EP2

July 10, 2020

1 MAC0219 EP2 - Cuda & OpenMPI

Nome	NUSP
Eduardo Freire de Carvalho Lima	10262627
Kaique Kazuyoshi Komata	10297741
Lucas Civile Nagamine	7083142
Victor Hugo Miranda Pinto	10297720
Yurick Yussuke Honda	10258984

1.1 Experimento

O objetivo do experimento é utilizar a experiência com análise de desempenho de programas paralelos, adquirida no EP1, para planejar e analisar experimentos para determinar parâmetros de configuração de implementações CUDA e OMPI do cálculo do fractal de Mandelbrot.

O experimento é composto pelos códigos implementados no EP1 (sequencial, Pthreads e OpenMP) e três novas versões: Cuda, OMPI e OMPI + OpenMP. Todas as execuções devem respeitar alguns parâmetros fixos: repetições = 15, tamanho da imagem = 4096 e região do conjunto de Mandelbrot = Triple Spiral Valley. Dessa maneira, as variáveis para cada uma das versões a ser comparadas são:

- Sequencial (mandelbrot_seq.c e mandelbrot_seq_sem.c): presença e ausência de alocação de memória e operações de entrada/saída;
- Pthreads (mandelbrot_pth.c): número de threads (2^0 a 2^5);
- OpenMP (mandelbrot_omp.c): número de threads (2^0 a 2^5);
- Cuda (mandelbrot_cuda.cu): dimensões quadradas do grid (x, y) e dos blocos ($(1, 1)$, $(2, 2)$, $(4, 4)$, $(8, 8)$, $(12, 12)$, $(16, 16)$, $(24, 24)$, $(32, 32)$, $(48, 48)$, $(64, 64)$);
- OMPI (mandelbrot_ompi.c): número de processos ($1, 8, 16, 32$ e 64);
- OMPI + OpenMP (mandelbrot_mpi_ompi.c): número de processos ($1, 8, 16, 32$ e 64) e número de threads ($2^0 a 2^5$).

Para automatizar alguns dos processos deste experimento, utilizamos os shell scripts run_measurements.sh, ompi_script.sh e cuda_script.sh.

Os arquivos referentes aos cálculos do conjunto de Mandelbrot (*.c e* .cu) que foram entregues são executados com alocação de memória e geração de imagem. Entretanto, para os experimentos foram usadas versões que não possuem a alocação de memória e a geração de imagem para que fosse medida apenas as performances dos cálculos.

Além disso, para compilar os arquivos basta executar o Makefile enviado com o comando make.

1.2 Relatório

Primeiramente, rodamos o comando na célula abaixo para instalar e atualizar os pacotes e dependências necessários para a execução dos scripts desse notebook:

[15]:
```
] up
```

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


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


  Updating
`~/Documents/College/EPs/MAC0219/EP2/Project.toml`
 [no changes]
  Updating
`~/Documents/College/EPs/MAC0219/EP2/Manifest.toml`
 [no changes]
```

Depois, rodamos a célula abaixo para verificar os status dos pacotes e ver se há algum problema neles:

[16]:
```
] st
```

```
Status `~/Documents/College/EPs/MAC0219/EP2/Project.toml`
  [336ed68f] CSV v0.7.3
  [a93c6f00] DataFrames v0.21.4
  [91a5bcdd] Plots v1.5.4
```

Após isso, entramos no modo shell para rodar o script run_measurements.sh, que irá compilar todos os arquivos em C que calculam o Conjunto de Mandelbrot, executar todos os experimentos necessários para os arquivos do EP1 e calcular os tempos de execução com o comando perf stat do Linux. Como o script roda todos os experimentos de uma única vez, sua execução pode demorar algum tempo:

; ./run_measurements.sh

Ainda no modo shell, rodamos o script ompi_script.sh para calcular o Conjunto de Mandelbrot e executar os experimentos necessários para o programa paralelizado com OMPI e para o programa paralelizado com OMPI + OpenMP. A execução dos experimentos é extensa e pode demorar algumas horas:

; ./ompi_script.sh

Devido à necessidade do uso da gpu, o script cuda_script.sh referente aos testes para o cálculo do Conjunto de Mandelbrot do código paralelizado com CUDA foi executado numa máquina da Rede Linux do IME-USP. Por conta disso, não foi possível adicionar a célula da execução para esse script

como feito com os outros. Além disso, diferentemente dos outros, a execução dos testes da versão CUDA não levou uma quantidade grande de tempo.

Na célula abaixo é definida a função confidence_interval, que faz o cálculo de intervalo de confiança e irá ser utilizada na manipulação dos resultados e na construção do DataFrame.

[17]:
```
function confidence_interval(sd)
    return Float64(1.96) * sd / Float64(sqrt(10))
end
```

[17]:
```
confidence_interval (generic function with 1 method)
```

O script run_measurements criará pastas com arquivos log que contém os resultados dos experimentos relativos às implementações do EP1. Para podermos analisar esses resultados e produzir gráficos, executamos o código da célula abaixo para ler todos os arquivos log, extrair as informações que queremos e criar um CSV (results_ep1.csv) a partir delas.

```
using Printf

function mean_and_ci_from_line(line)
    mean, ci, count = 0, 0, 0
    for token in split(line, " ")
        num = tryparse(Float64, replace(replace(token, "%" => ""), "," => "."))
        if num !== nothing
            count += 1
            if count == 1
                mean = num
            else
                sd = mean * num / Float64(100)
                ci = confidence_interval(sd)
            end
        end
        if count >= 2 break end
    end
    return mean, ci
end

function log_to_csv_ep1(script, csv_file, threads_enabled)
    log_path = @sprintf("results/%s.log", script)
    threads = 1
    mean, ci = 0, 0

    open(log_path) do log_file
        for (i, line) in enumerate(eachline(log_file))
            if (i + 1) % 5 != 0 continue end

            mean, ci = mean_and_ci_from_line(line)

            write(csv_file, @sprintf("%s,%d,%.9F,%.9F\n", script, threads, mean, ci))
```

```
            if threads_enabled threads *= 2 end
        end
    end
end

function log_to_csv_handler_ep1(scripts)
    csv_file = open("results_ep1.csv", "w")
    write(csv_file, "script,threads,mean,ci\n")

    for script in scripts
        threads_enabled = !startswith(script, "mandelbrot_seq")
        log_to_csv_ep1(
            script,
            csv_file,
            threads_enabled
        )
    end

    close(csv_file)
end

scripts = ["mandelbrot_seq", "mandelbrot_seq_sem", "mandelbrot_pth", "mandelbrot_omp"]
log_to_csv_handler_ep1(scripts)
```

Então, podemos obter um dataframe a partir do CSV gerado results_ep1.csv.

[18]:
```
using CSV
using DataFrames

df = DataFrame!(CSV.File("results_ep1.csv"))
```

[18]:

	script	threads	mean	ci
	String	Int64	Float64	Float64
1	mandelbrot_seq	1	24.0919	0.0268782
2	mandelbrot_seq_sem	1	23.1015	0.0114548
3	mandelbrot_pth	1	23.3108	0.00433445
4	mandelbrot_pth	2	12.7285	0.00394459
5	mandelbrot_pth	4	8.21261	0.0066173
6	mandelbrot_pth	8	7.77238	0.0101165
7	mandelbrot_pth	16	7.79411	0.0251204
8	mandelbrot_pth	32	7.7193	0.00526292
9	mandelbrot_omp	1	23.2023	0.00287619
10	mandelbrot_omp	2	12.6656	0.00235507
11	mandelbrot_omp	4	7.69942	0.0167025
12	mandelbrot_omp	8	7.62807	0.0189117
13	mandelbrot_omp	16	7.59968	0.00753653
14	mandelbrot_omp	32	7.63577	0.00804559

E, então, repetimos o mesmo processo para a implementação OMPI:

```
using Printf

function log_to_csv_ompi(script, csv_file)
    log_path = @sprintf("results/%s.log", script)
    processes = [1, 8, 16, 32, 64]
    mean, ci = 0, 0
    j = 1

    open(log_path) do log_file
        for (i, line) in enumerate(eachline(log_file))
            if (i + 1) % 5 != 0 continue end

            mean, ci = mean_and_ci_from_line(line)

            write(csv_file, @sprintf("%s,%d,%.9F,%.9F\n", script, processes[j], mean, ci))
            j += 1
        end
    end
end

function log_to_csv_handler_ompi(scripts)
    csv_file = open("results_ompi.csv", "w")
    write(csv_file, "script,processes,mean,ci\n")

    for script in scripts
        log_to_csv_ompi(
            script,
            csv_file
        )
    end

    close(csv_file)
end

scripts = ["mandelbrot_ompi"]
log_to_csv_handler_ompi(scripts)
```

E geramos o dataframe relativo aos dados do OMPI presentes no arquivo results_ompi.csv:

[19]:
```
using CSV
using DataFrames

df_ompi = DataFrame!(CSV.File("results_ompi.csv"))
```

[19]:

	script	processes	mean	ci
	String	Int64	Float64	Float64
1	mandelbrot_ompi	1	30.7906	0.0229011
2	mandelbrot_ompi	8	18.0055	0.161819
3	mandelbrot_ompi	16	17.9517	0.164674
4	mandelbrot_ompi	32	19.1304	0.216986
5	mandelbrot_ompi	64	23.8729	0.250062

O mesmo processo para OMPI + OMP:

```
using Printf

function log_to_csv_ompi_omp(script, csv_file)
    log_path = @sprintf("results/%s.log", script)
    processes = [1, 8, 16, 32, 64]
    threads = [1, 2, 4, 8, 16, 32]
    mean, ci = 0, 0
    k = 1
    j = 1

    open(log_path) do log_file
        for (i, line) in enumerate(eachline(log_file))
            if (i + 1) % 5 != 0 continue end

            mean, ci = mean_and_ci_from_line(line)

            write(csv_file, @sprintf("%s,%d,%d,%.9F,%.9F\n", script, processes[j], threads[k],
            k += 1
            if k == 7
                k = 1
                j += 1
            end
        end
    end
end

function log_to_csv_handler_ompi_omp(scripts)
    csv_file = open("results_mpi_ompi.csv", "w")
    write(csv_file, "script,processes,threads,mean,ci\n")

    for script in scripts
        log_to_csv_ompi_omp(
            script,
            csv_file
        )
    end

    close(csv_file)
end
```

```
scripts = ["mandelbrot_mpi_ompi"]
log_to_csv_handler_ompi_omp(scripts)
```

Obtemos o dataframe a partir do arquivo results_mpi_ompi.csv.

[20]:
```
using CSV
using DataFrames

df_mpi_ompi = DataFrame!(CSV.File("results_mpi_ompi.csv"))
```

[20]:

	script	processes	threads	mean	ci
	String	Int64	Int64	Float64	Float64
1	mandelbrot_mpi_ompi	1	1	26.3924	0.376238
2	mandelbrot_mpi_ompi	1	2	25.2983	0.00627201
3	mandelbrot_mpi_ompi	1	4	25.2817	0.00470093
4	mandelbrot_mpi_ompi	1	8	25.2812	0.00470083
5	mandelbrot_mpi_ompi	1	16	25.2759	0.00469986
6	mandelbrot_mpi_ompi	1	32	25.2386	0.00312861
7	mandelbrot_mpi_ompi	8	1	9.27202	0.0155165
8	mandelbrot_mpi_ompi	8	2	9.343	0.034166
9	mandelbrot_mpi_ompi	8	4	9.29478	0.0207395
10	mandelbrot_mpi_ompi	8	8	9.26943	0.0298753
11	mandelbrot_mpi_ompi	8	16	9.22227	0.0131469
12	mandelbrot_mpi_ompi	8	32	9.22651	0.021159
13	mandelbrot_mpi_ompi	16	1	9.01874	0.0670785
14	mandelbrot_mpi_ompi	16	2	8.88542	0.0363478
15	mandelbrot_mpi_ompi	16	4	9.05891	0.0662543
16	mandelbrot_mpi_ompi	16	8	8.99984	0.0513191
17	mandelbrot_mpi_ompi	16	16	9.05741	0.0538929
18	mandelbrot_mpi_ompi	16	32	8.89437	0.0154358
19	mandelbrot_mpi_ompi	32	1	9.30472	0.0415233
20	mandelbrot_mpi_ompi	32	2	9.38614	0.0343238
21	mandelbrot_mpi_ompi	32	4	9.35379	0.0347852
22	mandelbrot_mpi_ompi	32	8	9.18694	0.0335953
23	mandelbrot_mpi_ompi	32	16	9.17926	0.0421013
24	mandelbrot_mpi_ompi	32	32	9.30503	0.047292
25	mandelbrot_mpi_ompi	64	1	10.2949	0.0338184
26	mandelbrot_mpi_ompi	64	2	10.4464	0.0284888
27	mandelbrot_mpi_ompi	64	4	10.3538	0.0391458
28	mandelbrot_mpi_ompi	64	8	10.4752	0.0292166
29	mandelbrot_mpi_ompi	64	16	10.3244	0.0332755
30	mandelbrot_mpi_ompi	64	32	10.3024	0.0325659

E, por fim, para Cuda:

```
using Printf

function log_to_csv_cuda(script, csv_file)
    log_path = @sprintf("results/%s.log", script)
```

```
    grid = [1, 2, 4, 8, 12, 16, 24, 32, 48, 64]
    mean, ci = 0, 0
    j = 1

    open(log_path) do log_file
        for (i, line) in enumerate(eachline(log_file))
            if (i + 1) % 5 != 0 continue end

            mean, ci = mean_and_ci_from_line(line)

            write(csv_file, @sprintf("%s,%d,%.9F,%.9F\n", script, grid[j], mean, ci))
            j += 1
        end
    end
end

function log_to_csv_handler_cuda(scripts)
    csv_file = open("results_cuda.csv", "w")
    write(csv_file, "script,grid,mean,ci\n")

    for script in scripts
        log_to_csv_cuda(
            script,
            csv_file
        )
    end

    close(csv_file)
end

scripts = ["mandelbrot_cuda"]
log_to_csv_handler_cuda(scripts)
```

E obtemos o dataframe a partir do arquivo results_cuda.csv.

[21]:
```
using CSV
using DataFrames

df_cuda = DataFrame!(CSV.File("results_cuda.csv"))
```

[21]:

	script	grid	mean	ci
	String	Int64	Float64	Float64
1	mandelbrot_cuda	1	2.25133	2.2047e-5
2	mandelbrot_cuda	2	1.19167	8.937e-6
3	mandelbrot_cuda	4	0.68676	2.3922e-5
4	mandelbrot_cuda	8	0.62615	3.1862e-5
5	mandelbrot_cuda	12	0.98198	1.7407e-5
6	mandelbrot_cuda	16	1.25094	2.8455e-5
7	mandelbrot_cuda	24	1.72821	5.4415e-5
8	mandelbrot_cuda	32	2.26355	4.9665e-5
9	mandelbrot_cuda	48	3.27187	6.6516e-5
10	mandelbrot_cuda	64	4.38169	0.000157516

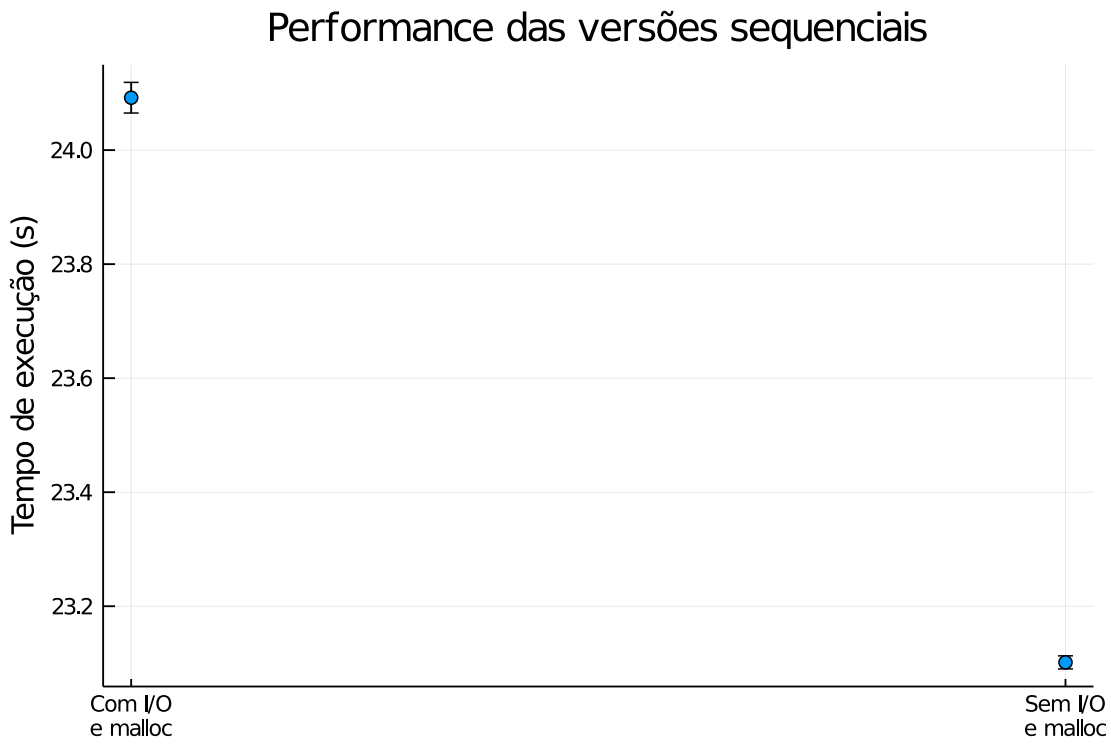
1.3 Análise

Após construir os DataFrames com os dados, podemos iniciar a construção dos gráficos e análise deles.

Primeiramente será feita uma análise separada para cada código executado, comparando os resultados nos diferentes parâmetros e depois, será feita uma análise comparando os resultados entre os códigos, relacionando-os.

1.3.1 Sequencial

[22]:
```
using Plots

function plot_seq(data)
    plt = plot(
        title = "Performance das versões sequenciais",
        ylabel = "Tempo de execução (s)",
    )
    scatter!(
        ["Com I/O\ne malloc", "Sem I/O\ne malloc"],
        data.mean,
        yerror = data.ci,
        label = nothing,
        position = :center
    )
    display(plt)
end

plot_seq(filter(row -> startswith(row.script, "mandelbrot_seq"), df))
```



Entre as duas versões sequenciais testadas, há apenas uma diferença: a presença/ausência de alocação de memória e operações de entrada e saída. Antes de executarmos os testes, e com base na experiência que adquirimos no EP1, previmos que a versão com essas operações naturalmente apresentaria um tempo de execução ligeiramente maior.

Nossa previsão mostrou-se correta: a versão com malloc e I/O apresentou um tempo médio de 24.0919 segundos, enquanto a implementação sem malloc e I/O levou um tempo médio 4.11% menor, 23.1015 segundos.

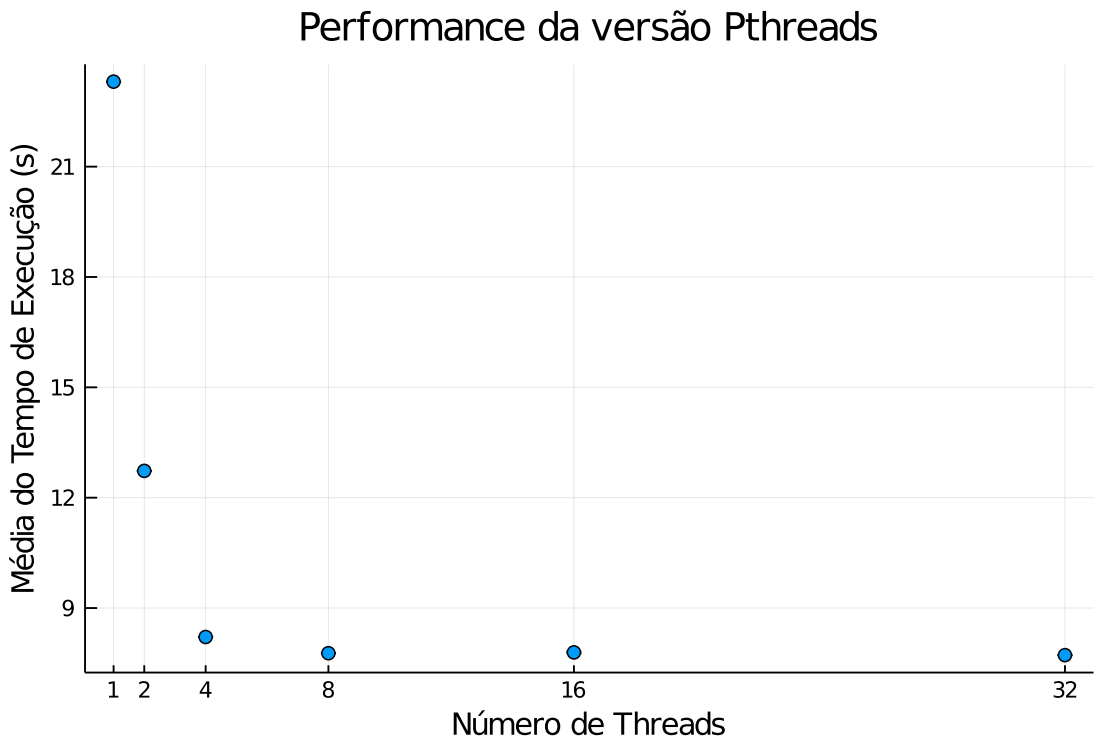
1.3.2 Pthreads

Como os parâmetros para o experimento são fixos dessa vez, com exceção do número de threads, iremos analisar os resultados do programa paralelizado com pthreads focando apenas nas médias do tempo de execução e no número de threads.

A função plot_pth definida abaixo recebe um conjunto de dados e constrói um gráfico que relaciona a média do tempo de execução no eixo Y com o número de threads no eixo X:

[23]:
```
using Plots
function plot_pth(data, name)
    fig = plot(xlabel = "Número de Threads", ylabel = "Média do Tempo de␣
 ↪Execução (s)",
```

```
        legend = :topright, title = name, xticks=[1, 2, 4, 8, 16, 32])
    scatter!(
        data.threads,
        data.mean,
        yerror = data.ci,
        label = nothing,
    )
    display(fig)
end

plot_pth(filter(row -> startswith(row.script, "mandelbrot_pth"), df),␣
 ↪"Performance da versão Pthreads")
```



Para a implementação com Pthreads, as execuções variaram apenas no número de threads. Os valores foram 1, 2, 4, 8, 16 e 32. Com base na experiência que adquirimos no EP1, sabíamos que, **até certo limite**, quanto mais threads utilizarmos, menos tempo será gasto no total. No entanto, após uma determinada quantidade de threads, é possível chegar a um ponto em que o overhead de tempo para gerenciá-las ultrapassa o ganho de tempo obtido com seu uso.

Pelos experimentos anteriores, sabíamos que o valor máximo de threads testado, 32, não atingiria esse limite e, portanto, conforme aumentamos o número de threads, o tempo médio gasto com a execução do programa seria menor.

Ao executarmos os testes, essas predições se confirmaram: de fato, a cada aumento

no número de threads, o tempo de execução foi reduzido. Com 1 thread, gastou-se 23.3108 segundos, enquanto com 32 threads chegamos ao valor mínimo de 7.7193 segundos. Uma exceção a essa "escala decrescente" foi de 8 para 16 threads, em que o tempo aumentou de 7.77238 s para 7.79411 s. No entanto, se levarmos em consideração o intervalo de confiança 95% dos dois casos, é possível (e provável) que o valor para 16 threads seja de fato menor que para 8 threads.

É válido destacar que, a cada aumento no número de threads, o ganho de tempo se torna menor, conforme mostra a tabela abaixo:

Aumento no número de threads	Variação de tempo
De 1 para 2	-44.96%
De 2 para 4	-35.48%
De 4 para 8	-5.36%
De 8 para 16	+0.28%
De 16 para 32	-0.96%

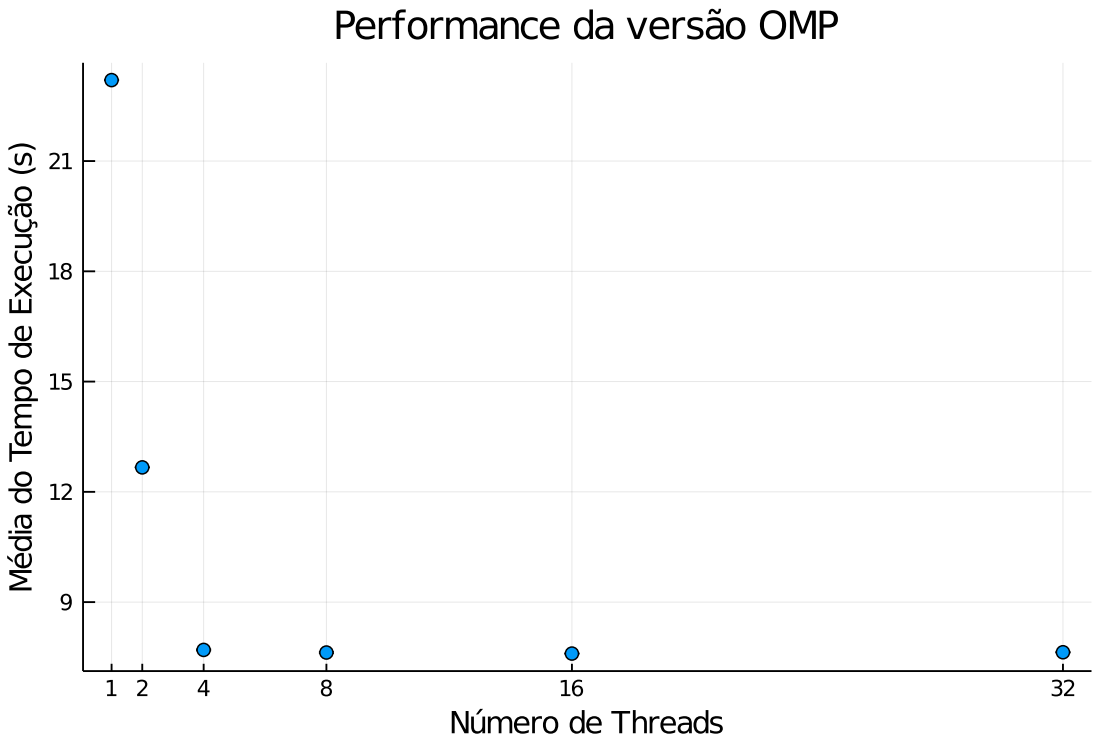
Isso se dá devido a alguns motivos, dentre os quais se destacam o overhead de tempo obtido quando é preciso gerenciar um número maior de threads e o fato de o número de threads exceder a quantidade de núcleos físicos, o que não permite um ganho de mesma proporção no tempo.

1.3.3 OpenMP

Similarmente à seção de Pthreads, nosso experimento com OpenMP se restringe aos testes com diferentes quantidades de threads.

A função plot_omp definida abaixo recebe um conjunto de dados e constrói um gráfico que relaciona a média do tempo de execução no eixo Y com o número de threads no eixo X:

[24]:
```
using Plots
function plot_omp(data, name)
    fig = plot(xlabel = "Número de Threads", ylabel = "Média do Tempo de␣
 ↪Execução (s)",
        legend = :topright, title = name, xticks=[1, 2, 4, 8, 16, 32])
    scatter!(
        data.threads,
        data.mean,
        yerror = data.ci,
        label = nothing,
    )
    display(fig)
end

plot_omp(filter(row -> startswith(row.script, "mandelbrot_omp"), df),␣
 ↪"Performance da versão OMP")
```



Como já comentamos na seção de Pthreads sobre nossas expectativas quanto à relação entre quantidade de threads utilizada e o tempo de execução, nos abstemos de repeti-las aqui. O que vale ser destacado é que os resultados se mostraram bastante semelhantes, com reduções nos tempos de execução observados conforme aumentamos a quantidade de threads.

Dois pontos merecem ser abordados à parte:

1. É interessante observar a performance ligeiramente melhor da versão com OMP comparada a Pthreads, como podemos observar nesta tabela:

Número de threads	Tempo de OMP em relação a Pthreads
1	-0,47%
2	-0,49%
4	-6,24%
8	-1,86%
16	-2.49%
32	-1,08%

2. A implementação com OpenMP registrou uma piora de tempo de 16 para 32 threads (um aumento médio de 7.59968 s para 7.63577 s), que se mantém mesmo quando levamos os intervalos de confiança em consideração. Isso pode significar que, a partir de 32 threads, a versão com OMP já começa a apresentar uma piora em desempenho - dados os motivos listados no fim da

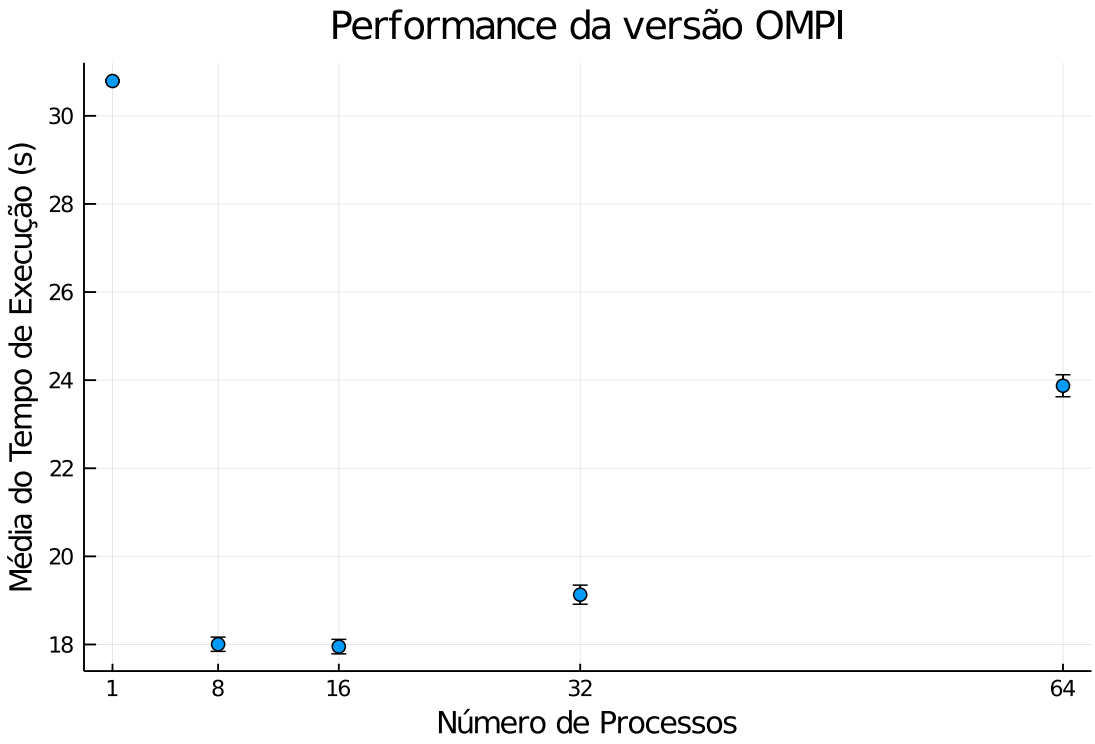
última seção. É possível, mas improvável, que esse aumento de tempo seja devido apenas a um desvio atípico dos dados em nosso experimento.

1.3.4 OMPI

Na versão OMPI, assim como nas outras, a maioria dos parâmetros experimentados é fixa. Os testes se restringem, portanto, à quantidade de processos utilizados.

A função plot_ompi definida abaixo recebe um conjunto de dados e constrói um gráfico que relaciona a média do tempo de execução no eixo Y com o número de processos no eixo X:

[25]:
```
using Plots
function plot_ompi(data, name)
    fig = plot(xlabel = "Número de Processos", ylabel = "Média do Tempo de␣
 ↪Execução (s)",
        legend = :topright, title = name, xticks=[1,8,16,32,64])
    scatter!(
        data.processes,
        data.mean,
        yerror = data.ci,
        label = nothing,
    )
    display(fig)
end

plot_ompi(df_ompi, "Performance da versão OMPI")
```



Nos resultados do experimento, a melhor performance no cálculo do conjunto de mandelbrot com base nas médias dos tempos de execução apresentada foi a execução com 16 processos com 17,95 segundos, enquanto a pior foi a execução com 1 processo com 30,79 segundos.

Observando os resultados dos gráficos é possível perceber um resultado peculiar. Diferentemente do esperado, a performance do cálculo do conjunto de mandelbrot não se mostra proporcional à quantidade de processos usados no cálculo.

É possível visualizar que a média do tempo de execução vai diminuindo à medida que a quantidade de processos vai aumentando. Entretanto, esse comportamento se altera depois de 16 processos, com a média do tempo de execução tendendo a aumentar à medida que o número de processos também aumenta.

Observando a tabela abaixo com as porcentagens das variações das médias dos tempo de execução do cálculo do conjunto de mandelbrot nas transições entre os números de processos, é possível observar o comportamento descrito anteriormente:

Aumento no número de processos	Variação da média de tempo
De 1 para 8	-41,52%
De 8 para 16	-0,29%
De 16 para 32	+6,56%
De 32 para 64	+24,79%
De 1 para 16	-41,69%

Aumento no número de processos	Variação da média de tempo
De 16 para 64	+32,98%

Na tabela acima também é possível observar a variação no tempo médio de execução do pior parâmetro (1 processo) para o tempo médio de execução do melhor parâmetro (16 processos). Além disso, pode se observar que a variação na performance da execução com 16 processos comparada com a de 64 processos é consideravelmente alta, com um aumento em segundos de ~33%.

O comportamento de aumento no tempo médio de execução em números de processos maiores pode se dar devido à um overhead de comunicação maior entre os processos (um processo precisa se comunicar/esperar por vários outros processos) ou às características e especificações da máquina onde os testes foram feitos (a máquina pode não lidar muito bem com uma paralelização muito alta).

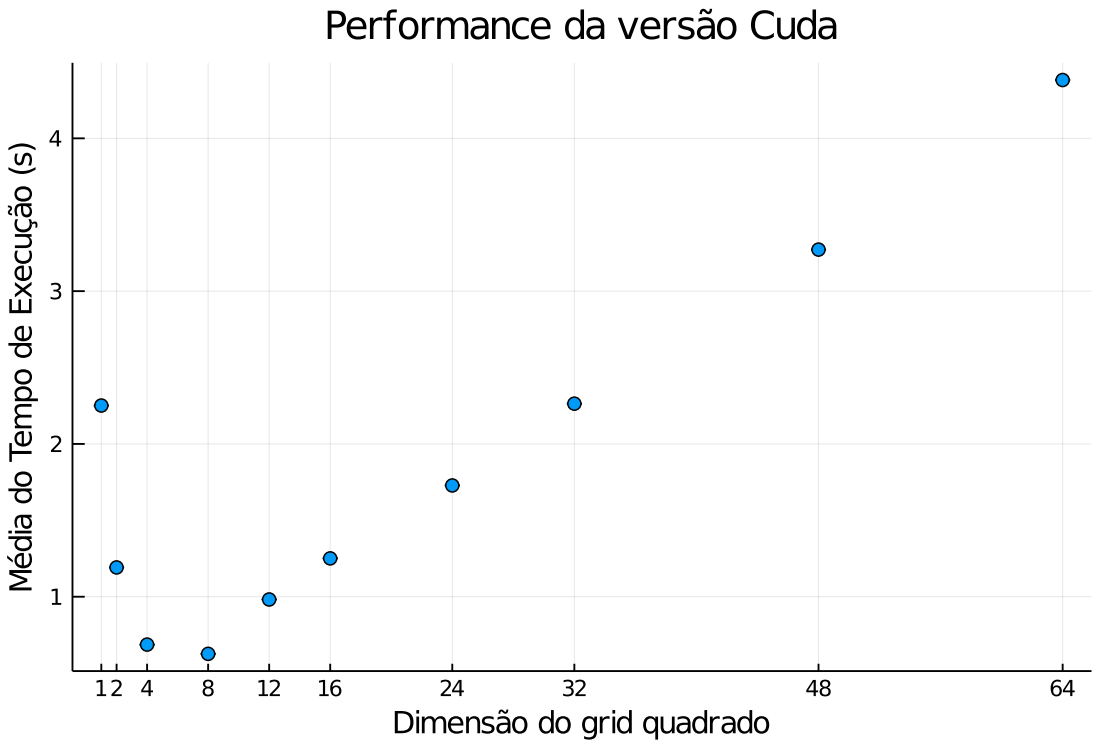
É válido ressaltar que mesmo com a performance piorando nos números de processos maiores, esta ainda se mostra melhor do que a performance na execução com apenas 1 processo, tendo uma média de 23,87 segundos para a execução com 64 processos comparado com os 30,79 segundos da execução usando 1 processo.

1.3.5 CUDA

Na versão CUDA, a região do conjunto e o tamanho da imagem são fixos. A variável de execução, portanto, é apenas a dimensão do grid. Os grids que utilizamos possuem sempre as dimensões X e Y iguais (ou seja, é quadrado).

Dessa maneira, a função plot_cuda definida abaixo recebe um conjunto de dados e constrói gráficos que relacionam as médias dos tempos de execução com a dimensão do grid utilizado:

[26]:
```
using Plots
function plot_cuda(data)
    fig = plot(xlabel = "Dimensão do grid quadrado", ylabel = "Média do Tempo␣
 ↪de Execução (s)",
        legend = :topright, title = "Performance da versão Cuda", xticks=[1, 2,␣
 ↪4, 8, 12, 16, 24, 32, 48, 64])
    scatter!(
        data.grid,
        data.mean,
        yerror = data.ci,
        label = nothing,
    )
    display(fig)
end

plot_cuda(df_cuda)
```



O CUDA possui as dimensões (x,y) do grid como parâmetro, o qual decidimos admití-lo como um quadrado, logo x=y.

Nos resultados do experimento, a melhor performance no cálculo do conjunto de mandelbrot com base nas médias dos tempos de execução apresentada foi a execução com o grid (8.8) com 0,62 segundos, enquanto a pior foi a execução com grid (64,64) com 4,38 segundos.

É possível visualizar que a média do tempo de execução vai diminuindo à medida que o tamanho do grid começa a aumentar. Entretanto, esse comportamento se altera depois do grid (8,8), com a média do tempo de execução tendendo a aumentar à medida que o tamanho do grid também aumenta.

Observando a tabela abaixo com as porcentagens das variações das médias dos tempo de execução do cálculo do conjunto de mandelbrot nas transições entre os números de processos, é possível observar o comportamento descrito anteriormente:

Aumento no número de processos	Variação da média de tempo
De 1 para 2	-47,21%
De 2 para 4	-42,02%
De 4 para 8	-10,15%
De 8 para 12	+55,56%
De 12 para 16	+27,55%
De 16 para 24	+38,41%
De 24 para 32	+30,63%

Aumento no número de processos	Variação da média de tempo
De 32 para 48	+44,69%
De 48 para 64	+33,94%
De 1 para 8	-72,18%
De 8 para 64	+595,24%
De 1 para 64	+94,67%

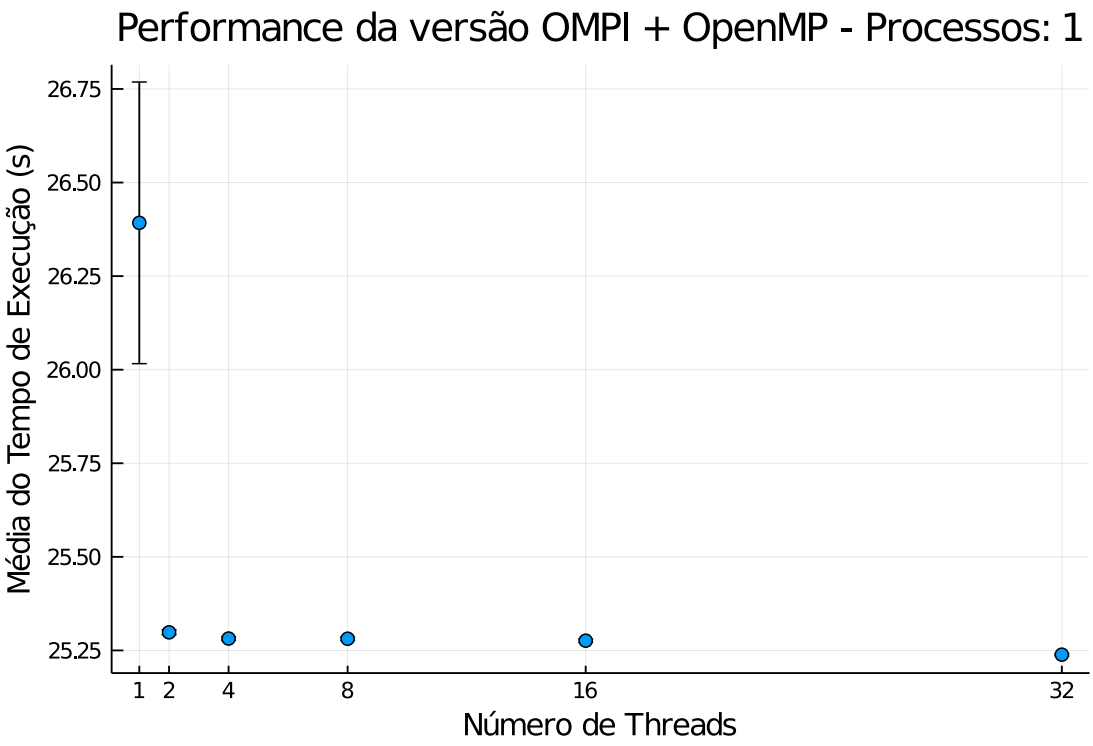
Na tabela acima, também calculamos algumas variações que consideramos relevantes para nossa avaliação. Inicialmente podemos observar uma queda, atingindo variação de ~72% no tempo em segundos entre os grids (1,1) e (8,8). Em seguida, observamos um crescimento contínuo no tempo de execução, chegando a ~600% de aumento entre os grids (8,8) e (64,64), valor extremamente alto. A título de curiosidade, calculamos também a variação entre os grids (1,1) e (64,64) e obtivemos ~95%.

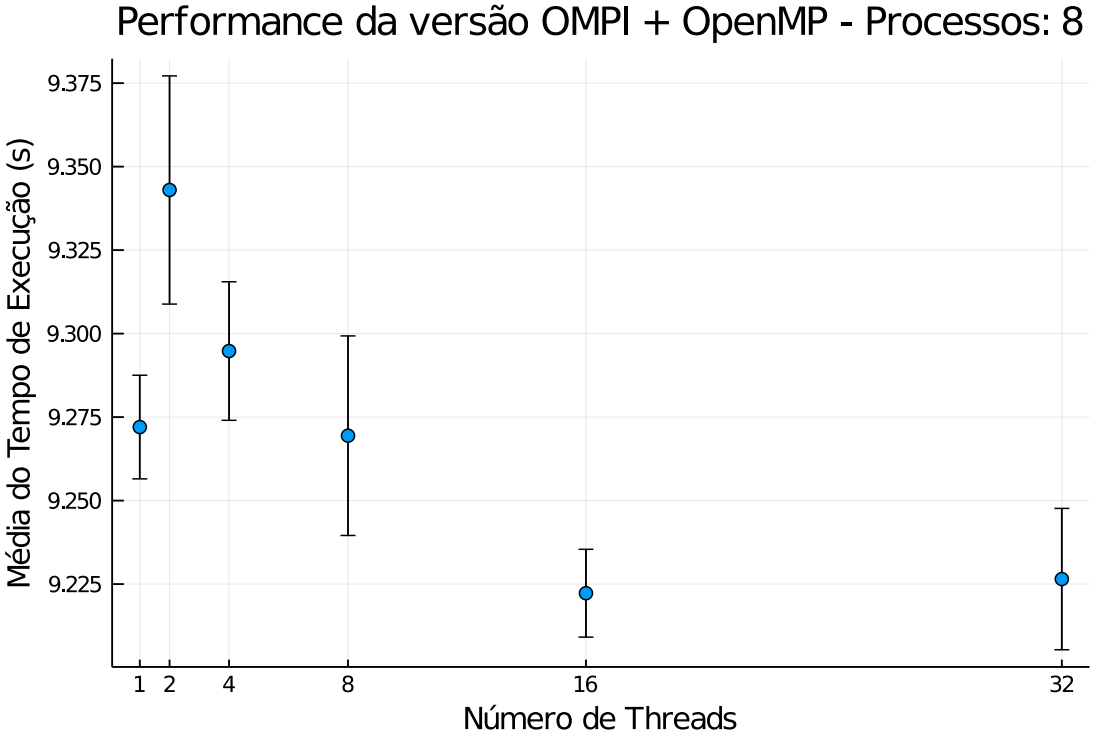
1.3.6 OMPI + OMP

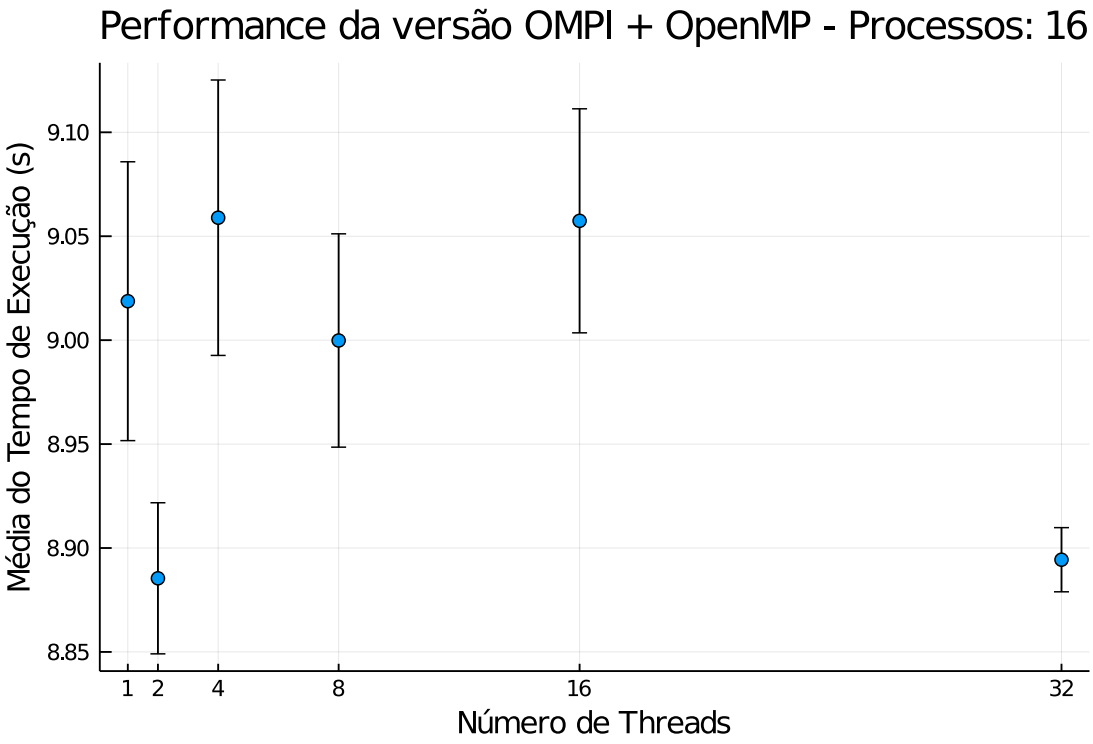
Na versão OMPI + OMP, é levado em consideração dois parâmetros variáveis para os testes: o número de processos e o número de threads. Fora eles, os outros parâmetros são fixos, como nos outros programas.

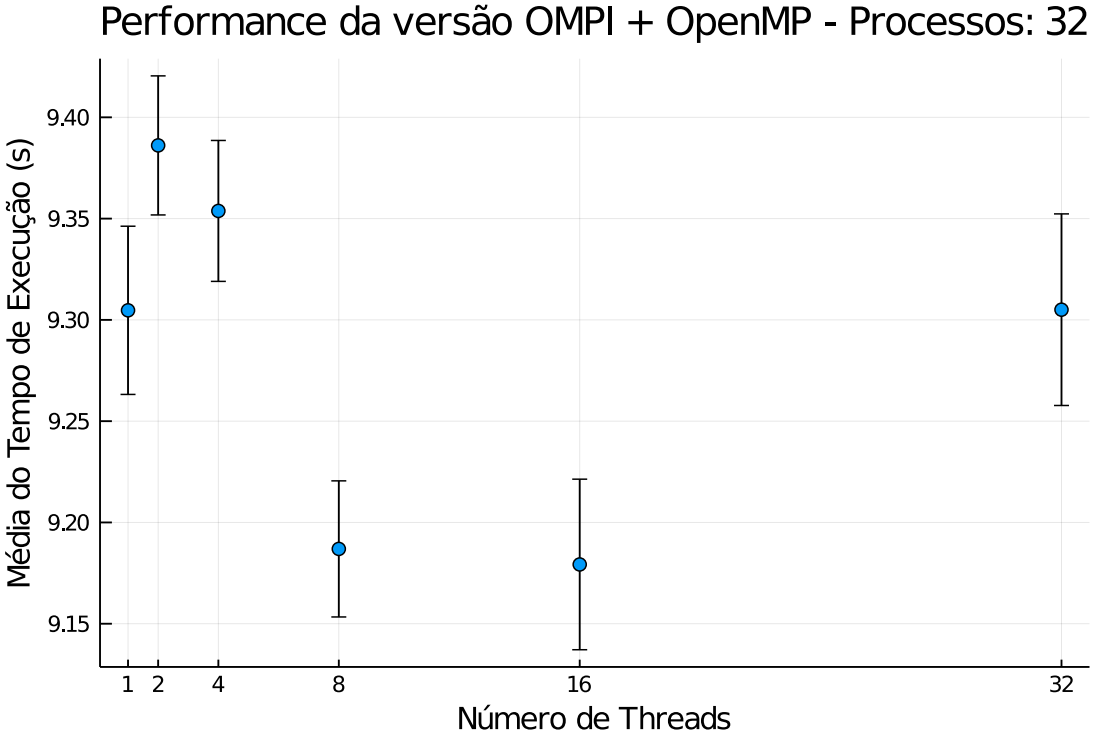
A função plt_mpi_omp definida abaixo recebe um conjunto de dados e constrói gráficos que relacionam as médias dos tempos de execução com o número de threads para cada valor no número de processos:

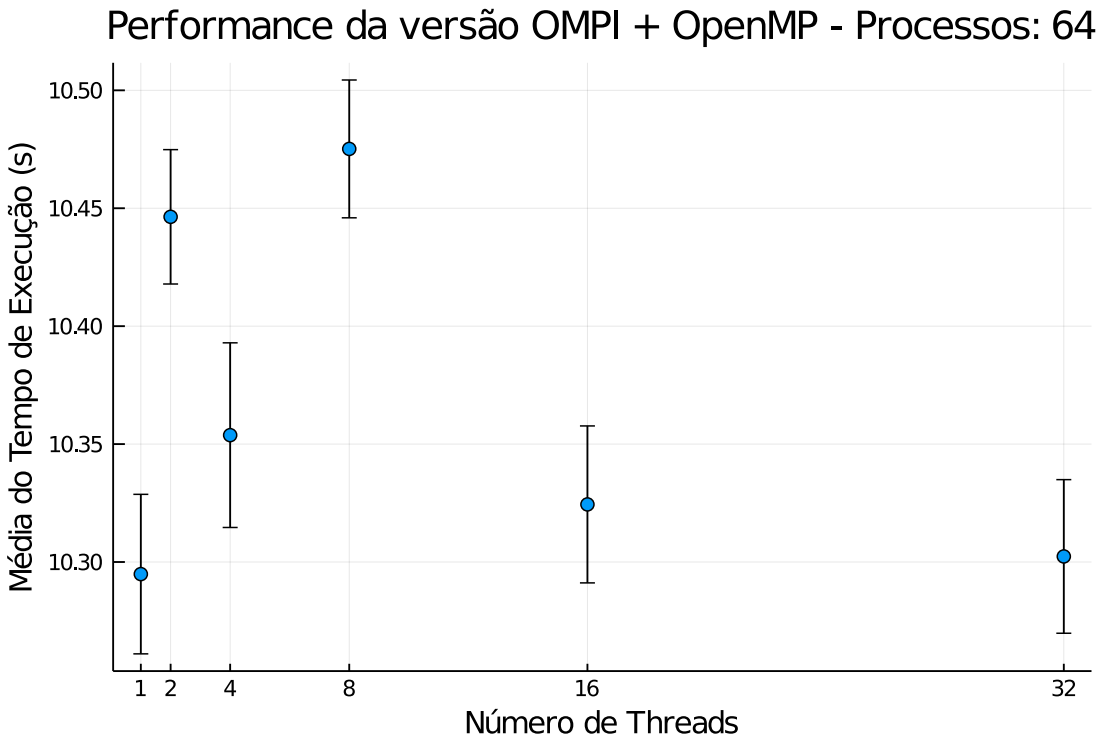
[27]:
```
using Plots
function plot_mpi_omp(data)
    for p in [1,8,16,32,64]
        d = data[data[!, :processes] .== p, :]
        fig = plot(xlabel = "Número de Threads", ylabel = "Média do Tempo de␣
 ↪Execução (s)",
            legend = :topright, title = string("Performance da versão OMPI +␣
 ↪OpenMP - Processos: ", p), xticks=[1,2,4,8,16,32])
        scatter!(
            d.threads,
            d.mean,
            yerror = d.ci,
            label = nothing,
        )
        display(fig)
    end
end

plot_mpi_omp(df_mpi_ompi)
```


Performance da versão OMPI + OpenMP - Processos: 1
Média do Tempo de Execução (s)
26.75
26.50
26.25
26.00
25.75
25.50
25.25
1
2
4
8
16
32
Número de Threads


Performance da versão OMPI + OpenMP - Processos: 8
Média do Tempo de Execução (s)
9.375
9.350
9.325
9.300
9.275
9.250
9.225
1
2
4
8
16
32
Número de Threads


Performance da versão OMPI + OpenMP - Processos: 16
Média do Tempo de Execução (s)
9.10
9.05
9.00
8.95
8.90
8.85
1
2
4
8
16
32
Número de Threads


Performance da versão OMPI + OpenMP - Processos: 32
Média do Tempo de Execução (s)
9.40
9.35
9.30
9.25
9.20
9.15
1
2
4
8
16
32
Número de Threads



Primeiramente, observando em conjunto os resultados dos gráficos, é possível observar que os valores das médias dos tempos de execução das execuções usando mais de 1 processo são bem menores que as médias para as execuções usando apenas 1 processo, para todos os valores de número de threads. Também é possível observar que apenas o gráfico para a execução com 1 processo apresenta um certo tipo de padrão nos resultados, enquanto os outros gráficos possuem seus dados mais dispersos e diferentes.

No gráfico referente à execução com apenas 1 processo, é possível ver o mesmo padrão encontrado na análise do OpenMP, de que o tempo médio de execução é proporcional ao valor do número de threads, diminuindo à medida que o número de threads aumenta. Esse comportamento é condizente com o esperado visto que a execução com 1 processo equivale à execução sem o uso da paralelização com OMPI, se tornando apenas uma execução paralelizada com OpenMP.

Nos gráficos referentes aos outros valores testados de números de processos, os resultados se apresentam de forma mais caótica e não padronizada com variações oscilantes. Algumas performances na execução com números de threads maiores chegam a se mostram piores do que execuções com números de threads mais baixos.

Utilizando o gráfico referente ao experimento com 32 processos como exemplo, é possível visualizar que a performance melhora com 8 e 16 threads comparado com os valores de threads menores, mas que piora consideravelmente ao chegar no valor de 32 threads. Enquanto isso, no gráfico referente ao experimento com 16 processos, temos que as melhores performances são atingidas nas execuções com 2 e 32 threads

e as piores com 4 e 16 threads.

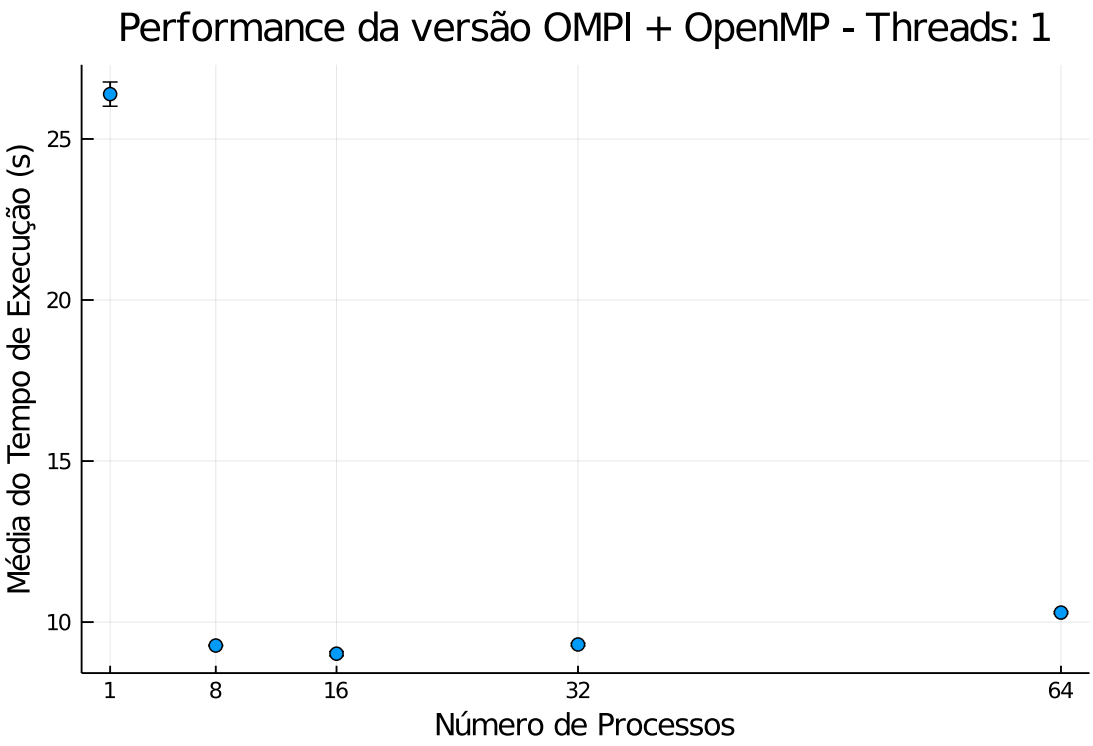
Observando a tabela abaixo, podemos observar que a melhor performance observada foi de 8,88 segundos e foi obtida na execução com 16 processos e 2 threads:

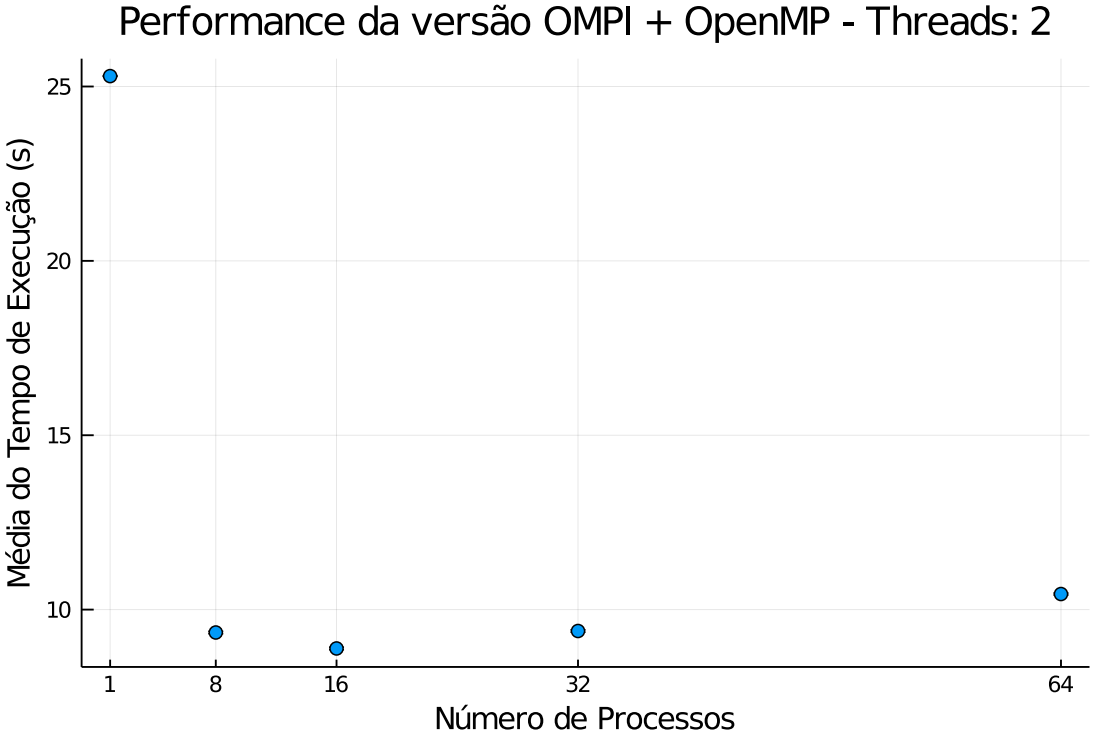
Nº de Processos	Melhor Média de Tempo Observada (s)	Nº de Threads
1	25,23	32
8	9,22	16
16	8,88	2
32	9,17	16
64	10,29	1

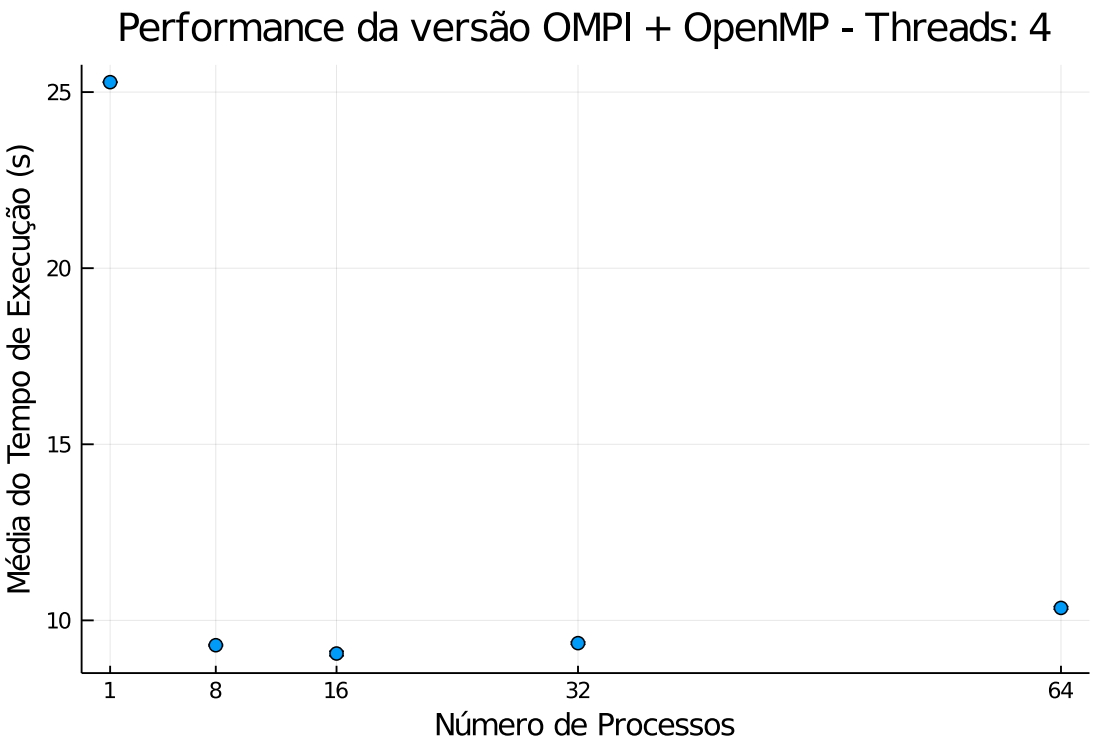
Uma conclusão importante que pode ser obtida desses resultados é que mesmo que as execuções com altos níveis de paralelização com OMPI (números altos de processos) não se mostrem ser as melhores no quesito performance, a diferença delas para as execuções sem nenhum nível dessa paralelização ainda é muito grande. No caso dessa execução OMPI + OpenMP, podemos ver que a pior performance aplicando um número de processos maior que 1 foi de 10,47 segundos, enquanto a melhor performance utilizando apenas 1 processo foi de 25,23 segundos, um aumento de 140,97%.

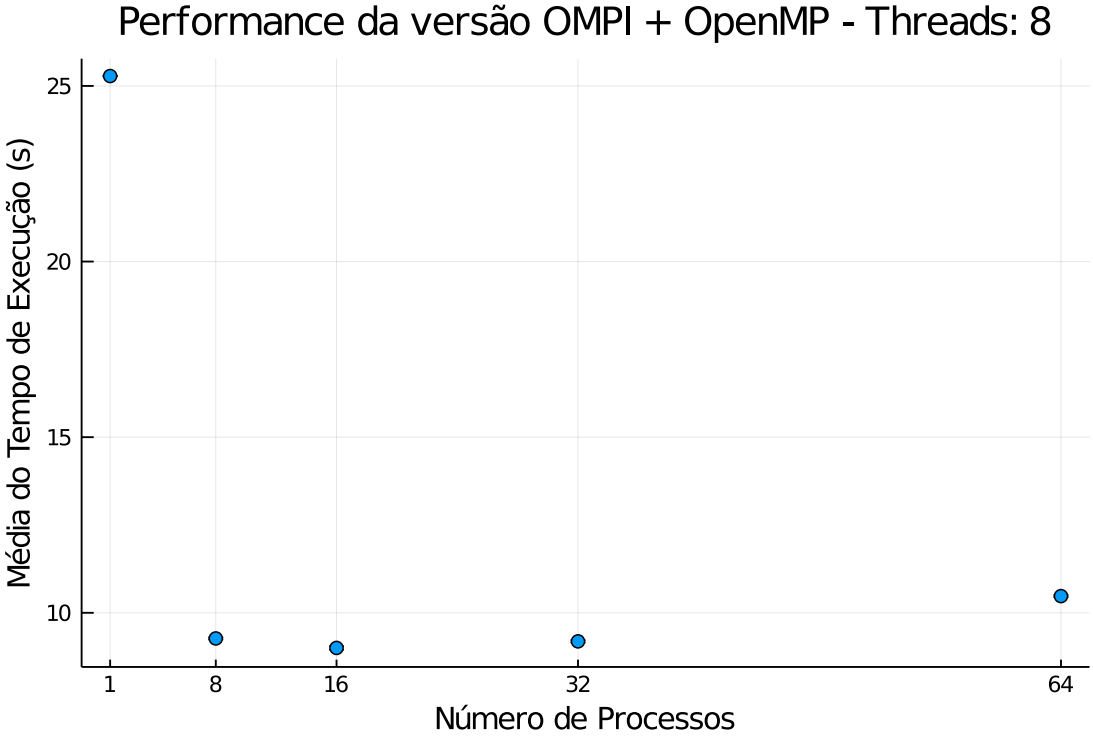
É possível também inverter a coluna de threads com a coluna de processos, e assim analisar os gráficos que relacionam a médias dos tempos de execução com o números de processos para cada número de threads. A função plot_mpi_omp_2 definida abaixo constrói esses novos gráficos:

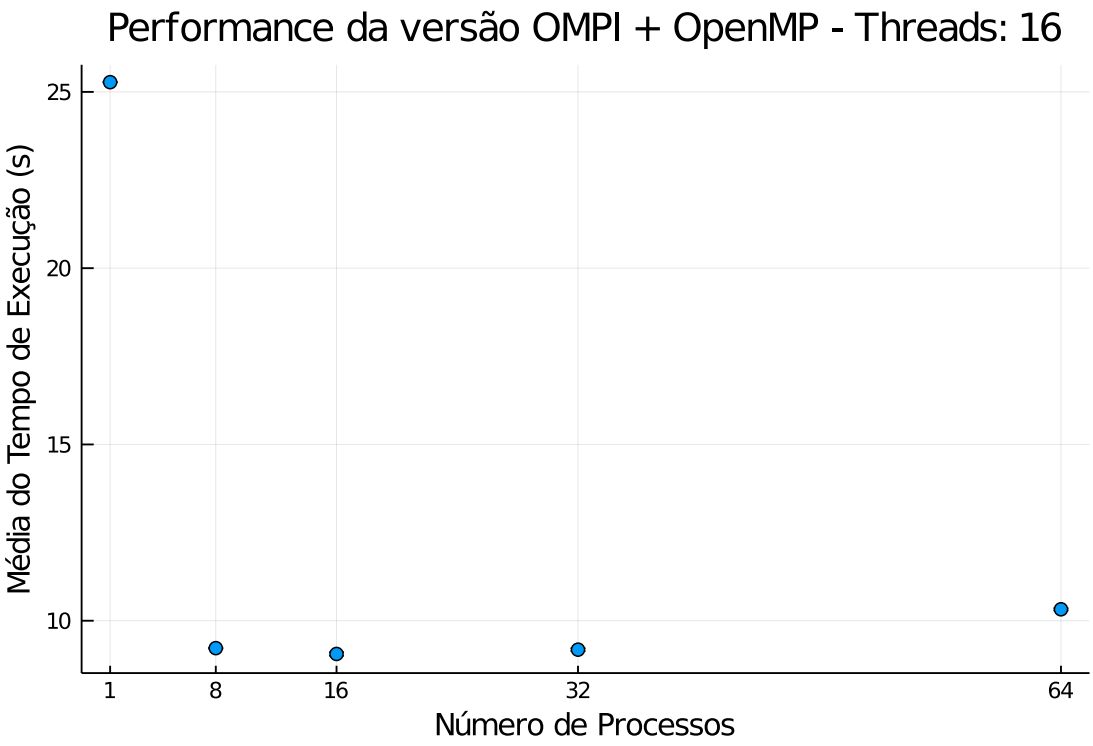
[28]:
```
using Plots
function plot_mpi_omp_2(data)
    for t in [1,2,4,8,16,32]
        d = data[data[!, :threads] .== t, :]
        fig = plot(xlabel = "Número de Processos", ylabel = "Média do Tempo de␣
 ↪Execução (s)",
            legend = :topright, title = string("Performance da versão OMPI +␣
 ↪OpenMP - Threads: ", t), xticks=[1,8,16,32,64])
        scatter!(
            d.processes,
            d.mean,
            yerror = d.ci,
            label = nothing,
        )
        display(fig)
    end
end

plot_mpi_omp_2(df_mpi_ompi)
```

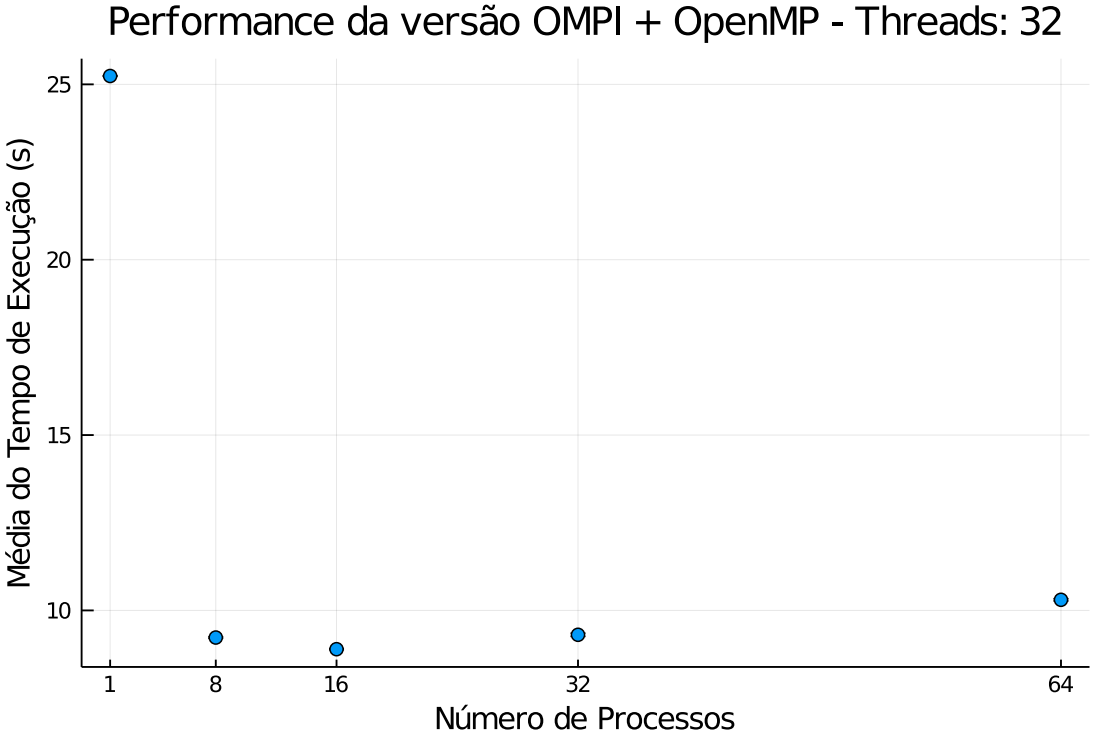

Performance da versão OMPI + OpenMP - Threads: 1
Média do Tempo de Execução (s)
25
20
15
10
1
8
16
32
64
Número de Processos


Performance da versão OMPI + OpenMP - Threads: 2
Média do Tempo de Execução (s)
25
20
15
10
1
8
16
32
64
Número de Processos


Performance da versão OMPI + OpenMP - Threads: 4
Média do Tempo de Execução (s)
25
20
15
10
1
8
16
32
64
Número de Processos


Performance da versão OMPI + OpenMP - Threads: 8
Média do Tempo de Execução (s)
25
20
15
10
1
8
16
32
64
Número de Processos


Performance da versão OMPI + OpenMP - Threads: 16
Média do Tempo de Execução (s)
25
20
15
10
1
8
16
32
64
Número de Processos


Performance da versão OMPI + OpenMP - Threads: 32
Média do Tempo de Execução (s)
25
20
15
10
1
8
16
32
64
Número de Processos

Observando os novos gráficos é possível perceber que, diferentemente da execução anterior, todos os gráficos possuem um padrão na apresentação dos seus dados e esse padrão é o mesmo para todos eles. Além disso, esse padrão é o mesmo visto na análise das execuções feitas com paralelização OMPI apenas. Assim, temos que a média do tempo de execução vai diminuindo à medida que o número de processos vai aumentando até a marca de 16 processos. A partir daí, a média do tempo de execução começa a aumentar junto com o valor do número de processos mas não chegando aos valores altos da execução com apenas 1 processo.

Portanto, com esses resultados, podemos de uma certa forma concluir que para qualquer valor de número de threads, as execuções utilizando 16 processos possuem a melhor performance baseada no tempo de execução.

1.4 Conclusão

De acordo com os gráficos e dados obtidos pelos scripts, é válido relemberar que a versão sequencial é mais demorada que as versões paralelizadas. Utilizando os dados do EP2, levamos em torno de 24s para terminar a execução com io/malloc e 23s sem io/malloc.

Já analisando o Pthreads e o OpenMP, fica perceptível uma diminuição no tempo de execução quando utilizamos diversas threads. Como exemplo, podemos citar o tempo de execução para 32 threads, em que atingimos 7,8s e 7,6s nas versões Pthreads e OpenMP, respectivamente, em comparação com os 24s da versão sequencial. Também podemos observar que esse tempo decai à medida que aumentamos o número de threads, como explicado na seção específica de cada um. Fazendo uma breve comparação entre essas duas, o OpenMP foi levemente mais rápido que o Pthreads para qualquer quantidade de thread, como podemos perceber no exemplo citado acima para 32 threads.

Analisando agora o OMPI, utilizamos os números de processos que são múltiplos de 8 como sugerido no enunciado. É válido ressaltar que a performance não se mostra proporcional à quantidade de processos, onde obtivemos 30,8s para 1 processo, 18s para 16 processos (tempo mínimo) e 24s para 64 processos. Analisando esses valores, podemos perceber que, inicialmente, houve uma queda no tempo de execução, mas que esse valor voltou a crescer quando utilizamos mais de 16 processos.

Se tentarmos aplicar uma simples ideia de proporção, podemos relacionar os diferentes métodos de paralelização vistos até agora. Assim, podemos perceber que o OpenMP ainda continua sendo o mais rápido para 1 processo (dado que os métodos de paralelização em threads executam em um único processo). Por outro lado, como o OpenMP executa apenas 1 processo, é possível observar que seu tempo de execução seria superior caso executássemos, por exemplo, 16. Para 32 threads, obtivemos 7,6s, o que resultaria em 121,6s para todos os 16 processos, tempo muito superior aos 18s obtivos pelo OMPI. Esse resultado também se observa para as outras quantidades utilizadas, porém ressaltamos essa comparação apenas para uma delas. Logo, conseguimos observar que o OMPI é superior caso executássemos múltiplos processos em paralelos.

O CUDA possui as dimensões (x,y) do grid como parâmetro, o qual decidimos admití-lo como um quadrado, logo x=y. De acordo com o gráfico obtido, podemos perceber que há uma diminuição no tempo de execução quando vamos aumentando o tamanho do grid de dimensão (1,1) até dimensão (8,8). Para a dimensão (1,1), obtivemos tempo de execução igual a 2.2s, chegando ao tempo mínimo de 0.62s na dimensão (8,8). Após isso, o tempo de execução vai aumentando, chegando a 4.38s para a dimensão (64,64).

Ao compararmos o CUDA com os outros processos, percebemos que existe uma diferença bastante notória no tempo de execução, com predominância de tempo abaixo de 2s, chegando a marca dos 0.62s. É válido mencionar que uma comparação com os outros métodos é injusta, pois obtivemos esses resultados em uma máquina da rede Linux, como sugerido pelo professor, máquina com maior capacidade de processamento que os nossos computadores pessoais.

Por fim, iremos analisar a versão OMPI+OMP. Como mencionado em sua seção, observamos uma queda no tempo de execução ao compararmos múltiplos processos sendo executados com apenas 1. Para 1 processo, também observamos um padrão no tempo de execução quando aumentamos o número de threads, que diminui lentamente, atingindo seu mínimo com 32 threads.

Ao compararmos os gráficos do OMPI+OMP e do OMPI, observamos que, para uma mesma quantidade de processos em ambos os gráficos, obtemos um tempo de execução muito menor pro OMPI+OMP, independente do número de threads. Como mencionado acima, no OMP, atingimos tempo mínimo de execução de 18s quando executamos 16 processos. Para essa mesma quantidade no OMPI+OMP, chegamos a resultados abaixo dos 9s (para 2 e 32 threads). É válido ressaltar que mesmo a diferença sendo pequena, também atingimos os menores tempos no OMPI+OMP quando executamos 16 processos.

Em relação ao número de threads, não houve muito diferença para um mesmo número de processos, além de não ter um padrão para número de processos maiores que 1. A variação no tempo de execução não superava 1s entre quaisquer pares de threads. Já mantendo fixo o número de threads fixo e variando o número de processos, percebemos que começamos com uma leve queda no tempo de execução ate os 16 processos, voltando a crescer até os 64 processos.

Ao compararmos o OMPI+OMP com o OpenMP, percebemos que, para 1 processo, temos que o OpenMP é mais rápido que o OMPI+OMP para qualquer quantidade de threads. Como o OMPI+OMP paraleliza processos, executando-os ao mesmo tempo, podemos perceber rapidamente que o OMPI+OMP é superior em performance, levando menos tempo para executar todos os processos que o OMP caso este precisasse executar múltiplos processos sem paralelizá-los. Este fato é notado ao pegarmos, por exemplo, o tempo de execução para 16 threads do OpenMP(7.6s). Caso fosse necessário executar 8 processos, teríamos um tempo superior a 1min, enquanto que alcançamos 9.2s na versão OMPI+OMP para o mesmo número de threads.