Programação Paralela com MPI

Programação Paralela utilizando o modelo Pipeline

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, o modelo Pipeline e o algoritmo Insertion Sort.**

***Abstract*—This paper relates the difficulties, problems, solutions and results found during the development of a software biased to work using parallel processing. For this project, the MPI communication structure is used, as well as the Pipeline model and the Insertion Sort Algorithm.**

***Palavras Chave*—MPI; Programação Paralela; Insertion Sort; Pipeline; Ordenação; Estágios.**

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 (Pipeline) 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 Pipeline. Este sistema teria como propósito a leitura de um arquivo texto contendo até 100.000 números aleatoriamente organizados, e a organização dos mesmos de forma crescente utilizando processos paralelos. O algoritmo apresentado para a organização foi o *Insertion Sort*, o qual será abordado posteriormente.

B. *Modelo Pipeline*

O modelo Pipeline é caracterizado pelo fato de implementar processos encadeados, onde há um fluxo de dados de um para outro, ou seja, vários estados/estágios[1] executando paralelamente e transferindo dados para o seu seguinte de forma encadeada.

Não há estado de maior ou menor prioridade, todos possuem a mesma importância no programa. Caso um estado falhe, todo o sistema entrará em colapso. A imagem abaixo ilustra o modelo básico de Pipeline.

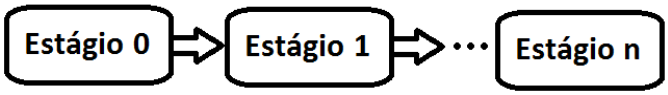

Fig. 1 - Modelo Pipeline.

C. *Ordenação*

No caso deste projeto, o algoritmo de ordenação é trivial para o funcionamento da ideia principal (Pipeline). Este algoritmo é o *Insertion Sort*. Utilizando esta implementação, permitimos que um valor A seja comparado com elementos de um vetor, caso haja algum valor menor que A, digamos, B, A posiciona-se no local de B e este segue a comparação no lugar de A até o fim do vetor. Caso o mesmo não esteja cheio, é possível adicionar o valor que restar no seu final, porém, caso o vetor esteja cheio, um valor sempre irá sobrar, seja ele o que entrou (A) ou o que foi excluído (B).

Desta forma, sempre retornando um valor, o *Insertion Sort* é ideal para a implementação utilizando o modelo Pipeline. A forma como o modelo foi elaborado é descrita na seção abaixo (II.D.).

D. *Implementação e Soluções*

Para a solução do desafio proposto, foi implementado o modelo Pipeline caracterizando três estágios separadamente.

Estágio 0 – Primeiro nodo. Realiza a leitura do arquivo de dados de entrada e utiliza a função *Insertion Sort* para inserir em seu vetor resultado. O valor que sobrar é enviado para o próximo processo, caso não haja sobras, o valor enviado é -1. Este estágio segue o processamento até o fim dos elementos do vetor de entrada.

Estágios intermediários – Todos os estágios entre o primeiro e o último. Eles recebem um valor através de uma mensagem MPI e utilizam mesmo algoritmo de ordenação do primeiro estágio para a inserção do valor em seu vetor, o que sobrar é mandado para o próximo.

[1] Idealmente os estágios/estados/nodos devem executar funções inteiramente diferentes uns dos outros, sendo cada estágio, único. Porém, para nossa implementação, esta regra não é seguida à risca.

Estágio np-1 – Seria o último estágio, onde *np* é o número de processadores rodando o programa. Este nodo tem a tarefa extra de informar o primeiro sobre o fim do processamento, assim formando o momento ideal para realizar o término da contagem de tempo.

Todas as etapas sabem perfeitamente quando encerrar seu processamento, pois elas sabem o número exato de elementos que cada uma deve receber. Este é calculado pelo número de elementos totais do programa (informado pelo usuário na chamada do sistema) dividido pelo número de processadores (input_elements/np).

Em adição, para a escrita dos dados em um arquivo resultado, foi implementado um modelo de passagem de *Token*. Assim que o primeiro nodo for informado do fim do processamento, ele irá imprimir seu vetor resultado em um arquivo texto[2] e mandar uma mensagem (*Token*) para que o próximo processo faça sua impressão. Esta mensagem segue em cadeia até o último processo, gerando um vetor ordenado em sequência no arquivo de saída.

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[2]:
   ladrun –np <n_processos> exec <n_elementos>

III. TESTES E RESULTADOS

Esta seção apresenta variados testes e resultados do programa desenvolvido. Estes testes foram realizados utilizando **quatro** nós no cluster *Atlântica*[3], o qual possui 8 núcleos em cada nó, ou seja, 16 threads pra cada processador.

Em adição, foi implementado um script para facilitar os testes, este realiza automaticamente cada teste para todos os valores de elementos e processos duas vezes. Desta forma, os valores dos tempos apresentados neste documento são uma média entre cinco amostras geradas pelo script[4].

A. *Interface*

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

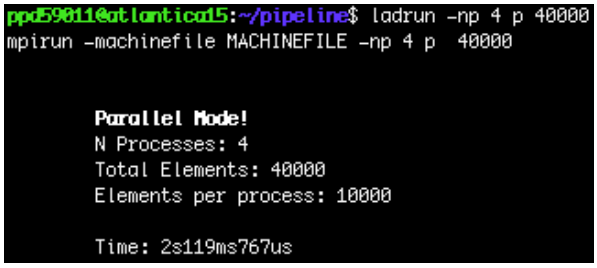

Fig. 2 - Execução para 40.000 elementos com 4 processos.

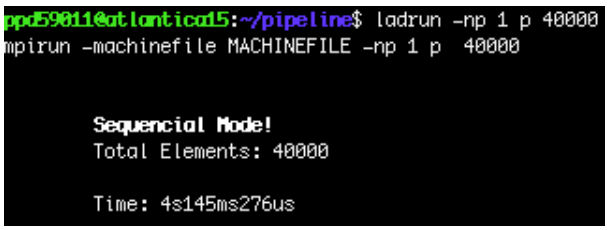

Fig. 3 - Execução para 40.000 elementos com 1 processo (sequencial).

B. *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: Nsegundos Nmilissegundos Nmicrossegundos)

[2] Pela forma como foi implementado o programa, é aconselhável excluir o arquivo texto de saída (*vetor_ordenado.txt*), ou renomear o já criado, a cada nova execução. A forma de acesso ao arquivo é feita por atualizações, ou seja, o arquivo não é sobrescrito, mas sim, texto é adicionado seu final. Podendo gerar uma desorganização, caso haja vários resultados na mesma sessão.

[3] É importante ressaltar que, apesar do cluster utilizado ser o mais rápido dentre todos, ele apresentava muita inconstância em seus dados, mostrando, a cada 10 testes, no mínimo, 5 valores completamente fora da média, ou seja, muito acima do comum.

[4] O script foi executado várias vezes, porém somente os cinco valores mais constantes e plausíveis foram utilizados para o cálculo da média, descartando os outros.

Tabela 1 - Tempo: n_elementos x processos.

-	1000	5000	10000	20000
1	2ms825us	72ms254us	271ms768us	1s096ms732us
2	4ms119us	59ms551us	213ms958us	753ms632us
4	6ms946us	57ms499us	168ms521us	566ms983us
6	10ms440us	53ms311us	139ms527us	506ms194us
8	9ms863us	53ms728us	128ms196us	443ms869us
10	11ms467us	55ms163us	126ms385us	348ms046us
-	40000	60000	80000	100000
1	5s143ms621us	10s206ms990us	17s094ms057us	26s968ms037us
2	3s436ms227us	7s310ms823us	12s697ms138us	19s502ms392us
4	2s109ms622us	5s053ms197us	8s220ms159us	13s235ms180us
6	2s073ms808us	4s305ms062us	6s799ms915us	10s071ms676us
8	1s340ms476us	3s077ms774us	5s051ms468us	8s221ms195us
10	1s141ms347us	2s410ms443us	4s200ms170us	7s109ms781us

Para 10.000 elementos ou mais, é possível analisar o decrescimento gradativo do valor do tempo em relação ao aumento de processos, assim reforçando o motivo do uso de programação paralela. Porém, para menos elementos, nota-se uma piora. Esta pode ser explicada pela diminuição da granulosidade, aumentando a frequência de comunicação e gerando pequenas e custosas tarefas. Fazendo assim com que a execução sequencial, ou com menos processos, seja, as vezes, a mais rápido.

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

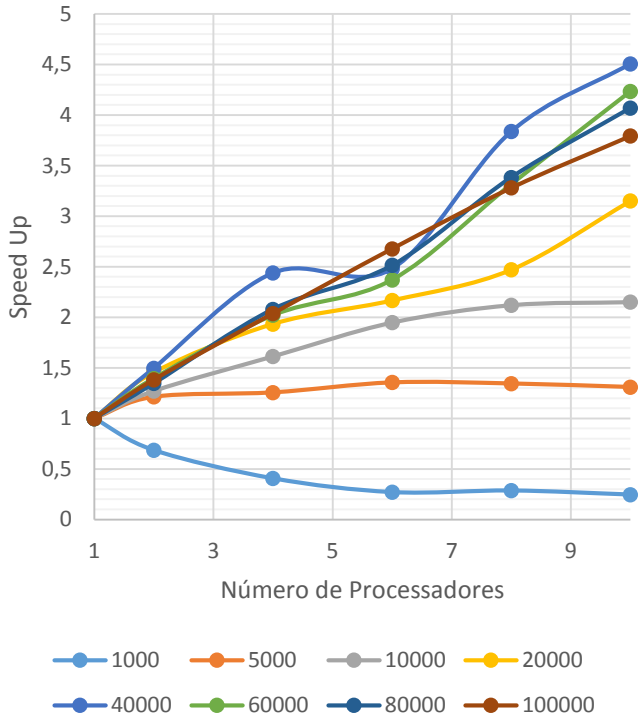

Fig. 4 - Gráfico Speed Up.

C. 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.

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

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.

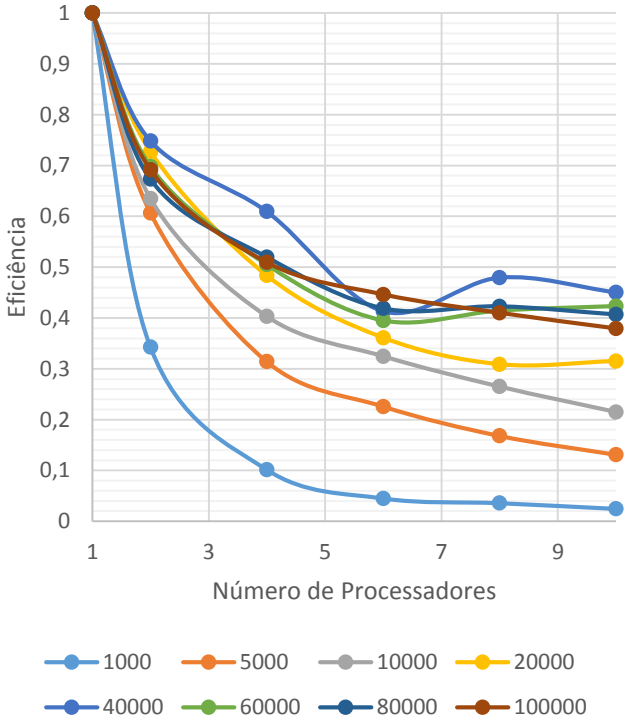

Fig. 5 - Gráfico Eficiência.

Como comentado na análise dos tempos (III.B.), poucos elementos reproduzem a menor temporização perante o aumento do número de processos, assim apresentando as menores eficiências no gráfico.

Além disso, é possível analisar individualmente a execução de 40.000 elementos, a qual possui a melhor eficiência e Speed Up. Isso pode provar que, para nosso ambiente de teste, a execução deste número de elementos, no algoritmo *Insertion Sort*, apresenta a melhor resposta sobre o aumento do número de processos, ou seja, a maior diferença de tempo entre um teste e outro.

Independentemente dos fatores citados anteriormente, o algoritmo *Insertion Sort* em conjunto com o modelo Pipeline é

consideravelmente ineficiente, ou seja, todos os seus cálculos de eficiência apresentados no gráfico da Fig.7 resultaram em valores abaixo de 1.

D. Extra – Exagerando nos processos

Um teste extra foi realizado para analisar o comportamento do programa desenvolvido quando executado usando 20 e 40 processos com 40.000 e 60.000 elementos. Os resultados são apresentados na tabela abaixo.

Tabela 2 - Execução para muitos processos

-	40000	60000
10	1s141ms347us	2s410ms443us
20	1s221ms191us	2s314ms392us
40	1s207ms328us	1s428ms717us

Analisando a tabela, percebe-se que a evolução do tempo sobre a execução com metade dos processos é muito baixa. Isso mostra que, após 10 processos, o tempo de execução apresenta um ângulo de crescimento muito pequeno, mantendo os valores praticamente constantes.

No cluster, as tarefas são executadas e divididas igualmente, tendo o poder de processamento restringido pela capacidade de cada processador (todos são iguais), independentemente do aumento do número de processos, a velocidade vai ser sempre limitada.

REFERÊNCIAS

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

[2] 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