Programação Paralela: Paradígma Mestre-Escravo

Filipe Franciel Utzig
Engenharia de Computação
PUC-RS
Porto Alegre 90619-900
Email: filipeutzig *at* gmail.com

Abstract*—*The present work implements the sorting algorithm known as Rank Sort. The algorithm will be implemented in two forms: parallel, using openMPI library with the master/slave paradigm, and sequential, so that, at the end, we can have a better knowledge regarding the performance and efficiency of the parallel version.

Keywords*—*MPI; Rank-Sort; parallelism;

I. Descrição do problema

Dado um vetor desordenado, para cada elemento x desse vetor, conta-se o número de elementos que são menores que o mesmo. Este número será o índice de x no vetor ordenado. O procedimento é repetido para todos os valores da lista. O algoritmo com que estamos trabalhando não aceita um vetor com possíveis repetições de valores.

II. Atacando o problema

O *Rank Sort* (1) (2) tem o seu funcionamento sequencial descrito no Algoritmo 1, onde é verificado a quantidade de valores que são menores que o valor selecionado, esta contagem provê a posição no vetor ordenado.

```
for (i=0; i < n; i++) do
    for (j=0; j < n; j++) do
        if (a[i]>a[j]) then
            x++;
        end
    end
    b[x] = a[i];
end
```

Algoritmo 1: *Rank Sort* sequencial

Como é possível notar, o algoritmo *Rank Sort* é de ordem $O(n^2)$, para cada elemento adicional no vetor o tempo de execução cresce de forma quadrática. Para implementar o Algoritmo 1 foi utilizado a linguagem programação C e, na versão paralela deste, a biblioteca openMPI (3), rodando sobre o sistema operacional Linux.

A. Descrição da Implementação Sequencial

A implementação sequencial espera receber como entrada o caminho para um arquivo contendo os valores embaralhados e o número de elementos a serem ordenados. Ao começar programa é feita a leitura de todos valores para um vetor alocado em memória, evitando assim que chamadas de sistemas sejam executadas durante a medição de tempo do algoritmo.

Essa versão ordena o vetor inteiro de uma vez só. Para cada elemento Y do vetor, o algoritmo simplesmente guarda numa variável x, o numero de elementos que são menores do que Y, feita essa comparação, o programa coloca o elemento Y na posição x. Esse laço é executado para cada elemento do vetor, o resultado ao final é o vetor ordenado de forma crescente, que é escrito num arquivo texto ao final da execução.

B. Descrição da Implementação Paralela

O funcionamento da implementação paralela segue a mesma lógica da versão sequencial, porém é necessária uma logica de controle de paralelismo e divisão de tarefas.

No início da execução o Mestre divide o vetor desordenado em diversos pedaços, que serão mandados para os Escravos. O número de elementos que cada escravo recebe é dado por ***N/4*M*** onde **N** é igual ao número total de elementos do vetor desordenado e **M** equivale ao número de escravos que executarão a tarefa.

Inicialmente, o Mestre atribui um pedaço do vetor desordenado para cada escravo. Como o número de pedaços é maior que o número de escravos, o processo Mestre fica aguardando um Escravo terminar de ordenar o pedaço que lhe foi atribuído. Quando o primeiro escravo termina o seu ordenamento, ele o envia o resultado parcial ao mestre, que o recebe e envia um novo pedaço à esse escravo. A cada fatia recebida pelo Mestre, uma sub-rotina concatena de forma ordenada dos pedaços recebidos. Esse processo é repetido até que não haja mais pedaços a serem enviados.

III. Resultados

Todos os testes foram executados no *cluster* Amazônia que consiste em uma *enclosure* HP BladeSystem C3000 com 4 Blades BL620c G7 e uma *storage* dedicada com acesso via Fiber Channel Protocol(8 Gib/s). Cada máquina possui dois processadores Intel Xeon E7 - 2850 2.0 GHz Hyper-Threading e 80GB de memória, totalizando 20 núcleos (40 *threads*) por nó e 80 núcleos (160 *threads*) no *cluster*. Os nós estão interligados por 4 redes *Gigabit-Ethernet* chaveadas e 2 redes InfiniBand (para comunicação entre os nós) (4).

A Tabela I contém os tempos de execução dos testes, cada teste foi executado 3 vezes a fim de aumentar a confiabilidade dos dados adquiridos, e os resultados apresentados é a média simples das execuções. Podemos observar um decréscimo de tempo em todos os casos com a paralelização dos processos, inclusive quando somente um escravo realiza

a tarefa, esse comportamento ocorre pois o escravo ordena uma fatia 4x menor por vez, não percorrendo toda a extensão do vetor e devido a complexidade quadrática do algoritmo fica explicado os resultados vantajosos. Podemos observar melhor esse comportamento através da Figura 1.

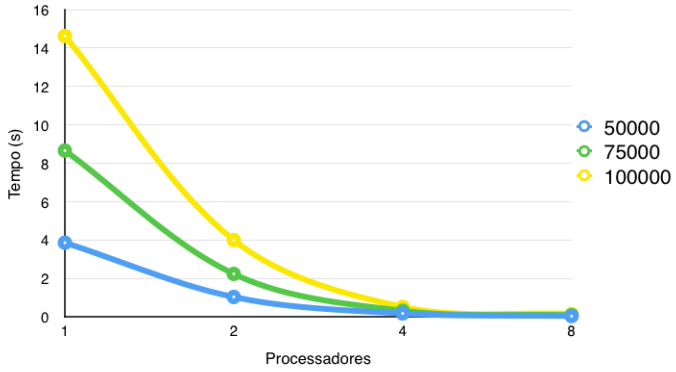

Fig. 1. Tempo de execução dos testes em função do número de elementos do vetor.

TABLE I. Tempo de Execução dos testes.

Número de Processadores	50000 Elementos	75000 Elementos	100000 Elementos
1	3,861 s	8,662 s	14,632 s
2	1,033 s	2,236 s	3,997 s
4	0,176 s	0,318 s	0,514 s
8	0,039 s	0,085 s	0,150 s

O aumento na velocidade de execução foi notável, de forma a quantificar o aumento, foi calculado o *Speed Up* (2) de cada teste, que é obtido de acordo com a Equação 1 onde Ts é o tempo do algoritmo sequencial e Tn é o tempo do algoritmo paralelo, a Tabela II contém todos os valores referentes a *Speed Up*, nela podemos observar que nesse caso o desempenho possuí um comportamento super linear, pois $Sp > Np$. Podemos observar o comportamento do *Speed Up* através da Figura 2.

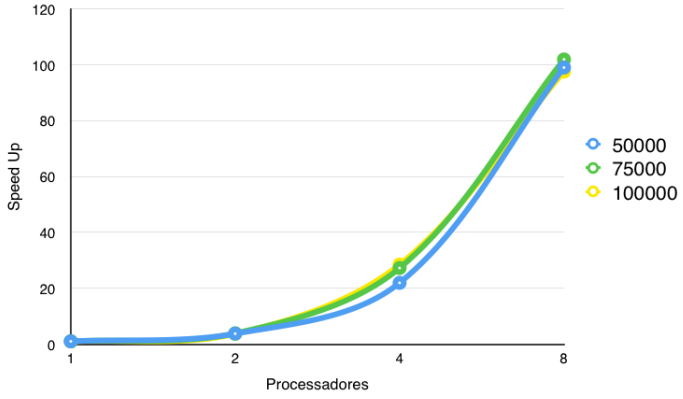

Fig. 2. Speed up.

$$SpeedUp = \frac{Ts}{Tn} \quad (1)$$

TABLE II. Speed Up dos testes.

Número de Processadores	50000 Elementos	75000 Elementos	100000 Elementos
1	1	1	1
2	3,74	3,87	3,66
4	21,94	27,24	28,47
8	99,00	101,91	97,55

Outro método de aferir o ganho da implementação paralela é através da eficiência, calculada a partir da Equação 2, todos os dados encontram-se na Tabela III. Observamos que os casos com 4 e 8 processadores obtiveram um coeficiente de eficiência superior ao número de processadores utilizados, com isto é possível verificar que o desempenho melhora substancialmente na forma paralelizada, e este comportamento se deve à característica da ordem quadrática do algoritmo, no qual quanto menor for o número de elementos processados por cada escravo, muito mais rápida será a execução do ordenamento. O comportamento da eficiência dessa paralelização está ilustrada na Figura 3.

$$Eficiência = \frac{SpeedUp}{n} \quad (2)$$

TABLE III. Eficiência dos testes.

Número de Processadores	50000 Elementos	75000 Elementos	100000 Elementos
1	1	1	1
2	1,87	1,94	1,83
4	5,48	6,81	7,12
8	12,38	12,74	12,19

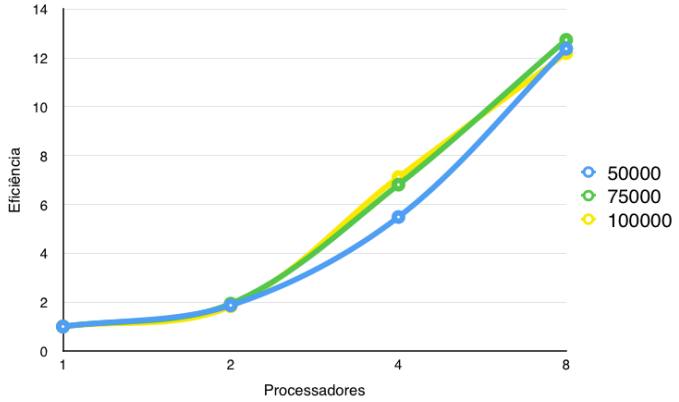

Fig. 3. Eficiência.

IV. Conclusão

Podemos observar que com a paralelização do problema obtivemos um ganho de performance significativo. Nos cenários simulados foi possível verificar que conforme o número de processadores era aumentado o *Speed Up* também aumentava, porém esse comportamento não é linear, pois se aumentarmos muito o número de processadores chegaremos um ponto em que o gargalo será a comunicação entre esses diversos escravos.

References

[1] M. Johnson, "Parallel sorting algorithms," 2015. [Online]. Available: http://www.massey.ac.nz/ mjjohnso/notes/59735

[2] B. Beattie, "Parallel algorithms," 2015. [Online]. Available: http://www2.cs.uregina.ca/ beattieb/CS340

[3] S. L. Hamilton, *An Introduction to Parallel Programing*, 2013.

[4] IDEIA, "IDEIA - instituto de pesquisa e desenvolvimento," 2015. [Online]. Available: http://www.lad.pucrs.br