# Fases Paralelas Usando MPI

## Trabalho 4 – Programação Paralela e Distribuída

Maiki Buffet
Estudante de Engenharia de Computação
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brasil
maiki.buffet@acad.pucrs.br

***Resumo* — Este trabalho apresenta a implementação de um algoritmo paralelo usando MPI e o paradigma de Fases Paralelas para ordenar um vetor de inteiros usando o algoritmo *BubbleSort*. Isto foi feito para mensurar o *Speed Up* e a eficiência das diferentes técnicas de programação paralela com diferentes tamanhos de dados e números de processos.**

***Palavras-chave* — *Programação Paralela; MPI; Fases Paralelas; InsertionSort; Speed Up; Eficiência.***

## I. INTRODUÇÃO

Não é fácil a criação de algoritmos paralelos por um número de razões. Uma delas é que a maioria das pessoas tendem a pensar de maneira sequencial; outra razão é o fato do tratamento de áreas críticas e troca de mensagens entre processos paralelos não ser trivial. O *Message Passing Interface* (MPI) [1] é um sistema de transmissão de mensagens padronizado que ajuda o programador a realizar estas tarefas de forma facilitada, uma vez que cria e administra a interface na criação de algoritmos paralelos. Neste documento está descrito os testes realizados com um algoritmo paralelo de ordenamento usando o paradigma de Fases Paralelas e um algoritmo sequencial, que foram criados com a finalidade de avaliar as discrepâncias de performance entre eles. O paradigma de Fases Paralelas consiste em cada processo ser atribuído a uma fase par ou ímpar, onde serão intercalados até que a tarefa esteja concluída. A comunicação entre as tarefas ocorre entre a execução de cada fase.

## II. FUNCIONAMENTO

### A. Algoritmo Sequencial

O algoritmo sequencial cria um vetor e o preenche com números inteiros positivos e não repetidos, através da leitura de um arquivo. Após isto começa a contagem do tempo do processamento do método *BubbleSort* [3][4] – responsável pela ordenação crescente dos valores; até o ponto em que o método é finalizado e a contagem encerrada, computando o tempo gasto no processamento; além da entrega dos valores já ordenados.

### B. Algoritmo Divisão e Conquista

No algoritmo Fases Paralelas, o primeiro processo cria o vetor com os valores a serem ordenados através da leitura do arquivo escolhido. Após isto, inicia medindo o tempo e envia o vetor dividido com o tamanho de cada tarefa para outros processos. O tamanho da tarefa é calculado da seguinte forma:

$$Ts = N / NP$$

Onde N é o número de elementos a serem ordenados e NP é o número de processos. O primeiro processo envia um número adicional de inteiros para cada processo diferente – com exceção do último processo – sendo que estes inteiros são compartilhados entre processos adjacentes com a finalidade de comunicar as mudanças no vetor a ser ordenado. Estes inteiros serão referenciados como inteiros conectados. Então a parte que fará a ordenação inicia a execução, sendo que esta parte é dividida em dois laços:

Fase Par:

1) Roda a rotina *BubbleSort* com o vetor do processo.
2) Envia os inteiros conectados para a fase Ímpar.
3) Sinaliza a *flag* de ordenação com o método *MPI_Allreduce* [5].
4) Se todas as fases estiverem ordenadas, encerra o laço.
5) Caso contrário, recebe os inteiros conectados da fase Par.

Fase Ímpar:

1) Recebe os inteiros conectados da fase Par.
2) Executa a rotina *BubbleSort* com o vetor do processo.
3) Sinaliza a *flag* de ordenação com o método *MPI_Allreduce*.
4) Se todas as fases estiverem ordenadas, encerra o laço.
5) Caso contrário, recebe os inteiros conectados da fase Ímpar.

Quando todas as fases tiverem concluído os processos de ordenação, a primeira termina a execução da contagem da medida de tempo, imprime o seu vetor ordenado e envia o tempo gasto para o último processo. Então cada processo imprimirá o seu próprio vetor ordenado e o último processo, além da impressão do seu vetor também imprimirá o tempo total gasto.

## III. EXECUÇÃO

### A. *Execução*

São necessários o envio de quatro parâmetros para a execução de cada algoritmo – ambos integrados no mesmo arquivo fonte:

- O número de processos a serem executados;
- O número de elementos a serem ordenados;
- O nome do arquivo com os elementos a serem lidos;
- O número de inteiros conectados entre cada fase.

## IV. SAÍDA

```
79990: 99984
79991: 99986
79992: 99987
79993: 99988
79994: 99989
79995: 99992
79996: 99993
79997: 99994
79998: 99996
79999: 99997
80000: 100000
A versão fases paralelas demorou 11.878313 segundos para concluir a tarefa.
```

Fig. 1. Saída padrão com elementos ordenados e tempo gasto neste processamento. Na figura foram utilizados 8 processos na ordenação de 80000 valores com 10000 inteiros conectados entre cada fase.

## V. RESULTADOS

### A. *Tempo de Execução*

Três conjuntos diferentes de testes foram feitos para validar a performance:

1. 2000, 4000 e 8000 elementos com o número de elementos conectados igual a 1;
2. 20000, 40000 e 80000 elementos com o número de elementos conectados igual a Ts/2;
3. 20000, 40000 e 80000 elementos com o número de elementos conectados igual a Ts.

Para cada conjunto de testes foram utilizados 2, 4 e 8 processos. A discrepância entre o número de elementos entre o primeiro e os outros testes pode ser explicada pela grande quantidade de tempo que o algoritmo levou para ser executado com o número de elementos conectados igual à 1.

Os resultados foram computados por uma única máquina do cluster Cerrado – composto por 2 enclosures Dell PowerEdge M1000e com 16 blades Dell PowerEdge M610 e 15 blades Dell PowerEdge M620, sendo que cada máquina possui dois processadores Intel Xeon Six-Core E5645 2.4GHz Hyper-Threading e 24GB de memória, totalizando 12 núcleos (24 threads) por nó e 372 núcleos (744 threads) no cluster. Estas máquinas que estão presentes no Laboratório de Alto Desempenho (LAD) [2], executaram cada teste três vezes (contando com três conjuntos de teste para cada experimento) e uma média entre eles fora feita.

Fig. 2. Imagem frontal do cluster Cerrado.

Os tempos de execução para o conjunto 1 foram retratados na Tabela 1.

| Casos de Teste | Sequencial | 2 Processos | 4 Processos | 8 Processos |
|---|---|---|---|---|
| **2000** | 0,014595 | 1,075544 | 0,657359 | 0,261726 |
| **4000** | 0,058237 | 8,932322 | 5,419241 | 1,848973 |
| **8000** | 0,225475 | 80,37158 | 41,745402 | 14,348899 |

Tab. 1. Tempo de Execução em segundos do conjunto 1.

O tempo de execução está ilustrado no Gráfico 1.


Gráf. 1. Gráfico do tempo de execução em função da quantidade de processos do conjunto 1.

Os tempos de execução para o conjunto 2 foram retratados na Tabela 2.

| Casos de Teste | Sequencial | 2 Processos | 4 Processos | 8 Processos |
|---|---|---|---|---|
| **20000** | 1,459915 | 2,773418 | 1,242364 | 0,589273 |
| **40000** | 5,982604 | 11,230482 | 4,913252 | 2,12993 |
| **80000** | 24,577337 | 45,755914 | 19,867853 | 8,566835 |

Tab. 2. Tempo de Execução em segundos do conjunto 2.

O tempo de execução está ilustrado no Gráfico 2.


Gráf. 2. Gráfico do tempo de execução em função da quantidade de processos do conjunto 2.

Os tempos de execução para o conjunto 3 foram retratados na Tabela 3.

| Casos de Teste | Sequencial | 2 Processos | 4 Processos | 8 Processos |
|---|---|---|---|---|
| **20000** | 1,458922 | 2,704717 | 1,275457 | 0,620893 |
| **40000** | 5,992393 | 10,989221 | 5,002012 | 2,442084 |
| **80000** | 24,53955 | 44,362644 | 20,389839 | 9,670367 |

Tab. 3. Tempo de Execução em segundos do conjunto 3.

O tempo de execução está ilustrado no Gráfico 3.


Gráf. 3. Gráfico do tempo de execução em função da quantidade de processos do conjunto 3.

## *B. Speed Up*

O *Speed Up* para um determinado número de processos é calculado pela fórmula:

$$Spn = Ts / Tn$$

Onde *Ts* é o tempo sequencial e *Tn* é o tempo com *n* processos.

Os *Speed Ups* para o conjunto 1 foram retratados na Tabela 4.

| Casos de Teste | 2 Processos | 4 Processos | 8 Processos |
|---|---|---|---|
| **2000** | 0,01357 | 0,022202 | 0,055764 |
| **4000** | 0,00652 | 0,010746 | 0,031497 |
| **8000** | 0,002805 | 0,005401 | 0,015714 |

Tab. 4. *Speed Up* do conjunto 1.

O *Speed Up* do conjunto 1 está ilustrado no Gráfico 4.


Gráf. 4. Gráfico do *Speed Up* em função da quantidade de processos do conjunto 1.

Os *Speed Ups* para o conjunto 2 foram retratados na Tabela 5.

| Casos de Teste | 2 Processos | 4 Processos | 8 Processos |
|---|---|---|---|
| **20000** | 0,526396 | 1,175111 | 2,477485 |
| **40000** | 0,532711 | 1,217646 | 2,808827 |
| **80000** | 0,53714 | 1,23704 | 2,868893 |

Tab. 5. *Speed Up* do conjunto 2.

O *Speed Up* do conjunto 2 está ilustrado no Gráfico 5.


Gráf. 5. Gráfico do *Speed Up* em função da quantidade de processos do conjunto 2.

Os *Speed Ups* para o conjunto 3 foram retratados na Tabela 6.

| Casos de Teste | 2 Processos | 4 Processos | 8 Processos |
|---|---|---|---|
| **20000** | 0,539399 | 1,143843 | 2,349716 |
| **40000** | 0,545297 | 1,197997 | 2,453803 |
| **80000** | 0,553158 | 1,203519 | 2,537603 |

Tab. 6. *Speed Up* do conjunto 3.

O *Speed Up* do conjunto 3 está ilustrado no Gráfico 6.


Gráf. 6. Gráfico do *Speed Up* em função da quantidade de processos do conjunto 3.

### C. *Eficiência*

A eficiência para um determinado número de processos é calculada pela fórmula:

$$Ef_n = Sp_n / n$$

Onde *Spn* é o *Speed Up* com um determinado número de processos e *n* é o número de processos.

A eficiência do conjunto 1 foi retratada na Tabela 7.

| Casos de Teste | 2 Processos | 4 Processos | 8 Processos |
|---|---|---|---|
| **2000** | 0,006785 | 0,005551 | 0,006971 |
| **4000** | 0,00326 | 0,002687 | 0,003937 |
| **8000** | 0,001403 | 0,00135 | 0,001964 |

Tab. 7. Eficiência do conjunto 1.

A eficiência do conjunto 1 está ilustrada no Gráfico 7.


Gráf. 7. Gráfico da eficiência em função da quantidade de processos do conjunto 1.

A eficiência do conjunto 2 foi retratada na Tabela 8.

| Casos de Teste | 2 Processos | 4 Processos | 8 Processos |
|---|---|---|---|
| **2000** | 0,263198 | 0,293778 | 0,309686 |
| **4000** | 0,266356 | 0,304412 | 0,351103 |
| **8000** | 0,26857 | 0,30926 | 0,358612 |

Tab. 8. Eficiência do conjunto 2.

A eficiência do conjunto 2 está ilustrada no Gráfico 8.


Gráf. 8. Gráfico da eficiência em função da quantidade de processos do conjunto 2

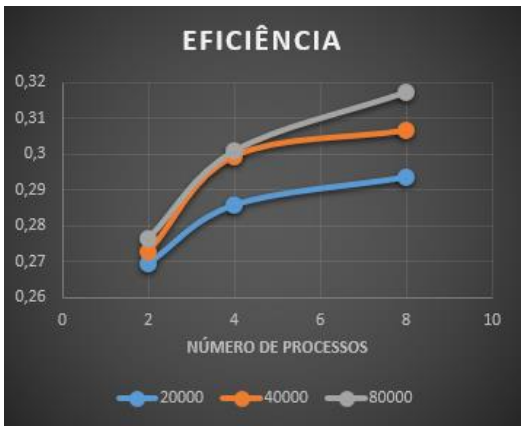
.

A eficiência do conjunto 3 foi retratada na Tabela 9.

| Casos de Teste | 2 Processos | 4 Processos | 8 Processos |
|---|---|---|---|
| **2000** | 0,2697 | 0,285961 | 0,293714 |
| **4000** | 0,272649 | 0,299499 | 0,306725 |
| **8000** | 0,276579 | 0,30088 | 0,3172 |

Tab. 7. Eficiência do conjunto 1.

A eficiência do conjunto 3 está ilustrada no Gráfico 9.




Gráf. 9. Gráfico da eficiência em função da quantidade de processos do conjunto 3.

## VI. CONCLUSÃO

Pode-se concluir através das análises anteriores que este algoritmo funciona melhor com um considerado número de inteiros conectados e ao menos com quatro processos. Isto pode ser visto nos resultados do primeiro conjunto se comparados aos outros – mesmo com diferente número de elementos conectados – o primeiro conjunto apenas atingiu *Speed Ups* menores que 1. A explicação para isto pode ser explanada pelo número de iterações necessárias para ordenar o vetor – o qual é incrementado pelo número de inteiros conectados sendo transferidos, neste caso 1. Conclui-se também que o aumento do número de inteiros conectados tem resultados decrescentes no desempenho. O motivo disto é mostrado nos resultados do segundo conjunto em comparação ao terceiro, uma vez que

eles são quase os mesmos, sendo explicados pelo aumento do tempo de comunicação por conta do número de inteiros conectados sendo transferidos. Também é possível de se concluir que este algoritmo possui uma fraca eficiência com qualquer possibilidade de combinação dos números de processos e inteiros conectados. Nos gráficos 7, 8 e 9 é visto que pela segmentação do algoritmo de ordenação sendo paralelizado, o algoritmo *BubbleSort* é ineficiente e o segmentando com a função de se buscar melhor performance irá necessariamente aumentar a ineficiência do mesmo.

## REFERÊNCIAS


[1] *Message Passing Interface (MPI)*, Blaise Barney, Lawrence Livermore National Laboratory, Livermore, CA, EUA. Disponível em: computing.llnl.gov/tutorials/mpi/

[2] *Laboratória de Alto Desempenho*, Faculdade de Informática, PUCRS, Porto Alegre, RS, Brasil. Disponível em: www3.pucrs.br/portal/page/portal/ideia/Capa/LAD/LADInfraestrutura/LADInfraestruturaHardware

[3] *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers - Chapter 9 Sorting Algorithms*, Barry Wilksinson and Michael Allen, Prentice Hall, Upper Saddle River, NJ, EUA. Disponivel em: grid.cs.gsu.edu/~cscyip/csc4310/Slides9.pdf

[4] *Bubble Sort and its variants - Introduction to Parallel Computing*, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar

[5] *PI Allreduce - MPICH - High-Performance Portable MPI.* Disponível em: www.mpich.org/static/docs/v3.1/www3/MPI Allreduce.html

*Available at: http://parallelcomp.uw.hu/ch09lev1sec3.html*