Universidade Federal de Santa Catarina
Centro de Florianópolis
Departamento de Informática
e Estatística

Lucas P. Tonussi

MPI Bucket Sort T3

Florianópolis
25 de novembro de 2019

Lucas P. Tonussi

MPI Bucket Sort T3

Trabalho de casa. Sistemas de Informação. UFSC.

Orientador: Prof. Dr. Odorico Machado Mendizabal

Universidade Federal de Santa Catarina
Centro de Florianópolis
Departamento de Informática
e Estatística

Florianópolis
25 de novembro de 2019

Lucas P. Tonussi

MPI Bucket Sort T3

Trabalho de casa. Sistemas de Informação. UFSC.

Comissão Examinadora

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina
Orientador

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina

Florianópolis, 25 de novembro de 2019

Dedico este trabalho a todos aqueles que, de alguma forma,
auxiliaram para a concretização desta etapa.

Agradecimentos

Agradeço a meu pai, minha mãe e a você.

"Quando a festa já estava pela metade,
Jesus subiu ao templo e se pôs a ensinar.
Admirados, os judeus comentavam:
"Como é que este homem sabe letras,
sem ter estudado?"
(João 7.14-5)

Resumo

Resumo em português.

Palavras-Chave: 1. mpi. 2. bucket sort.

Abstract

Abstract in english.
Keywords: 1. mpi. 2. bucket sort.

Lista de figuras

Lista de tabelas

Lista de Siglas e Abreviaturas

PCS	*Parallel Computing Systems*
MPI	*Message Passing Interface*
UFSC	*Universidade Federal de Santa Catarina*

Sumário

1 Introdução

Caso o leitor já conhece o Open MPI então pode ir para a próxima subseção 1.0.1, pois o texto a seguir é meramente uma tradução da página inicial do open mpi <https://www.open-mpi.org>.

O Projeto Open MPI nos serve com: biblioteca, documentação, que implementa uma interface de passagem de mensagem (inglês MPI). Está em constante desenvolvimento por: acadêmicos, pesquisadores, e parceiros da indústria de software/hardware. Open MPI é um arcabouço muito bom para estudar computação distribuída.

Características do OPEN MPI:

1. Totalmente de acordo com MPI 3.1
2. Thread-safe e concorrência integrada
3. Geração dinâmica de processos
4. Tolerância a falha para rede e processos
5. Suporte a heterogeneidade da rede
6. Uma única biblioteca suporta todas as redes
7. Instrumentação em tempo real
8. Suporte a muitos escalonadores de processo
9. Suporte a muitos sistemas operacionais (32, 64 bits)
10. Produção com qualidade de software
11. Alta performance em todas as plataformas
12. Portável e de fácil manutenção
13. Ajustável por instaladores e usuários finais
14. Modelagem baseada em componentes
15. APIs documentadas
16. Lista de emails da comunidade bastante ativa
17. Código aberto licenciado com a BSD

1.0.1 Objetivos desse trabalho

Esse trabalho procura explicar a implementação de Bucket Sort, que é um algoritmo de disposição de valores de um vetor em classes de comprimento, então aplica-se um outro algoritmo de ordenação em cada classe para que o vetor esteja todo ordenado ao final. Mas será aplicando esse algoritmo tendo em vista a computação paralela. A implementação da solução do problema usa Open MPI para distribuir o problema em várias linhas de processamento paralelo.

2 Implementações

2.0.1 Alguns conceitos básicos

Rank

A identificação de um procresso no sistema MPI. Começa em zero (processo raiz/root) até n-1 processos [1, p. 30].

Communicator

O comunicador define uma coleção de processos (grupo), que poderão se comunicar entre si atraves de troca de mensagens [1, p. 30].

Group

Grupo é um conjunto de processos. Todo e qualquer grupo é associado a um "communicator"(MPI_COMM_WORLD) [1, p. 30].

Mundo

Mundo MPI é um mundo virtual que a biblioteca MPI cria para poder gerenciar os processos e suas trocas de mensagens [1, p. 30].

2.0.2 Bucketsort

Figura 1 – Funcionamento do Bucketsort. Fonte da Imagem: Programiz <https://www.programiz.com/dsa/bucket-sort>.

O bucket sort separa, os elementos de um array de números aleatórios, em classes, como se estivesse criando um histograma. E então cada classe (bucket) tem um comprimento

de $x_1 \rightarrow x_2$, $x_3 \rightarrow x_4$. Os elementos de cada bucket compem um novo sub-array, esse sub-array deve ser ordenado, usando-se um ótimo algoritmo de ordenamento (i.e. qsort). Quando esses buckets são reagrupados na ordem disposta pelas classes, o vetor como um todo estará ordenado.

2.0.3 Complexidade do Bucket sort

O pior caso do Bucket Sort, para essa implementação desse trabalho, é $O(n)+O(n^2)+O(n)+O(n^2)$. Veja abaixo:

```
Bucket ** semi_bucket_sort(int *array, Stride **strides, Configuration *p)
{
    Bucket **buckets_rand_nums = (Bucket **)malloc(sizeof(Bucket) * p->
    world_size);

    for (int i = 0; i < p->world_size; i++)
    {
        buckets_rand_nums[i] = (Bucket *)malloc(sizeof(Bucket));
        buckets_rand_nums[i]->index = 0;
        buckets_rand_nums[i]->size = 0;
    }

    for (int s = 0; s < p->array_size; s++) {
        for (int j = 0; j < p->world_size; j++) {
            if (array[s] >= strides[j]->range_min &&
                array[s] < strides[j]->range_max) {
                buckets_rand_nums[j]->size++;
            }
        }
    }

    for (int i = 0; i < p->world_size; i++)
    {
        buckets_rand_nums[i]->array = (int *) malloc
        (sizeof(int) * buckets_rand_nums[i]->size);
    }

    for (int s = 0; s < p->array_size; s++) {
        for (int j = 0; j < p->world_size; j++) {
            if (array[s] >= strides[j]->range_min &&
                array[s] < strides[j]->range_max) {
                bucket_insert(buckets_rand_nums[j], array[s]);
            }
        }
    }
```

Esse método acima, dispõe os valores aleatórios de um arranjo em buckets. Para esse trabalho optou-se não desperdiçar memória atoa, calculando os tamanhos dos buckets antes de alocar memória para eles.

O caso médio é $O(n^2)_{master} + (k * O(n \log n)))_{slaves}$ onde k é igual ao número de buckets disponíveis. Lembrando que essa complexidade é para esse trabalho, pois se está usando distribuição dos trabalhos de ordenamentos para processos. Mas em um sistema real o tempo será $O(n^2)_{master} + (k * O(n \log n)))_{slaves} + k * (MPI\_ISend\_Time) + k * (MPI\_Recv\_Time)$.

Entretanto, no caso médio é onde, esperamos, que o paralelismo usando Open MPI irá nos fornecer com algum aumento do Speedup.

O algoritmo usado para ordenar cada vetor dentro de um bucket pode ser o qsort do c que é garantidamente $O(n \log n)$, por praticidade, pois já vem implementado na biblioteca padrão do C.

2.0.4 Sobre o código em anexo

O número de ranks será definido por MPIRUN -N NRO_PROCESSOS. Cada um desses escravos poderá executar em uma máquina remota, referenciados por algum IP.[1]. É importante notar que o processo raiz também ordena 1 dos buckets criados (ou seja, ele envia para sí mesmo um bucket).

Junto com esse pdf tem um código em C chamado *openmpi_bucket_sort.c* para compilar esse código basta seguir as instruções abaixo:

```
mpicc openmpi_bucket_sort.c -o openmpi_bucket_sort -lm -O2
mpirun -np <NUMERO DE PROCESSOS> openmpi_bucket_sort -t -s <TAMANHO DO ARRAY>
```

Tamanho do Array e Número de Processos

Uma falha do código implementado é não conseguir lidar bem com qualquer número de processos. Esforços foram aplicados para tentar deixar o código pronto para qualquer possibilidade de combinação de número de processos no MPI World e qualquer tamanho de array, mas não foi possível. Se possível abra os arquivos das saídas do programa (S*.TXT).

A seguir o leitor poderá ver um diagrama que procura ensinar como o código foi pensado e implementado. O código em si pode ser visto no arquivo *openmpi_bucket_sort.c*.

[1] Para fins de simulações o software foi executado localmente.

2.0.5 Como foi implementado usando Open MPI

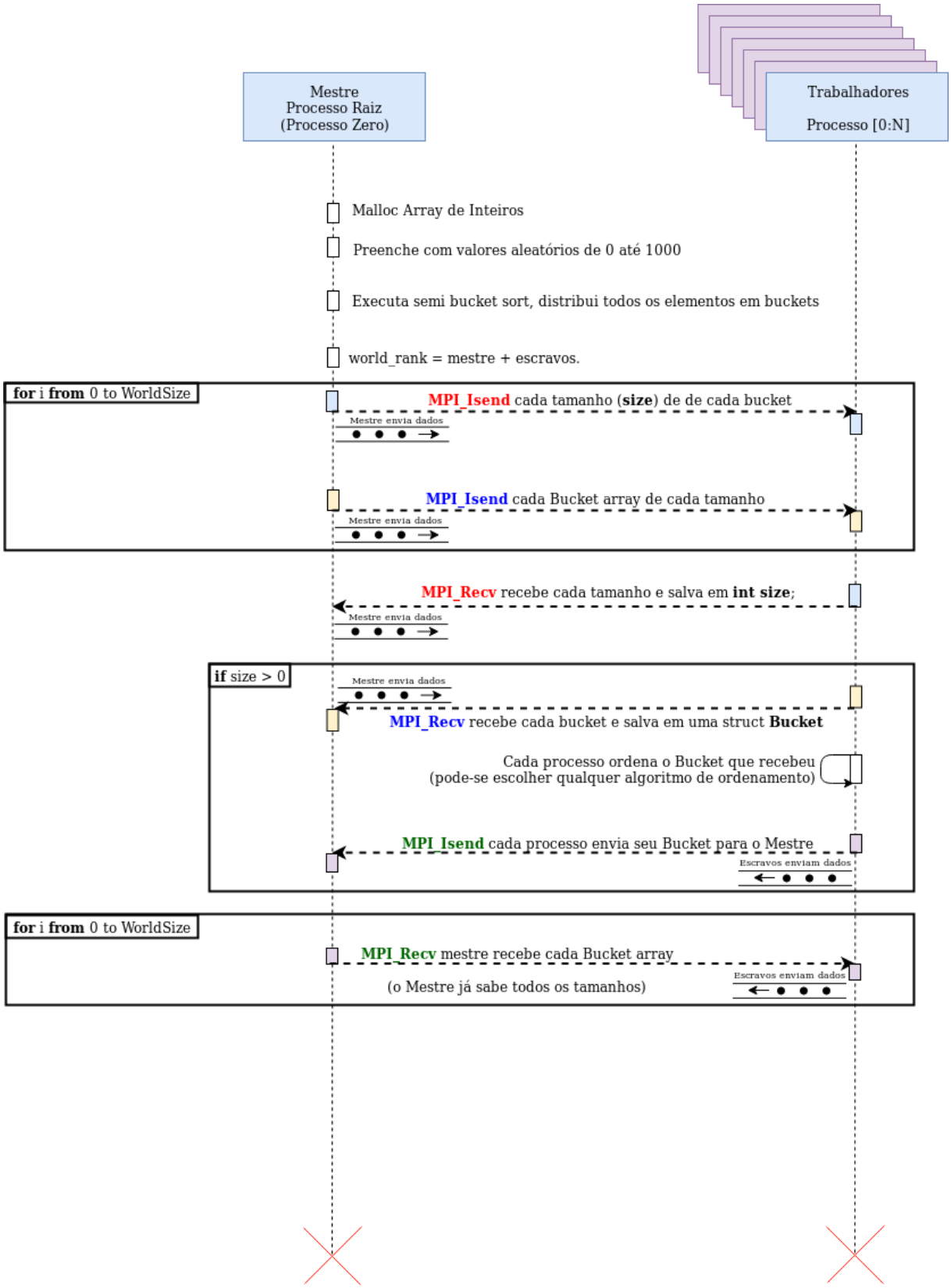

Figura 2 – Esquematização do funcionamento da aplicação em c usando *Sends e Receives*.

A figura [2], acima, mostra a ideia em alto nível do código do Bucket sort distribuído. O arquivo .c tem comentários que explicam o uso dos métodos escolhidos para compor a implementação. Mas vale também mostrar dois métodos que não foram usados, e o por que:

O Host precisa preparar a dispersão de valores em classes. Caso contrário ficaria muito complexo mapear N dispersões e depois juntar todas as sub-dispersões de cada processo para colocar em ordem tudo novamente.

O algoritmo usado para ordenar cada vetor dentro de um bucket pode ser o qsort do c, por praticidade, pois já vem implementado na biblioteca padrão do C.

3 Conclusões

Semáforos e *Mutexes*

Não foi necessário utilizar semáforos, pois não há região de memória sendo acessada por múltiplos processos, concorrentemente, nessa implementação para esse trabalho. Como o método usado é troca de mensagens entre processos, não há risco de mesmas regiões de memória sendo acessada por múltiplos processos, é diferente do caso quando se usa memória compartilhada em que dois processos podem acessar uma mesma região de memória através de uma palavra chave.

Deadlocks

Ocorre quando a passagem de mensagem não pode ser completada por algum motivo. Para evitar deadlock quando usando MPI_Send é preciso tomar cuidado ao garantir que todo MPI_Send terá um correspondente MPI_Recv. Mas no caso dessa implementação estamos usando apenas MPI_Isend então não haveria necessidade de haver um correspondente, pois Isend é não bloqueante.

Rede com ping alto

Um outro problema a se preocupar com um sistema distribuído mestre-escravos é com a latência da rede, se tiver uma latência muito alta isso posso acarretar em demora na resposta dos cálculos. Exemplo: Em alguns momentos a resposta pode estar indisponível para uma série de usuários de um sistema por que os escravos daquela região não estão respondendo a tempo.

Tempo de Execução

Processo	**Tempo**	**World size**	**Qnt. elementos.**
0	0.000142	6	132
0	0.000253	6	231
0	0.000280	6	531
0	0.001051	7	31
0	0.001883	7	132
0	0.000567	7	331
0	0.000274	7	448
0	0.000385	7	777
0	0.000355	7	888

Tabela 1 – Conjunto de testes e o tempo que o mestre demorou para calcular tudo. Usou-se MPI_Time() para registrar o tempo.

1. O tempo de execução tende a diminuir a medida que o tamanho do array aumenta, e também a medida que o WORLD_SIZE aumenta.

2. O tempo de ordenação aumenta com o aumento do número de ranks. Mesmo que o número de ranks ultrapasse o número de cores da máquina.

Referências Bibliográficas

1 Contribuidores da do CNPAD SP (Unicamp). *Apostila de Treinamento: Introdução ao MPI. Centro Nacional de Processamento de Alto Desempenho São Paulo.* 2019. [Acessado em: Novembro, 5, 2019.]. Disponível em: <https://www.cenapad.unicamp.br/servicos/treinamentos/apostilas/apostila_MPI.pdf>. 14