Programação Paralela com MPI

Programação Paralela utilizando o modelo Divisão e Conquista

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 Divisão e Conquista, e o algoritmo Merge Sort de forma recursiva.**

***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, it is used the MPI communication structure, as well as the Divide and Conquer model, and the recursive version of the Merge Sort Algorithm.**

***Palavras Chave*—MPI; Programação Paralela; Merge Sort; Divisão; Conquista; Ordenação; Recursão.**

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 (Divisão e Conquista) e sua forma de implementação.

II. PROJETO E DESENVOLVIMENTO

A. Problema Inicial

O desafio proposto inicialmente seria de implementar um sistema capaz de ordenar paralelamente um vetor de tamanho variável contendo números aleatoriamente ordenados. Esta implementação foi feita utilizando um modelo de *Divisão e Conquista*, e um algoritmo recursivo de ordenação, baseado no *Merge Sort*.

A ideia principal seria, contendo um vetor de *n* elementos, dividi-lo exatamente pela metade entre seu nodo (pai) e seu filho. Está divisão aconteceria até um certo ponto, onde os nodos começariam a organizar o vetor em sua posse, e retorná-lo a seus pais.

B. Modelo Divisão e Conquista

O modelo de Divisão e Conquista é caracterizado pela existência de nodos pais e filhos, além de uma divisão de trabalho entre estes. Os pais recebem um certo trabalho, e eles dividem entre seus filhos, ou entre ele mesmo e o filho (forma implementada neste sistema), desta forma, o(s) filho(s) fica(m) encarregado(s) de realizar(em) o trabalho também. A etapa de divisão de trabalho é dada até um certo ponto, após, inicia-se a conquista, que neste caso, é a ordenação do vetor.

C. Ordenação

Neste projeto, a ordenação foi realizada a função *Merge Sort*, esta função tem como principal característica não ser uma função de ordem quadrática, porém, linear, tornando uma ordenação muito mais rápida. Contudo, ela somente organiza dois vetores já ordenados, necessitando da definição do início, meio (elemento que separa os dois vetores) e fim do vetor.

Para sua implementação, uma versão recursiva foi desenvolvida, de forma que ela vá dividindo e chamando-se recursivamente até que os elementos do vetor se adequem à função *Merge* e sejam organizados pela mesma. Além da ordenação dos elementos, a mesma implementação é usada para a organização dos vetores quando retornados dos filhos para seus pais.

D. Implementação e Soluções

Foi implementado um algoritmo deveras simples e reutilizável para ambos os pais e filhos. Esta ideia é rapidamente apresentada a seguir.

1. Enquanto não for N_elementos/N_Processos
2. Divide Vetor
3. Atualiza tamanho
4. Envia vetor para o próximo
5. Ordena
6. Recebe do filho
7. Organiza
8. Se for filho e não for menor que nível de início, envia para pai

Analisando o método acima, pode-se perceber que uma das únicas diferenças entre os nodos seria que o nodo 0 (matriz) receberia seus elementos pelo vetor, e não através de mensagens MPI, e não retornaria o vetor para ninguém, ao invés disso, ele imprime em um arquivo de saída.

Enfim, é importante ressaltar que os cálculos de nível e próximo nodo (filho) na árvore é feito através das equações a seguir:

$$nível = log_2(\frac{total\ de\ elementos}{elementos\ recebidos})$$

$$próximo\ nodo = ID\ atual + 2^{nível}$$

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[1]:
   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 apenas **um** nó no cluster *Atlântica*, o qual possui 8 núcleos em cada nó, e o recurso de *hyper-threading*, 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 seis vezes. Desta forma, os valores dos tempos apresentados neste documento são uma média entre quatro, cinco ou seis amostras geradas pelo script.

A. *Script*

O arquivo desenvolvido para auxiliar o teste é descrito abaixo, e acompanha o programa em um arquivo *.zip*.

```
1   #!/bin/bash
2   if [ $# -eq 0 ]; then
3       echo "Usage:"
4       echo "       $0 'file.c'"
5       exit 1
6   fi
7
8   ladcomp -env mpiCC $1 -o dec
9
10  for i in 1 2 4 8 16
11  do
12      for j in 10000 20000 40000 60000 80000 100000
13      do
14          for k in {1..6}
15          do
16              ladrun -np $i dec $j >> results4_$i.txt
17          done
18      done
19  done
```

B. *Interface*

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

```
ppd59011@atlantica07:~/dec$ ladrun -np 16 dec 100000
mpirun -machinefile MACHINEFILE -np 16 dec  100000


        Parallel Mode!
        Depth: 4
        Total Elements: 100000
        Elements per node: 6250

        Time: 0s14ms431us
```

Fig. 1 - Execução para 16 processos com 100000 elementos.

```
ppd59011@atlantica07:~/dec$ ladrun -np 1 dec 100000
mpirun -machinefile MACHINEFILE -np 1 dec  100000


        Sequencial Mode!
        Total Elements: 100000

        Time: 0s32ms894us
```

Fig. 2 - Execução para 1 processo (sequencial) com 100000 elementos.

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: Nmilissegundos Nmicrossegundos)

-	10000	20000	40000
1	2ms676us	5ms715us	12ms202us
2	2ms312us	4ms853us	10ms165us
4	2ms372us	2ms855us	5ms748us
8	1ms188us	1ms951us	3ms918us
16	1ms810us	2ms560us	5ms358us
-	60000	80000	100000
1	19ms056us	25ms876us	32ms872us
2	15ms643us	15ms852us	24ms946us
4	8ms810us	11ms872us	15ms189us
8	5ms843us	7ms696us	9ms698us
16	8ms162us	9ms075us	9ms895us

Fig. 3 - Tabela de tempos

[1] Aconselha-se que a execução seja feita com um número de elementos divisível pelo número de processos, e um número de processos resultante de uma potência de 2, ou seja, 1, 2, 4, 8 ou 16 processos.

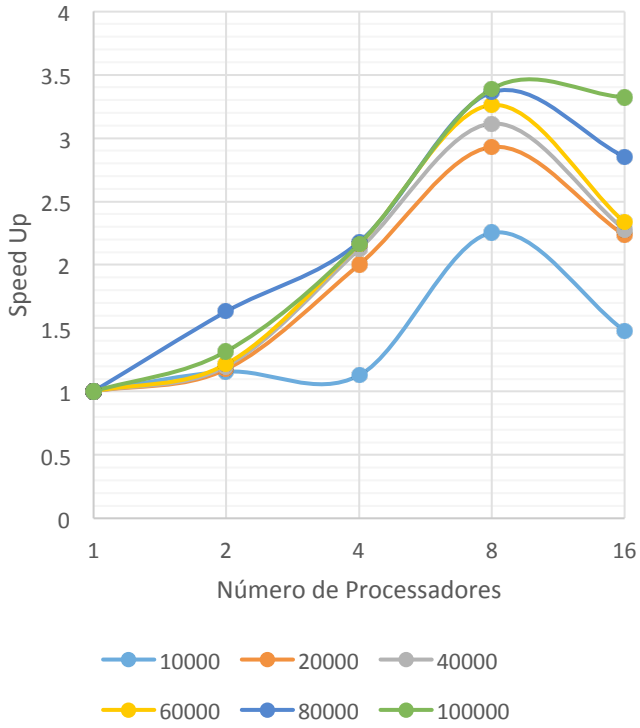

Fig. 4 - Gráfico Speed Up[2].

É possível ressaltar que o ponto de maior *Speed Up* no gráfico é a execução com 8 processos. Sua razão entre o tempo de execução sequencial pelo tempo de execução paralela resulta nos maiores valores para aquele número de processos, independentemente do número de elementos.

Outro fator de interessante ressalte é a pequena queda que existe na execução com 16 processos, esta queda pode ser explicada pela limitação do cluster utilizado. Este, como definido anteriormente (seção III.0) possui 8 processadores com 2 threads cada, logo, utilizando mais de 8 processos em um mesmo nodo, uma pequena parte do processador é reservada para a comunicação entre as threads, causando a queda do tempo apresentada.

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 processos.

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

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.

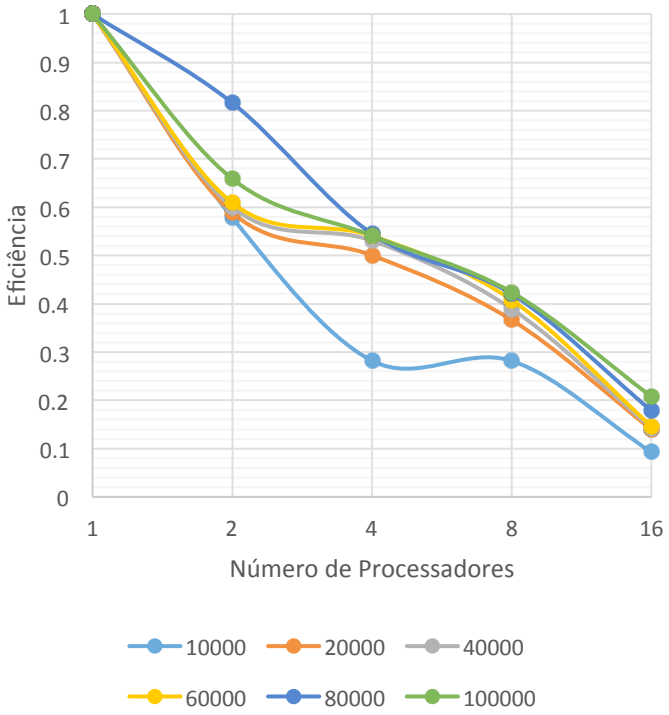

Fig. 5 - Gráfico Eficiência[2].

Por fim, é notória a grande ineficiência da execução paralela quando comparada com a execução sequencial. Com uma queda quase linear, o gráfico apresenta o comportamento de baixo desempenho deste modelo.

Pode-se explicar esta característica devido à utilização da função *Merge Sort* de ordem linear, e o alto número de trocas e cálculos executados pelos processos. Quanto mais processos, mais recursos são utilizados do cluster.

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

[3] Website: Trustees of Indiana University. LAM/MPI Parallel Computing. Bloomington , IN, EUA. "One-step Tutorial: MPI: It's easy to get started.". Reference: http://www.lam-mpi.org/tutorials/one-step/ezstart.php

[2] Eixo horizontal dos gráficos é representado de forma logarítmica com base 2.