OpenMP - Shell sort

Universidade Federal do Rio Grande do Norte

Amanda Freire de Albuquerque - 20160116488

Tiago Onofre Araujo - 2018014455

Dezembro de 2020

Sumário

1 Introdução

Este trabalho apresenta a implementação do shell sort de forma serial e paralela, e tem como objetivo comparar a eficiência de cada paradigma utilizando o tempo de execução como parâmetro. O trabalho também fará a análise de speedup, eficiência e escalabilidade apresentado em gráficos para uma melhor visualização.

O shell sort é um algoritmo de ordenação onde sua complexidade ainda não é conhecida, não há uma fórmula fechada para sua função assintótica. Apesar de ser considerado instável, o algoritmo é bastante eficiente sendo considerado o melhor dentre os de complexidade quadrática.

O funcionamento do algoritmo se baseia na ordenação por inserção, porém de forma mais eficiente, a lista original de números é quebrada em sublistas e assim sendo aplicada a ordenação por inserção em cada uma.

1.1 Compilação

- Serial

```
$ g++ -g -Wall -std=C++11 serial/shellsort_serial.cpp -o shells
```

- Paralelo

```
$ g++ -g -Wall -fopenmp -std=C++11 parallel/shellsort_prl.cpp -o
    shellp
```

1.2 Execução

- Serial

```
$ ./shells <tamanho_do_vetor>
```

- Paralelo

```
$ ./shellp <num_de_cores> <tamanho_do_vetor>
```

1.3 Cálculo das Médias dos Tempos

O respositório consta com um script em python para automatizar o cálculo das médias aritmeticas após o uso do shellscript:

```
$ python3 script.py arquivo_com_tempo.txt
```

2 Desenvolvimento

2.1 Solução Serial

```
void shell( long int * arr, long int size )
{

    //GAP -> the distance between the values that will be swapped like
        an insertion sort.

    for( long int gap = size/2; gap > 0; gap /= 2 )
    {
        // Insertion sort
        for( long int i = gap; i < size; i++ )
        {

            long int aux = arr[i];

            long int itr;

            for( itr = i; itr >= gap && arr[ itr - gap ] > aux; itr -=
                gap )
            {
                arr[itr] = arr[ itr - gap ];
            }

            // Correct position of the arr[i].
           arr[itr] = aux;
        }
    }

}
```

Variáveis

- gap: Representa a distância usada para aplicar o insertion sort no vetor.
- `itr`: Iterador de controle para o laço onde ocorre o insertion sort.
- `i`: Iterador que inicializa a partir da posição do gap atual.

Lógica

- A condição de parada é depois de aplicado o insertion sort em todas as "fatias" do vetor.
- O shell sort é, de maneira resumida, a otimização do insertion sort, pois este tem seu melhor caso quando o vetor possui alguma ordem parcial (no caso do shell sort é comumente dito que o vetor está gap-ordenado a cada iteração).

2.2 Solução Paralela

```
void shell( long int * arr, long int size, int threads )
{
    long int i, itr;


    //GAP -> the distance between the values that will be swapped like
        an insertion sort.

    for( long int gap = size/2; gap > 0; gap /= 2 )
    {
        #pragma omp parallel for private(i, itr) shared(arr, gap, size) \
        default(none) num_threads(threads)
        // Insertion sort
        for( i = gap; i < size; i++ )
        {

            long int aux = arr[i];

            for( itr = i; itr >= gap && arr[ itr - gap ] > aux; itr -=
                gap )
            {
                arr[itr] = arr[ itr - gap ];
            }

            // Correct position of the arr[i].
           arr[itr] = aux;
        }
    }

}
```

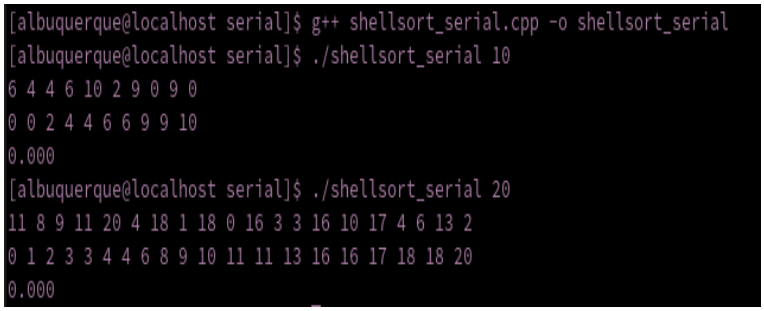
Lógica

- A flexibilidade do OpenMP permitiu que o código serial não fosse alterado, sendo necessário apenas a escolha da área paralela, número de threads e o controle do escopo das variáveis.
- A área definida foi no uso do insertion sort, diferente do MPI, o vetor maior não é divido em vetores locais, mas sim iterado paralelamente indicando quais variáveis devem ser vistas por todos os cores (o vetor a ser ordenado, o valor da distância de troca e o tamanho do vetor) e quais devem ser vistas por valores locais (iteradores de laço de cada core).

3 Resultados

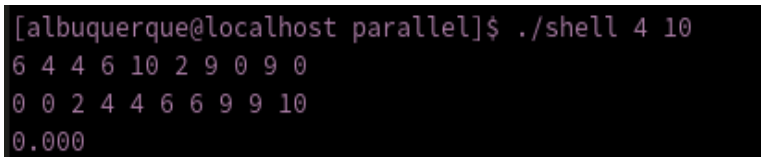
3.1 Exemplos de Saída

Serial

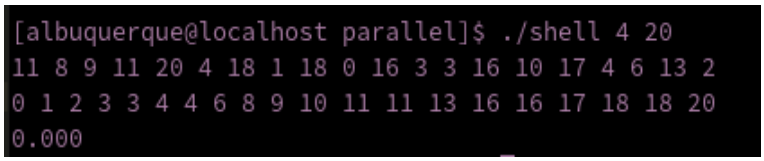


A imagem acima mostra um vetor desordenado e outro ordenado pelo algoritmo serial do shell sort.

- **Paralelo**



A imagem acima mostra um vetor desordenado e outro ordenado pelo algoritmo paralelo do shell sort.



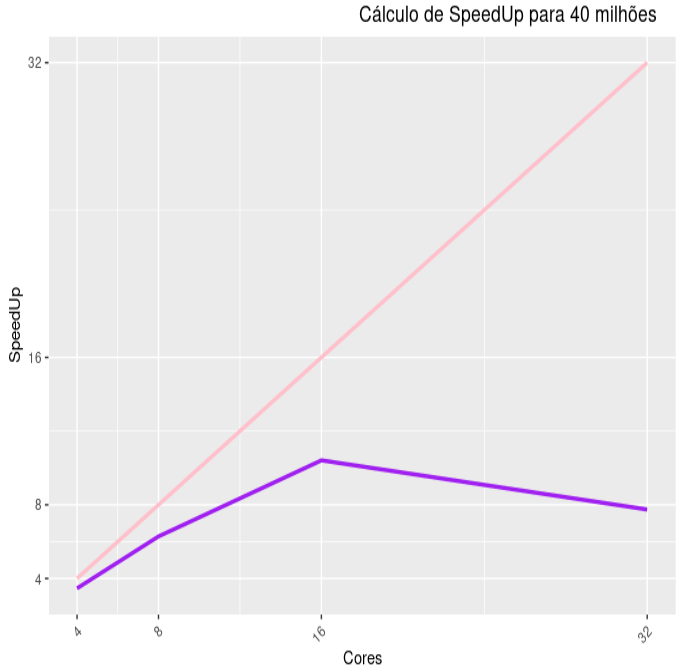
A imagem acima mostra um vetor desordenado e outro ordenado pelo algoritmo paralelo do shell sort.

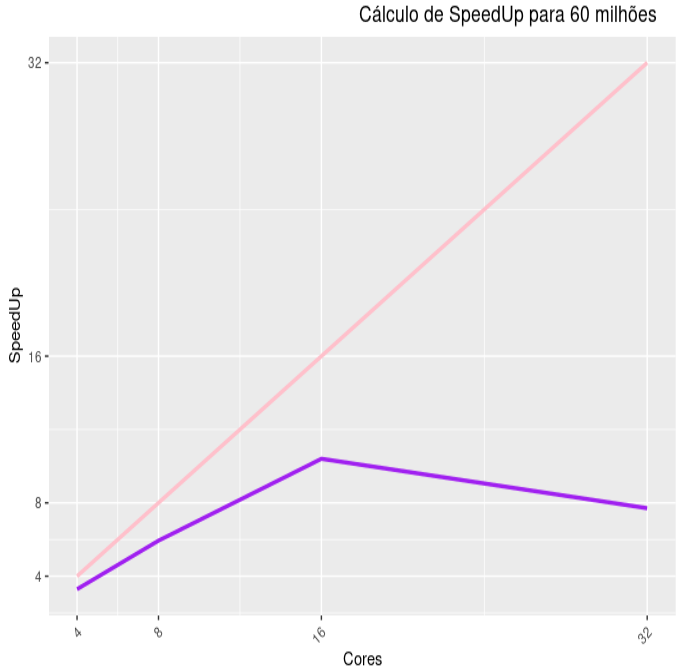
4 Análises

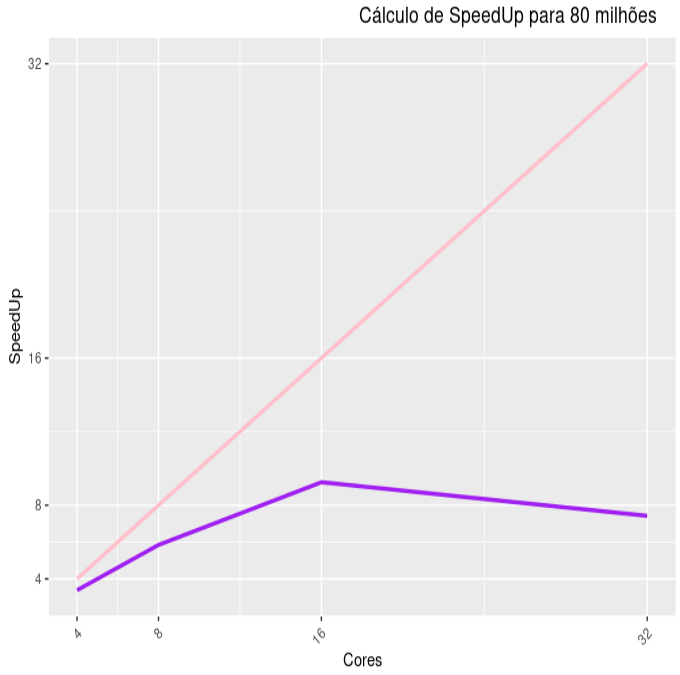
4.1 Speed Up

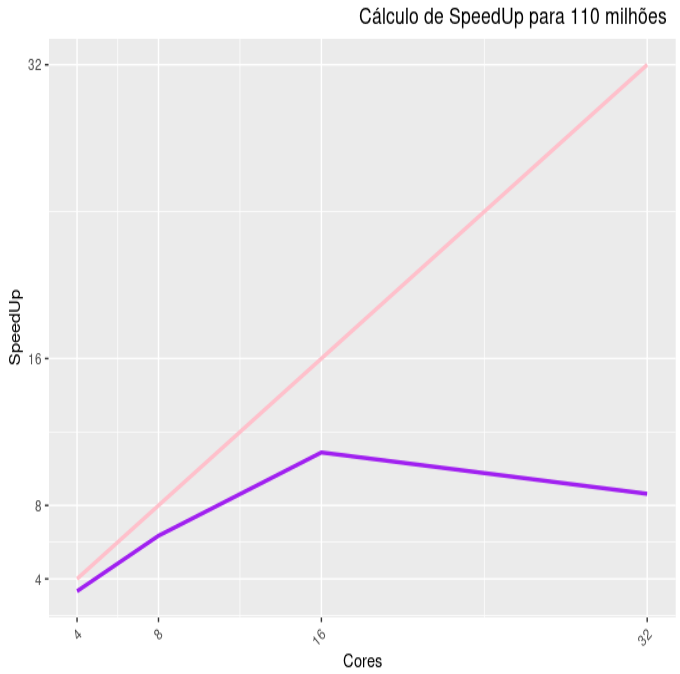
As imagens abaixo mostram os speedups das quatro amostras: 40 milhões, 60 milhões, 80 milhões e 110 milhões. O cálculo do é feito dividindo o tempo do código serial pelo código paralelo.

- Desempenho esperado
- Desempenho em análise




Cálculo de SpeedUp para 60 milhões
SpeedUp
32
16
8
4
4
8
16
32
Cores


Cálculo de SpeedUp para 80 milhões
SpeedUp
32
16
8
4
4
8
16
32
Cores

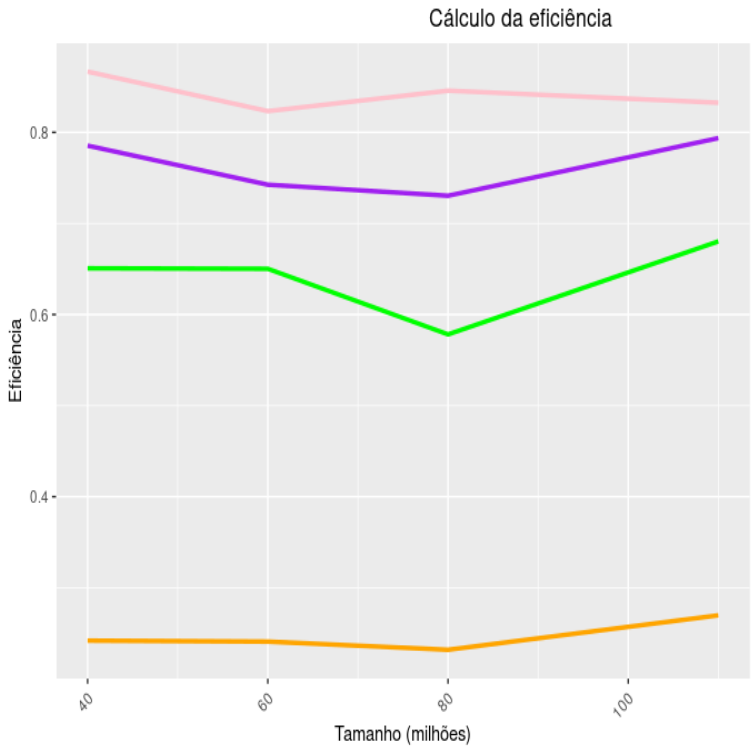


Após analisar os gráficos, vemos que todos seguem o mesmo padrão, a linha começa crescente e a partir de 16 cores, o speed up começa a cair. O que se espera é que quanto maior o número de cores, maior o speedup, mesmo que se distancie do ideal (o mesmo valor do core utilizado), porém, como é visto no gráfico, depois de um certo core, não é alcançada a curva ideal.

Tendo o conhecimento dessas informações, concluímos que o algoritmo não é bom paralelamente, pois quando chega a um determinado número de cores ocorre queda no desempenho.

4.2 Eficiência

- 4 cores
- 8 cores
- 16 cores
- 32 cores



Tamanhos	Eficiencia4	Eficiencia8	Eficiencia16	Eficiencia32
40	0.8666512	0.7853903	0.6507548	0.2419849
60	0.8232987	0.7425045	0.6502092	0.2408822
80	0.8457533	0.7304111	0.5781689	0.2319374
110	0.8325314	0.7936870	0.6803032	0.2697602

Ao observar o gráfico e a tabela acima percebemos que a medida que o tamanho aumenta temos uma eficiência quase sempre constante ou crescente, e como não há proporção no crescimento dos cores e do tamanho do problema, classificamos o algoritmo como escalável.

5 Considerações finais

Após as análises dos resultados nas seções anteriores, observamos que o algoritmo paralelo deverá ser ajustado para obter uma melhor eficiência a medida que aumentamos a quantidade de cores. Concluímos que a utilização do openmp facilita a utilização do paralelismo nos códigos seriais e que apesar dos tempos com 16 cores serem melhores quando comparados com 32 cores, temos uma redução bastante considerável relacionado ao código serial.