UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

PROCESSAMENTO DISTRIBUÍDO USANDO MPI

Sadi Júnior Domingos Jacinto

Professor orientador: Odorico Machado Mendizabal

Florianópolis
2019

Sadi Júnior Domingos Jacinto

PROCESSAMENTO DISTRIBUÍDO USANDO MPI

Análise do uso de processamento distribuído utilizando MPI, requerido pelo professor da disciplina Programação Paralela e Distribuída, Odorico Machado Mendizabal, necessário para obtenção de nota.

Professor orientador: Odorico Machado Mendizabal

Florianópolis

2019

Conteúdo

1 RESUMO

Muitos problemas interessantes de otimização não podem ser resolvidos de forma exata, utilizando a computação convencional (sequencial) dentro de um tempo razoável, inviabilizando sua utilização em muitas aplicações reais.

Embora os computadores estejam cada vez mais velozes, existem limites físicos e a velocidade dos circuitos não pode continuar melhorando indefinidamente. Por outro lado nos últimos anos tem-se observado uma crescente aceitação e uso de implementações paralelas nas aplicações de alto desempenho como também nas de propósito geral, motivados pelo surgimento de novas arquiteturas que integram dezenas de processadores rápidos e de baixo custo.

Dito isso, o presente relatório tem por objetivo analisar o desempenho de uma aplicação de ordenação de vetores de inteiros em sua implementação paralela, utilizando o MPI para troca de mensagens entre diferentes processos.

2 DEFINIÇÃO DO ESTUDO

O presente relatório buscará analisar o desempenho de uma aplicação responsável por ordenar um vetor de inteiros em sua implementação paralela em comparação com sua implementação sequencial.

Para isso, os seguintes critérios foram seguidos:

- A aplicação foi inteiramente escrita na linguagem C.
- Foi utilizada a biblioteca OpenMPI para realizar a comunicação entre os diferentes processos.
- O algoritmo de ordenação usado foi o *Bucket Sort*, para a classificação dos dados, e a função *qsort()* para a ordenação dos mesmos.
- Tanto a versão paralela quanto a versão sequencial ordenam o mesmo conjunto de dados (copiados para ambas as versões) e utilizam o mesmo número de *buckets*, com o objetivo de os resultados não serem prejudicados por diferenças nas implementações[1].
- Foram escritos duas aplicação para realizar essa análise, uma delas dando prioridade para o processamento dos dados e outra priorizando a otimização de memória.

Tais critérios serão melhor explicados ao longo do relatório.

[1] afora o paralelismo, é claro.

3 MPI (*MESSAGE PASSING INTERFACE*)

O MPI é um padrão de interface voltado para a troca de mensagens em máquinas paralelas com memória distribuída. Apesar de alguns pensarem dessa forma, o MPI não é um compilador ou um produto específico. Além disso, o MPI também fornece suporte à troca de mensagens entre máquinas com memória compartilhada.

No padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado. Porém, esses processos podem executar diferentes programas. Por isso, o padrão MPI é algumas vezes referido como MPMD (*Multiple Program Multiple Data*).

Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. É importante frisar que o número de processos no MPI normalmente é fixo. Dito isso, tais processos podem usar mecanismos de comunicação ponto a ponto (operações para enviar mensagens de um determinado processo a outro), ou coletivas, na qual um grupo de processos pode invocar operações coletivas de comunicação para executar operações globais.

Sobre o MPI, o mesmo é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna.

Finalmente, os algoritmos que criam um processo para cada processador podem ser implementados, diretamente, utilizando-se comunicação ponto a ponto ou coletivas, sendo que os algoritmos que implementam a criação de tarefas dinâmicas ou que garantem a execução concorrente de muitas tarefas, num único processador, precisam de um refinamento nas implementações com o MPI.

Com relação à sua real implementação em um programa, a mesma só é possível através do uso de bibliotecas[2] que fornecem implementações das funções do padrão MPI. É importante salientar que apesar de seguirem a especificação, tais bibliotecas podem diferir em uso e desempenho.

3.1 FUNÇÕES USADAS

O MPI apresenta uma grande gama de funções que implementam diferentes formas de envio de mensagens entre processos. Dito isso, apenas duas dessas funções[3] foram utilizadas neste trabalho, melhor detalhadas nas subsubseções abaixo.

3.1.1 *MPI_Send(void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator)*

Função responsável por enviar mensagens de um processo para outro. Possuí os parâmetros:

- *void* data*:
  É um ponteiro para o dado que será enviado.

- *int count*:
  O tamanho total que será enviado.

[2] sendo as principais o MPICH(`https://www.mpich.org/`) e o Open MPI(`https://www.open-mpi.org/`)
[3] aqui não está sendo considerado as funções de inicialização do MPI, vide:
MPI_Init (&argc, &argv); MPI_Comm_size (comm, &size) e MPI_Comm_rank (comm, &rank)

- *MPI_Datatype datatype*:
  O tipo de dado que será enviado, podendo ser:
  - MPI_SHORT;
  - MPI_INT;
  - MPI_LONG;
  - MPI_LONG_LONG;
  - MPI_UNSIGNED_CHAR;
  - MPI_UNSIGNED_SHORT;
  - MPI_UNSIGNED;
  - MPI_UNSIGNED_LONG;
  - MPI_UNSIGNED_LONG_LONG;
  - MPI_FLOAT;
  - MPI_DOUBLE;
  - MPI_LONG_DOUBLE e
  - MPI_BYTE.

  Trata-se de um constante da própria especificação, usada para evitar incompatibilidade no tamanho dos tipos entre diferentes plataformas.

- *int destination*:
  Identificador do processo para o qual a mensagem será enviada.

- *int tag*:
  Variável personalizável, podendo ser usada para, de acordo com o valor, implementar diferentes ações no receptor. Uma questão importante é que a mensagem só será recepcionada se o valor da *tag* for o mesmo no emissor e no receptor da mensagem.

- *MPI_Comm communicator*:
  Identifica um grupo específico de nodos para o qual a mensagem será enviada.

3.1.2 *MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)*

Função responsável pelo recebimento de mensagens. Possuí os parâmetros:

- *void* data*:
  É um ponteiro que indica o local de memória no qual a mensagem recebida será armazenada.

- *int count*:
  Variável que armazenará o tamanho do dado recebido.

- *MPI_Datatype datatype*:
  Equivalente ao do *MPI_Send*.

- *int destination*:
  Identificador do processo do qual se espera receber uma mensagem.

- *int tag*:
  Equivalente ao do *MPI_Send*.
- *MPI_Comm communicator*: Equivalente ao do *MPI_Send*.
- *MPI_Status* status*:
  É uma estrutura que armazena detalhes da transmissão da mensagem.

4 *BUCKET SORT*

O *Bucket sort* é uma técnica de classificação que classifica os elementos primeiro os dividindo em vários grupos chamados baldes[4]. Os elementos dentro de cada grupo são classificados usando qualquer um dos algoritmos de classificação adequados ou chamando recursivamente o mesmo algoritmo.

Vários *buckets*[5] são criados. Cada *bucket* é preenchido com um intervalo específico de elementos, que então são classificados usando qualquer outro algoritmo. Por fim, os elementos de cada *bucket* são reunidos para obter a matriz ou o vetor original classificado.

O processo de classificação pode ser entendido como uma abordagem de coleta dispersa. Os elementos são primeiro dispersos em *buckets* e, em seguida, os elementos dos *buckets* são classificados. Finalmente, os elementos são reunidos em ordem.

Esse algoritmo é especialmente útil, e fácil de implementar, em aplicações paralelas, onde cada *bucket* pode ser enviado para um processo, onde o mesmo ordenará o *bucket* e o reenviará, já ordenado, para o processo de origem, o que acarretará em um aumento de desempenho com relação à sua implementação sequencial[6].

4.1 CLASSIFICAÇÃO DOS *BUCKETS*

Os elementos podem ser classificados através de uma função matemática que, dados os valores máximos e mínimos dos elementos, além do número de *buckets* disponível, determina para qual *bucket* tal elemento será enviado.

Essa classificação, apesar de adicionar mais processamento ao algoritmo, garante que, após a ordenação de todos os *buckets*, ao agrega-los, do *bucket* de menor índice (0) até o maior (n), o vetor já se encontre ordenado, uma vez que cada *bucket* conterá um intervalo de elementos que sempre será superior ao *bucket* anterior[7] e inferior ao *bucket* seguinte[8].

As imagens a seguir ilustram melhor isso:

A função matemática que é usada para calcular em qual *bucket* o elemento deve ser inserido é:

$$\frac{elem - min}{\frac{max - min + 1}{buckets}}$$

Onde:

- elem: é o elemento que se deseja inserir em algum *bucket*;
- min: menor elemento presente no vetor ou matriz;
- max: maior elemento presente no vetor ou matriz e
- buckets: número total de *buckets* existentes.

4.2 COMPLEXIDADE

- **Pior Complexidade ($\theta(n^2)$)**:
  Quando a distribuição dos elementos não for uniforme, os mesmos provavelmente

[4] ou *buckets* em inglês, daí o nome.
[5] deste ponto em diante, o termo *bucket* será utilizado ao invés do termo balde.
[6] pelo menos em teoria
[7] com exceção do *bucket* 0
[8] com exceção do último *bucket*

serão colocados no mesmo *bucket*. Isso pode resultar em alguns *buckets* com mais número de elementos do que outros, o que irá atrasar a execução consideravelmente.

Nesse cenário, a complexidade passa a depender do algoritmo de ordenação usado para ordenar os elementos do *bucket*.

A complexidade se torna ainda pior quando os elementos estão na ordem inversa.

- **Melhor Complexidade ($\theta(n+k)$)**:
  Ocorre quando os elementos são distribuídos uniformemente nos *buckets*, com um número quase igual de elementos em cada *bucket*.

  A complexidade se torna ainda melhor se os elementos dentro dos *buckets* já estiverem classificados.

  Se o algoritmo de ordenação utilizado for de tempo linear, a complexidade geral, na melhor das hipóteses, será linear, isto é, $\therefore (n+k)$m onde $\therefore (n)$ representa a complexidade para inserir os elementos nos *buckets* e $\theta(k)$ representa a complexidade para ordenar os elementos do *bucket*[9].

- **Caso Médio ($\theta(n)$)**:
  Isso ocorre quando os elementos são distribuídos aleatoriamente na matriz. Mesmo que os elementos não sejam distribuídos uniformemente, a classificação do *bucket* é executada em tempo linear.

[9] considerando que os algoritmos usados são de complexidade de tempo linear

5 qsort()

Trata-se de uma função disponibilizada pela biblioteca *stdlib.h* utilizada para ordenação de vetores. Possuí a sintaxe:

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void * p1, const void* p2))
```

Onde:

- **base*:
  Ponteiro para o primeiro elemento do vetor;
- *nitems*:
  Número de elementos do vetor;
- *size*:
  Tamanho em *bytes* de cada elemento do vetor, é, obrigatoriamente, um inteiro positivo e
- **compar*:
  Função criada para o problema e que compara dois elementos.

A função **compar* é o que garante o polimorfismo dessa função, uma vez que se trata de um ponteiro para outra função, que é definida pelo desenvolvedor e que, portanto, pode ser implementada para ordenar qualquer tipo de dado.

A função que o ponteiro da função **qsort** apontar será chamada sempre com dois elementos e seu retorno ditará se os elementos serão trocados de lugar ou não. Para ficar mais claro, os valores de retorno da função explicitada são:

- Menor que 0 (< 0):
  O elemento apontado por *p1* vai antes do elemento apontado por *p2*.
- 0:
  O elemento apontado por *p1* é equivalente ao elemento apontado por *p2*.
- Maior que 0 (> 0):
  O elemento apontado por *p1* vai depois do elemento apontado por *p2*.

6 SOBRE A IMPLEMENTAÇÃO

6.1 SOBRE A SOLUÇÃO GENÉRICA DO PROBLEMA

Essa sessão será dedicada à explicação da implementação do algoritmo de ordenação paralelo.

Primeiramente, através do *input* do usuário[10], é definido o tamanho do vetor. Em seguida, após alocar memória para esse vetor, o mesmo é preenchido com números aleatórios.

Aqui é importante frisar que tanto a versão paralela quanto a sequencial possuem os mesmos dados alocados nas mesmas posições de seus respectivos vetores. Isso foi feito para garantir que os resultados finais entre a versão paralela e sequencial não sejam afetados pelos elementos nos vetores de cada versão.

```
1 srand(time(NULL));
2 for (i = 0; i < size; i++) { //size é o tamanho do vetor
3   int random = rand() / 1000;
4   parallel_array[i] = random;
5   sequential_array[i] = random;
6 }
7
```

Como dito na sessão 2, o algoritmo de ordenação utilizado foi o *Bucket Sort*[11] junto com o algoritmo *Quick Sort*[12]. Para garantir que as versões paralela e sequencial criassem e ordenassem o mesmo número de *buckets* usando o mesmo método de inserção, foi definido que o número de *buckets* é equivalente ao número de processos disponíveis na versão paralela (incluso o mestre).

Isso foi feito para garantir que a versão sequencial não obtivesse um desempenho inferior por ficar, recursivamente, dividindo o vetor original em *buckets*, o que implica em alocar memória e inserir, através do cálculo apresentado na sessão 4, os elementos em cada novo *bucket* criado.

Além disso, para garantir a precisão dos resultados, tanto a versão paralela quanto a sequência tem, em suas implementações, a tarefa de encontrar o maior e o menor elemento em seus respectivos vetores, mesmo que esses elementos sejam, no final das contas, os mesmos[13].

De resto, tanto a versão paralela quanto a sequencial tratam de dividir o vetor original em *buckets*, ordenar tais *buckets* e depois reagrupá-los no vetor original, sendo que a diferença fundamental entre elas é que, enquanto na versão paralela cada *bucket* é ordenado por um processo, na versão sequencial os *buckets* são ordenados sequencialmente por um único processo. Apesar dessa ser a única diferença fundamental entre ambas as versões, a versão paralela possuí algumas características à mais que valem a pena serem comentadas.

6.1.1 IMPLEMENTAÇÃO PARALELA

Primeiramente, como era de se esperar, apenas o nodo mestre possuí o vetor original. Esse nodo também é o responsável, ao final da execução paralela, por executar a versão sequencial do algoritmo. Isso é garantido através da condicional:

[10] passado como parâmetro na execução da aplicação
[11] usado para classificação
[12] implementado usando a função *qsort()*
[13] dado o fato que ambos os vetores são iguais

```
if(rank == 0) {
  criação e população dos vetores (paralelo e sequencial)

  parallel_sort(...); // função que executa a versão paralela do algoritmo
  ...
  sequential_sort(...); /* função que executa a versão sequencial do
  algoritmo. Por estar dentro da condicional, existe a garantia de
  que apenas um processo irá executar essa função. */
}
```

O nodo mestre é o único responsável por criar os *buckets* e inserir os elementos neles. Para isso, o mesmo possuí duas variáveis, sendo uma delas um vetor de inteiros[14] cujo tamanho é equivalente ao número de processos existentes(incluso o nodo mestre), e a outra sendo um vetores com ponteiros para inteiros[15], novamente sendo esse vetor com tamanho equivalente ao número de processos.

O *bucket_sizes* serve para armazer o tamanho do *bucket* que cada processo irá receber. Em outras palavras, cada posição desse vetor(0 até n) é corresponde à um processo. Já o **short_bucket* é quem irá armazenar os *buckets* e seus elementos, sendo que a mesma lógica do *bucket_sizes* se aplica à ele.

Dessa forma, tem-se que o processo 1 irá ordenar o *bucket* armazenado na posição 1 do vetor **short_bucket*, sendo que o tamanho desse vetor está armazenado na posição 1 do vetor *bucket_sizes*.

Dados esse dados, primeiramente o nodo mestre irá enviar para os processos 1 até n, o tamanho de vetor que os mesmos irão receber e ordenar. Após o mestre receber a confirmação de que cada nodo escravo[16] recebeu sua mensagem, o mesmo, então, envia para os escravos seus respectivos *buckets*[17]. Após ter enviado para cada escravo seu respectivo *bucket*, o mestre então trata de ordenar o *bucket* 0.

Note que essa estratégia possibilita que, ao invés de o mestre, após o envio de todos os dados necessário para os escravos, ficar simplesmente ocioso, esperando que cada escravo finalize seu trabalho e envie para o mestre o *bucket* que o mesmo ficou responsável por ordenar, para só então realocá-los no vetor original e assim encerrar o processo.

Ao invés disso, como desde o princípio o mestre foi incluído no cálculo do tamanho e inserção de elementos nos *buckets*, o mesmo, após enviar todas as informações necessárias para os escravos começarem seu processamento, trata de ordenar um dos *buckets*, o que, além de impedir a ociosidade do mestre, acrescenta ao rol de nodos mais um para realizar o processamento dos dados, o que possibilita um aumento, pequeno ou grande[18], na velocidade de resposta do algoritmo.

O código abaixo exemplifica esse processo:

[14] a partir desse ponto, referenciado como *bucket_sizes*

[15] a partir desse ponto referenciado como **short_bucket*. Perceba que o símbolo * permanece, isso foi feito para ser um constante lembrete para o leitor de que tal variável se trata de vetor de ponteiros

[16] a partir desse ponto, o(s) termo(s) nodo mestre e/ou processo mestre será(ão) substituído(s) por apenas mestre e o(s) termo(s) nodo(s) escravo(s) e/ou processo(s) escravo(s) irá(ão) ser substituído(s) por escravo(s)

[17] **short_bucket*[1..n

[18] esse aumento depende do tamanho do vetor original

```
for (i = 1; i < numprocs; i++) { /* numprocs aqui representa o número total
    de processos existentes. */

  /* Mestre enviando para os escravos 1 à (numprocs - 1) a informação de
   qual será o tamanho do bucket que o mesmo irá receber. */
  MPI_Send(&bucket_sizes[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD);

  /* Mestre enviado para os escravos seus respectivos buckets.
  Para maiores informações sobre a função MPI_Send favor rever a sessão
   3.1.1 */
  MPI_Send(short_bucket[i], bucket_sizes[i], MPI_INT, i, 0,MPI_COMM_WORLD);

  quick_sort(short_bucket[0], bucket_sizes[0]); // Ordenação do bucket 0
}
```

Agora, no lado dos escravos, os mesmos inicialmente permanecem bloqueados, aguardando que o mestre envie o tamanho de seus respectivos *buckets*. Após receberem essa informação, cada escravo aloca memória para um ponteiro de inteiros cujo tamanho será aquele recebido anteriormente. Feito isso, os escravos voltam a esperar que o mestre os envie seus respectivos *buckets*, que, ao serem recebidos, serão armazenas no ponteiro anteriormente alocado.

Nesse ponto, os escravos já possuem todas as informações necessárias para iniciarem seu trabalho, então, cada um deles trata de ordenar seu *bucket*. Assim que um escravo termina de ordenar seu *bucket*, o mesmo trata de enviá-lo para o mestre. Assim que recebe a confirmação de que o mestre recebeu o *bucket*, o escravo é finalizado.

O código abaixo exemplifica esse processo:

```
if(tank == 0) {
  // Execução do mestre
  ...
} else {
  MPI_Recv(&size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  /* Escravo recebe do mestre a informação  de qual o tamanho de seu bucket
    e armazena essa informação na variável size.

  Para maiores informações sobre a função MPI_Recv favor rever a sessão
   3.1.2 */

  int *bucket = (int *)malloc(sizeof(int) * size);
  /* Escravo aloca uma área de memória para armazenar seu bucket quando o
   receber do mestre */

  MPI_Recv(bucket, size, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  /* Escravo recebe do mestre seu bucket e o copia para a área de memória
   previamente criada.*/

    quick_sort(bucket, size); /* Escravo ordena seu bucket*/

    MPI_Send(bucket, size, MPI_INT, 0, 0, MPI_COMM_WORLD); /* E o envia, já
     ordenado, de volta para o mestre */
}
```

Voltando para o mestre, após terminar de ordenar seu *bucket*, o mesmo irá aguardar receber dos escravos os *buckets* para eles enviados. É importante ressaltar que o mestre recebe os *buckets* na mesma ordem em que ele os enviou, ou seja, primeiramente o mestre recebe o *bucket* enviado ao processo 1, por mais que outro processo qualquer tenha terminado sua tarefa mais cedo e esteja a mais tempo tentando enviar seu *bucket* para o

mestre. Além disso, o mestre não apenas recebe os *buckets* na ordem na qual os envio, como também armazena cada vetor recebido na exata posição do vetor *short_ bucket* no qual o mesmo estava originalmente. Isso é possível graças ao fato de que cada processo recebe o *bucket* equivalente à seu rank e, portanto, ao enviar o *bucket* de volta ao mestre, o mesmo só precisa usar o rank do emissor da mensagem para determinar a posição do *bucket* recebido no vetor de *buckets* original.

Finalmente, após receber todos os *buckets*, o mestre concatena os *buckets* em um único vetor, que, graças ao método de inserção do *Bucket Sort*4.1, já estará ordenado. Feito tudo isso, o mestre calcula quanto tempo durou todo esse processamento e, então, passa a executar, sozinho, a versão sequencial do programa.

O código abaixo exemplifica esse processo:

```
for (i = 1; i < numprocs; i++) {
  MPI_Recv(short_bucket[i], bucket_sizes[i], MPI_INT, i, 0, MPI_COMM_WORLD,
    &status);
  /* Recebe o bucket de cada escravo e o armazena em sua posição
  correspondente do vetor de buckets original.
  */
}

/* Percorre cada elemento de cada bucket, em ordem, e o adiciona em sua
   posição final no vetor original */
int index = 0;
for (i = 0; i < numprocs; i++) {
  for (j = 0; j < bucket_sizes[i]; j++) {
        parallel_array[index++] = short_bucket[i][j];
  }
}
```

6.2 SOBRE AS SOLUÇÕES PERSONALIZADAS DO PROBLEMA

Conforme mencionado na seção2, foram realizadas duas implementações da aplicação. Em ambas foram implementadas a versão sequencial e paralela. Além disso, o funcionamento de ambas as implementações, em síntese, é o mesmo na seção anterior a essa. A diferença entre essas implementações consiste no modo como os *buckets* são alocados.

A primeira implementação[19], aloca, para cada *bucket*, o tamanho equivalente ao vetor original. Ao fazer isso, adquire-se um pequeno desempenho, uma vez que, mesmo não sabendo qual o tamanho de cada *bucket*, se sabe que o mesmo não pode ser maior que o vetor original. Assim, é garantido que o cálculo para inserir os elementos nos *buckets*, que, conforme explicado na seção4.2, possuí complexidade $\therefore (n)$, será executado apenas uma única vez. Essa solução, apesar de aumentar o desempenho da aplicação, se torna inviável em casos no qual o vetor é extremamente grande e existem dois ou mais processos, uma vez que a memória necessária para alocar os *buckets* pode ser superior à memória real da máquina.

A segunda implementação[20], aloca a memória dos *buckets* apenas após saber o tamanho exato de cada *bucket*. Para isso, o cálculo de inserção é executado duas vezes, a primeira para determinar o tamanho dos *buckets*, e a segundo para alocar os elementos em seus respectivos *buckets*. Ao tratar a alocação de *buckets* dessa forma, observa-se uma notável perda de desempenho da aplicação, mas ao mesmo tempo, se torna possível realizar a

[19] contida no arquivo **mpi_process_priority.c**
[20] contida no arquivo **mpi_memory_priority.c**

ordenação de vetores maiores do que o suportado pela primeira implementação. Essa solução é simplesmente essência em sistemas que operam com pouca memória, mas é completamente inútil em situações nas quais o sistema possuí uma quantidade absurda de memória e/ou o tamanho do vetor não é extremamente grande.

O código abaixo exemplifica isso:

```
/* Implementação que aloca cada bucket com o tamanho do vetor original */
for (i = 0; i < numprocs; i++) { // numprocs é o número total de buckets
  short_bucket[i] = (int *)malloc(sizeof(int) * size); // size é o tamanho
    total do vetor original.
}

/* Implementação que aloca cada bucket com seu tamanho exato
  Envolve calcular primeiro a inserção dos elementos
  para determinar o tamanho dos buckets para,
  só depois, alocar a memória e recalcular a inserção
  de elementos
*/

unsigned int max = get_max(array, size);
unsigned int min = get_min(array, size);

for (i = 0; i < numprocs; i++) {
  bucket_sizes[i] = 0;


double pivot = (((double)max - min + 1) / (numprocs));
for(i = 0; i < size; i++) {
      int idx = (array[i] - min) / pivot;
      bucket_sizes[idx]++;
}

for (i = 0; i < numprocs; i++) {
  short_bucket[i] = (int *)malloc(sizeof(int) * bucket_sizes[i]);
}
```

6.3 DEMAIS CONSIDERAÇÕES

Algumas observações que são importantes frisar:

- A inicialização do MPI é realizada antes de se iniciar a análise do tempo de processamento.
- A alocação de memória para o vetor original assim como a inserção de elementos aleatórios no mesmo, tanto na versão paralela quanto na sequencial, são realizadas antes de se iniciar a análise do tempo de processamento
- Todos os *outputs* da aplicação estão inseridos em partes do código nas quais ou a análise de tempo de processamento já encerrou ou a mesma ainda não foi iniciada, de forma que tais *outputs* em nada interferem no desempenho final da aplicação.
- Os *outputs* que mostravam o vetor antes e depois de ser ordenado, tanto na versão paralela quanto na sequencial, apesar de estarem em partes não críticas do

código[21] foram comentados para evitar poluição visual[22]. Os mesmos podem ser descomentados à vontade.

- O código fonte contém alguns comentários que explicam, de forma breve, as partes principais da aplicação. Tais comentário, para facilitar a localização, estão no estilo de múltiplas linhas(*/*comentário*/*).
- Com o intuito de facilitar a leitura do código fonte, todas as partes do código que foram possíveis de serem convertidas em funções, o foram.
- O código abaixo explica como foi implementado o cálculo de inserção dos elementos nos *buckets*:

```
unsigned int max = get_max(array, size); // Encontra o maior elemento
    do vetor.
unsigned int min = get_min(array, size); // E o menor

// Calcula qual é o melhor pivo
double pivot = (((double)max - min + 1) / (numprocs)); // numprocs é a
     quantidade de buckets
for (i = 0; i < size; i++) {
  int index = (array[i] - min) / pivot; // Encontra o índice do bucket
     no qual esse elemento será inserido
  short_bucket[index][bucket_sizes[index]] = array[i]; // Insere o
     elemento no bucket
  bucket_sizes[index]++;
}
```

[21] vide item superior
[22] em casos de vetores com tamanho elevado

7 RESULTADOS

Após todo esse detalhamento, ambas as implementações foram submetidas à uma bateria de testes. O ambiente de testes possuí as seguintes configurações:

```
OS
Linux s-pc 4.19.85-1-MANJARO #1 SMP PREEMPT Thu Nov 21
   10:38:39 UTC 2019 x86_64 GNU/Linux

CPU
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits
   virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              1
Core(s) per socket:              8
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           158
Model name:                      Intel(R) Core(TM) i7-9700K
   CPU @ 3.60GHz
Stepping:                        12
CPU MHz:                         800.041
CPU max MHz:                     4900,0000
CPU min MHz:                     800,0000
BogoMIPS:                        7202.00
Virtualization:                  VT-x
L1d cache:                       256 KiB
L1i cache:                       256 KiB
L2 cache:                        2 MiB
L3 cache:                        12 MiB
NUMA node0 CPU(s):               0-7

RAM
              total        used        free      shared  buff/cache   available
Mem:       32880560     4269840    14884996      220864    13725724    27972388
Swap:      17408220           0    17408220
```

Além disso, o seguinte *script* foi utilizado para automatizar os testes:

```
#!/bin/bash

if [ ! -f "memory" ]; then
        mpicc -o memory mpi_memory_priority.c
fi
```

```
if [ ! -d "process" ]; then
        mpicc -o process mpi_process_priority.c
fi

for i in "memory" "process"; do
        for j in {1..8}; do
                for k in 10 100 1000 10000 100000 1000000 10000000; do
                        mpirun -np "$j" "$i" "$k" &>> "$i"_results.txt
                done
        done
done
```

8 CONCLUSÕES

Como era de se esperar, a implementação paralela só obteve melhor desempenho sobre a sequencial quando o tamanho do vetor era relativamente grande, o que confirma que o processamento paralelo de dados só é vantajoso quando existe uma quantidade absurda de dados para serem processados. Além disso, outras confirmações já amplamente conhecidas foram, novamente, confirmadas:

- Uma quantidade absurda de processos não adiciona nenhum ganho de desempenho quando o vetor é pequeno, muito pelo contrário, acarretando, não raramente, em perda de desempenho. Conclusão já conhecida: *speedup* linear é algo quase, se não, impossível de ocorrer na prática.

Fora isso, em todas as execuções foi comprovado que a implementação que se utiliza da estratégia de alocação de *buckets* "não precisa"6.2 possuí desempenho superior ao da implementação com alocação exata de *buckets*. Em compensação, foi comprovado que a implementação que utiliza a estratégia com melhor gerência de memória é capaz de processar uma quantidade maior de dados sem resultar em estouros de memória ou travamento da máquina, cenário não tão incomum para a outra solução.

Para maiores informações a respeito dos resultados obtidos, favor ler os logs das aplicações, que se encontram no arquivo anexado.