Programação Paralela com MPI

Programação Paralela utilizando o modelo Fases Paralelas

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 focado em execução paralela. Utilizou-se a estrutura MPI, o modelo Fases Paralelas, e dois algoritmos de ordenação/organização diferentes, o Merge Sort e o Odd-Even Transposition Sort.**

***Abstract*—This paper relates the difficulties, problems, solutions and results found during the development of a software focused to work using parallel processing. For this project, it is used the MPI communication structure, as well as the Parallel Phases model, and two different sorting algorithms, the Merge Sort and the Odd-Even Transpositon Sort.**

***Palavras Chave*—MPI; Programação Paralela; Merge Sort; Fases Paralelas; Ordenação; Recursão, Odd-Even Transposition Sort; Par; Ímpar.**

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 (Fases Paralelas) e sua forma de implementação.

Para fins de análise de desempenho, o projeto proposto foi implementado duas vezes utilizando dois algoritmos de ordenação diferentes, os quais possuem características bem distintas. A implementação e comparação dos resultados dos dois programas será apresentada posteriormente.

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 e números aleatoriamente ordenados. Esta implementação foi feita utilizando um modelo de Fases Paralelas, e um algoritmo de ordenação baseado no *Bubble Sort*, ou seja, o *Odd-Even Transposition Sort (OETN)*.

A ideia principal seria, após lido um vetor de *n* elementos, dividi-lo pelo número total de processos em execução, e ordená-lo, fazendo a comunicação necessária entre os processos. Para isso, o programa teria duas fases, e dependendo de qual for, seriam trocados elementos entre os processos ativos naquela fase. O objetivo seria transferir os elementos de menor valor para os processos de menor *ID* e os elementos de maior valor para os processos de maior *ID*.

B. *Modelo Fases Paralelas*

O modelo de Fases Paralelas é caracterizado pela existência de duas ou mais fases ocorrendo paralelamente e de forma sincronizada, sendo que, ao fim de cada fase, haja uma interação entre os processos para troca de informações/dados. Ou seja, os processos recebem um pedaço do vetor para ordenação, após, dependendo da fase, passa um número *x* de elementos de uma de suas extremidades para um determinado nodo, podendo ser seu vizinho.

No caso desta implementação, as fazes são divididas em *PAR* e *ÍMPAR*, na fase par, os processos de *ID* par enviam seus maiores elementos para os processos de *ID* ímpar, enquanto estes, recebem e enviam seus elementos de menor valor para o nodo do qual recebeu a mensagem anteriormente; já na faze ímpar, o comportamento dos processos inverte, tendo os processos ímpares enviando os maiores elementos para os pares, e estes, os menores para os ímpares. A imagem abaixo ilustra o comportamento deste modelo.

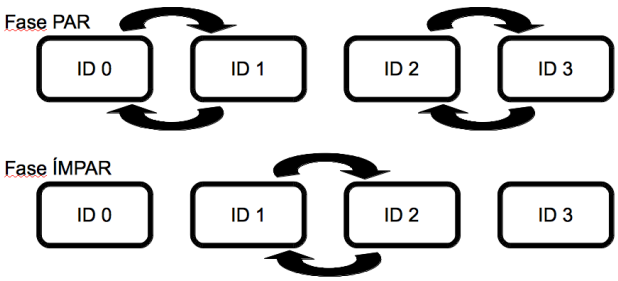

Fig. 1 - Modelo de Fases Paralelas utilizando quatro processos.

C. *Ordenação*

Como informado na Seção I, foram desenvolvidas duas soluções para o problema proposto. Na primeira, foi utilizando o algoritmo *Odd-Even Transposition Sort* e na segunda, uma

função implementada de forma recursiva que chama o algoritmo de organização *Merge Sort*.

O algoritmo *OETN* é baseado na função de ordenação *Bubble Sort*, porém, neste caso, percorre o vetor dado de dois em dois elementos, tendo um loop para os elementos pares e outro para os elementos ímpares. Mesmo com essa diferença, este algoritmo ainda possui complexidade de ordem $O(n^2)$, logo, necessita de muitos recursos, processamento e tempo para sua execução com um elevado número de elementos.

Em compensação, o algoritmo *Merge Sort* possui complexidade O(n), logo, é claramente mais rápido, porém, este não é considerado um algoritmo de ordenação, mas sim de organização, logo, ele simplesmente organiza dois vetores ordenados, tendo noção do elemento que divide estes vetores. Uma função recursiva foi implementada, de forma que controle as variáveis passadas por parâmetro na função *Merge*, chamando-a até que os elementos trocados por ela possam ordenar o vetor. Desta forma recursiva, a execução da função basta a deixar de ser linear para ter uma complexidade logarítmica, ou seja, O(log n).

D. *Implementação e Soluções*

Inicialmente, todos os nodos possuem acesso ao arquivo de entrada, lendo-o e guardando seus elementos em um vetor. Como definido anteriormente, cada processo possui um vetor próprio de tamanho definido pelo total de elementos sobre o número de nodos em execução. Porém, um estudo foi feito e concluído que há uma expressiva diferença de tempo na execução quando passados diferentes quantidades de valores a cada etapa da fase (resultados apresentados na Seção III.D), logo, um buffer variável foi concatenado ao vetor resultado de cada nodo.

Antes do início da etapa de troca de fases, todos os nodos utilizam a função de ordenação para deixar o vetor pronto para troca de dados. O comportamento dos processos em cada fase foi fiel ao modelo apresentado na Seção II.B.. Para a definição de parada da comunicação entre os processos, foi utilizada a função *MPI_Allreduce* (Referência [4]) que possibilita que todos os processos saibam se alguém ainda está ordenando, no momento em que não há mais ordenações, a execução está completa.

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

Esta seção apresenta variados testes e resultados dos programas desenvolvidos. Estes testes foram realizados utilizando **um** nó no cluster *Atlântica*, o qual possui 8 núcleos em cada nó, ou seja, 16 threads pra cada processador.

Em adição, foram implementados três scripts para maior praticidade e facilitação nos testes, estes realizam 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. Os scripts podem ser encontrados no arquivo *.zip do projeto, juntamente com os códigos-fonte.

A. *Interface*

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

Fig. 2 - Execução Sequencial.

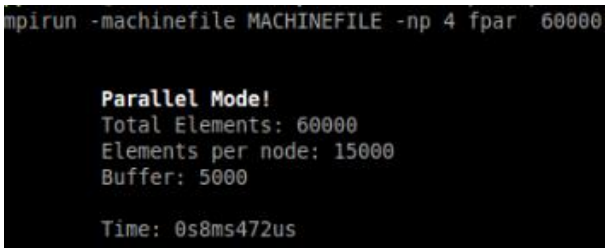

Fig. 3 - Execução Paralela.

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 são mostradas duas tabelas com algumas capturas de tempo em relação ao número de elementos e o número de processos.

(Modelo: Nsegundos Nmilissegundos Nmicrossegundos)

Table 1 - Tempos de Execução para o modelo utilizando *Merge Sort*.

-	10000	20000	40000
1	2ms492us	5ms349us	11ms385us
2	2ms383us	5ms323us	10ms440us
4	1ms911us	2ms575us	6ms275us
8	1ms339us	2ms800us	4ms743us
16	1ms408us	2ms838us	4ms635us
-	60000	80000	100000
1	17ms645us	24ms124us	30ms585us
2	16ms158us	21ms205us	27ms961us
4	8ms530us	13ms306us	16ms119us
8	7ms938us	10ms446us	13ms027us
16	7ms035us	9ms611us	12ms091us

Table 2 - Tempos de execução para o modelo utilizando *OETS*.

-	10000	20000	40000
1	391ms549us	1s040ms759us	6s145ms590us
2	250ms955us	1s046ms746us	4s315ms280us
4	325ms890us	1s210ms029us	5s138ms582us
8	205ms902us	759ms917us	3s119ms779us
16	186ms065us	1s257ms880us	2s945ms046us
-	60000	80000	100000
1	14s040ms907us	24s996ms868us	39s024ms952us
2	9s165ms383us	15s485ms684us	23s852ms486us
4	7s581ms997us	16s179ms492us	18s485ms122us
8	5s113ms329us	8s472ms770us	19s948ms567us
16	6s199ms701us	8s227ms547us	17s402ms227us

Analisando ambas as tabelas acima é possível perceber a grande diferença entre os tempos de execução para cada algoritmo de ordenação apresentado. Este claro contraste entre os tempos pode ser explicado pelo já descrito na Seção II.C., os algoritmos possuem ordens diferentes, sendo a ordem logarítmica (*Merge Sort* recursivo) melhor e de menor consumo quando comparada a ordem quadrática (*OETS/Bubble Sort*).

As figuras Fig. 4 e Fig. 5 apresentam graficalmente o comportamento do *Speed Up* para cada implementação com diversas quantidades de elementos como entrada.

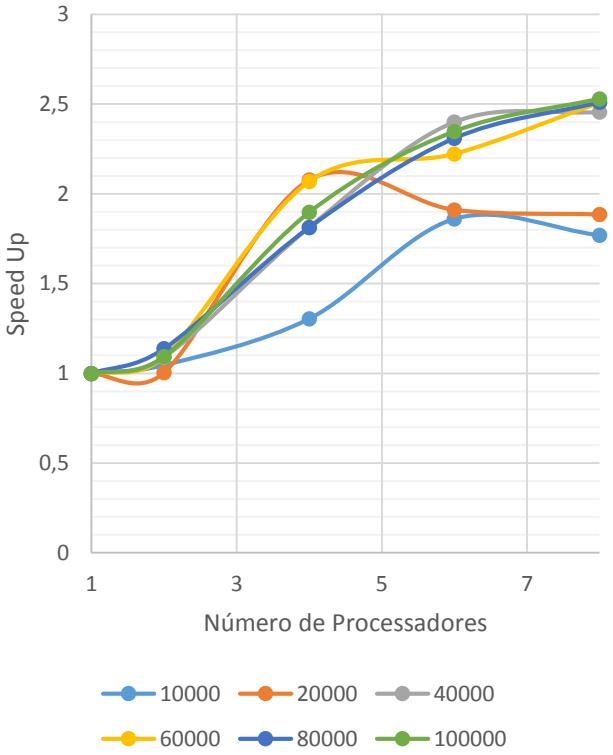

Fig. 4 - Gráfico Speed Up para *Merge Sort*.

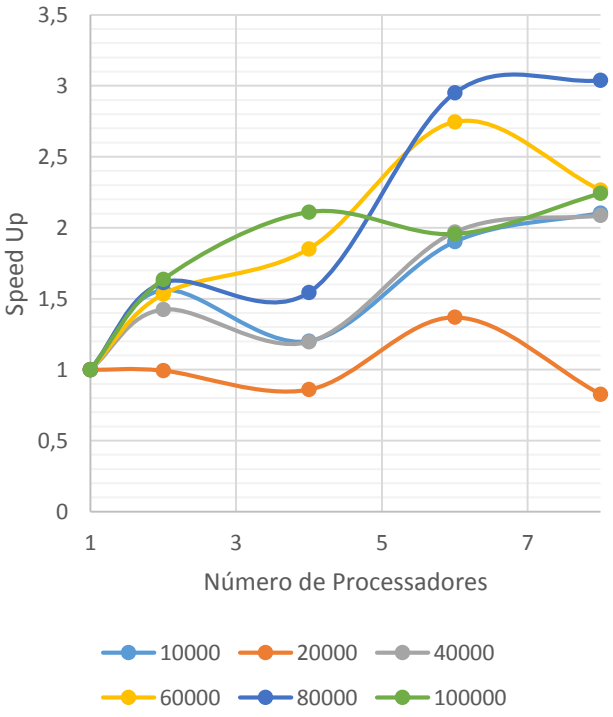

Fig. 5 - Gráfico Speed Up para modelo *OETS*.

Tendo ordens de complexidade tão distintas, os algoritmos apresentam tempos de execução muito variados, porém, a representação gráfica de ambos possui algumas semelhanças. Ambos possuem um *Speed Up* que cresce na medida em que o número de processadores aumenta, porém, enquanto para o *Merge Sort*, este crescimento é muito próximo do linear, para o *OETN*, ele é mais truncado, tendo uma leve queda para 4 processos.

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.

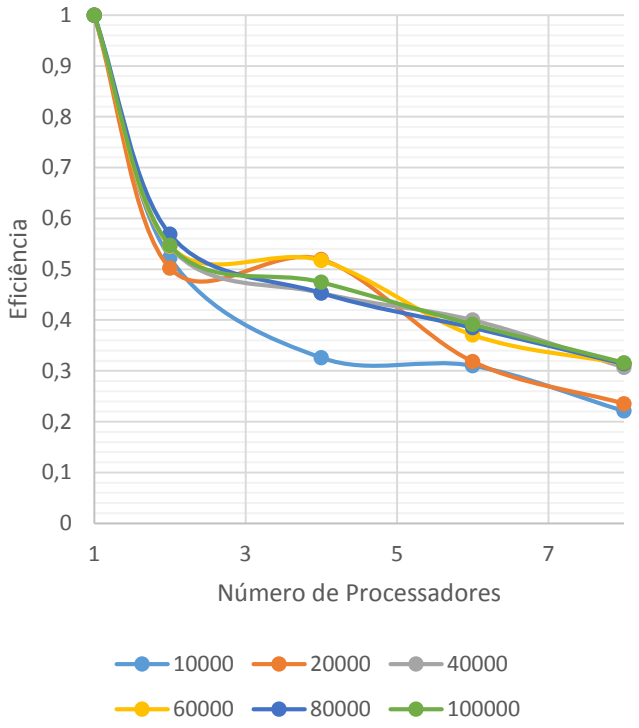

Fig. 6 - Gráfico do desempenho para o modelo *Merge Sort.*

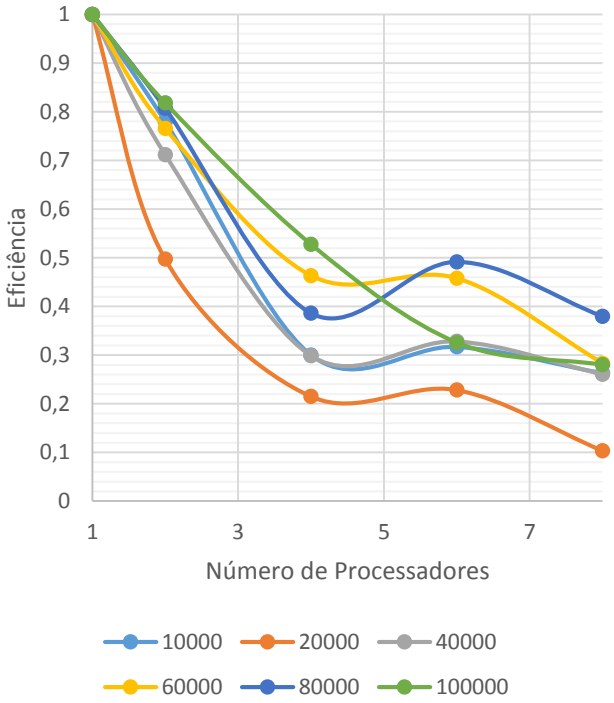

Fig. 7 - Gráfico do desempenho para o modelo *OETS.*

Assim como no *Speed Up*, na análise gráfica de desempenho, a função da Fig. 8 apresenta um comportamento muito semelhante, independentemente do número de elementos de entrada, já na Fig. 9, notamos que, apesar de todos apresentarem uma queda de eficiência, a quantidade de valores do vetor de entrada influência diretamente na *imagem* de cada linha.

Por fim, é importante ressaltar que em ambos os casos, apesar de os valores de tempo em execuções paralelas serem menores que em execuções sequenciais, os algoritmos paralelos possuem menor eficiência em relação ao uso dos recursos de processamento quando comparados as suas execuções com apenas um processo.

D. *Extra*

Um teste para verificar a influência do número de elementos transmitidos por cada processo (*Buffer*) nas fases par ou ímpar foi feito, para isso, foi necessária a implementação de um script que calculasse várias porcentagens baseadas no tamanho do vetor de cada processo. O objetivo deste teste seria definir a porcentagem, ou tamanho de *Buffer* ideal para adquirir a execução mais rápida do programa.

Aplicando este teste no programa utilizando *Merge* Sort, para 60.000 elementos, pode-se verificar os resultados apresentados na Fig. 10, onde os melhores tempos são capturados quando o *Buffer* é em média 33%~50% do tamanho do vetor de cada processo. Ou seja, para 60.000 elementos com 4 processos, cada processo teria um vetor de 15.000 elementos para ordenação, e um buffer de 5.000 elementos para a troca.

É possível entender este comportamento fazendo a seguinte analogia:

- Cada processo possui um número *x* de elementos, sendo uma parte destes, considerada elementos de valor pequeno, e outra parte, elementos de valor grande. Dentre estes, estão alguns elementos que realmente fazem parte daquele nodo.
- Se o processo passar um número baixo de elementos para seu vizinho, serão necessários vários ciclos de comunicação para que todos os elementos pertencentes àquele vizinho sejam transmitidos.
- Se o processo passar um número alto de elementos, ou seja, mais do que o vizinho necessita guardar, na hora da ordenação, este irá ter que percorrer vários elementos desnecessários, pois eles não pertencem àquele nodo.
- Logo, percebeu-se que entre 33~50% do vetor é o tamanho ideal do *Buffer* para que passem os valores corretos de cada vetor, onde estes possam ser reordenados, e não existam muitos valores não pertencentes àquele nodo para serem analisados e descartados (gastando tempo e processamento).

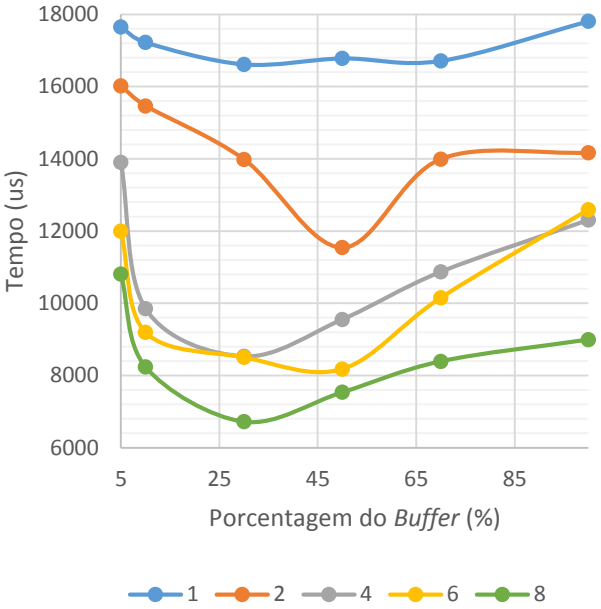

Fig. 8 – Execução da implementação com *Merge Sort*, tendo 60.000 elementos e variados tamanhos de buffer.

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: Kendall, Wes. Beginner MPI. University of Tennessee, Knoxville. "MPI Reduce and Allreduce". EUA. http://mpitutorial.com/mpi-reduce-and-allreduce/

[4] Livro: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. & Stein, Clifford (2001). "Introduction to Algorithms. Chapter 1: Foundations (Second ed.)". Cambridge, MA: MIT Press and McGraw-Hill. pp. 3–122. ISBN 0-262-03293-7.