Programação Paralela: Paradigma Pipeline

Filipe Franciel Utzig
Engenharia de Computação
Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS
Porto Alegre 90619-900
Email: filipeutzig *at* gmail.com

Abstract*—*The present work presents the implementation of sorting algorithm known as Insertion Sort, this algorithm will be presented in two forms: parallel, using openMPI library with the pipeline 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; Insertion-Sort; parallelism; pipeline; performance;

I. Introdução

No cenário computacional atual o uso de sistemas multiprocessados é algo comum, mesmo entre sistemas embarcados como *smartphones*. Neste contexto explorar o uso de programação paralela torna-se muito interessante. Entre as diversas técnicas de programação paralela existentes, neste trabalho, será explorado o paradigma de pipeline sobre o problema o problema de ordenação de dados, utilizando o *Insertion Sort*.

Este algoritmo funciona construindo o vetor ordenado de um item por vez, ao receber um valor x, ele é comparado a cada um dos valores já posicionados no vetor de saída, de forma a verificar qual a posição em que x deve ser inserido. Feita a inserção, o algoritmo deve agora recolocar todos os elementos seguintes uma posição adiante. O processo se repete para *x+1, x+2, etc...* até que todo o vetor esteja ordenado. O Insertion Sort, ao contrário de outros algoritmos, passa através do vetor uma só vez. Esse algoritmo, também é comumente comparado à organizar, de maneira crescente, as cartas de um baralho. [3].

II. Detalhamento do problema

A rotina de funcionamento acima mencionada pode ser visualizada no Algoritmo (1).

```
for (i = 0; i < n; i++) do
    val = unsorted[i];
    for (j = 0; j < i; j++) do
        if (val < sorted[j] then
            aux = sorted[j];
            sorted[j] = val;
            val = aux;
        end
    end
    sorted[i] = val;
end
```

Algoritmo 1: *Insertion Sort*

Para implementar o algoritmo 1 foi utilizado a linguagem programação C e, na versão paralela deste, a biblioteca openMPI [1], rodando sobre o sistema operacional Linux.

A. Detalhamento da versão sequencial

A versão sequencial do algoritmo segue o mesmo principio de funcionamento da versão paralela, que será apresentada a seguir, porém seu desenvolvimento é simplificado quando comparado a versão paralela. O algoritmo utiliza dois vetores para realizar o ordenamento: O primeiro contém os valores desordenados e o segundo vetor é serão armazenados os valores ordenados.

Para cada elemento x retirado do vetor desordenado, o algoritmo percorre o vetor ordenado até achar um elemento y tal que $(x < y)$. Ao encontrar este elemento, seu valor é armazenado em uma variável auxiliar e x é inserido na posição em que estava y. O algoritmo repete os passos acima, desta vez para o elemento y. O processo se repete até que não haja mais elementos no vetor desordenado. Caso não encontre, no vetor ordenado, nenhum elemento y que satisfaça a condição $(x < y)$, ele simplesmente adiciona o valor x no final do vetor ordenado, visto que x é o maior elemento deste vetor.

B. Detalhamento da versão paralela

A versão paralela funciona da seguinte maneira: Cada processo possui um vetor com tamanho definido por **n/P** onde **n** é o número total de elementos a serem ordenados, e **P** é o número de processos utilizados. O primeiro estágio do *pipeline* tem um funcionamento levemente diferente dos outros, é o primeiro estágio que contém o vetor de entrada armazenado em memória. O primeiro estágio inicia a inserção ordenada na sua fatia do vetor resultado, quando a fatia do vetor resultado do primeiro estágio estiver cheio, a inserção de um valor resulta na saída de outro, o valor removido é então enviado para o próximo estágio do *pipeline*, este processo de inserção é repetido até que não tenha haja mais nenhum valor para ser ordenado.

O processo acaba assim que o último estágio terminar de ordenar o seu vetor, neste momento é feita a tomada de tempo para avaliação dos resultados.

III. Resultados

Todos os testes foram executados no *cluster* Atlântica que é composto por 16 máquinas Dell PowerEdge R610. Cada máquina possui dois processadores Intel Xeon Quad-Core E5520 2.27 GHz Hyper-Threading e 16GB de memória, totalizando 8 núcleos (16 threads) por nó e 128 núcleos (256 threads) no custer. Os nós estão interligadas por 4 redes Gigabit-Ethernet chaveadas. O ultimo nó (atlantica16), dispõe de uma NVIDIA Tesla S2050 Computing System, com 4

NVIDIA Fermi computing processors (448 CUDA cores cada) divididos em 2 host interfaces e 12GB de memória. [2]

A Tabela I contém os tempos de execução dos testes, cada teste foi executado 29 vezes a fim de aumentar a confiabilidade dos dados adquiridos, para isso foi desenvolvido um *script* de forma a automatizar a execução e criar uma relação de todos os tempos obtidos, após esse processo a média de todas as execuções foi calculada. Podemos observar um decréscimo de tempo em todos os casos com a paralelização dos processos, esse comportamento ocorre pois cada estágio do pipeline ordena uma fatia menor, não percorrendo toda a extensão do vetor. Podemos observar melhor esse comportamento através da Figura 1.

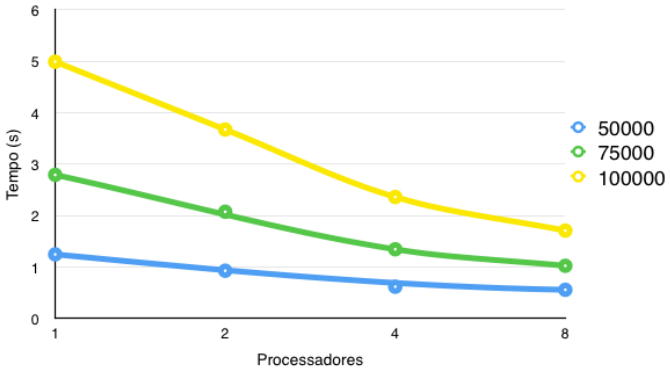

Fig. 1. Tempo de Execução.

TABELA I. TEMPO DE EXECUÇÃO DOS TESTES.

Número de Processadores	50000 Elementos	75000 Elementos	100000 Elementos
1	1,245	2,791	4,992
2	0,927	2,074	3,673
4	0,616	1,344	2,361
8	0,556	1,023	1,707

O aumento de velocidade foi observado nos estudos de caso, para quantificar o tempo de resposta de cada caso, foi calculado o *Speed Up* de cada cenário de teste, que é obtido de acordo com a Equação (1) onde Ts é o tempo do programa sequencial e Tn é o tempo do programa paralelo, a Tabela II contém todos os valores referentes a *Speed Up*, nela podemos observar que neste caso o desempenho possui um comportamento linear para os cenários de teste analisados.

Analisando os resultados é possível verificar uma diminuição no crescimento do valor de *Speed Up* a partir de 8 processadores com 50000 elementos, este comportamento é justificado pois o número de elementos processados por cada estágio do pipeline diminuí bastante e o *overhead* de comunicação passa a ser mais perceptível. Podemos observar comportamento das avaliações de *Speed Up* através da Figura 2.

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

A eficiência com a paralelização também foi calculada e é obtida através da Equação (2), todos os dados encontram-se na Tabela III. Observamos que ocorre uma redução no

TABELA II. SPEED UP

Número de Processadores	50000 Elementos	75000 Elementos	100000 Elementos
1	1	1	1
2	1,34	1,35	1,36
4	2,02	2,08	2,11
8	2,24	2,73	2,92

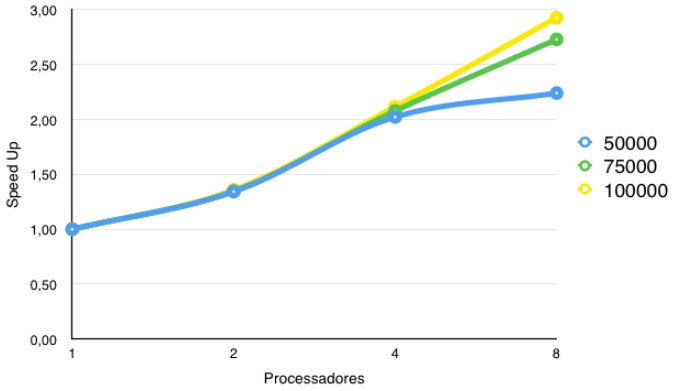

Fig. 2. Speed up.

ganho de eficiência do programa ao aumentarmos o número de processadores, isto é, ao aumentarmos em n vezes os processadores usados, o desempenho não aumentará na mesma proporção, já que a otimização do algoritmo ao aumentar o paralelismo não é linear.

O comportamento da eficiência dessa paralelização está ilustrada na Figura 3.

TABELA III. EFICIÊNCIA DOS TESTES.

Número de Processadores	50000 Elementos	75000 Elementos	100000 Elementos
1	1	1	1
2	0,67	0,67	0,68
4	0,51	0,52	0,53
8	0,28	0,34	0,37

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

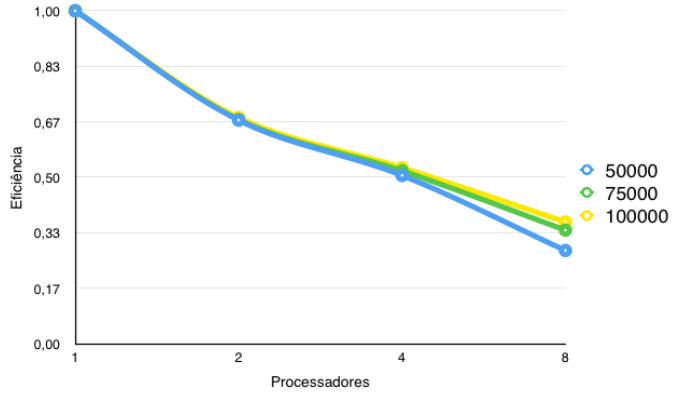

Fig. 3. Eficiência.

IV. Conclusão

Podemos observar que com a paralelização do problema obtivemos um ganho de desempenho significativo. Nos cenários simulados foi possível verificar que conforme o número de processadores foi aumentando 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 processos, comportamento este que começou a ser observado com o uso de 8 processadores e um número menor de elementos processados.

Referências

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

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

[3] MATHBITS. (2015) Beggining java, unit 6 - insertion sort. [Online]. Available: http://mathbits.com/MathBits/Java/arrays/InsertionSort.htm