Pipeline Usando MPI

Trabalho 3 – 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 Divisão e *Conquista* para ordenar um vetor de inteiros usando o algoritmo MergeSort. 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; Divisão e Conquista; 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 Divisão e Conquista e um algoritmo sequencial, que foram criados com a finalidade de avaliar as discrepâncias de performance entre eles. O paradigma de Divisão e Conquista consiste em cada processo fazer uma avaliação se a tarefa que foi lhe atribuída é pequena o bastante para ser processada (conquistada), caso contrário, o processo divide a tarefa e envia partes menores da mesma para os processos filhos, a fim de descobrir se eles podem processá-la ou não.

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 *MergeSort* [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 Divisão e Conquista, 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 avalia se a tarefa é menor que o tamanho limite. O limite pode ser calculado pela fórmula abaixo, respeitando apenas a utilização de valores cujo resultado seja um número inteiro:

$$L = N / NP$$

Onde *L* é o limite da tarefa, *N* é o número de elementos a serem ordenados e *NP* é o número de processos. Até que que o número da tarefa seja menor ou igual ao limite, o primeiro processo divide a tarefa por 2 e a envia a um processo filho. Para avaliar qual processo receberá a tarefa, pode-se utilizar a seguinte fórmula:

$$C_{rank} = (F_{rank} + 2^a)$$

Onde F_{rank} é o rank da fonte e *a* é o nível da árvore. Então o primeiro processo ordena seu próprio vetor e aguarda o retorno de um processo filho com a sua parte ordenada. O processo filho faz a mesma avaliação e cria qualquer processo filho necessário para conquistar a tarefa. Toda que um processo filho retorna um vetor ordenado de volta ao pai, este faz um *merge* do vetor recebido com o seu próprio. Quando o primeiro processo finaliza as recepções e os *merges*, ele finaliza a contagem do tempo de execução das tarefas e entrega o vetor ordenado e as medições.

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 divisão e conquista demorou 0.010008 segundos para concluir a tarefa.
```

Fig. 1. Saída padrão com elementos ordenados e tempo gasto neste processamento. Na figura foram utilizados 16 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	16 Processos
500000	0,182733	0,110852	0,068867	0,052782	0,050309
1000000	0,381686	0,224974	0,144442	0,104693	0,085298
2000000	0,79893	0,46533	0,289892	0,208445	0,169216

Tab. 1. Tempo de Execução em segundos.

Os resultados foram computados por duas máquinas 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:

$$Spn = Ts / Tn$$

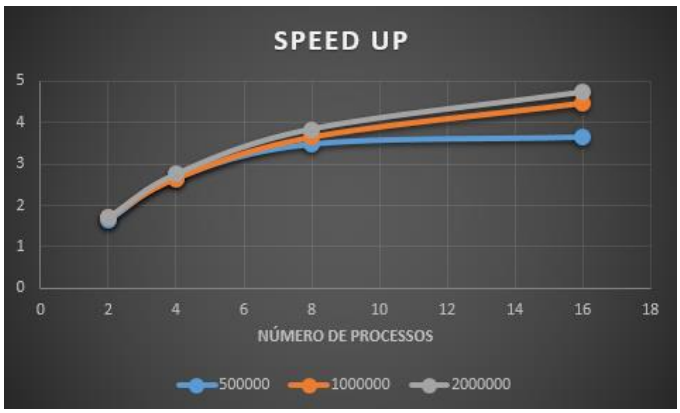
Onde *Ts* é o tempo sequencial e *Tn* é o tempo com *n* processos.

Os *Speed Ups* foram retratados na Tabela 2.

Casos de Teste	2 Processos	4 Processos	8 Processos	16 Processos
500000	1,648441	2,653419	3,462033	3,632213
1000000	1,696578	2,642486	3,645764	4,474736
2000000	1,716911	2,755957	3,83281	4,721362

Tab. 2. Speed Up

O *Speed Up* está ilustrado no Gráfico 2.



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

Através da análise dos resultados do tempo de execução e do *Speed Up*, entende-se que quanto menores forem os valores utilizados nem paradigma, menores serão os valores relativos do *Speed Up*. Isto ocorre devido ao fato que na execução sequencial, não temos nenhum *overhead* de comunicação entre os processos. Porém com valores altos, este resultado se inverte, e o *Speed Up* se torna muito maior. Neste caso, pode-se averiguar que a divisão do processamento

é bastante eficiente – em termos de tempo de execução e paralelismo.

C. *Eficiência*

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

$$Efn = Spn / n$$

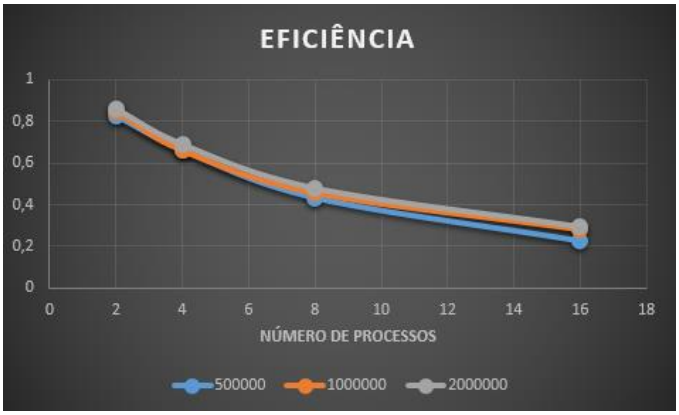
Onde *Spn* é o *Speed Up* com um determinado número de processos e *n* é o número de processos.

A eficiência foi retratada na Tabela 3.

Casos de Teste	2 Processos	4 Processos	8 Processos	16 Processos
500000	0,824221	0,663355	0,432754	0,227013
1000000	0,848289	0,660622	0,455721	0,279671
2000000	0,858455	0,688989	0,479101	0,295085

Tab. 3. Eficiência.

A eficiência está ilustrada no Gráfico 3.



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

Após analisarmos os resultados anteriores e compararmos com os dados sobre a eficiência, pode-se constatar que – mesmo com valores grandes – a inclusão de inúmeros processos a fim de dividir ainda mais as tarefas não tornará a execução melhor – se comparar o tempo de execução, *Speed Up* e os recursos necessários para a inclusão de novos processadores.

VI. Conclusão

Pode-se concluir que o algoritmo é melhor utilizado com uma grande quantidade de elementos. Isto pode ser visto no gráfico de eficiência (Gráfico 3), quando utilizados 2000000 elementos se comparado à execução com 500000. Isto pode ser explicado pelo custo de comunicação e expansão dos níveis da árvore, apenas valendo o uso com grandes valores. É importante notar que com um grande número de elementos e um pequeno limite das tarefas, a máquina executante precisará de um hardware para rodar uma quantidade suficiente de *threads* ao mesmo tempo (isto não ocorreu nesta implementação em função do modo que se calculou este limite). Dessa forma, este algoritmo pode ser muito dependente de hardware dependendo dos parâmetros do mesmo. Após obtermos os resultados das execuções das versões Mestre-Escravo, Pipeline e Divisão e Conquista, iremos aplicar a técnica de paradigma Fases Paralelas a fim de mensurar qual das técnicas abordadas se destaca mais em termos de tempo de execução, *Speed Up*, eficiência, complexidade de implementação e custo de processamento.

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] *Merge Sort – Sorting Algorithm Animations*, David R. Martin. Disponível em: www.sorting-algorithms.com/merge-sort