Pipeline Usando MPI

Trabalho 2 – 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 *Pipeline* para ordenar um vetor de inteiros usando o algoritmo InsertionSort. 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; Pipeline; 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. Para descrever isto, um algoritmo paralelo de ordenamento usando o paradigma de *Pipeline* e um algoritmo sequencial, em contrapartida, foram criados com a finalidade de avaliar as discrepâncias de performance entre eles. O paradigma de *Pipeline* consiste na criação de um número de estágios de execução – barreiras – a fim de paralisar o algoritmo, desde que cada estágio seja atribuído a um processo diferente.

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 *InsertionSort* [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 Pipeline

No algoritmo *Pipeline*, o primeiro estágio cria um vetor e o preenche com números inteiros positivos e não repetidos, através da leitura de um arquivo. Após isto inicia o tempo de computação e começa o ordenamento dos elementos, inserindo-os num vetor menor – cuja função será a de ser o vetor responsável pelo ordenamento de cada estágio. O tamanho deste vetor foi definido como:

$$S = N / NP$$

Onde S é o tamanho do vetor de cada estágio, N é o número de elementos a serem ordenados e NP o número de processos utilizados. Quando o vetor do estágio está cheio, é enviado o elemento excedente ao próximo estágio, a fim de ser ordenado e inserido no seu próprio vetor. Quando o último estágio termina o ordenamento, é enviado um sinal ao primeiro estágio para encerrar a computação do tempo gasto neste processo. Cada estágio imprime o seu vetor ordenado e após isto o primeiro estágio envia ao último o tempo gasto no processo, com a finalidade de o imprimir na tela e encerrar a execução do programa.

III. Execução

A. Execução

São necessários o envio de três 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.

IV. Saída

```
99990: 99990
99991: 99991
99992: 99992
99993: 99993
99994: 99994
99995: 99995
99996: 99996
99997: 99997
99998: 99998
99999: 99999
100000: 100000
A versão pipeline demorou 7.93 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 100000 valores.

V. RESULTADOS

A. *Tempo de Execução*

Os tempos de execução foram retratados na tabela 1.

Casos de Teste	Sequencial	2 Processos	4 Processos	8 Processos
25000	1.52	1.29	0.85	0.59
50000	5.99	5.1	3.3	2.08
100000	24.01	20.38	13.08	7.93

Tab. 1. Tempo de Execução em segundos.

Os resultados foram computados por uma única máquina do *cluster* Atlântica – composto por 16 máquinas Dell PowerEdge R610. Casa máquina possui dois processadores Intel Xeon Quad-Core E5520 2.27GHz Hyper-Threading e 16 GB de memória, totalizando 8 núcleos (16 threads) por nó e 128 núcleos (258 threads) no *cluster*. Os nós estão interligados por 4 redes Gigabit-Ethernet chaveadas. O último nó (atlantica16), dispõe de uma NVIDIA Tesla S2050 Computing System, com 4 NVIDIA Fermi Computing Processors (448 CUDA cores cada) divididos em 2 hosts interfaces e 12GB de memória (3GB por GPU). Estas máquinas que estão presentes no Laboratório de Alto Desempenho (LAD) [2], executaram cada teste três vezes e uma média entre eles fora feita.

Fig. 2. Imagem frontal do cluster Atlântica.

O tempo de execução está ilustrado no gráfico 1.

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

B. *Speed Up*

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

$$Sp_n = Ts / Tn$$

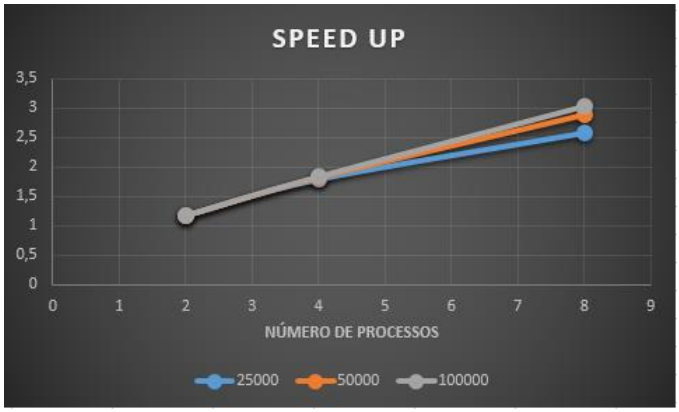
Onde *Ts* é o tempo sequencial e *Tn* é o tempo com *n* processos.

Os *Speed Ups* foram retratados na tabela 2.

Casos de Teste	2 Processos	4 Processos	8 Processos
25000	1.178	1.788	2.576
50000	1.175	1.815	2.88
100000	1.178	1.836	3.028

Tab. 2. Speed Up

O *Speed Up* está ilustrado no gráfico 2.



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

C. *Eficiência*

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

$$Ef_n = Sp_n / n$$

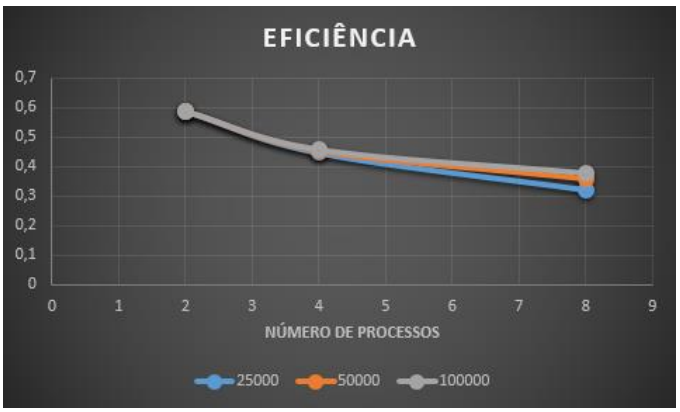
Onde *Spn* é o *Speed Up* com um determinado número de processos e *n* é o número de processos.

A eficiência foi retratada na tabela 3.

Casos de Teste	2 Processos	4 Processos	8 Processos
25000	0.589	0.447	0.322
50000	0.587	0.454	0.36
100000	0.589	0.489	0.379

Tab. 3. Eficiência.

A eficiência está ilustrada no gráfico 3.



Gráf. 3. Gráfico da eficiência em função da quantidade de processos.

VI. CONCLUSÃO

Os resultados de performance/eficiência não foram tão bons quanto a versão paralela utilizando o paradigma de *Master-Slave*. Embora a complexidade do algoritmo de ordenamento seja:

$$O(n^2)$$

Significa que, por exemplo, fazendo a divisão do vetor com os elementos por quatro, na realidade o resultado do custo computacional está sendo dividido por dezesseis:

$$(n / 4)^2 = n^2 / 16$$

Conclui-se que existem alguns problemas com o paradigma *Pipeline* para este algoritmo: a execução é muito simplificada pois os estágios só ordenam um elemento por vez e os elementos precisam passar por vários estágios desnecessários de comunicação. Isto é mostrado no gráfico de eficiência (Gráfico 3); mesmo que os valores de *Speed Up* continuem crescendo, os valores de eficiência caem a medida que mais estágio/elementos são adicionados.

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] *Insertion Sort*, Core War, Reino Unido. Disponível em: http://corewar.co.uk/assembly/insertion.htm