Programação Paralela com MPI

Programação Paralela utilizando o modelo Mestre-Escravo

Marcelo Melo Linck
Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS
Faculdade de Informática - FACIN
Porto Alegre, Brasil
marcelo.linck@acad.pucrs.br

***Resumo*—Este documento relata as dificuldades, problemas, soluções e resultados encontrados durante o desenvolvimento de um programa desenvolvido para execução paralela. Utilizou-se a estrutura MPI e o algoritmo modelo Mestre-Escravo.**

***Abstract*—This paper relates the difficulties, problems, solutions and results found during the development software biased to work using parallel processing. For this project, the MPI communication structure was used, as well as the model Master-Slave algorithm.**

***Palavras Chave*—MPI; Programação Paralela; Merge Sort; Rank Sort; Mestre-Escravo; Ordenação.**

I. INTRODUÇÃO

Este documento apresenta o fluxo de desenvolvimento de um Programa cuja execução utiliza processos em paralelo para uma possível melhoria de desempenho. Neste relatório, são apresentadas comparações entre a execução do programa desenvolvido utilizando diversos números de processos, assim como o mesmo de forma sequencial. Além dos resultados, é apresentado o modelo utilizado (Mestre-Escravo) e sua forma de implementação

II. PROJETO E DESENVOLVIMENTO

A. Problema Inicial

O problema inicial apresentado foi o desafio de implementar um sistema paralelo utilizando o modelo Mestre-escravo. Este sistema teria o propósito de realizar a leitura de um arquivo texto contendo até 100.000 números aleatoriamente organizados, e organizá-los de forma crescente utilizando processos paralelos. O algoritmo apresentado para a organização foi o *Rank Sort*, o qual será abordado mais adiante.

B. Modelo Mestre-Escravo

Este modelo tem como principal característica, como o nome já diz, a existência de um processo mestre e *n* processos escravos. É possível imagina uma fila de processos lado a lado, sendo todos controlados por um só, o qual seria o mestre, desta maneira é possível a implementação da solução de vários problemas utilizando programação paralela.

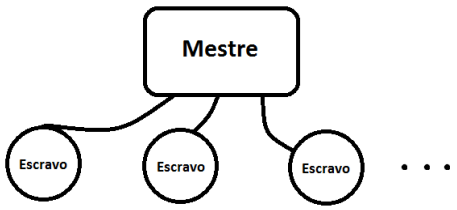

Fig. 1 - Modelo Mestre-Escravo.

C. Implementação e Soluções

Para a solução do desafio proposto foi implementado um sistema utilizando o modelo apresentado no tópico anterior (II.B.). A estrutura foi organizada da seguinte maneira: O mestre é encarregado de calcular o número de tarefas de ordenação que devem ser mandadas para cada escravo, além disso, ele é o responsável pela fusão e organização dos vetores de retorno de cada escravo, envio e recepção de informações do escravo, e por fim, ele é o único processo que realiza impressões na tela. Já o escravo apenas recebe a posição que precisa iniciar a ordenação, já sabendo até que ponto deve ser ordenado (calculado no início do programa), ele ordena e retorna o vetor ao mestre.

D. Ordenação e Organização

A parte de ordenação do vetor, por cada escravo (paralelo) ou pelo programa sequencial, é feita da mesma maneira. O algoritmo utilizado é o *Rank Sort*, este, funciona da seguinte maneira: Em um loop, pega a primeira posição do vetor de entrada e verifica o número de valores menores que aquele, para o número de valores menores, ele incrementa uma variável, que após a verificação, vira a posição em que o valor será inserido em um novo vetor (vetor ordenado).

Para a organização, o algoritmo utilizado é o *Merge Sort*, que pega dois vetores organizados em uma única variável vetor, divididos por um elemento no meio. O primeiro elemento do vetor é salvo e entra em um loop, verificando a existência de um elemento menor que ele, no momento que há, passa para o outro lado do vetor, assim, de um lado para o outro, o vetor total é organizado crescentemente.

O algoritmo *Merge Sort* é mais eficiente pois ele é de ordem linear, já um algoritmo de ordenação, como o *Rank Sort*, possui tempo de ordem quadrática, ou seja, O(n^2).

E. Compilação e Execução

A compilação é recomendada ser feita, por acesso remoto ao LAD (Laboratório de Alto Desempenho). Para isso, seguem os passos abaixo:

1. Cópia do arquivo:
   scp arquivo.c usuario@marfim.lad.pucrs.br:
2. Acesso ao LAD:
   ssh usuario@marfim.lad.pucrs.br
3. Compilação:
   ladcomp -env mpiCC arquivo.c -o exec
4. Verificação de slot livre:
   ladqview
5. Alocação:
   ladalloc –c cluster -n <n_processos> -t <tempo>
   -e(exclusivo)/-s(compartilhado)
6. Execução:
   ladrun –np <n_processos> exec <n_elementos>

III. TESTES E RESULTADOS

A. Interface

Abaixo são apresentados algumas imagens mostrando a interface do programa após sua execução.

```
ppd59011@gates16:~/sort$ ladrun -np 13 s 80000
mpirun -machinefile MACHINEFILE -np 13 s  80000


        Parallel Mode!
        N Slaves: 12
        Total Elements: 80000
        Elements per task: 1666

        Time: 0s140ms837us
```

Fig. 2 - Execução para 80.000 elementos com 13 processos.

```
ppd59011@gates16:~/sort$ ladrun -np 1 s 5000
mpirun -machinefile MACHINEFILE -np 1 s  5000


        Sequencial Mode!
        Total Elements: 5000


        Time: 0s257ms660us
```

Fig. 3 - Execução para 5.000 elementos com 1 processo (sequencial).

B. Tarefas por escravo

Para fins de teste, o programa foi modificado de forma que os escravos imprimissem, ao final do programa, o número de tarefas que cada um executou. Idealmente, de acordo com a especificação, cada processo deveria executar exatamente 4 tarefas, mas verificando as imagens abaixo, vemos que esta regra não se aplica perfeitamente.

```
ppd59011@atlantica16:~/sort$ ladrun -np 5 s 60000
mpirun -machinefile MACHINEFILE -np 5 s  60000


        Parallel Mode!
        N Slaves: 4
        Total Elements: 60000
        Elements per task: 3750
        Slave 4 performed 4 tasks.
        Slave 3 performed 4 tasks.
        Slave 1 performed 5 tasks.
        Slave 2 performed 3 tasks.
```

Fig. 4 - Execução para 5 processos com 60.000 elementos.

```
ppd59011@atlantica16:~/sort$ ladrun -np 3 s 1000
mpirun -machinefile MACHINEFILE -np 3 s  1000


        Parallel Mode!
        N Slaves: 2
        Total Elements: 1000
        Elements per task: 125
        Slave 1 performed 5 tasks.
        Slave 2 performed 3 tasks.
ppd59011@atlantica16:~/sort$ ladrun -np 3 s 100000
mpirun -machinefile MACHINEFILE -np 3 s  100000


        Parallel Mode!
        N Slaves: 2
        Total Elements: 100000
        Elements per task: 12500
        Slave 1 performed 4 tasks.
        Slave 2 performed 4 tasks.
```

Fig. 5 - Execução com 3 processos para 1.000 e 100.000 elementos.

Analisando a fig.4, verificamos que o escravo 3 e 4 possuem o mesmo número de tarefas, e logo, executam cada tarefa em aproximadamente o mesmo tempo, já os escravos 1 e 2 são visivelmente diferentes em relação ao tempo de execução de cada tarefa, sendo o escravo 1 mais rápido por executar mais tarefas que o escravo 2.

Em relação à fig. 5, podemos verificar o mesmo número de processos sendo executados utilizando um número muito alto de elementos e um número muito baixo. Para o número muito alto de elementos, ambos escravos executam o mesmo número de tarefas, já para um número menor, o escravo 1 é mais rápido. Assim, podemos concluir que, para tarefas mais simples, o processador utilizado no escravo 1 é mais eficiente, porém, para tarefas mais demoradas, ambos processadores convergem sua velocidade no mesmo valor, assim levando praticamente o mesmo tempo para a execução.

C. Tempo e Speed Up

Speed Up é o fator de aceleração do tempo de um programa paralelo em relação ao número de processos em que ele é executado. Ele é calculado verificando a variação do tempo sequencial pelo tempo paralelo. Um programa paralelo ideal, é um programa com *Speed Up* igual ao número de processos em execução, ou seja, seu gráfico *N_Processos x Speed Up* é inteiramente linear.

$$Sp_n = \frac{T_s}{T_n}$$

Equação 1 - Speed Up.

Existem outros diferentes comportamentos do *Speed Up*, i.e., o Super Linear, onde Sp>Np, e a mais comum no mundo real, Sp<Np.

A seguir é mostrada uma tabela com algumas capturas de tempo em relação ao número de elementos e o número de processos. (Modelo: NsegundosNmileNmicro)

Tabela 1 - Tempo: n_elementos x processos.

	100	1000	5000	20000
1	0s0ms112us	0s10ms825us	0s257ms607us	4s122ms821us
3	0s1ms347us	0s2ms148us	0s18ms565us	0s264ms842us
5	0s2ms30us	0s2ms413us	0s6ms776us	0s81ms320us
7	0s3ms139us	0s3ms791us	0s5ms522us	0s34ms378us
9	0s3ms839us	0s3ms967us	0s5ms933us	0s26ms404us
13	0s5ms384us	0s6ms69us	0s8ms377us	0s19ms427us

	40000	80000	100000
1	16s654ms835us	67s371ms533us	104s127ms719us
3	1s34ms82us	4s119ms146us	6s435ms838us
5	0s267ms705us	1s41ms871us	2s369ms961us
7	0s125ms484us	0s473ms784us	1s264ms203us
9	0s76ms773us	0s278ms774us	0s429ms5us
13	0s42ms369us	0s140ms774us	0s213ms131us

Abaixo, é apresentado um gráfico do comportamento do *Speed Up* no programa desenvolvido utilizando os valores de tempo, processos e elementos apresentados na tabela 1.

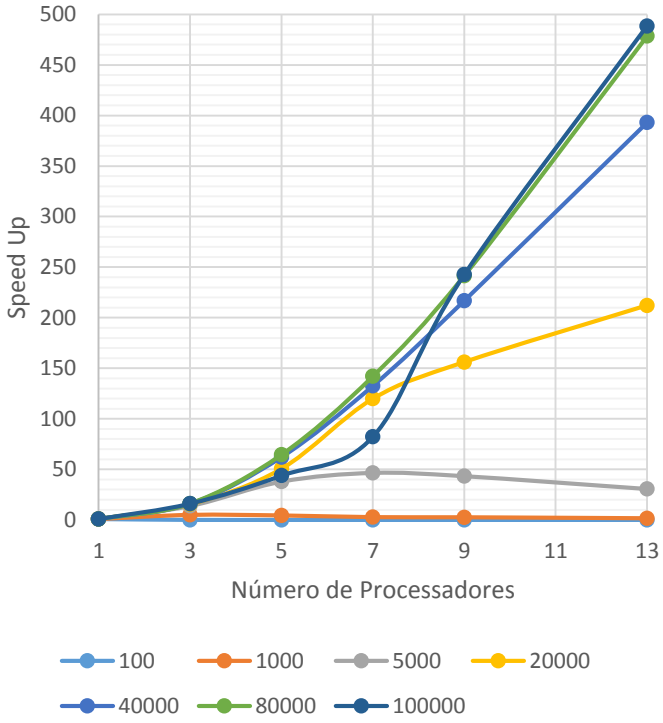

Fig. 6 - Gráfico Speed Up.

Verificando o gráfico e a tabela apresentados anteriormente, podemos concluir que, para um número grande de elementos, o aumento do número de processadores aumenta o valor do *Speed Up*. Porém, para um valor pequeno de elementos, a afirmação anterior não pode ser considerada. Se analisarmos a tabela, verificamos que para 100 elementos, rodando sequencialmente, o tempo foi muito menor que para 1000 ou mais. Para um número tão pequeno de elementos, o tempo de comunicação entre processos, juntamente com o tempo de organização do vetor e o tempo da ordenação paralela, se tornam maiores que o tempo de uma simples ordenação sequencial.

Outra exceção à regra é, novamente nos menores valores, de 100 à 5.000. É possível notar que, nestes casos, após 7 processadores (6 escravos), o tempo total do programa vai aumentando. Este comportamento indica que os processadores começam a encarar cada tarefa como custosa, e entrando no mesmo motivo indicado anteriormente, no caso de 100 elementos sequenciais.

D. Análise de Desempenho

A análise de desempenho de um sistema de execução paralela é medida pela eficiência do mesmo, ou seja, é medida pela utilização do processador. A mesma, é calculada pelo *Speed Up* sobre o número de processadores rodando.

$$E_n = \frac{Sp_n}{n}$$

Equação 2 - Eficiência.

O Gráfico a seguir mostra a relação da eficiência pelo número de processadores em execução, utilizando os valores de elementos dos dados anteriores.

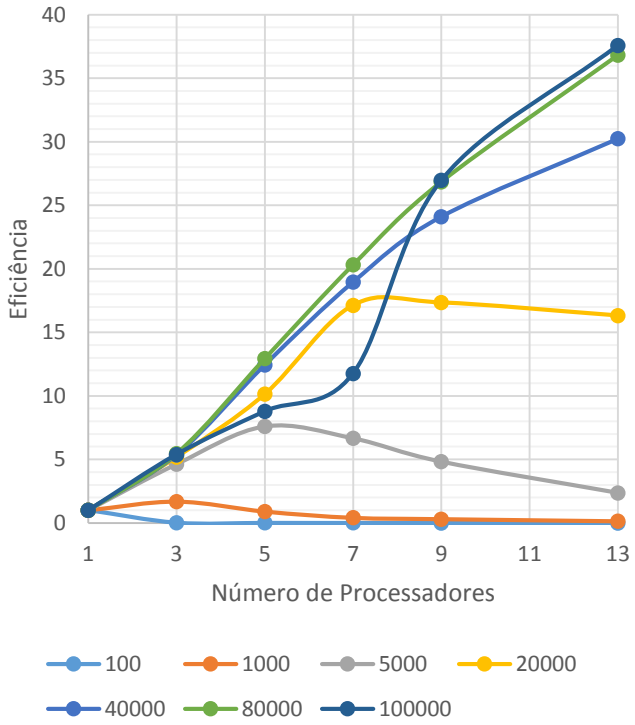

Fig. 7 - Gráfico Eficiência.

Pela análise de desempenho indicada acima, e especialmente do modelo Mestre-Escravo, podemos concluir que, para tarefas com alto número de processamento (ordenação de muitos elementos com muitos processos), este modelo se comporta muito bem, nota-se que para um aumento gradativo do valor dos elementos e processos, a eficiência vai aumentando também. É possível notar uma estupenda diferença de eficiência entre 5.000 e 20.000, assim como de 20.000 a 40.000. Porém, de 80.000 para 100.000, a diferença é muito baixa, podendo indicar que a partir dali, aumentando o número de elementos e processadores, os valores podem começar a se estabilizar.

Por fim, é importante ressaltar a ineficiência do modelo no tratamento de tarefas com poucos elementos, o tempo de comunicação e ordenação se tornam um estorvo para uma execução com muitos processadores e um baixo número de elementos.

E. Extra – Sequencial x Um Escravo

Como uma análise de desempenho extra, foi verificado o comportamento do programa se rodado sequencialmente, e com apenas dois processos, um mestre e um escravo, e o resultado foi o seguinte:

Tabela 2 - Sequencial x um escravo.

-	Sequencial	Um Escravo
20000	4s122ms821us	1s32ms663us
80000	67s371ms533us	16s501ms954us

Logo, apenas um escravo é significativamente mais rápido que sequencialmente. Mas por que? Pois, como indicado na seção II.D., a função de ordenação *Rank Sort* é uma função de ordem quadrática, logo, se dividindo cada tarefa de *n* elementos para *n/4* elementos, adquirimos uma resposta até 16 vezes melhor.

REFERÊNCIAS

[1] Paper: Economy Informatics Department, A.S.E. Bucharest. “Parallel Rank Sort “. 2005. http://www.economyinformatics.ase.ro/content/EN5/alecu.pdf

[2] Website: Zorzo, Avelino. “Slides das Aulas”. 2013. Moodle PUCRS. http://moodle.pucrs.br/mod/folder/view.php?id=651115

[3] Website: Daniel Thomasset. Michael Grobe. Academic Computing Services. The University of Kansas. “An introduction to the Message Passing Interface (MPI) using C”. EUA. http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml