



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Trabalho da Segunda Unidade
Programação Concorrente e Distribuída

Aluno: Jaime Cristalino Jales Dantas
Matrícula: 2016008362

NATAL
Novembro de 2016

Sumário

1. Introdução	4
2. Considerações Iniciais	5
3. Soma em Árvore.....	7
3.1 Implementação	7
3.2 Resultados	7
4. Soma Prefixada	12
4.1 Implementação	12
4.2 Resultados	12
5. Matriz Distribuição Coluna.....	16
5.1 Implementação	16
5.2 Resultados	16
6. Conclusão	23
7. Referências	24

Lista de ilustrações

Figura 1 - Htop na máquina de testes - Servidor do DCA	5
Figura 2 - Gráfico de testes com media e mediana	6
Figura 3 - Erro com muitos processos MPI	9
Figura 4 - Gráfico de run time de comparação soma em árvore.....	10
Figura 5 - Gráfico de run time de comparação soma em árvore com escala reduzida	10
Figura 6 - Gráfico de run time de comparação soma prefixada	14
Figura 7 - Gráfico de run time de comparação soma prefixada com escala reduzida	14
Figura 8 - Run time matriz colunas.....	18
Figura 9 - Run time matriz linhas	18
Figura 10 – Eficiência matriz linha	19
Figura 11 - Eficiência matriz coluna	19
Figura 12 - Speedup matriz colunas	20
Figura 13 - Speedup matriz linhas.....	20
Figura 14 - Comparação para 8 processos da multiplicação matriz x vetor	21
Tabela 1 - Teste de execução soma em árvore.....	9
Tabela 2 - Teste soma prefixada.....	13
Tabela 3 - Teste de run time Matriz x Vetor	17
Tabela 4 - Eficiência Matriz Coluna	22

1. Introdução

Este trabalho apresenta uma análise detalhada de desempenho de execução das funções da funcionalidade Message Passing Interface (MPI) e faz uma comparação direta com uma implementações das mesmas funcionalidades usando comunicação ponto a ponto MPI_Send e MPI_Recv. Além disso, é apresentado um algoritmo de multiplicação de uma matriz por um vetor com distribuição de colunas ao invés de linhas.

2. Considerações Iniciais

Quando estamos testando um software que executa paralelamente, é de fundamental importância realizamos diversos testes ao invés de apenas um teste. Devemos levar em consideração que o computador que estamos realizando os testes está executando outros processos em background paralelamente a execução do teste.

Os testes aqui apresentados foram realizados no servidor *labgrad* do DCA. O computador possui um processador de 8 núcleos modelo intel Xeon com 16GB de RAM. Os resultados aqui apresentados não são os ideais pois apresentam erros associados a medição de tempo de execução. Como o teste foi realizado em uma máquina não dedicada, outros processos estavam sendo executados concorrentemente a execução de nosso programa, disputando assim por tempo de execução de CPU. A imagem abaixo mostra um overview dos processos que estão sendo executados na máquina de testes.

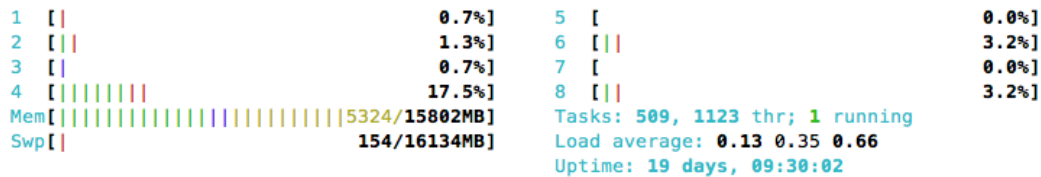


Figura 1 - Htop na máquina de testes - Servidor do DCA

Para que nosso teste ficasse mais representativo, escolhemos realizar diversos testes e ao final calcular a mediana dos resultados de tempo de execução. Quando usamos a mediana em nosso cálculo, desconsideramos pontos fora da curva que apresentam um tempo de execução muito superior ou muito inferior ao real. O Gráfico abaixo apresenta uma demonstração da execução de 30 testes com o algoritmo de soma em árvore usando MPI_Send e MPI_Recv para 16 processos. Neste gráfico é possível observarmos que a mediana teve valor de 5.14 e foi levemente superior ao valor da

média, que foi de 4.95. Isso se dá devido a influência sofrida pelo valor da média devido ao pontos fora da curva.

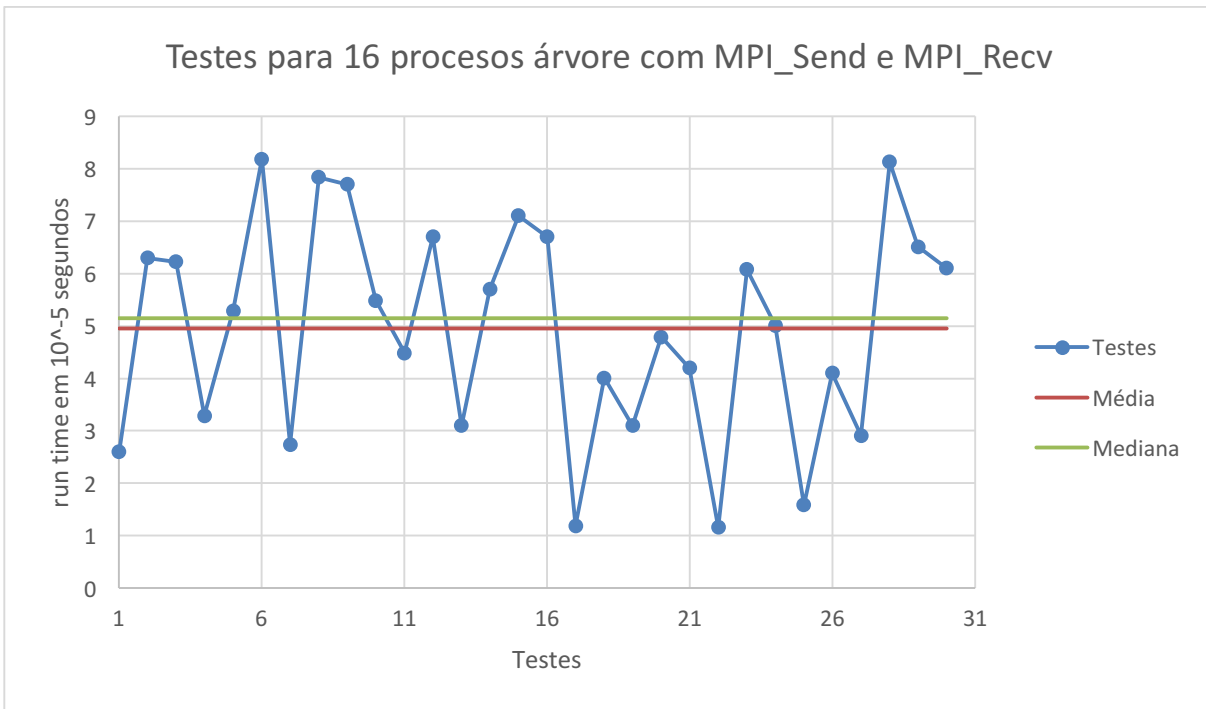


Figura 2 - Gráfico de testes com media e mediana

3. Soma em Árvore

3.1 Implementação

O `MPI_Reduce` implementa uma redução dos valores de entrada, ou seja, ele retorna a soma de todos os elementos. Foi implementado uma redução dos elementos usando as funções `MPI_Send` e `MPI_Recv` em forma de uma árvore binária. Cada processo armazena o valor de seu *rank* + 1 e ao final, o processo 0 irá mostrar o valor da soma total.

Para o desenvolvimento do algoritmo de soma em árvore, foi usando o raciocínio presente na questão 1.4 do capítulo 1 onde se usa o conceito de *bitmask* para realizar a operação de envio e recebimento das somas locais. Para mais detalhes, é necessário consultar a prova 1 onde está presente o desenvolvimento do algoritmo.

Já para o desenvolvimento da soma como o `MPI_Reduce`, foi usando um código disponível pelo autor do livro para o capítulo 3. O algoritmo final computa ambos os tempos de execução usando a função do MPI chamada de `MPI_Wtime`. Para usarmos essa função, criamos uma barreira em cada bloco que queríamos medir o tempo.

3.2 Resultados

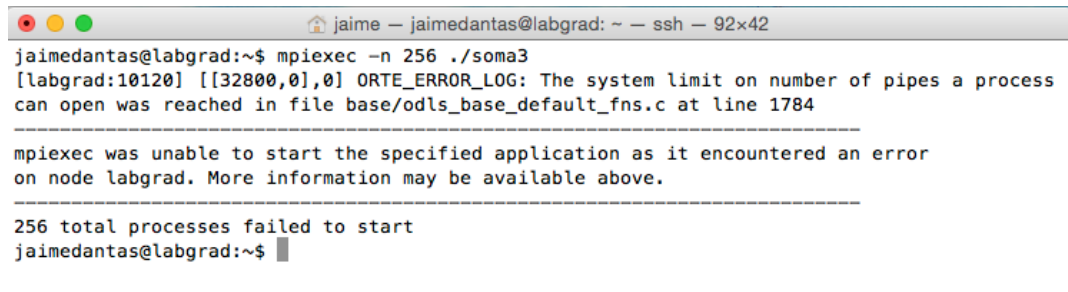
Foram realizados 5 testes para cada número de processos e calculado a mediana ao final do teste. A Tabela 1 abaixo mostra todos os testes que foram realizados.

		Run time test			
		MPI_Send MPI_Recv		MPI_Reduce	
		Run Time	Mediana	Run Time	Mediana
Processos	2	1.31E-05	1.1921E-05	3.10E-06	2.1458E-06
		1.31E-05		2.86E-06	
		8.82E-06		2.15E-06	
		6.91E-06		1.91E-06	
		1.19E-05		2.15E-06	
	4	1.31E-05	1.8835E-05	1.19E-05	1.4067E-05
		1.81E-05		1.62E-05	
		2.41E-05		1.41E-05	
		1.88E-05		2.10E-05	
		2.48E-05		1.41E-05	
	8	1.29E-05	2.5034E-05	1.46E-04	0.0001421
		2.50E-05		4.10E-05	
		3.29E-05		2.62E-04	
		8.58E-05		1.35E-04	
		1.19E-05		1.42E-04	
	16	3.70E-05	3.6955E-05	1.97E-04	0.00018501
		3.81E-05		1.54E-04	
		4.70E-05		1.68E-04	
		3.50E-05		1.85E-04	
		3.41E-05		7.49E-04	
	32	2.37E-02	0.0237329	2.52E-04	0.00025487
		1.50E-02		2.55E-04	
		6.70E-05		3.34E-04	
		4.60E-02		3.35E-04	
		2.70E-02		9.92E-05	
	64	3.61E-02	0.03721285	6.41E-04	0.00070786
		3.72E-02		7.08E-04	
		5.19E-04		4.35E-04	
		4.61E-02		1.00E-03	
		4.36E-02		9.52E-04	
	128	1.08E-02	0.01079607	1.38E-03	0.00143313

		8.47E-03		1.93E-03	
		7.85E-03		1.17E-03	
		6.13E-02		2.54E-03	
		2.27E-02		1.43E-03	

Tabela 1 - Teste de execução soma em árvore

Não foi possível realizar testes com mais de 128 processos pois o compilador MPI retornou um erro afirmando que a quantidade de processos era superior a permitida. O erro é apresentado na figura abaixo.



```

jaime — jaimesdantas@labgrad: ~ — ssh — 92x42
jaimesdantas@labgrad:~$ mpiexec -n 256 ./soma3
[labgrad:10120] [[32800,0],0] ORTE_ERROR_LOG: The system limit on number of pipes a process
can open was reached in file base/odls_base_default_fns.c at line 1784
-----
mpiexec was unable to start the specified application as it encountered an error
on node labgrad. More information may be available above.
-----
256 total processes failed to start
jaimesdantas@labgrad:~$

```

Figura 3 - Erro com muitos processos MPI

Usando os resultado obtidos, plotamos um gráfico de tempo de execução para os dois algoritmos. Os gráficos das Figura 4 e Figura 5 são os mesmos, diferindo apenas na escala do eixo x.

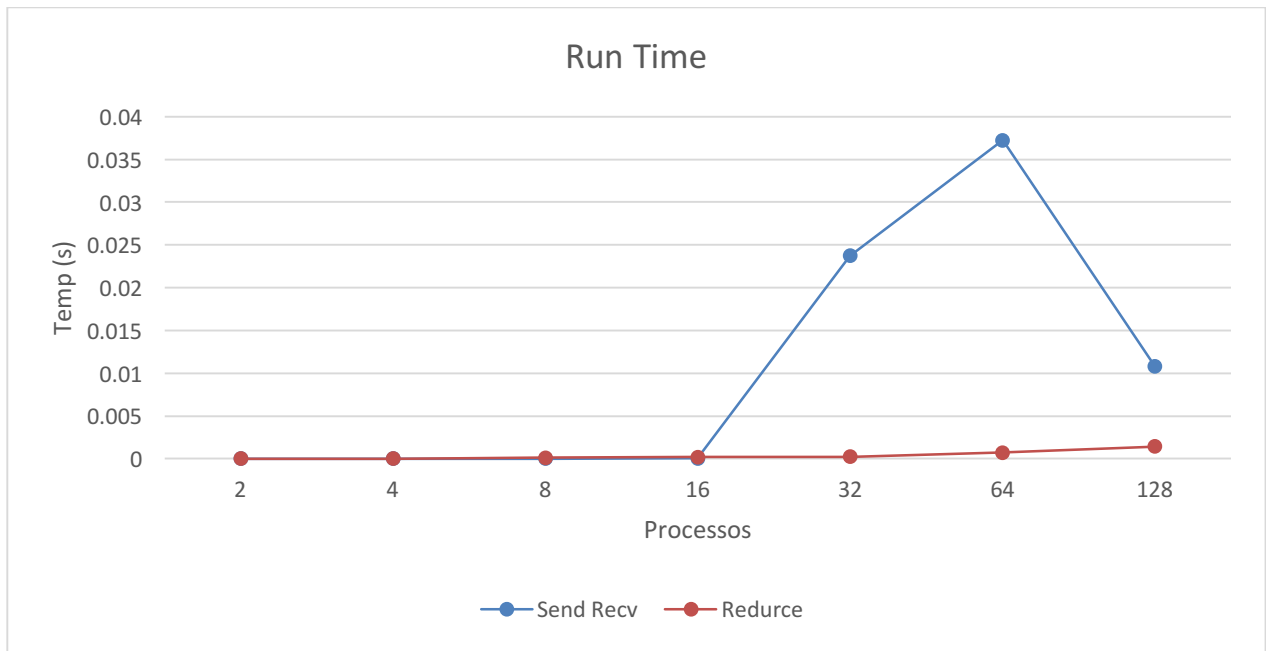


Figura 4 - Gráfico de run time de comparação soma em árvore



Figura 5 - Gráfico de run time de comparação soma em árvore com escala reduzida

Podemos observar que no gráfico da Figura 5 ambas as curvas apresentaram o mesmo comportamento. Obtivemos um resultado atípico para 8 e 16 processos onde o algoritmo desenvolvido com base de comunicação ponto a ponto obteve um desempenho melhor do que a função já implementada no MPI chamada de `MPI_Reduce`. Já para uma quantidade de processos superior a 16, a função `MPI_Reduce` apresentou um desempenho melhor. Vale salientar que como o computador de testes possui apenas 8 núcleos físicos, quando executamos mais de 8 processos, ele terá que executar paralelamente estes processos, tornando o teste para esses valores não representativos.

Além disso, podemos notar que o algoritmo desenvolvido apresenta um desempenho muito inferior a função `MPI_Reduce` quando o número de processos é muito grande. Entretanto, de uma forma geral, o desempenho de ambos os algoritmos são equivalentes. Isso constata que de fato a função `MPI_Reduce` implementa uma árvore, que por sua vez tem uma complexidade computacional de $O(\log(n))$.

4. Soma Prefixada

4.1 Implementação

Uma das maiores dificuldades para a implementação da soma prefixada foi o uso da função do MPI que implementa a redução parcial dos valores de entrada chamada `MPI_Scan`. Usando exemplos disponível em [2] foi possível montar um algoritmo que realiza a soma prefixada dos elementos de cada processos, que nesse caso foi utilizado o mesmo padrão da soma em árvore, $my_rank + 1$.

Já o algoritmo de soma prefixada usando funções de ponto a ponto do MPI foi desenvolvido em sala de aula.

4.2 Resultados

Assim como na soma em árvore, realizamos 5 testes para cada combinação e usamos a mediana como referencial padrão. A Tabela 2 abaixo mostra todos os testes que foram realizados.

Run time test					
		MPI_Send MPI_Recv		MPI_Scan	
		Run Time	Mediana	Run Time	Mediana
Processos	2	1.79E-05	3.7909E-05	3.81E-06	7.8678E-06
		4.70E-05		7.87E-06	
		3.70E-05		7.87E-06	
		4.51E-05		7.87E-06	
		3.79E-05		9.06E-06	
	4	5.89E-05	5.7936E-05	1.81E-05	1.7881E-05
		3.60E-05		5.01E-06	
		2.50E-05		5.01E-06	
		6.82E-05		1.79E-05	
		5.79E-05		1.81E-05	

	8	3.60E-05	3.7909E-05	9.89E-05	9.1076E-05
		3.79E-05		6.51E-05	
		3.91E-05		8.11E-05	
		4.98E-05		1.19E-04	
		3.10E-05		9.11E-05	
	16	8.48E-04	0.00084805	2.58E-04	0.00025797
		7.86E-04		2.58E-04	
		1.96E-03		3.84E-04	
		1.52E-03		1.71E-04	
		4.24E-04		1.87E-04	
	32	1.27E-03	0.01024508	5.17E-04	0.00051713
		3.20E-02		4.25E-04	
		1.78E-02		5.43E-04	
		6.16E-03		5.20E-04	
		1.02E-02		4.48E-04	
	64	1.49E-02	0.05197501	1.42E-03	0.0017221
		1.09E-01		1.76E-03	
		2.49E-02		1.71E-03	
		5.20E-02		1.72E-03	
		8.98E-02		1.82E-03	
	128	9.92E-02	0.0561769	5.38E-03	0.0053699
		2.51E-02		5.77E-03	
		1.64E-01		5.28E-03	
		4.65E-02		5.37E-03	
		5.62E-02		3.71E-03	

Tabela 2 - Teste soma prefixada

Usando os resultado obtidos, plotamos um gráfico de tempo de execução para os dois algoritmos. Os gráficos das Figura 4 e Figura 5 são os mesmo, diferindo apenas na escala do eixo x.

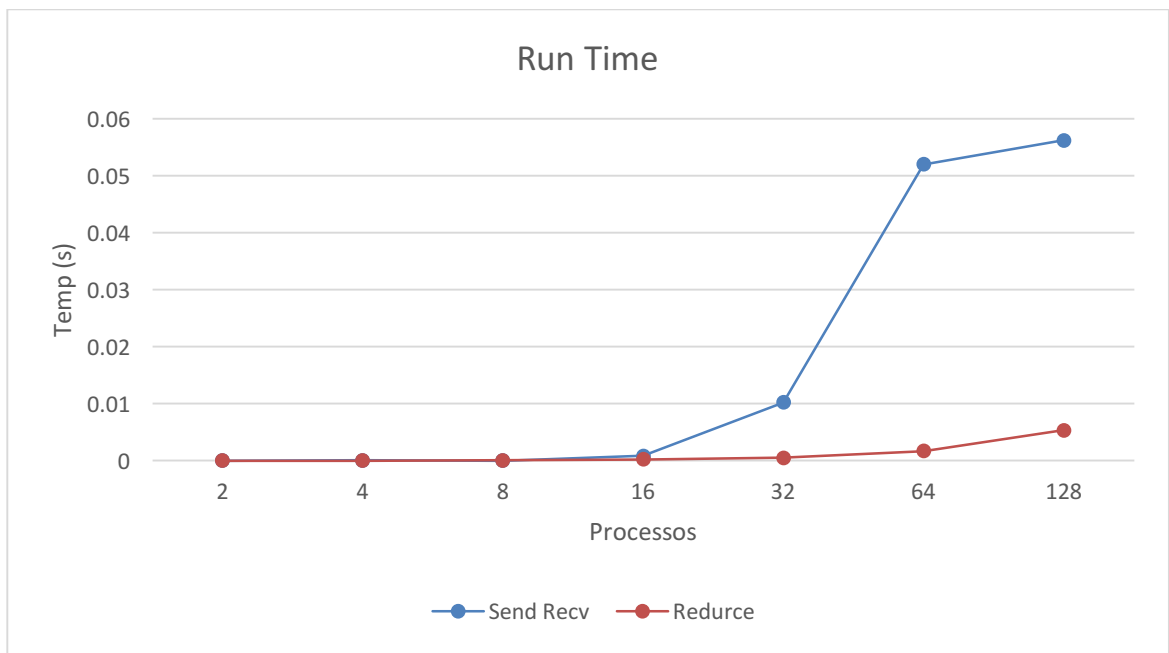


Figura 6 - Gráfico de run time de comparação soma prefixada

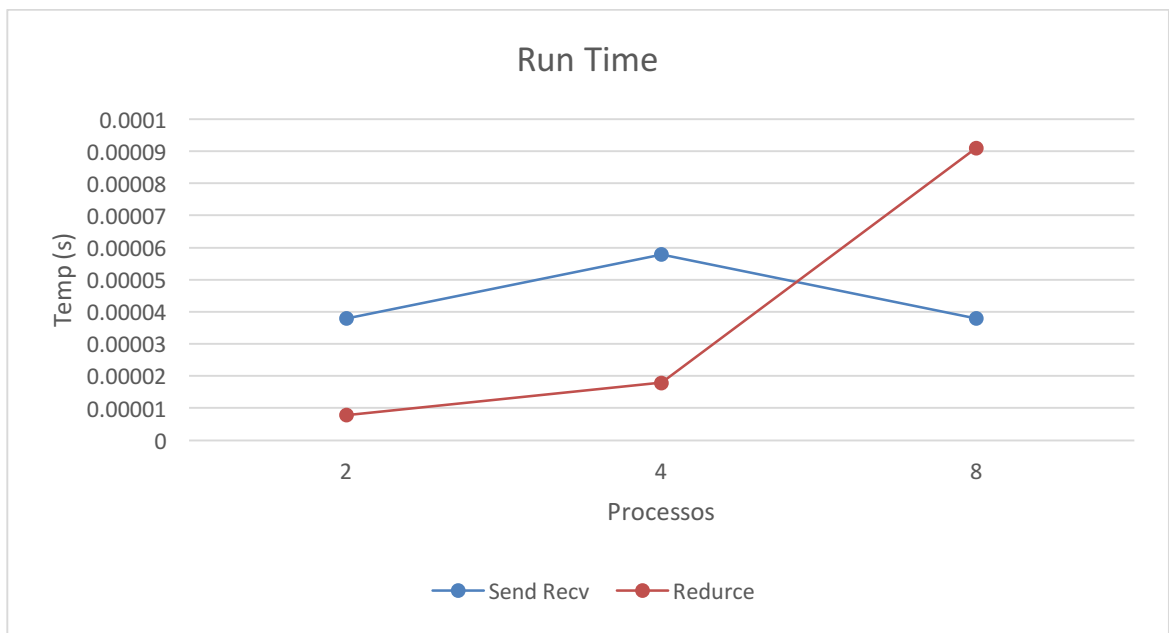


Figura 7 - Gráfico de run time de comparação soma prefixada com escala reduzida

Podemos observar que no gráfico da Figura 5 ambas as curvas apresentaram o mesmo comportamento de desempenho. Também obtivemos um resultado atípico para 8 processos onde nosso algoritmo desenvolvido com base de comunicação ponto a ponto obteve um desempenho melhor do que a função já implementado no MPI chamada de MPI_Scan. Já para as demais quantidade de processos, a função MPI_Reduce apresentou um desempenho melhor. Além disso, também constatamos que o algoritmo desenvolvido apresenta um desempenho muito inferior a função MPI_Scan quando o número de processos é muito grande.

5. Matriz Distribuição Coluna

5.1 Implementação

Foi usando o algoritmo fornecido pelo autor do livro [1] para o cálculo da multiplicação de uma matriz por um vetor usando as linhas da matriz e foram feitas alterações para que o algoritmo utilizasse as colunas ao invés das linhas.

Uma das maiores barreiras encontradas foi a utilização da função do MPI chamada `MPI_Type_vector`. Para que fosse calculado a soma usando as colunas da matriz, foi usando a função `MPI_Reduce_scatter`. Muitas horas foram gastas para implementar este algoritmo.

5.2 Resultados

Seguindo o raciocínio, foram realizados 5 testes para cada teste. Para que fosse possível analisar a escalabilidade do programa, realizados 4 testes com tamanhos de matrizes diferentes. Escolhemos um tamanho inicial da matriz de ordem 1024, de forma que o tempo de execução ficasse na ordem de milissegundos. Consequentemente, realizamos testes para matriz de ordem 2048, 4096 e 8192. A Tabela 3 abaixo mostra todos os testes realizados.

A partir dos resultados obtidos, foram plotados os gráficos de run time para cada algoritmo. A Figura 8 e a Figura 9 apresentam o tempo de execução para cada algoritmo. A eficiência e speedup são plotados logo em seguida nas Figura 10 até a Figura 13. Para o cálculo desses parâmetros, foram usadas as equações abaixo.

$$t_{speedup} = \frac{t_{serial}}{t_{paralelo}} \text{ AND } e = \frac{t_{serial}}{p \times t_{paralelo}}$$

		Run time test				Run time test			
		1024				2048			
		Colunas		Linhas		Colunas		Linhas	
		Run Time	Mediana	Run Time	Mediana	Run Time	Mediana	Run Time	Mediana
Processos	1	2.73E-03	0.0027809	2.72E-03	0.002764	1.09E-02	0.0122681	1.08E-02	0.010952
		2.77E-03		2.76E-03		1.23E-02		1.08E-02	
		3.06E-03		3.01E-03		1.23E-02		1.20E-02	
		2.78E-03		2.72E-03		1.09E-02		1.10E-02	
		3.25E-03		2.94E-03		1.23E-02		1.20E-02	
	2	2.10E-03	0.0015061	1.81E-03	0.001404	5.60E-03	0.0056019	5.48E-03	0.005486
		1.44E-03		1.40E-03		6.25E-03		6.19E-03	
		1.51E-03		1.40E-03		5.58E-03		5.49E-03	
		1.68E-03		1.55E-03		5.55E-03		5.48E-03	
		1.44E-03		1.40E-03		6.26E-03		6.30E-03	
	4	1.46E-03	0.0014341	1.41E-03	0.0014229	3.07E-03	0.0055711	2.93E-03	0.005543
		1.42E-03		1.43E-03		3.07E-03		2.95E-03	
		1.43E-03		1.42E-03		5.59E-03		5.54E-03	
		1.42E-03		1.46E-03		5.61E-03		5.58E-03	
		1.44E-03		1.40E-03		5.57E-03		5.62E-03	
	8	1.59E-03	0.0014892	1.49E-03	0.001179	3.05E-03	0.0030198	2.93E-03	0.0029521
		1.47E-03		1.54E-03		3.03E-03		2.95E-03	
		1.54E-03		1.18E-03		3.02E-03		2.97E-03	
		1.49E-03		1.16E-03		3.01E-03		2.96E-03	
		1.46E-03		1.15E-03		2.99E-03		2.95E-03	
	16	1.31E-03	0.0012801	1.14E-03	0.001183	5.24E-03	0.0052109	5.36E-03	0.0045161
		1.28E-03		1.18E-03		4.91E-03		4.37E-03	
		1.80E-03		1.18E-03		5.21E-03		4.52E-03	
		1.26E-03		1.21E-03		4.41E-03		4.11E-03	
		1.23E-03		1.76E-03		6.92E-03		5.17E-03	
		4096				8192			
Processos	1	4.34E-02	0.043344	4.34E-02	0.0433109	1.73E-01	0.1731122	1.73E-01	0.1732931
		4.33E-02		4.33E-02		1.73E-01		1.73E-01	
		4.33E-02		4.33E-02		1.73E-01		1.73E-01	
		4.33E-02		4.33E-02		1.73E-01		1.74E-01	
		4.33E-02		4.33E-02		1.73E-01		1.73E-01	
	2	2.21E-02	0.0220621	2.19E-02	0.0219431	9.56E-02	0.0879262	8.78E-02	0.087677
		2.21E-02		2.19E-02		8.78E-02		8.75E-02	
		2.20E-02		2.19E-02		9.90E-02		8.78E-02	
		2.21E-02		2.20E-02		8.78E-02		8.77E-02	
		2.37E-02		2.19E-02		8.79E-02		8.77E-02	
	4	2.23E-02	0.0130711	2.20E-02	0.0124381	8.91E-02	0.0889862	8.79E-02	0.0879059
		1.31E-02		1.24E-02		8.90E-02		8.79E-02	
		1.19E-02		1.18E-02		4.73E-02		4.70E-02	
		2.22E-02		2.20E-02		8.91E-02		8.79E-02	
		1.19E-02		1.18E-02		8.90E-02		8.79E-02	
	8	1.16E-02	0.011632	1.16E-02	0.01157	4.61E-02	0.0461321	4.61E-02	0.046056
		1.16E-02		1.16E-02		4.61E-02		4.60E-02	
		1.16E-02		1.16E-02		4.61E-02		4.60E-02	
		1.18E-02		1.16E-02		4.61E-02		6.17E-02	
		1.48E-02		1.16E-02		4.62E-02		4.62E-02	
	16	1.86E-02	0.0185752	1.85E-02	0.0167971	6.16E-02	0.0616128	5.19E-02	0.0502989
		1.70E-02		1.68E-02		5.00E-02		4.91E-02	
		2.76E-02		1.76E-02		6.51E-02		5.03E-02	
		1.75E-02		1.65E-02		6.74E-02		5.01E-02	
		2.17E-02		1.67E-02		5.01E-02		6.17E-02	

Tabela 3 - Teste de run time Matriz x Vetor

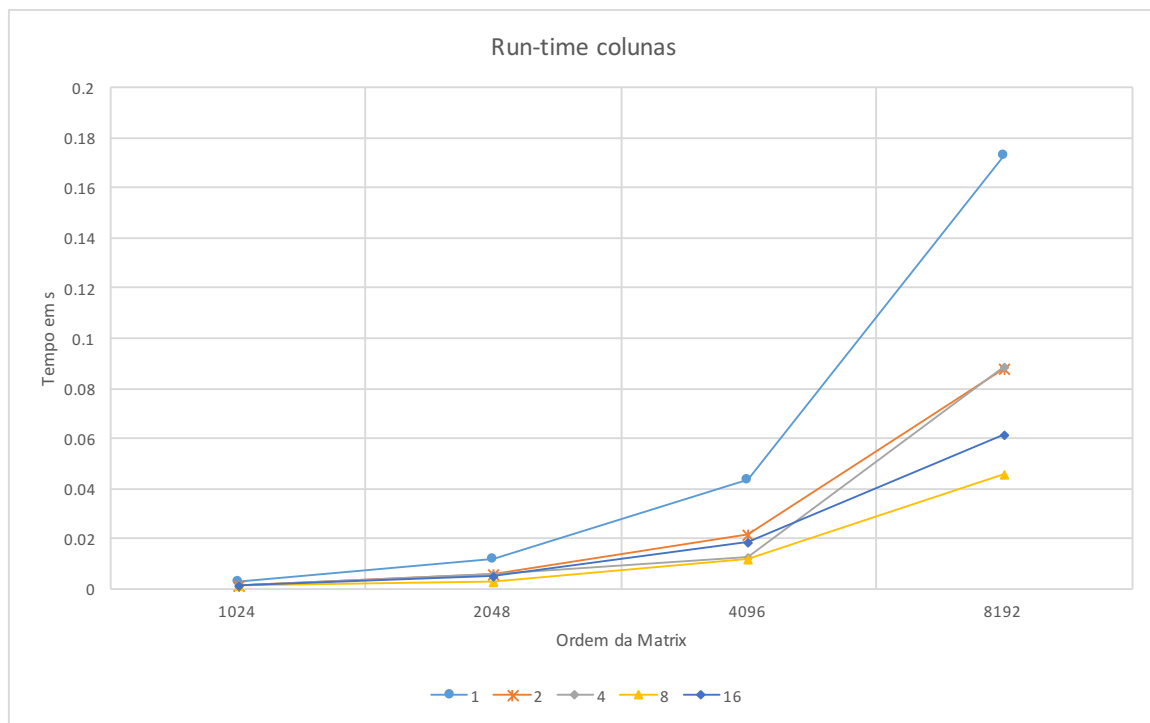


Figura 8 - Run time matriz colunas

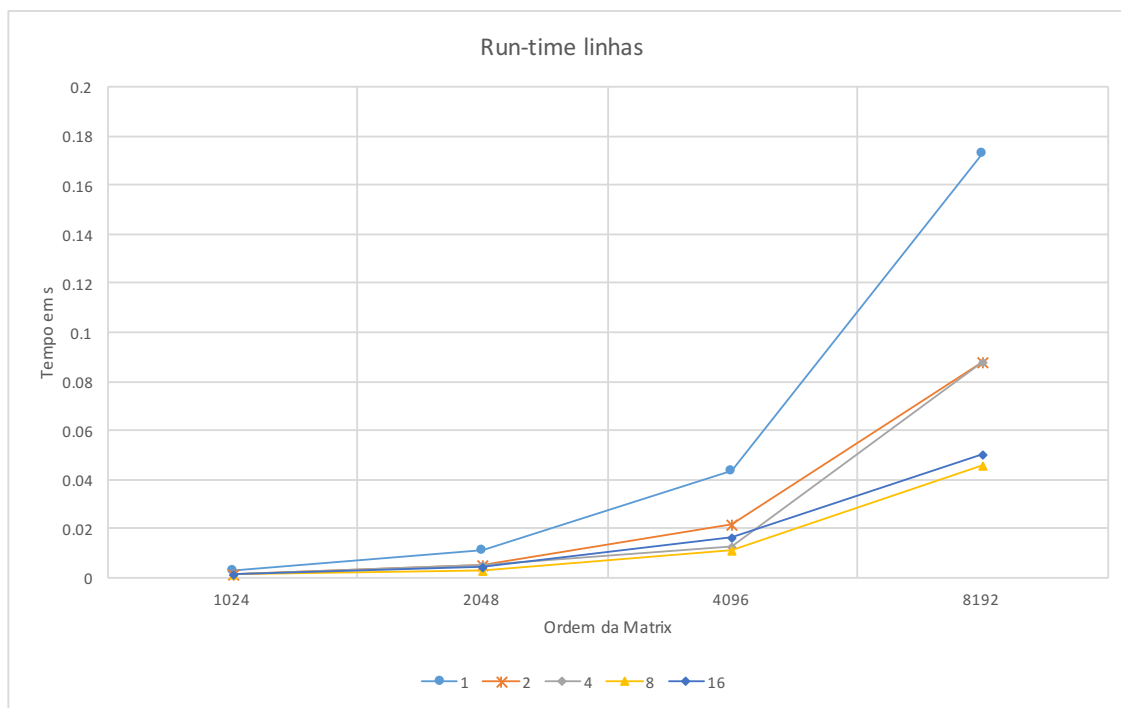


Figura 9 - Run time matriz linhas

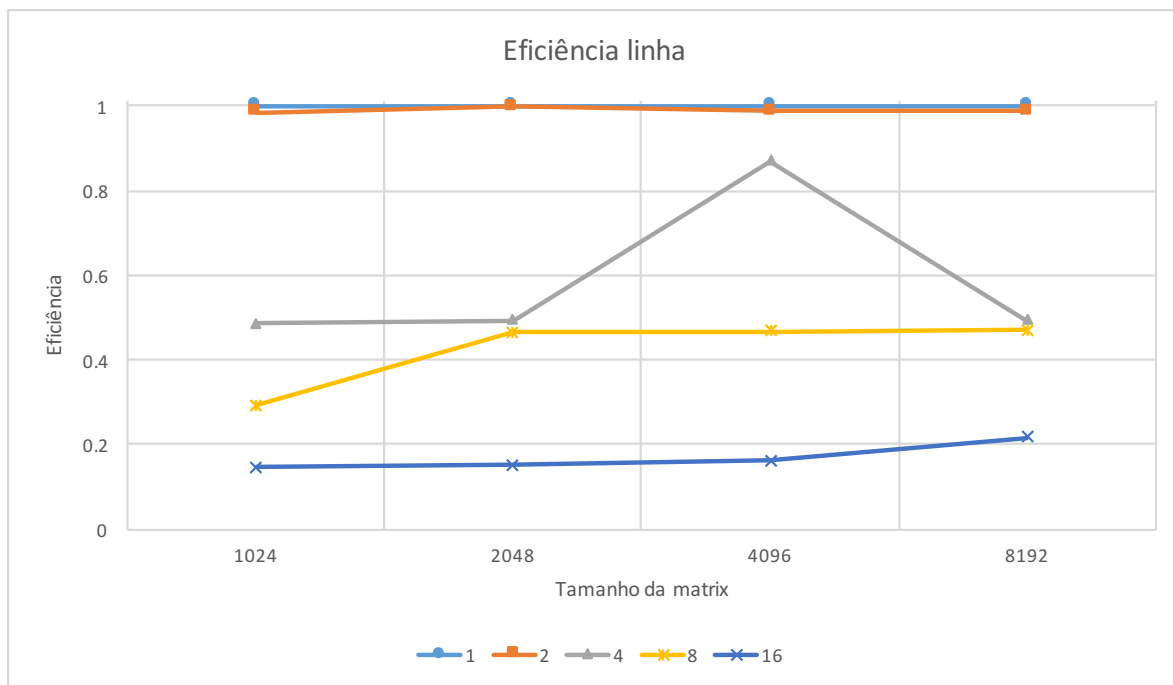


Figura 10 – Eficiência matriz linha

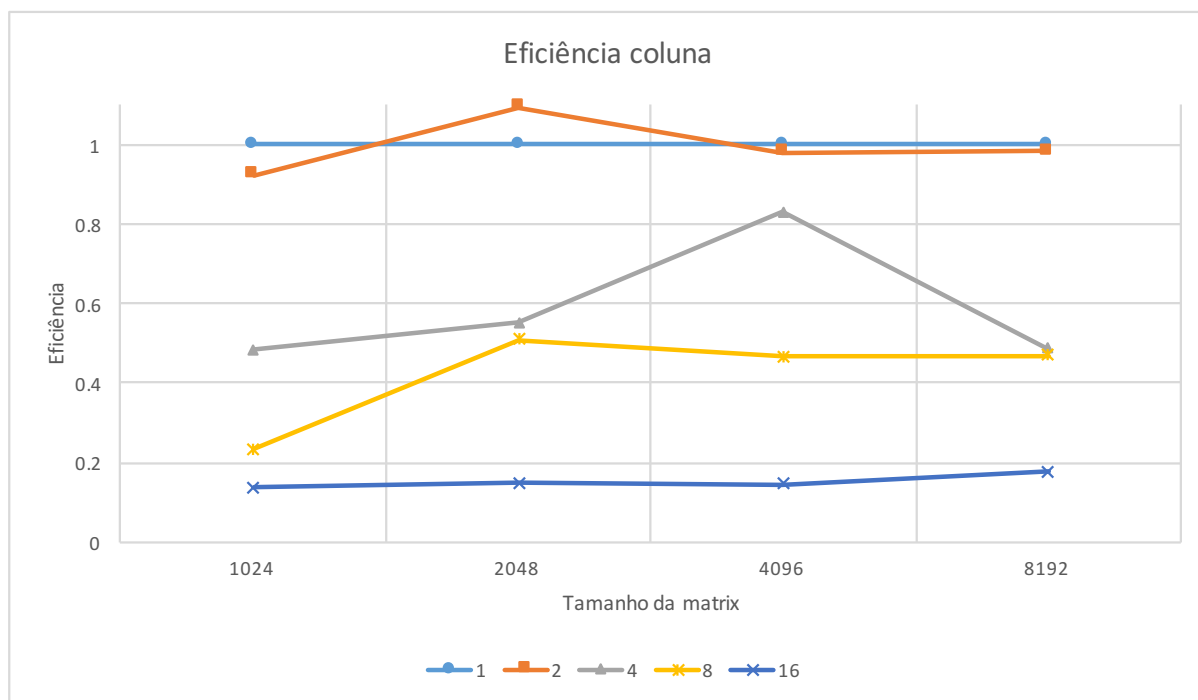


Figura 11 - Eficiência matriz coluna

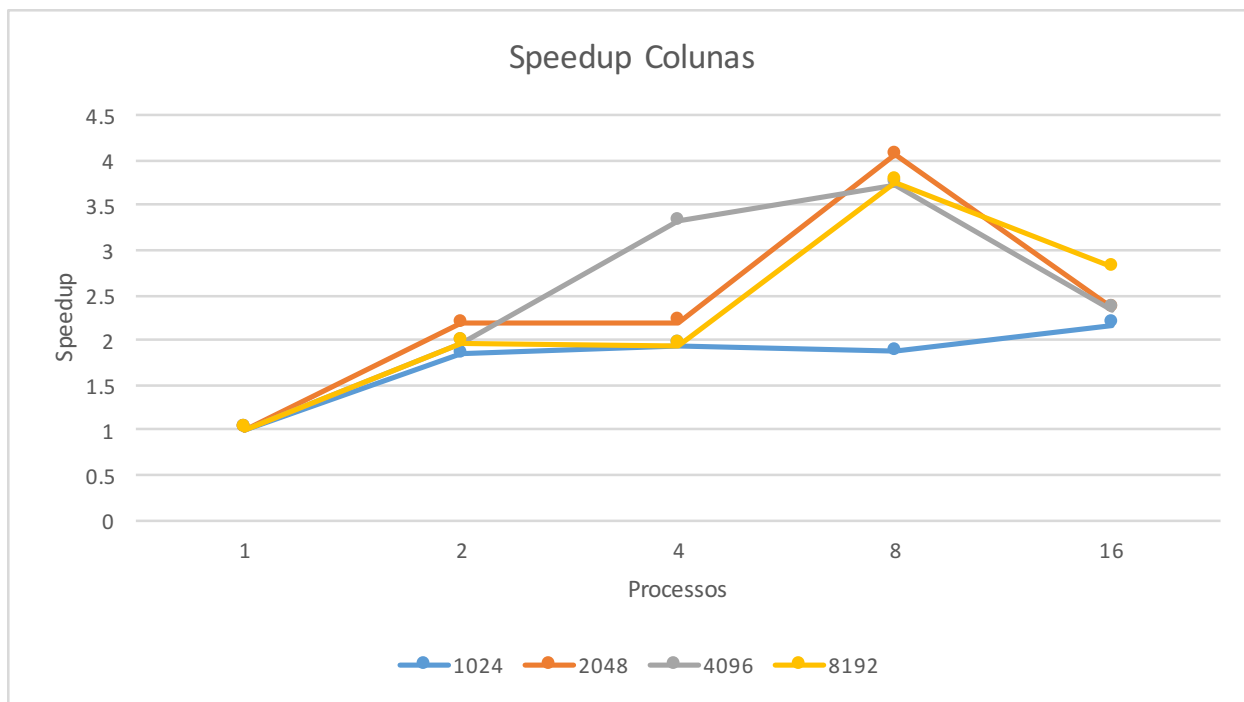


Figura 12 - Speedup matriz colunas

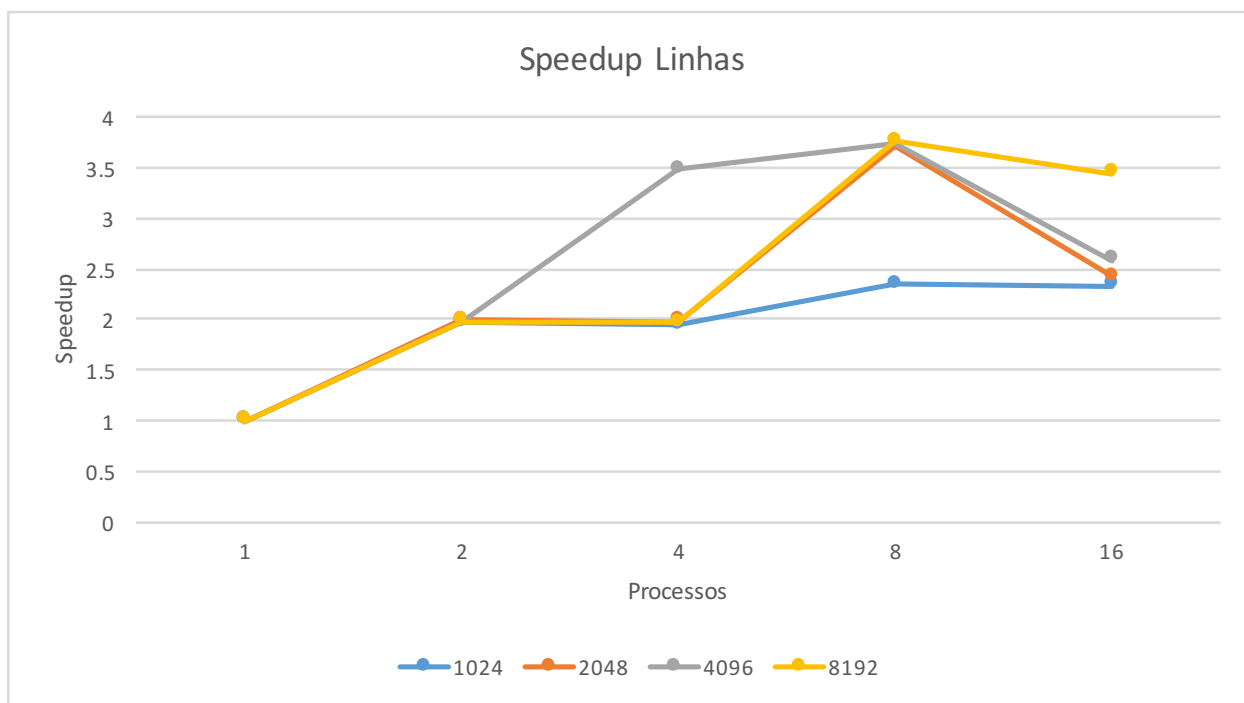


Figura 13 - Speedup matriz linhas

Podemos constatar, a partir dos gráficos que ambos os algoritmos apresentam um comportamento similar. É notório ainda que o algoritmo que implemente a multiplicação por colunas apresenta um tempo de execução um pouco superior ao de linhas. O gráfico da Figura 14 mostra que a diferença de execução entre os algoritmos é muito pequena. Para o tamanho da matriz de 2048, a multiplicação por colunas demorou 2,29% a mais do que a multiplicação por linhas. Essa diferença se dá a maneira como ocorre o armazenamento da matriz na memória cache, que para o caso de multiplicações por colunas, ocorrem mais cache misses do que o a multiplicação por linhas.

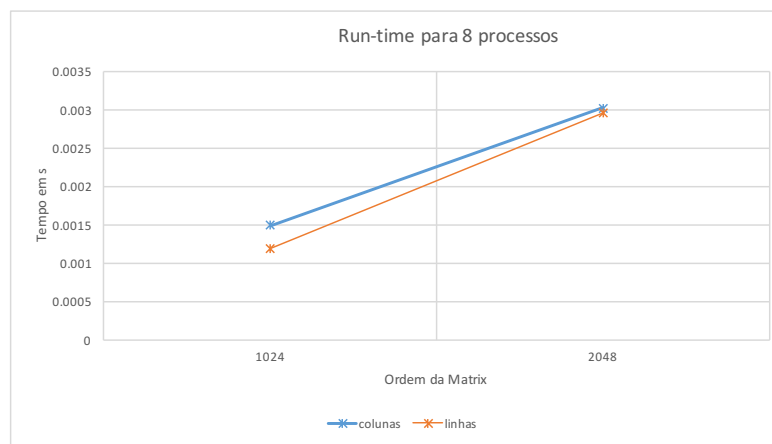


Figura 14 - Comparação para 8 processos da multiplicação matriz \times vetor

Analisando os resultados, vemos que conforme o tamanho do problema aumenta, que nesse caso é a ordem da matriz, o speedup e eficiência também aumentam para ambos os algoritmos. Nessa análise, desconsideramos alguns trechos onde o speedup/eficiência diminui a medida que aumentamos o tamanho do problema, porém, como esse teste não é totalmente representativo, desconsideramos esses trechos. Concluimos então que ambos os algoritmos são escaláveis. Programas fortemente escaláveis mantém a eficiência constante quando aumentamos a quantidade de

processos. Já fracamente escaláveis, mantém a eficiência constante a medida que aumentamos o problema e o número de processos. Para o nosso teste, o programa de multiplicação por colunas se apresentou fracamente escalável no trecho marcado de verde na Tabela 4. O nosso programa de multiplicação da matriz por coluna, obteve um speedup superlinear no trecho marcado de vermelho na Tabela 4. Esse resultado não é comum, visto que apenas em alguns casos quando barreiras físicas de hardware são superadas que é possível atingir uma eficiência maior que 1. No entanto, como o teste foi realizado executando o algoritmo diversas vezes e usando a mediana como valor de referencia, o nosso programa obteve de fato um speedup superliner para esses parâmetros em específico.

	Eficiência Matriz Coluna			
Processos	1024	2048	4096	8192
1	1	1	1	1
2	0.92322	1.09500	0.98232	0.98442
4	0.48479	0.55052	0.82901	0.48635
8	0.23343	0.50782	0.46579	0.46907
16	0.13578	0.14715	0.14584	0.17560

Tabela 4 - Eficiência Matriz Coluna

6. Conclusão

Esse trabalho apresentou uma análise detalhada de desempenho e escalabilidade de algoritmos desenvolvidos usando apenas funções do MPI de ponto a ponto e fez uma comparação com funções já implementadas na biblioteca MPI. Além disso, podemos constatar que estas funções do MPI tem como estrutura uma árvore binária, já a complexidade se mostrou próxima do algoritmo em árvore criado. Também foi constatato que é possível de se obter speedup superlinear quando superamos recursos de hardware. Ferramentas de paralelização de código como o MPI são de fundamental importância para o desenvolvimento de algoritmos e programas paralelos em uma era que a complexidade de problemas aumenta a cada dia. Em um futuro próximo, programação paralela e concorrente será primordial para qualquer programador.

7. Referências

- [1] **Pacheco**, P. S. (2011). An introduction to parallel programming. Amsterdam: Morgan Kaufmann.

- [2] **Mpich**. *Web pages for MPI Routines*. Disponível em <http://www.mpich.org/static/docs/v3.1/www3/>.