processos", "Tempo(s)", "Duração média e variação do intervalo de confiança"
             final_results_ompi.ci_duration)
```

Out[16]:



Análise e escolha de melhores parâmetros:

Podemos observar que, à partir de dois processos, o tempo médio de execução consistentemente diminui conforme aumenta a quantidade de processos envolvidos, e a partir de 16 processos os ganhos em tempo de execução passam a ser marginais relativamente as diminuições anteriores.

Vale chamar atenção para o fato de que os dados obtidos para os experimentos com 1 e 2 processos foram muito próximos, com valores médios para os tempos de execução e respectivos intervalos de confiança bastante similares. Buscando entender a causa para tal resultado, entendemos que tal similaridade se deve ao fato de que com 1 processo não há distribuição de tarefa (há simplemente o processamento total da imagem), e com 2 processos o primeiro processo tem um *overhead* de gerenciamento de mensagens e distribuição de tarefas (além de processar parte da imagem). A partir de 4 processos a divisão de trabalho vence o *overhead* de gerenciamento.

Os gráficos permitem observar ainda que o tempo individual de cada execução para uma determinada quantidade de processos se mostrou em geral consistente em todos os casos, isto é, constata-se pouca variação no tempo individual de cada umas das 15 execuções, levanto a valores para os respectivos tempos médio de execução com intervalos de confiança com amplitudes muito pequenas.

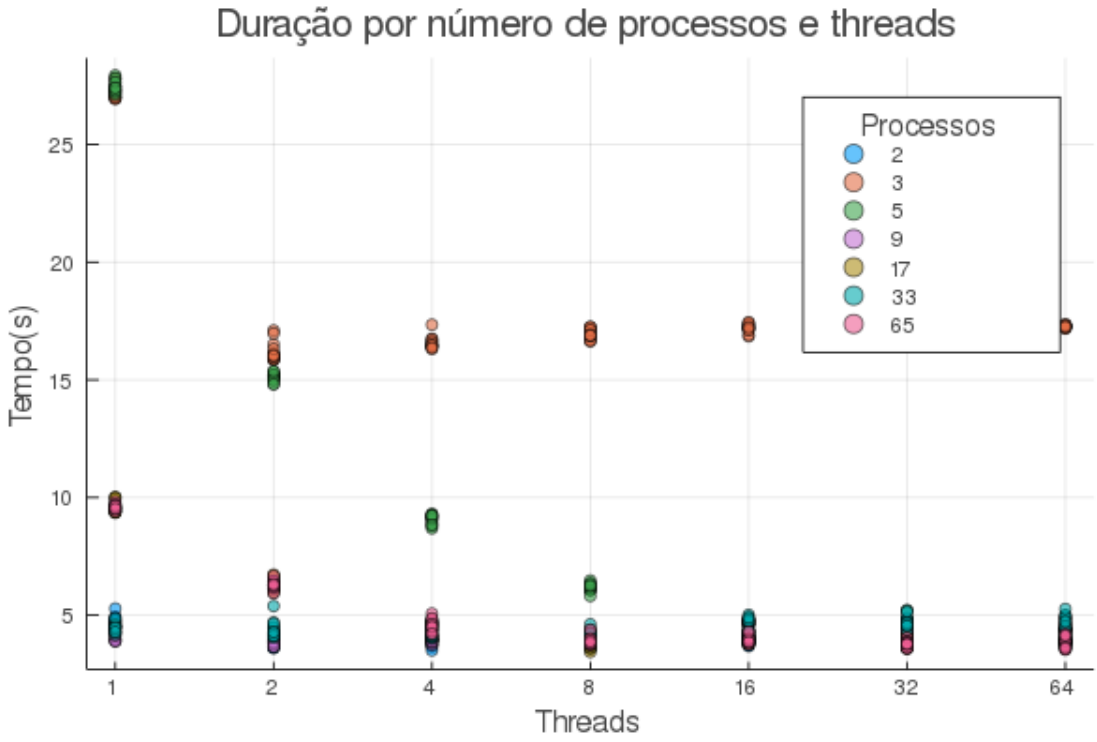
Como o tempo de execução diminui consistentemente e o intervalo de confiança é pequeno, comparativamente concluimos que o desempenho é ótimo para **64 processos**.

7.3 5.3 Resultados da versão OMPI + OMP

```
In [17]: experiments_ompi_omp[! , :threads_string] = [string(x) for x in experiments_ompi_omp[
         experiments_ompi_omp[! , :processes_string] = [string(x) for x in experiments_ompi_om
```

```
p = scatter(
        experiments_ompi_omp.threads_string,
        experiments_ompi_omp.duration,
        title = "Duração por número de processos e threads",
        xlabel = "Threads",
        legendtitle = "Processos",
        ylabel = "Tempo(s)",
        labels = processes,
        fmt = :png,
        alpha = 0.6,
        group = experiments_ompi_omp.processes_string,
        legend = :topright)
```

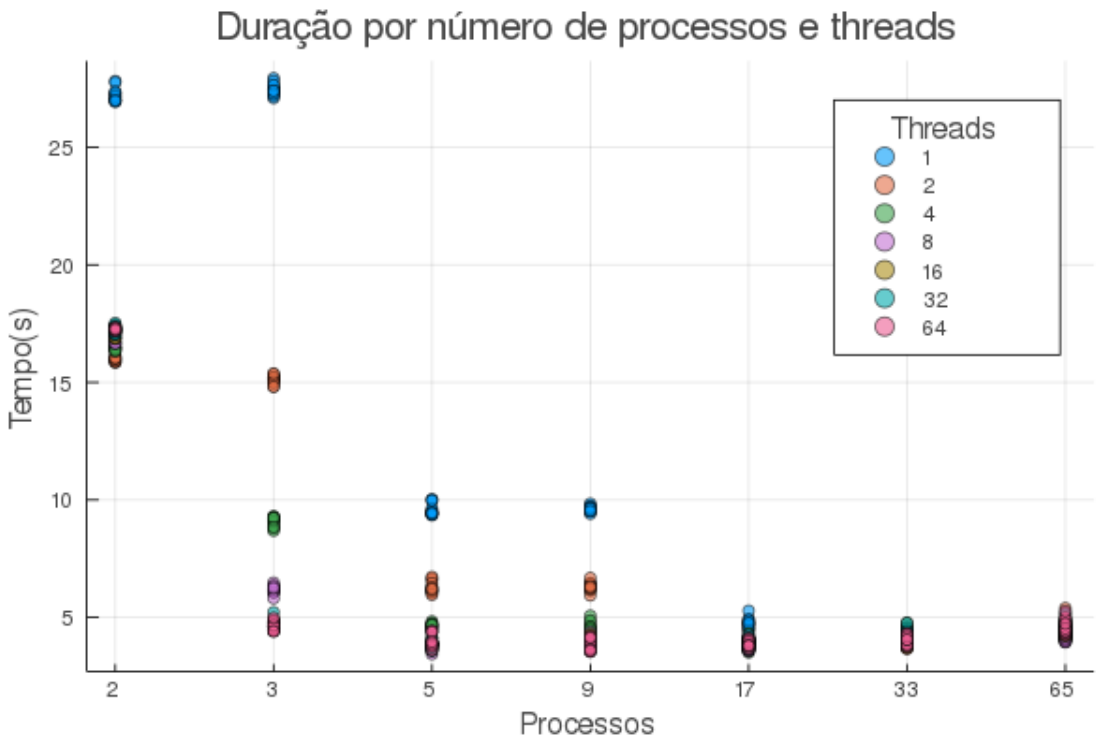
Out[17]:



In [18]:
```
p = scatter(
        experiments_ompi_omp.processes_string,
        experiments_ompi_omp.duration,
        title = "Duração por número de processos e threads",
        xlabel = "Processos",
        legendtitle = "Threads",
        ylabel = "Tempo(s)",
        labels = threads,
        fmt = :png,
```

```
        alpha = 0.6,
        group = experiments_ompi_omp.threads,
        legend = :topright)
```

Out[18]:



In [19]:
```
final_results_ompi_omp[! , :processes_string] = [string(x) for x in final_results_ompi
final_results_ompi_omp[! , :threads_string] = [string(x) for x in final_results_ompi_

fr_ompi_omp_with_64_processes = final_results_ompi_omp[(final_results_ompi_omp[:proces
```

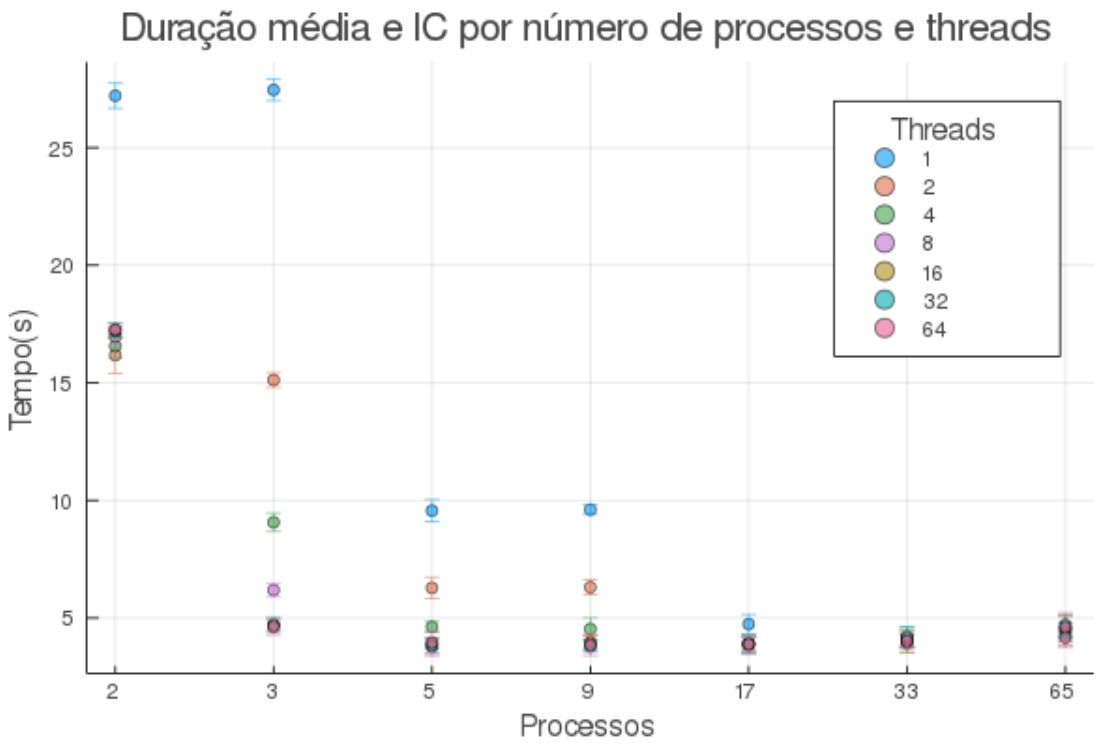
```
Warning: `getindex(df::DataFrame, col_ind::ColumnIndex)` is deprecated, use `df[!, col_ind]` i
  caller = top-level scope at In[19]:3
@ Core In[19]:3
```

Out[19]:

	threads	processes	mean_duration	ci_duration	processes_string	threads_string
	Int64	String	Float64	Float64	String	String
1	1	65	4.47534	0.319705	65	1
2	2	65	4.39513	0.658592	65	2
3	4	65	4.22526	0.356018	65	4
4	8	65	4.16466	0.349235	65	8
5	16	65	4.57533	0.534472	65	16
6	32	65	4.70935	0.434551	65	32
7	64	65	4.61237	0.614406	65	64

In [20]:
```
p = scatter(
        final_results_ompi_omp.processes_string,
        final_results_ompi_omp.mean_duration,
        title = "Duração média e IC por número de processos e threads",
        xlabel = "Processos",
        legendtitle = "Threads",
        ylabel = "Tempo(s)",
        labels = threads,
        fmt = :png,
        alpha = 0.6,
        yerror = final_results_ompi_omp.ci_duration,
        group = final_results_ompi_omp.threads,
        legend = :topright)
```

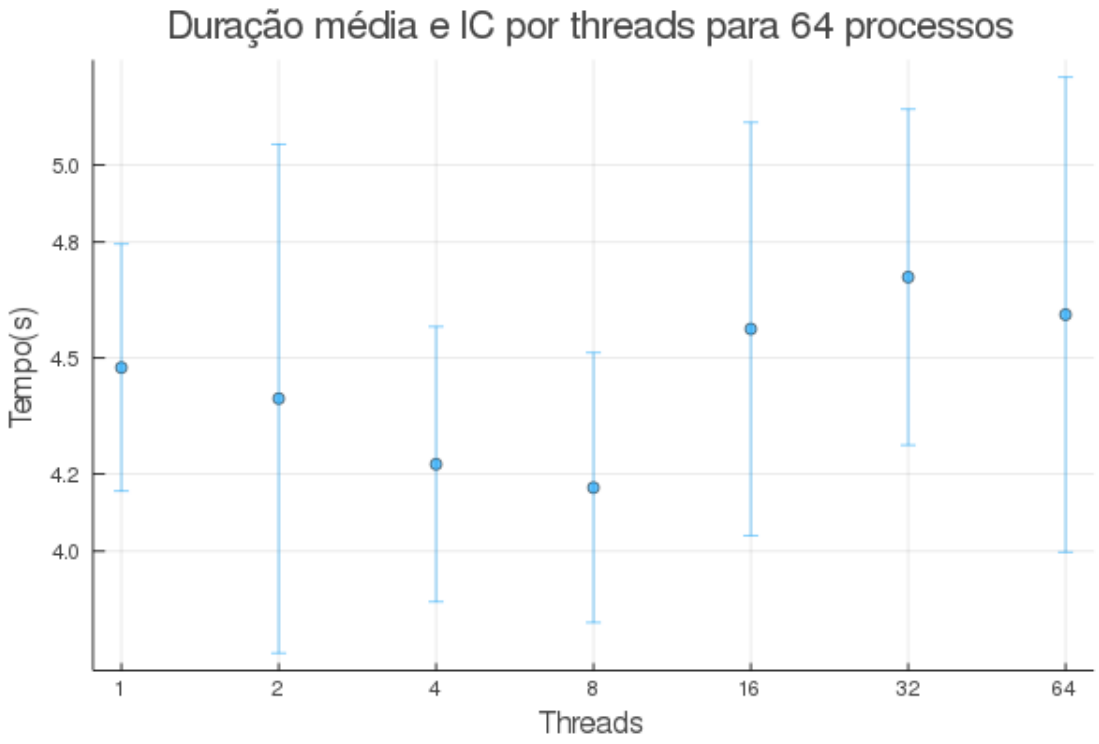
Out[20]:



In [21]:
```
p = scatter(
        fr_ompi_omp_with_64_processes.threads_string,
        fr_ompi_omp_with_64_processes.mean_duration,
        title = "Duração média e IC por threads para 64 processos",
        xlabel = "Threads",
        ylabel = "Tempo(s)",
        alpha = 0.6,
        fmt = :png,
```

```
        yerror = fr_ompi_omp_with_64_processes.ci_duration,
        legend=false)
```

Out[21]:



Análise e escolha de melhores parâmetros:

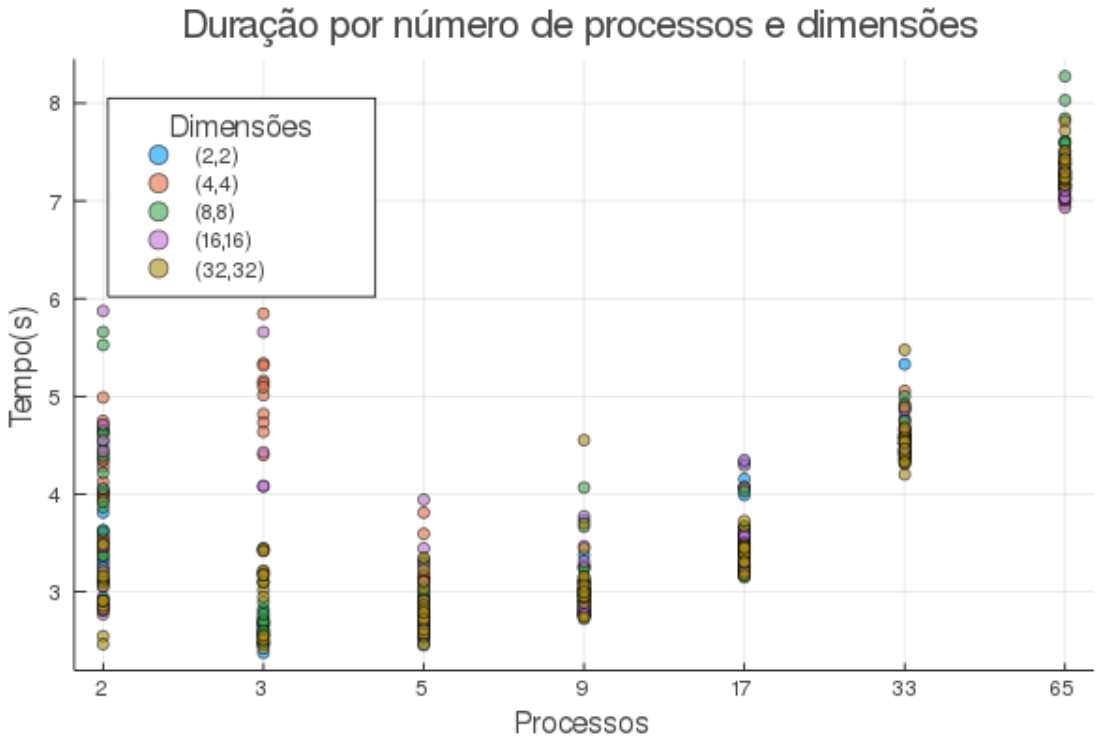
Baseado nos gráficos, podemos concluir que o tempo de execução médio decai consistemente com o aumento do número de processos lançados. Ademais, nota-se que o tempo de execução sofre uma influência maior do número de processos em relação ao número de threads. Percebemos que o tempo de execução é ótimo para 64 processos. Para decidir o número correspondente de threads, plotamos mais um gráfico fixando 64 processos e analisando a diferença no tempo de execução para diferentes threads e concluimos que o tempo de execução é ótimo para **8 threads**.

7.4 5.4 Resultados da versão OMPI + CUDA

```
In [22]: experiments_ompi_cuda[! , :processes_string] = [string(x) for x in experiments_ompi_cu

        p = scatter(
                experiments_ompi_cuda.processes_string,
                experiments_ompi_cuda.duration,
                title = "Duração por número de processos e dimensões",
                xlabel = "Processos",
                legendtitle = "Dimensões",
                ylabel = "Tempo(s)",
                labels = dimensions,
```

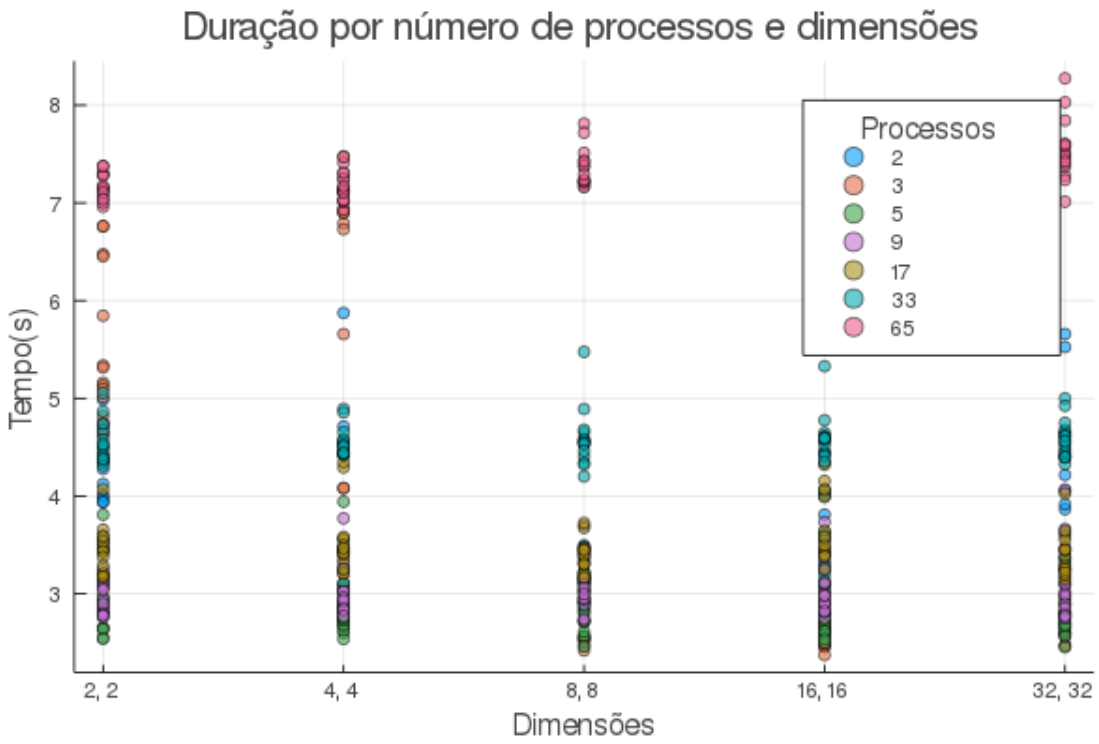
```
                fmt = :png,
                alpha = 0.6,
                group = experiments_ompi_cuda.dimensions,
                legend = :topleft)
```

Out[22]:



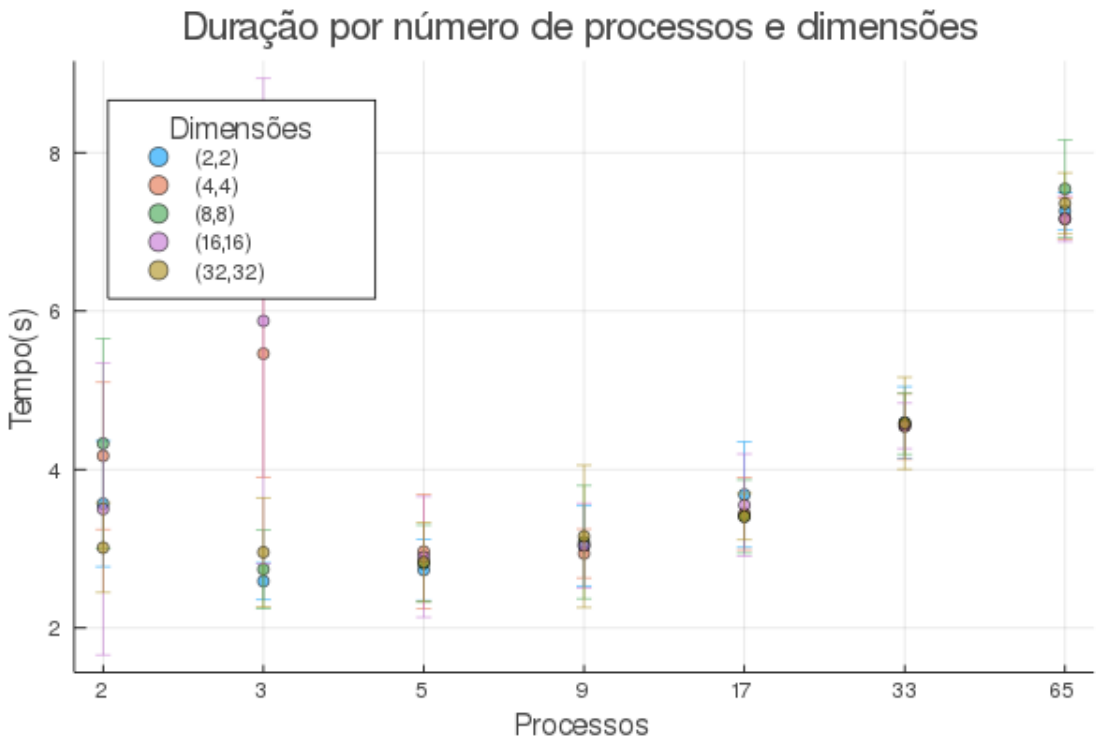
In [23]:
```
p = scatter(
                experiments_ompi_cuda.dimensions,
                experiments_ompi_cuda.duration,
                title = "Duração por número de processos e dimensões",
                xlabel = "Dimensões",
                legendtitle = "Processos",
                ylabel = "Tempo(s)",
                labels = processes,
                fmt = :png,
                alpha = 0.6,
                group = experiments_ompi_cuda.processes,
                legend = :topright)
```

Out[23]:



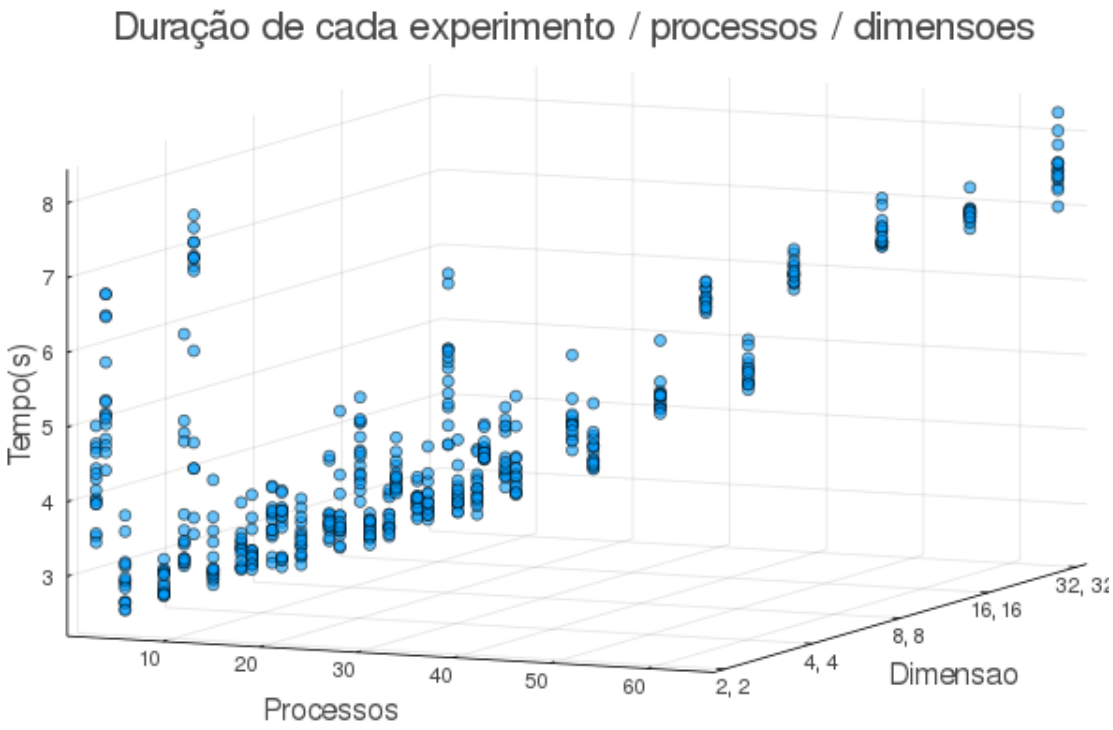

```
In [24]: p = scatter(
                    final_results_ompi_cuda.processes,
                    final_results_ompi_cuda.mean_duration,
                    title = "Duração por número de processos e dimensões",
                    xlabel = "Processos",
                    legendtitle = "Dimensões",
                    ylabel = "Tempo(s)",
                    labels = dimensions,
                    fmt = :png,
                    alpha = 0.6,
                    group = final_results_ompi_cuda.dimensions,
                    yerror = final_results_ompi_cuda.ci_duration,
                    legend = :topleft)
```

Out[24]:




```
In [25]: p = scatter3d(
                    experiments_ompi_cuda.processes,
                    experiments_ompi_cuda.dimensions,
                    experiments_ompi_cuda.duration,
                    title = "Duração de cada experimento / processos / dimensoes",
                    xlabel = "Processos",
                    ylabel = "Dimensao",
                    zlabel = "Tempo(s)",
                    seriestype = :scatter,
                    fmt = :png,
                    alpha = 0.6,
                    legend = false
                )
```

Out[25]:



Análise e escolha de melhores parâmetros:

O primeiro dos gráficos desta sub-seção já evidencia que para esta versão, o crescimento do número de processos implica em piora no desempenho, para todas as dimensões dos blocos da nossa região de interesse. O gráfico seguinte reforça essa conclusão ao mostrar que para todas as dimensões, o tempo de execução dos experimentos com maior quantidade de processos foram os mais elevados.

O terceiro gráfico nos ajuda a identificar a quantidade de processos com a qual temos melhor desempenho dentro da região de interesse da dimensão de bloco: 5 processos, onde vemos experimentos com menor tempo de execução. Fixado este parâmetro, o último dos gráficos nos permite, enfim, verificar que o tempo de execução é menos para menores dimensões dos blocos.

Nesse sentido, e baseado nisso, a escolha de melhores parâmetros se define como sendo: 5 **processos e blocos de dimensão** $(2, 2)$

8 6. Resultados do EP1 seguindo o padrão do EP2

Para que possamos fazer uma analise comparativa entre todas as versões que desenvolvemos para o Cálculo do Conjunto de Mandelbrot, através do script *run_experiments.py* executamos também experimentos relativos as versões entregues no EP1 (Sequencial, Pthreads e OpenMP) utilizando os mesmos parâmetros fixos considerados aqui: 15 **repetições** da geração de uma imagem de **tamanho** 4096 da região **Triple Spiral Valley**

Assim como nos casos anteriores, os arquivos .csv com os resultados destes experimentos foram fornecidos e nós os importamos abaixo:

8.0.1 Sequencial:

```
In [26]: experiments_sequencial = read_csv_results("seq_experiments.csv")
```

Out[26]:

	duration
	Float64
1	27.5763
2	27.6554
3	28.2566
4	29.741
5	28.4806
6	29.4961
7	28.7576
8	28.2968
9	28.4506
10	27.4238
11	29.6745
12	28.6952
13	29.916
14	27.5209
15	26.9703

```
In [27]: final_results_sequencial = DataFrame(mean_duration = mean(experiments_sequencial.durat
                                              ci_duration = 1.96 * std(experiments_sequencial.duratio
```

Out[27]:

	mean_duration	ci_duration
	Float64	Float64
1	28.4608	1.82404

8.0.2 Pthreads:

A implementação utilizada foi a mesma feita no EP1, e os experimentos tiveram a seguinte região de interesse:

Número de Threads: 1,2,4,8,16,32,64

Importamos os dados:

```
In [28]: experiments_pth = read_csv_results("pth_experiments.csv")
```

Out[28]:

	threads	duration
	Int64	Float64
1	1	28.8056
2	1	28.3448
3	1	29.0453
4	1	29.394
5	1	28.1015
6	1	28.3721
7	1	28.5022
8	1	28.2636
9	1	29.1992
10	1	30.22
11	1	28.5925
12	1	28.589
13	1	28.3946
14	1	29.0623
15	1	29.8218
16	2	26.9687
17	2	26.5998
18	2	26.2947
19	2	26.833
20	2	26.7021
21	2	26.8709
22	2	27.3269
23	2	26.7917
24	2	27.1039
25	2	26.5913
26	2	26.8068
27	2	26.6087
28	2	27.1613
29	2	26.4086
30	2	26.719
...	...	...

```
In [29]: final_results_pth = experiments_pth |>
                                @groupby({_.threads,}) |>
                                @map({threads = string(key(_).threads),
                                      mean_duration = mean(_.duration),
                                      ci_duration = 1.96 * std(_.duration)}) |>
                                DataFrame
```

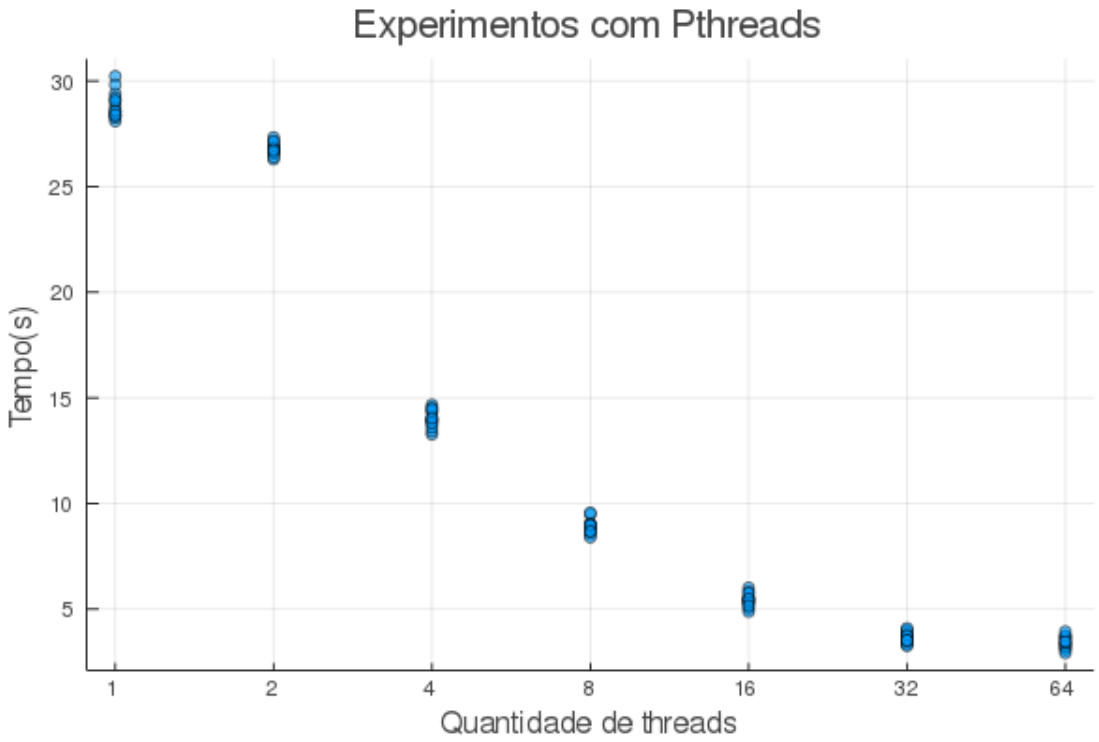
Out[29]:

	threads	mean_duration	ci_duration
	String	Float64	Float64
1	1	28.8472	1.19163
2	2	26.7858	0.543657
3	4	14.0143	0.79504
4	8	8.89757	0.616525
5	16	5.42304	0.579045
6	32	3.6455	0.463688
7	64	3.42551	0.548569

Temos então:

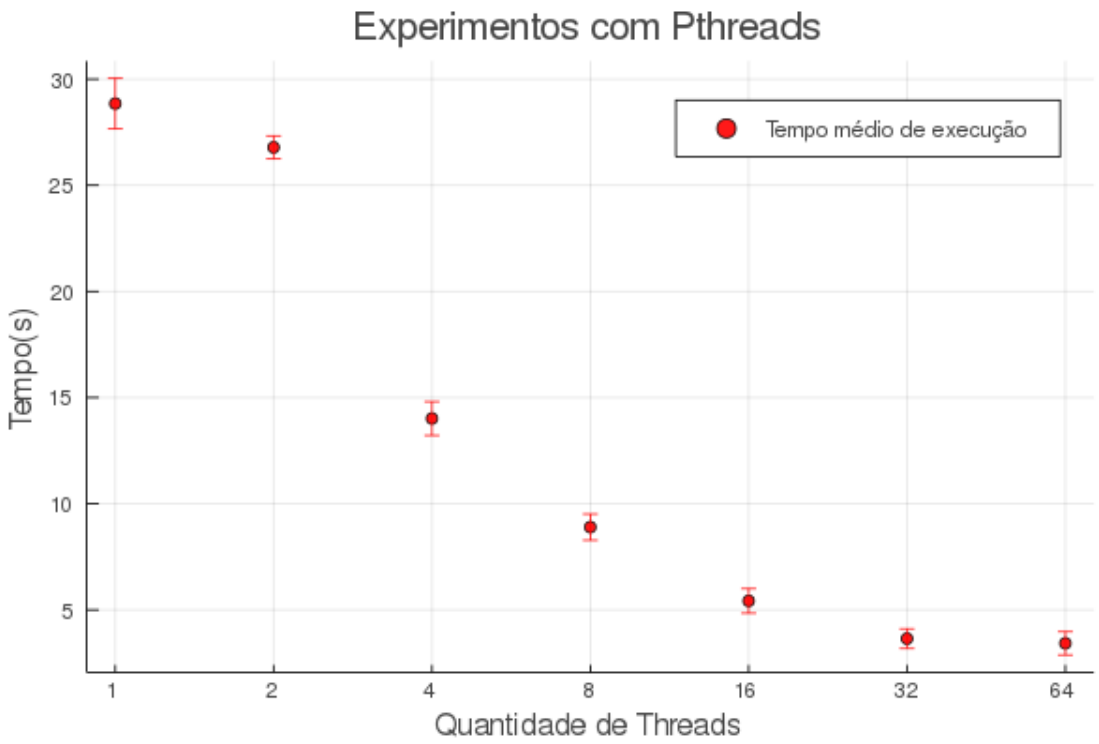
```
In [30]: experiments_pth[! , :threads_string] = [string(x) for x in experiments_pth[!, :threads

         plot_results(experiments_pth.threads_string, experiments_pth.duration,
             "", "Quantidade de threads", "Tempo(s)",
             "Experimentos com Pthreads", [])
```

Out[30]:




```
In [31]: plot_results(final_results_pth.threads, final_results_pth.mean_duration, "Tempo médio
```

Out[31]:



Podemos observar que ganhos significativos ocorrem até a marca de 32 threads, e depois disso aparentemente o overhead se torna tão grande quanto os ganhos de performance. Ademais, na comparação entre 32 e 64 threads, notamos que com 32 threads o intervalo de confiança é menor e as médias são muito similares entre si. Portanto, definimos 32 como o valor ideal deste parâmetro.

8.0.3 OpenMP:

Os experimentos foram realizados da mesma maneira que na implementação em pthreads, e com a mesma região de interesse.

```
In [32]: experiments_omp = read_csv_results("omp_experiments.csv")
```

```
Out[32]:
```

	threads	duration
	Int64	Float64
1	1	28.4714
2	1	29.1697
3	1	29.6925
4	1	28.6584
5	1	29.4521
6	1	28.7501
7	1	29.4946
8	1	28.8353
9	1	29.0405
10	1	29.4972
11	1	28.1914
12	1	29.2139
13	1	29.6678
14	1	29.4677
15	1	30.6828
16	2	14.9171
17	2	14.7917
18	2	14.7202
19	2	14.9838
20	2	14.4742
21	2	14.467
22	2	15.1323
23	2	14.5948
24	2	15.1095
25	2	14.3584
26	2	14.8756
27	2	14.4997
28	2	14.4835
29	2	14.8525
30	2	14.7609
...	...	...

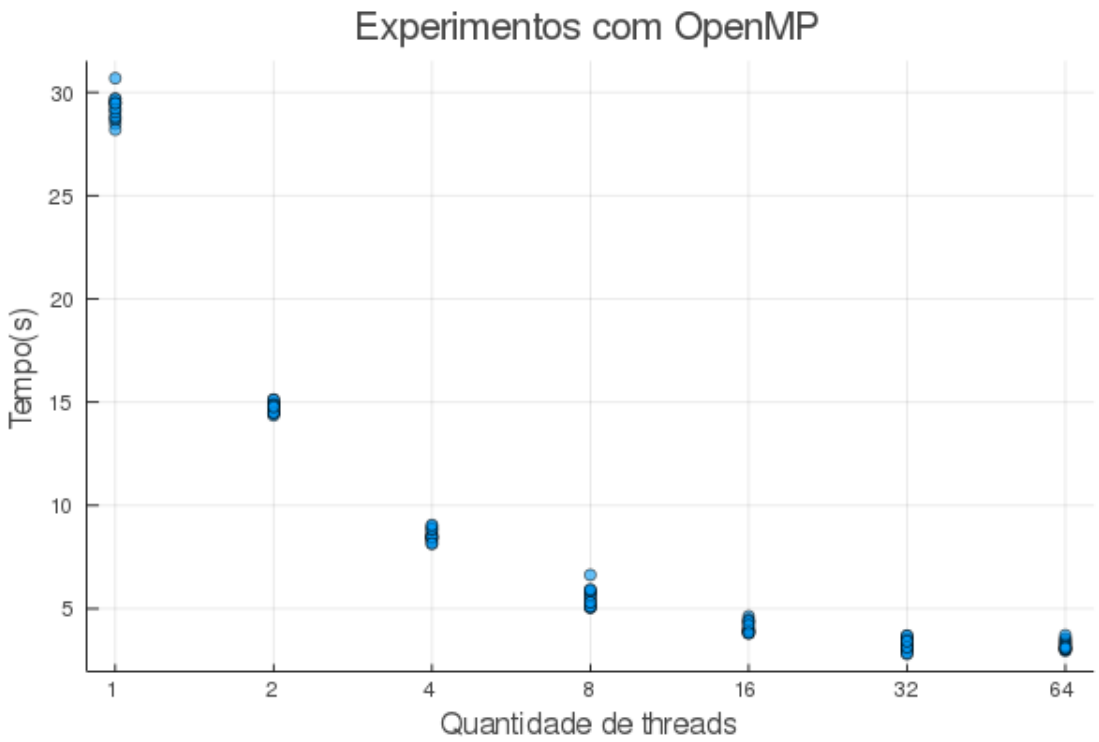
```
In [33]: final_results_omp = experiments_omp |>
                                @groupby({_.threads,}) |>
                                @map({threads = string(key(_).threads),
                                      mean_duration = mean(_.duration),
                                      ci_duration = 1.96 * std(_.duration)}) |>
                                DataFrame
```

Out[33]:

	threads	mean_duration	ci_duration
	String	Float64	Float64
1	1	29.219	1.1909
2	2	14.7347	0.483159
3	4	8.56224	0.55287
4	8	5.55285	0.847721
5	16	4.11329	0.534923
6	32	3.25592	0.557335
7	64	3.24436	0.44773

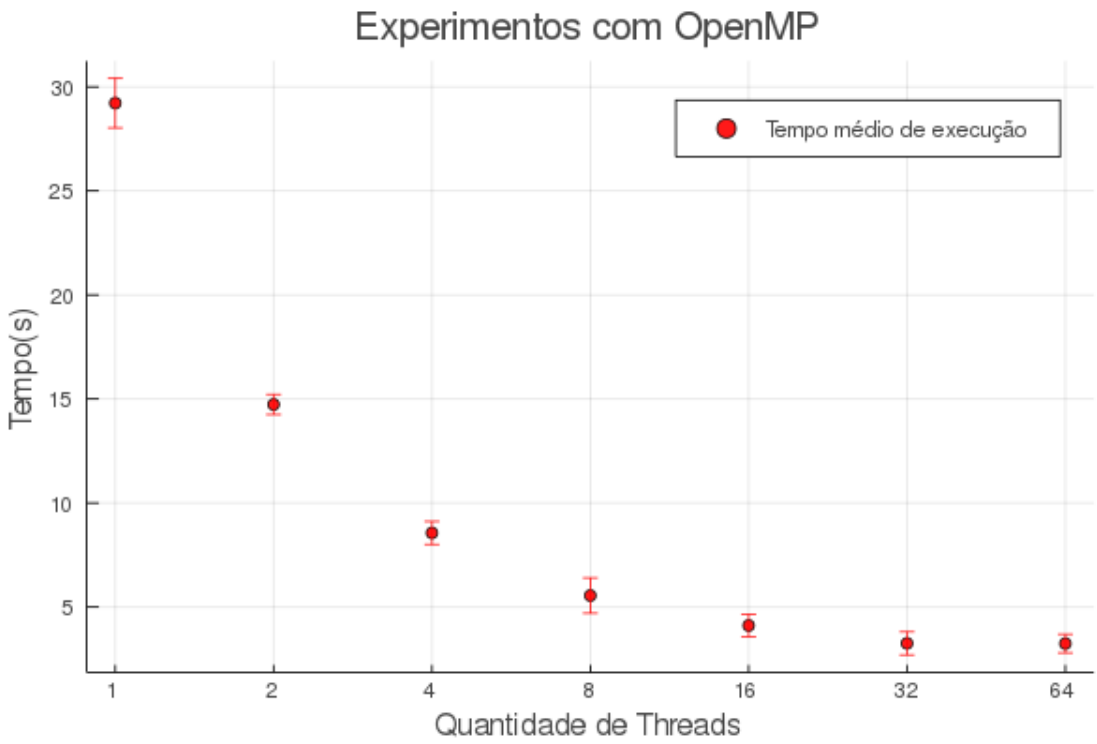
In [34]:
```
experiments_omp[! , :threads_string] = [string(x) for x in experiments_omp[!, :threads
plot_results(experiments_omp.threads_string, experiments_omp.duration, "", "Quantidade
```

Out[34]:



In [35]:
```
plot_results(final_results_omp.threads, final_results_omp.mean_duration, "Tempo médio
```

Out[35]:



Pela mesma argumentação que na versão com p-threads, temos que o valor ideal para o parâmetro é de 64 threads.

9 7. Comparação entre as diferentes versões

Segue um resumo das escolhas de parâmetros que fizemos (justificadas nas seções acima, junto às apresentações dos resultados):

Versão	Parâmetro escolhido
CUDA	bloco de dimensão $(8, 8)$
OMPI	64 processos
OMPI + OMP	64 processos e 8 threads
OMPI + CUDA	5 processos e blocos de dimensão $(2, 2)$
OMP	64 threads
Pthreads	32 threads

Abaixo, selecionamos as informações dos experimentos que foram feitos com os parâmetros escolhidos como melhores para cada uma das implementações, para em seguida gerar gráficos que nos possibilitam, enfim, realizar uma análise comparativa entre todas as implementações:

```
In [36]: names = ["Sequencial", "OMP", "Pthreads", "OMPI", "CUDA", "OMPI+CUDA", "OMPI+OMP"]

         duration = [ final_results_sequencial[1, :mean_duration],
```

```
            final_results_omp[7 , :mean_duration],
            final_results_pth[6 , :mean_duration],
            final_results_ompi[7, :mean_duration],
            final_results_cuda[3, :mean_duration],
            final_results_ompi_cuda[11, :mean_duration],
            final_results_ompi_omp[46, :mean_duration],
            ]

        ci_d = [ final_results_sequencial[1, :ci_duration],
            final_results_omp[7 , :ci_duration],
            final_results_pth[6 , :ci_duration],
            final_results_ompi[7, :ci_duration],
            final_results_cuda[3, :ci_duration],
            final_results_ompi_cuda[11, :mean_duration],
            final_results_ompi_omp[46, :ci_duration],
            ];
```

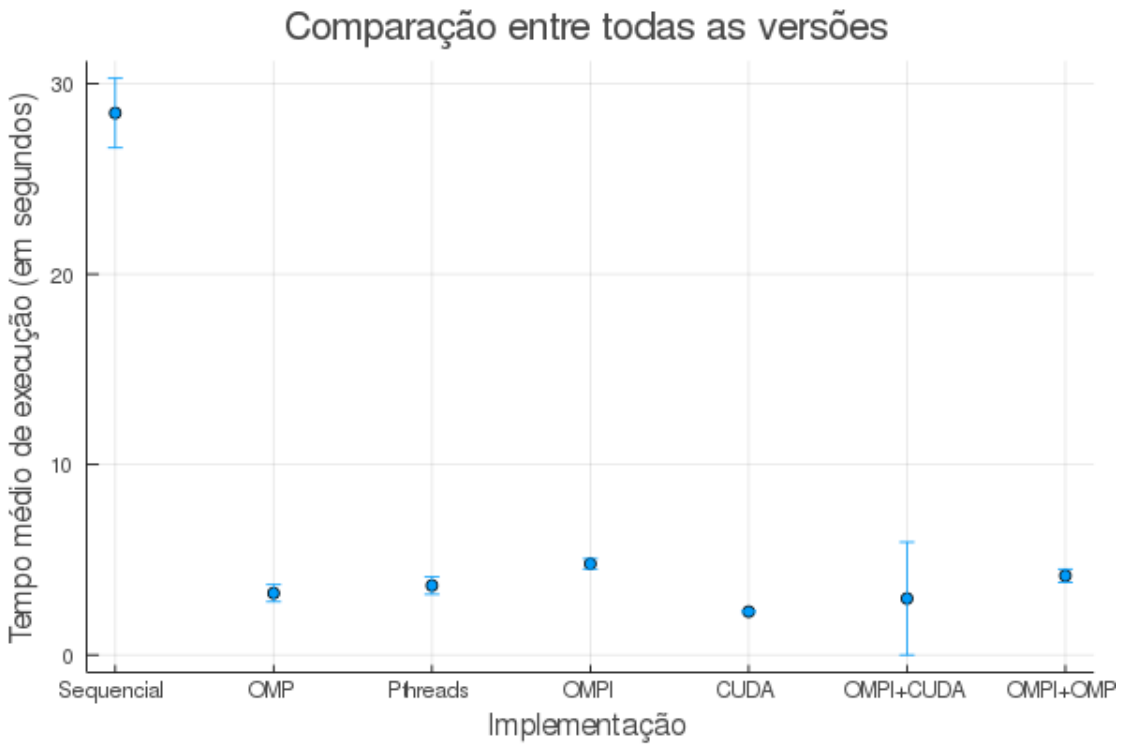
```
In [37]: melhores = DataFrame(nomes = names, duracao= duration, ci = ci_d)
```

Out[37]:

	nomes	duracao	ci
	String	Float64	Float64
1	Sequencial	28.4608	1.82404
2	OMP	3.24436	0.44773
3	Pthreads	3.6455	0.463688
4	OMPI	4.78659	0.268464
5	CUDA	2.27246	0.0727005
6	OMPI+CUDA	2.96337	2.96337
7	OMPI+OMP	4.16466	0.349235

```
In [38]: scatter(melhores.nomes,
                melhores.duracao,
                yerror = melhores.ci,
                legend =false,
                fmt = :png,
                ylabel = "Tempo médio de execução (em segundos)",
                xlabel="Implementação",
                title = "Comparação entre todas as versões")
```

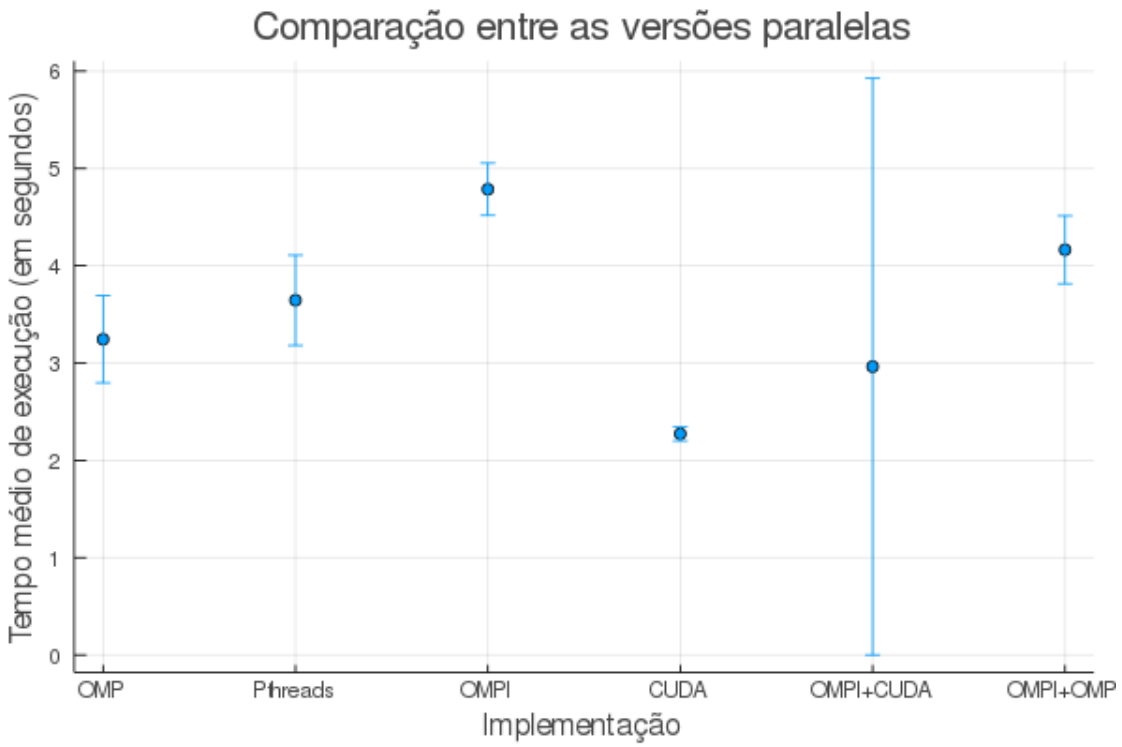
Out[38]:



Temos que todas as versões apresentam um ganho significativo de performance comparando-se com a versão sequencial, o que era esperado. Comparando melhor entre as versões paralelas, temos:

```
In [39]: paralelos = melhores[2:7, :]
         scatter(paralelos.nomes,
                 paralelos.duracao,
                 fmt = :png,
                 yerror = paralelos.ci,
                 legend =false,
                 ylabel = "Tempo médio de execução (em segundos)",
                 xlabel="Implementação",
                 title = "Comparação entre as versões paralelas")
```

```
Out[39]:
```



CONCLUSÃO:

Como esperado, e pode ser visto no primeiro dos dois gráficos acima, a versão sequencial possui um desempenho muito inferior ao das demais implementações.

Com o segundo gráfico, fica evidente que a versão com CUDA possui o melhor desempenho entre todas as implementações, sendo a que forneceu menor tempo médio de execução e, além disso, o menor intervalo de confiança associado a esta média. Tal constatação se sustenta quando consideramos que esta implementação é a que envolve a maior facilidade de adaptação da parte mais intensiva do problema. Isto pois a computação do valor de cada pixel da imagem, ao ser atribuida como tarefa de computação para a GPU, consegue processar milhares de operações simples como essas ao mesmo tempo, atingindo uma performance que não conseguimos alcançar mesmo com as diversas técnicas para retiramos o máximo de performance de uma CPU.

Temos que as versões de Pthreads e OMP implementadas no EP1 tem uma performance similar, com as otimizações implementadas na utilização de OMP em laços provavelmente sendo o fator diferencial que fazem com que a implementação em OMP tenha um desempenho ligeiramente melhor que a com Pthreads.

A versão de OMPI teve o pior desempenho entre as versões paralelas, possivelmente devido ao *overhead* da criação de vários processos ser superior ao *overhead* da criação de threads que é feita pelos outros métodos. Provavelmente se tivessemos utilizados várias máquinas, que juntas tivessem um poder computacional equivalente, ao invés de apenas uma, os ganhos de desempenho fornecidos pelo OMPI seriam ainda mais significativos. Cabe ressaltar também que essa é a única ferramenta que possibilitaria esse tipo de arquitetura fácilmente.

Entre as versões que se utilizaram de OMPI e outra ferramenta, temos que a combinação do OMPI com OMP gerou um pequeno ganho de performance devido a paralelilzação "interna" de cada processo em conjunto com os diversos processos criados. No entanto, a pequena escala do

problema provavelmente não possibilitou que os reais ganhos de desempenhos se mostrassem, visto que o *overhead* começa a se tornar significativo.

A versão de OMPI com CUDA implementada se mostrou instável, e os experimentos acabaram não sendo conclusivos devido á grande variância dos resultados - refletida na grande amplitude do intervalo de confiança associado ao tempo médio de execução. Podemos ver, no entanto, que o *overhead* da utilização de OMPI de certo modo compensou a utilização da GPU, o que provavelmente se deve a escala pequena do problema que estamos tratando

```
In [ ]:
```