Análise empírica do algoritmo ShellSort de ordenação

Augusto S. Carniel[1], **Gabriel C. Fermino**[1]

[1]Centro de Ciências Exatas e Tecnológicas (CCET) –
Universidade Estadual do Oeste do Paraná (UNIOESTE) – Cascavel – PR – Brasil

augusto.carniel@unioeste.br, gabriel.fermino@unioeste.br

Abstract. *This article aims to demonstrate an empirical analysis of the results from the comparison between a conventional and a parallel implementation of the ShellSort algorithm.*

Resumo. *Este artigo tem como objetivo demonstrar a análise empírica dos resultados provenientes da comparação entre a implementação convencional do algoritmo de ordenação ShellSort e uma implementação utilizando ferramentas que possibilitam a execução paralela deste mesmo método.*

1. Introdução

O modelo básico do ShellSort introduzido por D. L. Shell em 1959, foi um dos primeiros métodos de ordenação a ser descoberto e é um dos mais simples de se implementar. Baseado no método InsertionSort, ou seja, percorre os elementos da esquerda para a direita, teve como objetivo inicial ser uma alternativa mais viável de seu inspirador (SEDGEWICK, 1996).

A complexidade do método ShellSort é feita da seguinte maneira: Primeiramente um intervalo é definido, com tamanho inicial igual a metade do comprimento do vetor dos elementos a serem ordenados, ao fim de cada iteração divide-se, novamente, o valor atual do intervalo por 2. Basicamente o algoritmo passa diversas vezes pela lista de elementos, dividindo o grupo maior em grupos menores, iguais ao valor do intervalo. Nestes grupos menores, é aplicado o método InsertionSort (MISHRA, 2008).

Sua complexidade varia de acordo com a maneira em que os elementos estão dispostos, tendo em seu pior, e melhor caso, uma complexidade igual a $\mathcal{O}(n \log^2 n)$ e em seus casos médios uma complexidade de $\mathcal{O}(n \log n)$ (MISHRA, 2008). Tendo em vista este detalhes, resolve-se por tentar uma abordagem paralela na resolução deste problema, visando diminuir o tempo gasto na ordenação.

2. Metodologia PCAM

2.1. Particionamento

Seguindo a metodologia PCAM, o particionamento diz respeito a menor unidade de divisão do problema, que neste caso se trata de uma posição unitária do vetor de entrada.

2.2. Comunicação

Por se tratar de um algoritmo de ShellSort que utiliza como base o InsertionSort, a comunicação entre as partes é feita por meio de comparações entre os elementos anteriores do vetor, buscando encontrar a posição correta do elemento.

2.3. Agrupamento

Em relação ao agrupamento dos elementos, estes são divididos baseados na iteração do algoritmo, e da quantidade de threads especificado no código. Os elementos que estão sendo avaliados no momento da execução são divididos igualmente entre as threads criadas.

2.4. Mapeamento

O mapeamento foi feito no nó*cpu_dev*, do super-computador "Santos Dumont", que conta com 1 CPU's Intel Xeon de 2,4 GHZ, tendo 24 núcleos disponíveis para uso onde cada um desses receberá uma thread da execução do código.

3. Implementação

3.1. ShellSort Convencional

Para a implementação do ShellSort Convencional, primeiramente aloucou-se dinamicamente um vetor de tamanho *n* preenchendo-o com valores aleatórios. Após preencher o vetor, a função que realizará a ordenação é chamada. Dentro dentro função, faz-se as verificações e operações básicas do ShellSort e chama-se a função de ordenação InsertionSort, que é a base utilizada por D. L. Shell para a implementação do método ShellSort.

Code 1. ShellSort Convencional

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <time.h>

void insertionSort(int a[], int n, int auxM);
void shellSort(int a[], int n);

int main(){
    long int n = 850000000;
    int *vet;

    double start, fim;
    start = omp_get_wtime();
    vet = (int *)malloc(n*sizeof(int));
    srand(time(NULL));

    for (int i = 0; i < n; i++){
        vet[i] = rand() + 1;
    }

    shellSort(vet,n);

    fim = omp_get_wtime();

    printf("Tempo gasto: %lf segundos \n", fim - start);

    return 0;
}

void shellSort(int a[], int n){
```

```
32
33      int i, m;
34
35          while(m < n){
36          m = 3*m+1;
37      }
38      for(m = n/3; m > 0; m /= 3)
39      {
40              for(i = 0; i < m; i++)
41                  insertionSort(&(a[i]), n-i, m);
42      }
43  }
44
45  void insertionSort(int a[], int n, int auxM){
46
47      int j;
48      for (j=auxM; j<n; j+=auxM) {
49          int val = a[j];
50          int i = j - auxM;
51          while (i >= 0 && a[i] > val) {
52              a[i+auxM] = a[i];
53              i = i - auxM;
54          }
55          a[i+auxM] = val;
56      }
57  }
```

3.2. ShellSort Paralelo

A implementação do ShellSort Paralelo foi feita de forma análoga ao Convencional, porém foram adicionadas as funções referentes a biblioteca *"omp.h"*, para que a paralelização fosse possível.

Code 2. ShellSort Paralelo

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <omp.h>
4   #include <time.h>
5
6   void insertionSort(int a[], int n, int auxM);
7   void shellSort(int a[], int n);
8
9   int main(){
10      long int n = 850000000;
11      int *vet;
12
13      double start, fim;
14      start = omp_get_wtime();
15      vet = (int *)malloc(n*sizeof(int));
16      srand(time(NULL));
17
18      for (int i = 0; i < n; i++){
19          vet[i] = rand() + 1;
20      }
21
```

```
22      shellSort(vet,n);
23
24      fim = omp_get_wtime();
25
26      printf("Tempo gasto: %lf segundos \n", fim - start);
27
28      return 0;
29  }
30
31  void shellSort(int a[], int n){
32
33      int i, m;
34
35          while(m < n){
36          m = 3*m+1;
37      }
38      omp_set_num_threads(4);
39      for(m = n/3; m > 0; m /= 3)
40      {
41            #pragma omp parallel for shared(a,m,n) private (i)
42              for(i = 0; i < m; i++)
43                  insertionSort(&(a[i]), n-i, m);
44      }
45  }
46
47  void insertionSort(int a[], int n, int auxM){
48
49      int j;
50      for (j=auxM; j<n; j+=auxM) {
51          int val = a[j];
52          int i = j - auxM;
53          while (i >= 0 && a[i] > val) {
54              a[i+auxM] = a[i];
55              i = i - auxM;
56          }
57          a[i+auxM] = val;
58      }
59  }
```

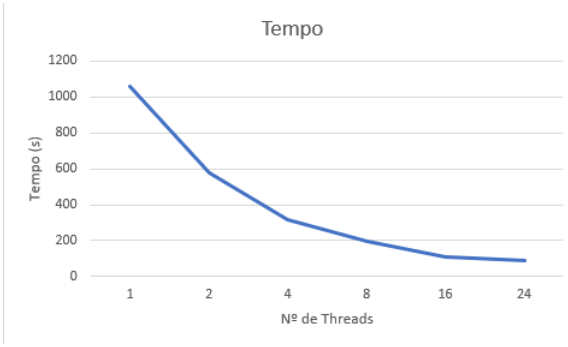
4. Resultados

Os testes foram realizados no super-computador ”Santos Dumont”, utilizando o nó nomeado de *cpu_dev*, que conta com 1 CPU’s Intel Xeon de 2,4 GHZ, tendo 24 núcleos para seu uso, com 64GB DDR3 de RAM. Nos testes foi definido um tamanho máximo pro vetor igual a 850000000 e os números de *Threads* foram variados entre 1 e 24. Nas figuras abaixo serão mostrados os resultados obtidos.

Figura 1. Tabela com os valores obtidos

DADOS						
num_threads	1	2	4	8	16	24
T(p)	1062	577	319	195	110	86
S(p)	1	1,840555	3,329154	5,446154	9,654545	12,34884
E(p)	1	0,920277	0,832288	0,680769	0,603409	0,514535

Figura 2. Tabela demonstrando o ganho em tempo de execução



Nota-se que conforme aumenta-se os recursos utilizados obtém-se uma ganho considerável nos tempos de execução.

Figura 3. Tabela demonstrando o Speedup

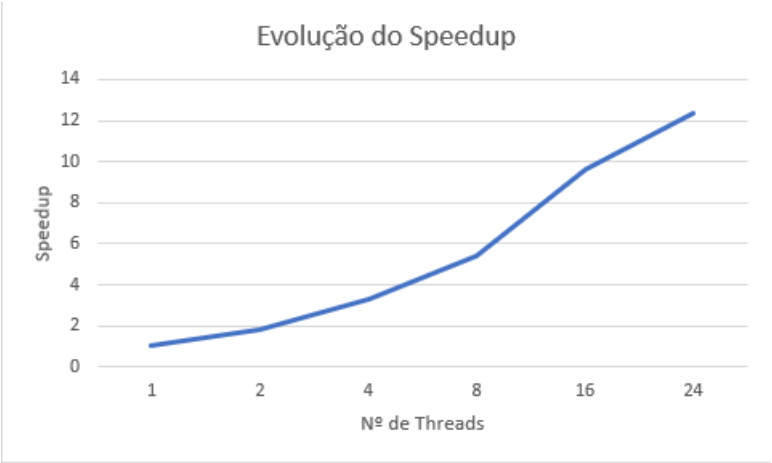
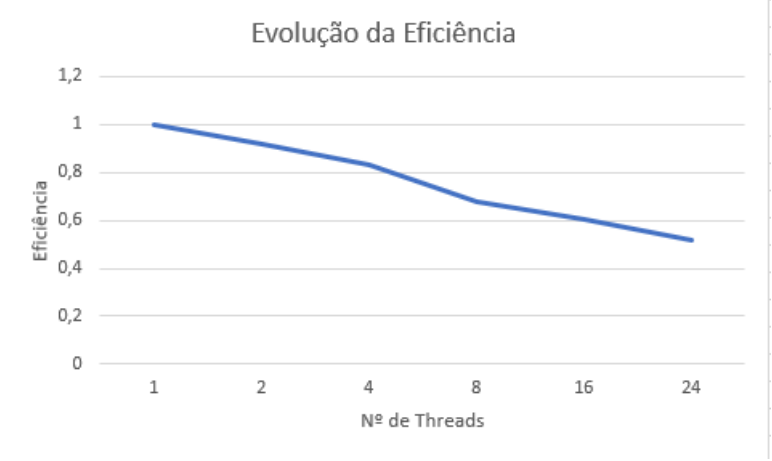


Figura 4. Tabela demonstrando a eficiência



5. Conclusão

Após uma análise aprofundada dos resultados obtidos, notou-se que o algoritmo possui uma boa escalabilidade, porém ao continuar aumentando os recursos, nota-se uma perca significativa na eficiência e uma diminuição no ganho de tempo. Utilizando como base a eficiência dos testes, observa-se que o resultado mais interessante se dá utilizando 8 Threads, obtendo um Speedup de aproximadamente 5 vezes o tempo original e mantendo uma eficiência próxima de 70%.

6. Referências

D. L. SHELL, Comm. Assoc. comp. Mach. 2, No. 7 (1959), 30-32.

SEDGEWICK, Robert. Analysis of Shellsort and related algorithms. In: European Symposium on Algorithms. Springer, Berlin, Heidelberg, 1996. p. 1-11.

NASCIMENTO, Antonieli et al. ESTUDO COMPARATIVO ENTRE OS MÉTODOS DE ORDENAÇAO INSERÇÃO DIRETA, BUBBLESORT, COMBOSORT E SHELLSORT. Plataforma de Submissão de Trabalhos e Anais de Eventos da Unicruz, 2019.

MISHRA, Aditya Dev; GARG, Deepak. Selection of best sorting algorithm. International Journal of intelligent information Processing, v. 2, n. 2, p. 363-368, 2008.