MPI - Odd even transposition sort

Universida Federal do Rio Grande do Norte

Tiago Onofre Araujo

20180144855

Outubro de 2020

Sumário

1 Introdução

O projeto consiste na análise do tempo de execução de um algoritmo de ordenação e dois diferente paradigmas de programação(serial e paralelo). Visando praticar a comunicação entre os processo do mpi, além da ordenação local de cada vetor, será aplicada a lógica par-ímpar entre a ligação e ordenação de cada processo e seus respectivos vetores. Na versão paralela fez-se uso de funções especificas do mpi para tratamento de vetores: `MPI Scatter` e `MPI Gather`.

1.1 Compilação

- Serial

```
$ g++ -g -Wall -std=C++11 arquivo_serial.cpp -o bin
```

- Paralelo

```
$ mpicxx -g -Wall -std=C++11 arquivo_paralelo.cpp -o bin
```

1.2 Execução sem script

- Serial

```
$ ./bin tamanho_do_vetor
```

- Paralelo

```
$ mpiexec -np <num_de_cores> ./bin tamanho_do_vetor
```

1.3 Compilação e execução com script

Os tópicos anteriores vão servir caso haja interesse em descobrir os limites de execução de sua máquina individualmente, uma vez descobertos o uso do script auxiliará as repetidas execuções para a análise empírica.

```
$ chmod 775 nome_do_script.sh
$ ./nome_do_script.sh
```

1.4 Cálculo das Médias dos Tempos

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

```
$ python3 mean.py arquivo_com_tempo.txt
```

2 Desenvolvimento

2.1 Especificações da máquina

MacBook Pro

- **SO**: macOS Catalina 10.15.6.
- **CPU**: Intel Core i5 Quad-Core 1.4GHz.
- **MEM**: 8Gb 2133 MHz LPDDR3.
- **Hyperthreading**: Ativado.

2.2 Solução Serial

A implementação serial foi feita baseada no pseudo-código do orientador desse projeto, o professor Dr. Kayo Gonçalves.

```
void odd_even_sort( long int * arr, long int size )
{
    long int phase, itr; // Kayo suggestions.

    for( phase = 0; phase < size; phase++ ) // Sorting loop
    {
        // Even
        if( phase % 2 == 0 )
        {
            for( itr = 1; itr < size; itr += 2 )
            {
                if( arr[itr - 1] > arr[itr] ) // iterator with iterator -
                    1
                {
                    swap( &arr[itr - 1], &arr[itr] );
                }
            }
        }
        else // Odd
        {
            for( itr = 1; itr < size - 1; itr += 2 )
            {
                if( arr[itr] > arr[itr + 1] ) // iterator with iterator +
                    1
                {
                    swap( &arr[itr], &arr[itr + 1] );
                }
            }

        }

    }
}
```

Variáveis

- `phase`: Iterador para controle das fases.
- `itr`: Iterador para realizar as comparações e trocas.
- `swap()`: Função que realiza a troca entre duas posições.

Lógica

- A condição de parada é a quantidade de fases totais que é igual ao tamanho do vetor.

- Nas fases pares as comparações começarão a partir dos indexes [0] e [1]
- 1. O itr começa com 1.
  2. Compara o valor com o index[itr - 1].
  3. O menor valor fica no index[itr - 1].
  4. Incrementa +2 no itr.
- Nas fases ímpares as comparações começarão a partir dos indexes [1] e [2]
- 1. O itr começa com 1.
  2. Compara o valor com o index[itr + 1]
  3. O menor valor fica no index[itr]
  4. Incrementa +2 no itr.

2.3 Solução Paralela

Aqui foi aproveitado o código serial para cada processo, entretanto a ordenação é apenas local e não total. Após a chamada, por todos os processos, da ordenação local há um laço de interação para aplicar a ordenação par-ímpar com os vetores de cada processo.

```
// MPI Vars
int my_rank, comm_sz;

// Starting MPI ------------------------------------------------
MPI_Init(NULL, NULL);

MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);


// Root vars ---------------------------------------------------
long int size = atol(argv[1]); // Array size.

long int * arr = gen_array(size); // main array
// -------------------------------------------------------------


// Local vars --------------------------------------------------
int local_size = size/comm_sz; // local list size

long int * local_arr = new long int[local_size]; // local list
// -------------------------------------------------------------
```

```
if( my_rank == 0 )
{
    //print( arr, size ); // printing.
}



// START
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Sending parts of the array to all procs.
MPI_Scatter( arr, local_size, MPI_LONG, local_arr, local_size,
    MPI_LONG, 0, MPI_COMM_WORLD );
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

delete[] arr;


// LOCAL SORT
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Sorting in local and starting timer.
std::chrono::steady_clock::time_point START =
    std::chrono::steady_clock::now();
odd_even_sort( local_arr, local_size ); // sorting.
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Variáveis e Funções

- `size`: Tamanho do vetor passado por linha de comando.
- `arr`: Vetor que terá tamanho `size` e números gerados aleatoriamente.
- `local size`: Tamanho dos vetores locais.
- `local arr`: Vetor local com tamanho `local size` e vazio.
- `MPI Scatter()`: Função do mpi que é responsável pela separação em partes iguais do vetor principal para cada vetor local.

Lógica

1. Chamada da função `Scatter()` apos instanciações das variavéis locais.
    - Cada processo recebe um vetor de tamanho `size÷comm sz`
2. Cada processo chama a função de ordenação para ordenar seu vetor local.
3. Após a chamada há ordenação local mas não total
4. Inicia-se um laço baseado no algoritmo Odd Even para ordenar os vetores locais.

5.
```
// PHASES
    //
        ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    for( int proc_itr = 1; proc_itr <= comm_sz; proc_itr++ )
    {
        if( ( my_rank + proc_itr ) % 2 == 0 ) // Odd
        {
           if( my_rank < comm_sz - 1 )
           {
               PHASE( my_rank, my_rank + 1, local_arr, local_size,
                   MPI_COMM_WORLD );
           }
        }
        else if( my_rank > 0 ) // Even
        {
           PHASE( my_rank - 1, my_rank, local_arr, local_size,
               MPI_COMM_WORLD );
        }
    }

    // Stoping timer.
    std::chrono::steady_clock::time_point STOP =
        std::chrono::steady_clock::now();
    //
        ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

6. **Variáveis e Funções**

   - `proc itr`: Iterador de controle para as fases dos processos.
   - `PHASE()`: Função que realizará a comparação e ordenação entre dois processos.

     –
     ```
     void PHASE( long int SEND_RANK, long int RCV_RANK, long
         int * arr, int size, MPI_Comm COMM )
     {
         // Get the current proc.
         int current_rank;
         MPI_Comm_rank(COMM, &current_rank);

         // temporary list to make possible the "merge"
         long int * temp_arr = new long int[size];

         // List with both proc values.
         long int * aux_arr = new long int[size*2];


         if( current_rank == SEND_RANK )
         {
     ```

```
    // Send with block
    // Will send the local list of the current send
        rank
    // This proccess will sleep until receive the
        sorted list.
    MPI_Send( arr, size, MPI_LONG, RCV_RANK, 0, COMM );

    // Receiving sorted list.
    MPI_Recv( arr, size, MPI_LONG, RCV_RANK, 1, COMM,
        MPI_STATUS_IGNORE );
}
else
{
    // Receiving from send rank his local list
    // Put the list in temporary list.
    MPI_Recv( temp_arr, size, MPI_LONG, SEND_RANK, 0,
        COMM, MPI_STATUS_IGNORE );


    // MERGE ZONE
    // ===========================================
    // Merge to aux_arr with order.

    long int * first = &aux_arr[0];
    long int * last = &aux_arr[size*2];

    long int * runner_1 = &arr[0];
    long int * runner_2 = &temp_arr[0];

    while( first != last )
    {
        // Case 1 and 2 --> complete the rest of the
            merged list
        // with the remaining values of other list.
        if( runner_1 == &arr[size] )
        {
            while( runner_2 != &temp_arr[size] ) //
                case 1
            {
                *first++ = *runner_2++;
            }
        }
        else if( runner_2 == &temp_arr[size] ) // case 2
        {
            while( runner_1 != &arr[size] )
            {
                *first++ = *runner_1++;
            }
        }
        else if( *runner_1 < *runner_2 )
```

```
            {
                *first++ = *runner_1++;
            }
            else
            {
                *first++ = *runner_2++;
            }

        }
        // ============================================


        // DIVIDE ZONE
        // ============================================
        int itr = size;

        for( int i = 0; i < size; i++ )
        {
            temp_arr[i] = aux_arr[i];

            arr[i] = aux_arr[itr];

            itr++;
        }
        // ============================================


        delete[] aux_arr;

        // Sending back the list with lowest values.
        MPI_Send( temp_arr, size, MPI_LONG, SEND_RANK, 1,
            COMM );

        delete[] temp_arr;

    }


}
```

– (a) É instanciada uma nova variável para pegar o processo atual.
  (b) Alocação de array temporário que será usado pelo processo de fase par com intuito de guardar o vetor local do processo ímpar(`SEND RANK`).
  (c) Condição para verificar o processo atual:
    i. Se o processo atual for igual ao processo que envia(`SEND RANK`), então temos uma fase ímpar: O processo atual en-

viará, para o `RCV RANK`, seu vetor local e enquanto aguarda pelo processo par ele bloqueia suas atividade até o `MPI Recv()` onde recebe de voltar um vetor local ordenado de acordo com a posição do processo atual.

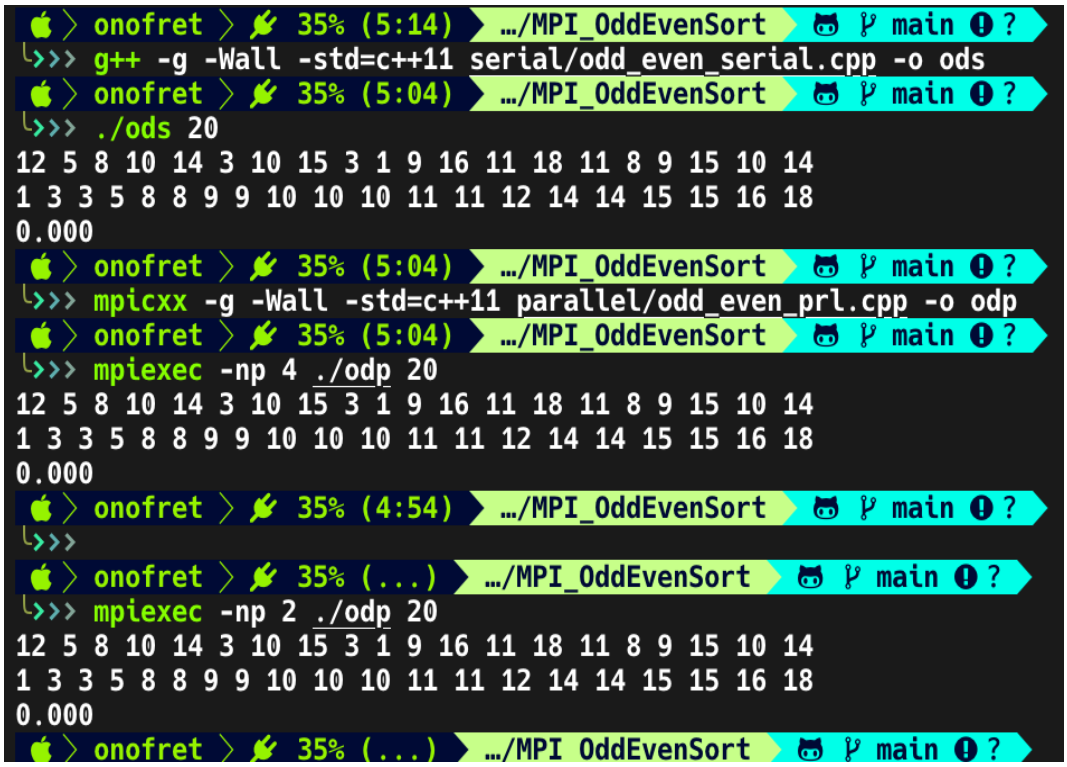
ii. Caso não seja, então temos uma fase par: o processo atual receberá o vetor local do `SEND RANK` e ficará encarregado de ordenar seu próprio vetor local com o vetor recebido(para otimizar este processo foi utilizado o merge sort), ao final retorna a menor metade do vetor merged para o processo ímpar e guarda a maior metade.

7. Com o fim do laço, os vetores locais estarão predispostamente ordenados de maneira que basta concatenar sequencialmente para obter uma ordem total: chamada função `MPI Gather()`.

8.
```
// GATHERING
    // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    MPI_Gather( local_arr, local_size, MPI_LONG, final_arr,
        local_size, MPI_LONG, 0, MPI_COMM_WORLD );
    // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

2.4 Corretude dos Algoritmos

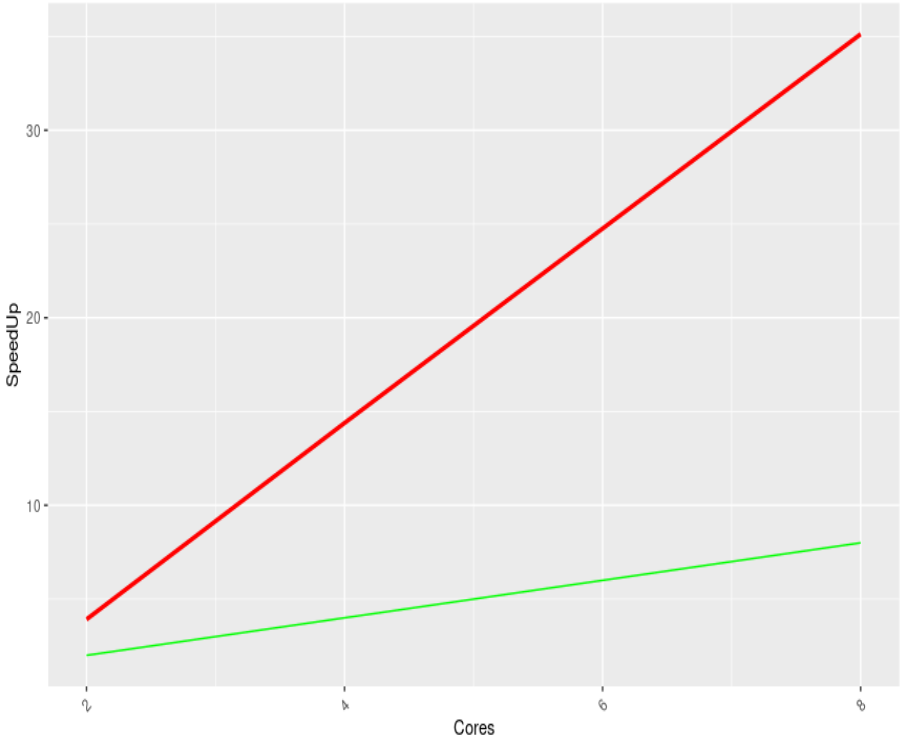
Na imagem a seguir, é demonstrada a corretude dos algoritmos usando como exemplo um mesmo vetor de tamanho 20 para todas as execuções:



Para corretude do paralelo vale notar que independente do número de cores a ordenação é a mesma(na demonstração so foram utilizados 2 e 4 cores).

2.5 Resultados

SpeedUp



- Desempenho esperado
- Desempenho em análise

O gráfico deixa distoante o problema de otimização no código serial, o esperado é que a linha vermelha esteja muito próxima da linha verde, porém, abaixo dela. Como o código serial seguiu uma implementação padrão de um odd even, é provável que a otimição do paralelo com o **merge sort** tenha causado essa disparidade e prejudicado a análise final tornando impraticável, a partir desse gráfico, a análise de eficiência e escalabilidade.

3 Conclusão

O intuito final, além do aprendizado da paralelização do código, é comparar os potenciais de cada paradigma e verificar o poder de escalabilidade do tempo de execução. Com o que pôde ser trabalhado acima, fica clara a necessidade de uma implementação "equilibrada" entre os dois paradigmas para conseguir uma

observação mais precisa, pois, mesmo com a corretude afirmando que está certo, uma otimização a mais pode "boostar" a execução paralela e ocasionar uma espécie de adulteração da análise.