

Fundamentos de Processamento Paralelo e Distribuídos -

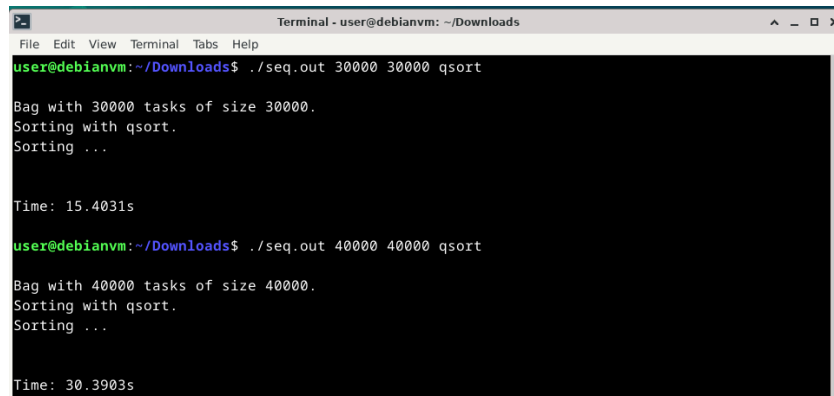
Trabalho 3

Leonardo Chou da Rosa

O Trabalho 3 da disciplina Fundamentos de Processamentos Paralelos e Distribuídos pretende avaliar o comportamento de um algoritmo executado de forma paralela, comparando seu desempenho com o mesmo algoritmo, só que de forma sequencial. Os códigos fornecidos contém dois tipos de algoritmos de arranjo de dados; o quicksort e o bubblesort.

Compilação e Execução:

Antes de executar o programa, é necessário indicar o tamanho do vetor sendo utilizado e a quantidade de vetores que serão criados. Após execução, o terminal irá informar ao usuário os valores escolhidos e a quantidade de tempo que o computador demorou para executar todas as tarefas. A imagem abaixo demonstra os resultados de alguns testes executados no terminal:



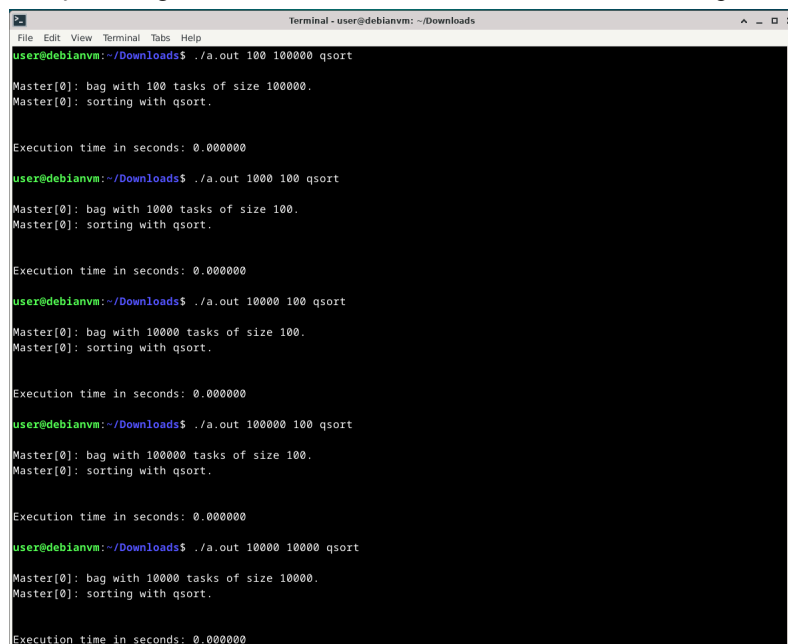
```
Terminal - user@debianvm: ~/Downloads
user@debianvm:~/Downloads$ ./seq.out 30000 30000 qsort
Bag with 30000 tasks of size 30000.
Sorting with qsort.
Sorting ...

Time: 15.4031s

user@debianvm:~/Downloads$ ./seq.out 40000 40000 qsort
Bag with 40000 tasks of size 40000.
Sorting with qsort.
Sorting ...

Time: 30.3903s
```

Para obter os resultados da versão MPI dos algoritmos, o processo de execução é o mesmo, porém algumas dependências são necessárias para compilar o código. No operador Linux, foi necessário instalar o pacote Linux de MPI, que contém o comando preciso para a sua compilação. Na hora de compilar, foi utilizado o comando ‘mpicc’, que garante a compilação de todos os componentes utilizados pelo código. Após execução do algoritmo, o terminal demonstra os mesmos valores esperados da versão sequencial:



```
Terminal - user@debianvm: ~/Downloads
user@debianvm:~/Downloads$ ./a.out 100 100000 qsort
Master[0]: bag with 100 tasks of size 100000.
Master[0]: sorting with qsort.

Execution time in seconds: 0.000000

user@debianvm:~/Downloads$ ./a.out 1000 100 qsort
Master[0]: bag with 1000 tasks of size 100.
Master[0]: sorting with qsort.

Execution time in seconds: 0.000000

user@debianvm:~/Downloads$ ./a.out 10000 100 qsort
Master[0]: bag with 10000 tasks of size 100.
Master[0]: sorting with qsort.

Execution time in seconds: 0.000000

user@debianvm:~/Downloads$ ./a.out 100000 100 qsort
Master[0]: bag with 100000 tasks of size 100.
Master[0]: sorting with qsort.

Execution time in seconds: 0.000000

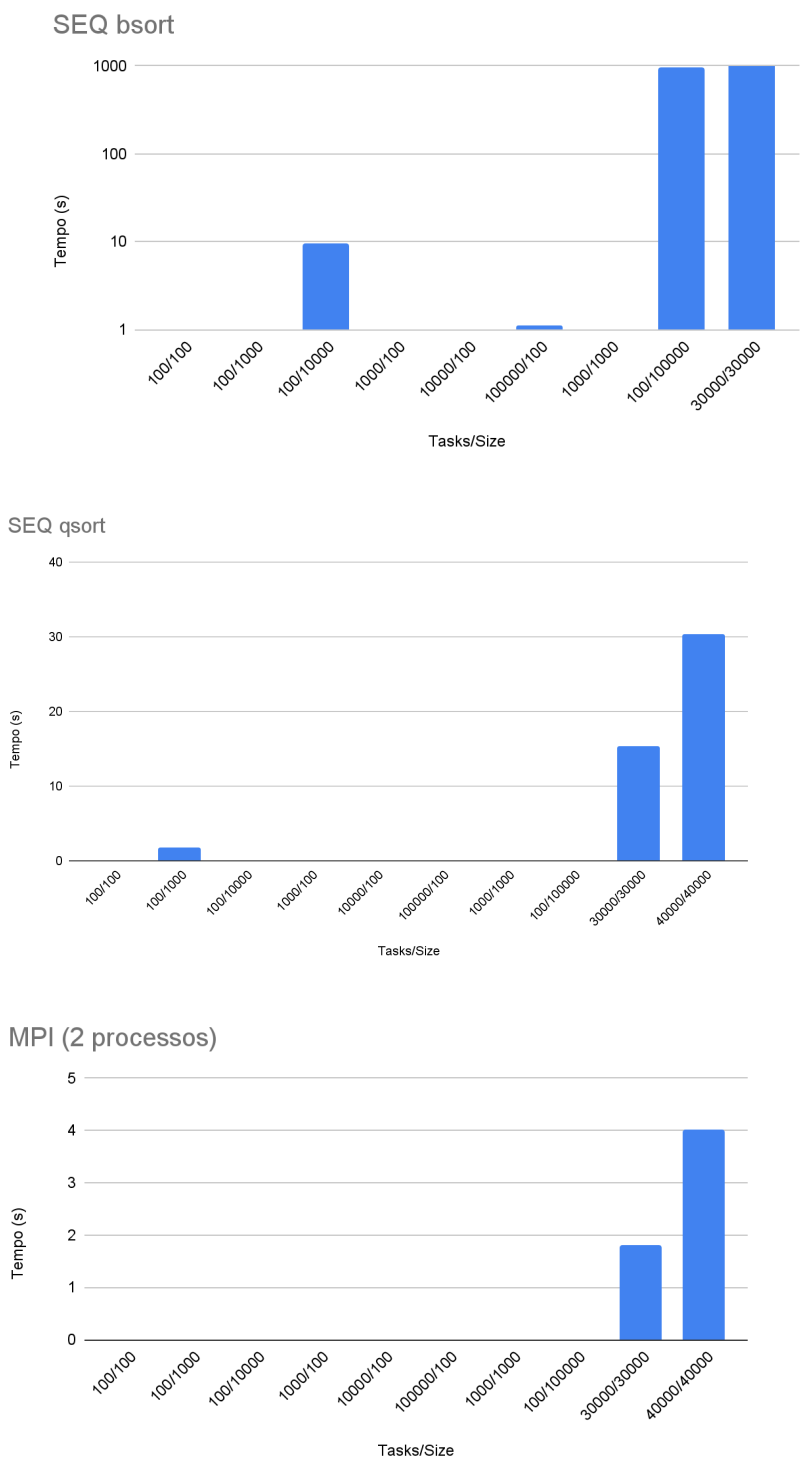
user@debianvm:~/Downloads$ ./a.out 10000 10000 qsort
Master[0]: bag with 10000 tasks of size 10000.
Master[0]: sorting with qsort.

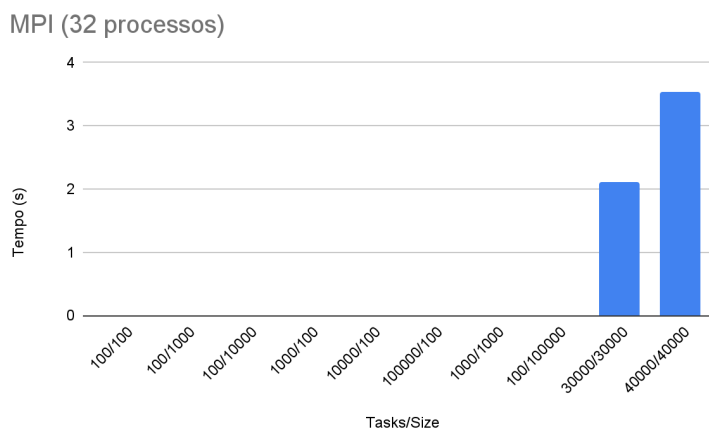
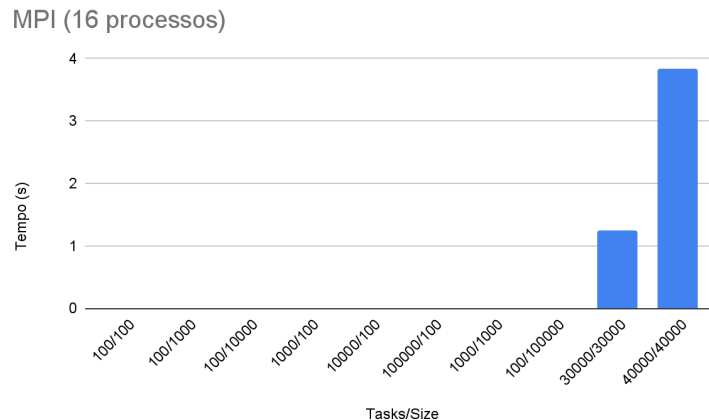
Execution time in seconds: 0.000000
```

Para obter diferentes quantidades de processos, é necessário alterar a variável “proc_n” no programa ms_mpi. Desta forma, é possível observar as diferenças de tempo de execução quando o número de processos aumenta.

Resultados Obtidos:

Para cada algoritmo diferente, foi desenvolvido diferentes gráficos que avaliam seu desempenho em questão de tarefas/tamanho de vetor sobre o tempo de execução da tarefa. No eixo vertical do gráfico, os valores de tempo em segundo estão presentes, e no eixo horizontal a quantidade de tarefas e o tamanho do vetor estão presentes:





Podemos observar que para valores menores o programa sequencial em bubblesort aparenta ser o mais eficiente para a tarefa, demonstrando tempos de execução muito rápidos em relação aos outros, porém após uma quantidade específica de entrada, a sua eficiência é inviabilizada. Quando o tamanho do vetor atinge 100.000 unidades, a quantidade de tempo que o bubblesort sequencial demora para executar aumenta exponencialmente, quase chegando aos 1.000 segundos, um valor muito mais alto do que em todos os outros testes realizados. No teste de 40.000/40.000, o programa sequencial em bubblesort não conseguiu terminar a execução. Em comparação com o programa sequencial em quicksort, o programa gradualmente aumenta em tempo de execução conforme a entrada esteja aumentando, porém, ele consegue executar todos os testes. Baseado no gráfico, podemos inferir que o programa sequencial quando está executando o quicksort se torna uma opção muito mais viável em comparação com o programa sequencial executando o bubblesort.

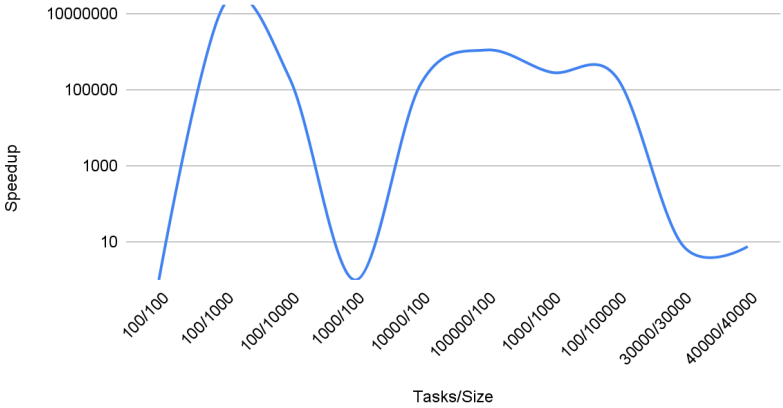
Os programas em MPI demonstraram resultados muito superiores aos programas sequenciais, porém entre eles as diferenças não são muito grandes. Em uma entrada que demorou 15 segundos na execução quicksort, os três programas MPI conseguiram resolver em 2 segundos ou menos. Entre eles, a quantidade de processos não aparentou muita diferença, onde o programa com 32 processos demonstrou uma leve superioridade de desempenho em relação aos outros.

Com os resultados obtidos, é possível avaliar o speedup, ou a quantidade de vezes que o programa em paralelo é mais rápido do que a versão sequencial em uma dada tarefa. Para obter esse valor, é necessário utilizar a fórmula:

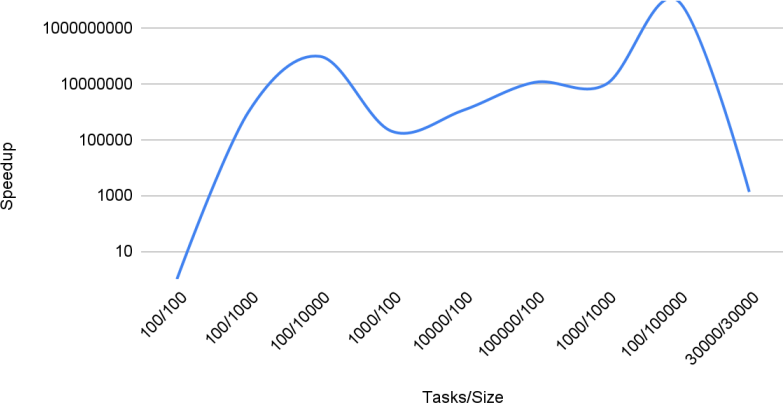
$$Speedup = Tempo de Execução Sequencial / Tempo de Execução Paralela$$

Através dessa fórmula, os seguintes gráficos foram desenvolvidos (para melhor visualização, o eixo vertical dos gráficos foram colocados em escala logarítmica):

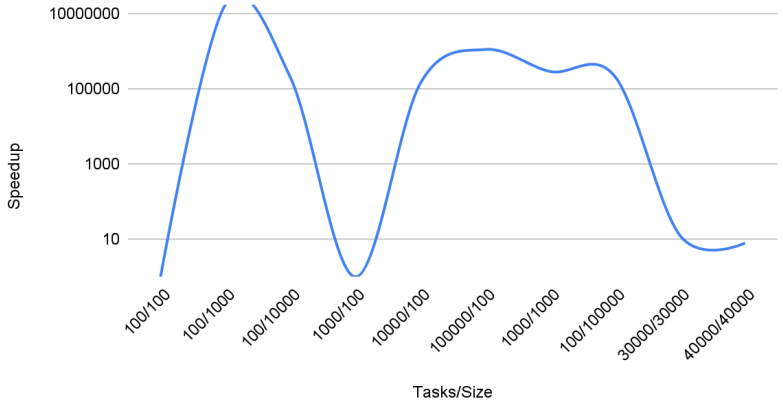
Speedup (2 processos x SEQ qsort)



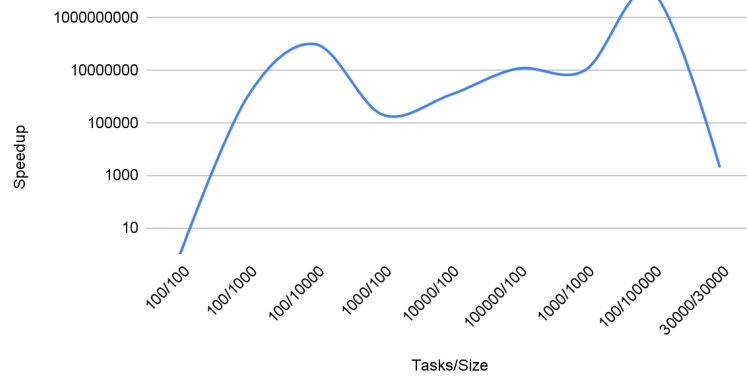
Speedup (2 processos x SEQ bsort)



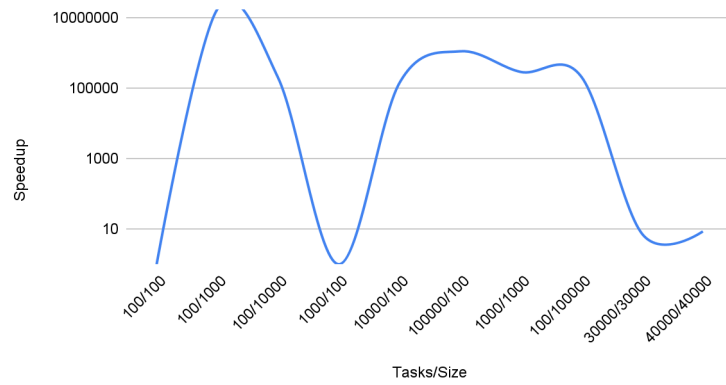
Speedup (16 processos x SEQ qsort)



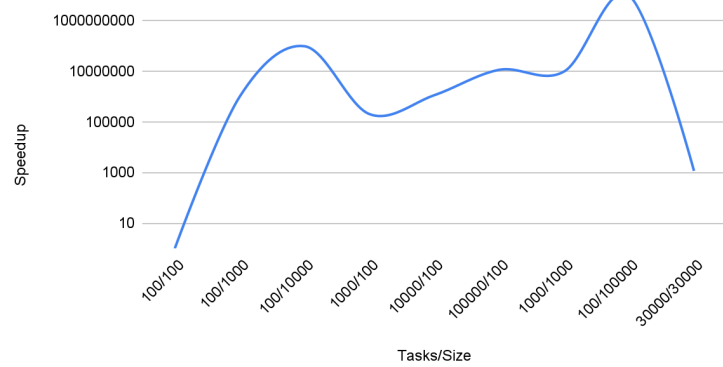
Speedup (16 processoss x SEQ bsort)

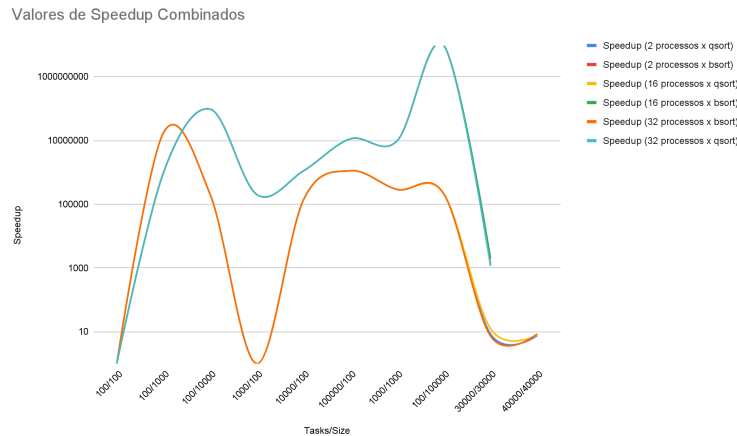


Speedup (32 processoss x SEQ qsort)



Speedup (32 processoss x SEQ bsort)





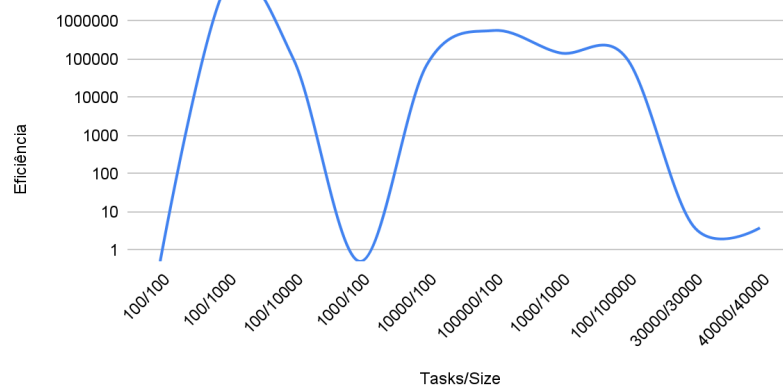
Em todos os gráficos apresentados, um padrão é evidente; o speedup do programa em paralelo aumenta de uma forma exponencial, conforme o tamanho de entrada aumenta, e após um certo ponto ele começa a decrescer novamente. O que chama mais atenção é que em números de entrada muito grande, o speedup do programa paralelo chega a ficar muito próximo de 1, ou seja, ele é minimamente mais rápido do que o programa sequencial. Outra observação interessante é que quando se observa o gráfico combinado, os valores de speedup para cada quantidade de processos são praticamente idênticos. Os valores somente mudam drasticamente quando os valores de entrada estão em seus máximos, onde é possível notar que os valores de speedup do programa com 32 processos é um pouco superior aos demais. Isso demonstra que nesse caso específico, a quantidade de processos não necessariamente causa uma grande diferença nos resultados.

Sabendo o speedup de cada quantidade de processos, podemos calcular a eficiência de cada programa. A fórmula utilizada para achar esses valores é o seguinte:

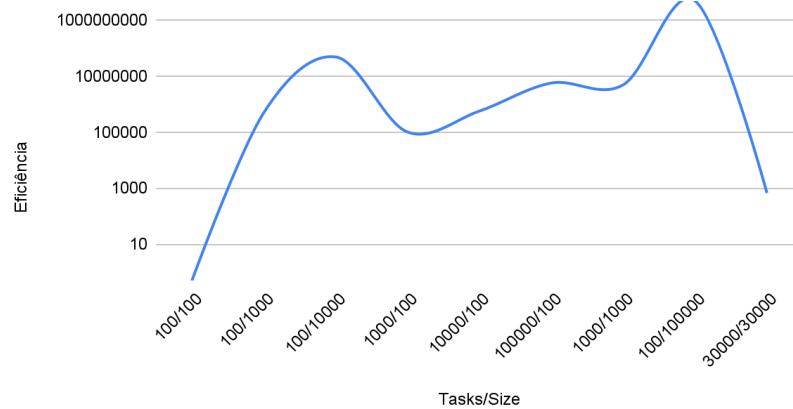
$$\text{Eficiência} = \text{Valor de Speedup dos Processos} / \text{Quantidade de Processos}$$

Com base nesta fórmula, os seguintes gráficos foram desenvolvidos:

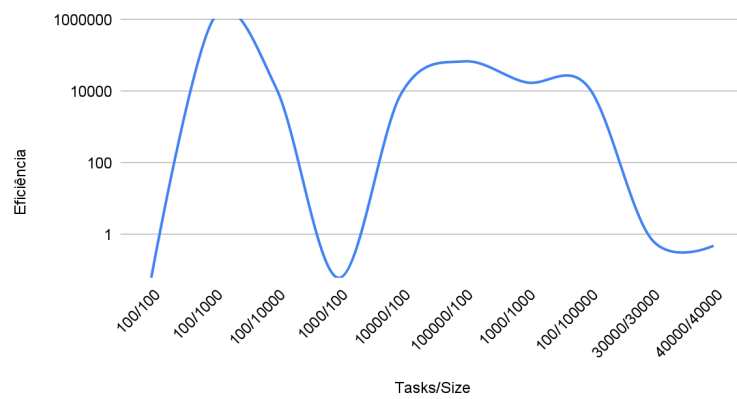
Eficiência (2 processos x qsort)



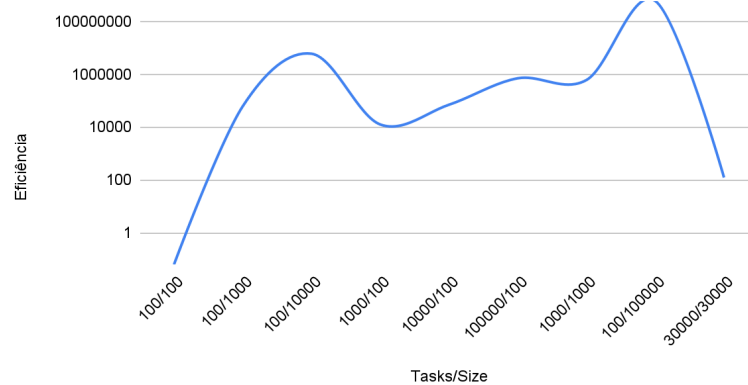
Eficiência (2 processos x bsort)



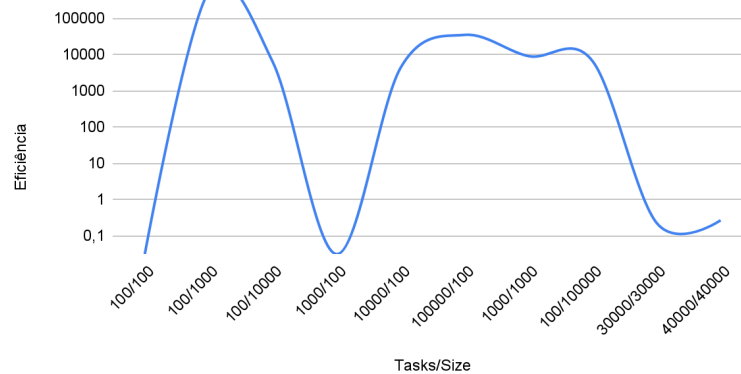
Eficiência (16 processos x qsort)



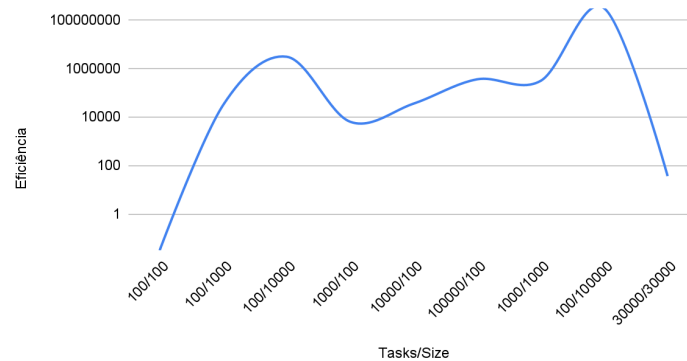
Eficiência (16 processos x bsort)



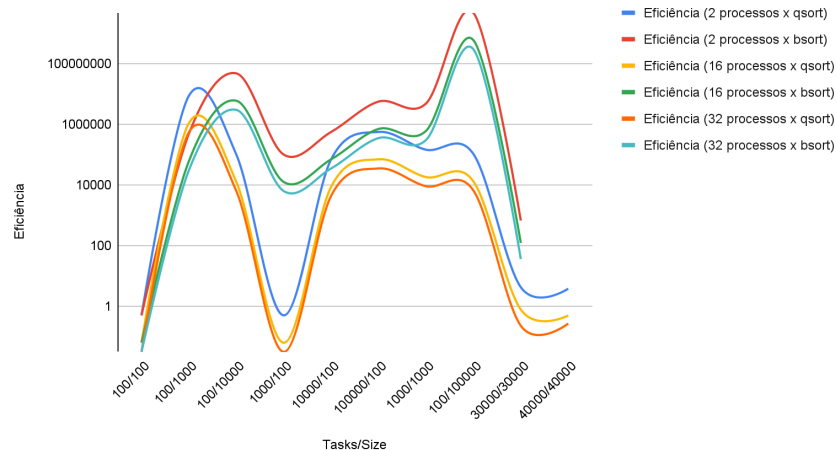
Eficiência (32 processos x qsort)



Eficiência (32 processos x bsort)



Valores de Eficiência Combinados



Similarmente ao gráfico de speedup, os valores individuais de cada gráfico não aparentam muita diferença, porém quando é observado todos os gráficos em conjunto é evidente que o programa com dois processos é o mais eficiente nesse contexto. O motivo para isso é que em tempos de execução, o programa com dois processos demonstrou resultados aceitáveis, porém minimamente inferiores aos demais programas paralelos. Essa diferença mínima é compensada quando se compara com o número de processos, onde a sua necessidade de recursos é muito inferior aos demais programas paralelos. Portanto, neste caso em específico, uma quantidade de dois processos demonstra melhor eficiência em comparações com programas que contêm mais do que dois processos executando a mesma tarefa.

A utilização de diversos processos para concluir uma tarefa é uma alternativa inteligente e eficiente em comparação a utilizar programas sequenciais. A dificuldade se apresenta quando é preciso determinar a quantidade de processos que devem ser usados na

execução da tarefa. Baseado nos resultados obtidos, a eficiência de um programa em paralelo aumenta conforme aumenta a quantidade de processos até um certo ponto, onde ela começa a cair de eficiência. Os resultados indicam que o programa com 16 processos demonstra uma execução de tempo muito semelhante ao programa com 32 processos, porém é mais eficiente devido à sua quantidade de processos reduzida. A utilização de processamento paralelo é considerada eficiente até um certo número de processos.