Universidade Federal do Rural do Semi-Árido
Centro Multidisciplinar de Pau dos Ferros
Departamento de Engenharias e Tecnologia
PEX1272 - Programação Concorrente e Distribuída
Docente: Ítalo Assis

Lista de Exercícios

Programação de memória distribuída com MPI

Atenção: as questões 12 e 13 devem ser executadas no supercomputador Santos Dumont começando com 1 processo em 1 nó e dobrando a quantidade de processos até atingir a quantidade de núcleos de 1 nó. Além disso, executar com 2 e 4 nós completos.

O(s) *script(s)* utilizado(s) para execução do programa deve(m) ser incluído(s) na resposta.

1. Modifique a regra trapezoidal para que ela estime corretamente a integral mesmo que `comm_sz` não divida n uniformemente. Você ainda pode assumir que $n \geq comm\_sz$.

2. Modifique o programa que apenas imprime uma linha de saída de cada processo (`mpi_output.c`) para que a saída seja impressa na ordem de classificação do processo: processe a saída de 0 primeiro, depois processe 1 e assim por diante.

3. Suponha que um programa seja executado com `comm_sz` processos e que x seja um vetor com n componentes. Como os componentes de x seriam distribuídos entre os processos em um programa que usasse uma distribuição:

   (a) em bloco?

   (b) cíclica?

   (c) bloco-cíclica com tamanho de bloco b?

   Suas respostas devem ser genéricas para que possam ser usadas independentemente dos valores de `comm_sz`, n e b. Ao mesmo tempo, as distribuições apresentadas nas respostas devem ser "justas", de modo que, se q e r forem dois processos quaisquer, a diferença entre o número de componentes atribuídos a q e a r seja a menor possível.

4. Escreva um programa MPI que receba do usuário dois vetores e um escalar, todos lidos pelo processo 0 e distribuídos entre os processos. O primeiro vetor deve ser multiplicado pelo escalar. Para o segundo vetor, deve-se calcular a sua norma. Os resultados calculados devem ser coletados no processo 0, que os imprime. Você pode assumir que n, a ordem dos vetores, é divisível por `comm_sz`.

5. Encontrar somas de prefixos é uma generalização da soma global. Em vez de simplesmente encontrar a soma de n valores,

$$x_0 + x_1 + \cdots + x_{n-1}, \tag{1}$$

   as somas dos prefixos são as n somas parciais

$$x_0, x_0 + x_1, x_0 + x_1 + x_2, \cdots, x_0 + x_1 + \cdots + x_{n-1}. \tag{2}$$

(a) Elabore um algoritmo serial para calcular as n somas de prefixos de um vetor com n elementos.

(b) Suponha $n = 2^k$ para algum inteiro positivo k. Crie um algoritmo paralelo para um sistema com n processos, cada um armazenando um dos elementos de x, que exija apenas k fases de comunicação.

- Sem utilizar `MPI_Scan`

(c) O MPI fornece uma função de comunicação coletiva, `MPI_Scan`, que pode ser usada para calcular somas de prefixos:

```
int MPI Scan(
void* sendbuf p /* in */,
void* recvbuf p /* out */,
int count /* in */,
MPI Datatype datatype /* in */,
MPI Op op /* in */,
MPI Comm comm /* in */);
```

Ela opera em arrays com `count` elementos; `sendbuf_p` e `recvbuf_p` devem se referir a blocos de `count` elementos do tipo `datatype`. O argumento `op` é igual ao `op` para o `MPI_Reduce`. Escreva um programa MPI que gere um vetor aleatório de `count` elementos em cada processo MPI, encontre as somas dos prefixos e imprima os resultados.

6. Uma alternativa para um `allreduce` estruturado em borboleta é uma estrutura de passagem em anel. Em uma passagem de anel, se houver p processos, cada processo q envia dados para o processo $q + 1$, exceto que o processo $p - 1$ envia dados para o processo 0. Isso é repetido até que cada processo tenha o resultado desejado. Assim, podemos implementar `allreduce` com o seguinte código:

```
sum = temp_val = my_val;
for (i = 1; i < p; i++) {
  MPI_Sendrecv_replace(&temp_val, 1, MPI INT, dest,
                       sendtag, source, recvtag, comm, &status);
  sum += temp_val;
}
```

(a) Escreva um programa MPI que implemente esse algoritmo para o `allreduce`. Como seu desempenho se compara ao `allreduce` estruturado em borboleta?

(b) Modifique o programa MPI que você escreveu na primeira parte para que ele implemente somas de prefixos.

7. As funções `MPI_Scatter` e `MPI_Gather` têm a limitação de que cada processo deve enviar ou receber o mesmo número de itens de dados. Quando este não for o caso, devemos utilizar as funções `MPI_Gatherv` e `MPI_Scatterv`. Consulte a documentação dessas funções e modifique seu programa da questão 4 para que ele possa lidar corretamente com o caso quando n não é divisível por `comm_sz`.

8. Suponha que $comm\_sz = 8$ e o vetor $x = (0, 1, 2, \cdots, 15)$ tenha sido distribuído entre os processos usando uma distribuição em bloco. Desenhe um diagrama ilustrando as etapas de uma implementação borboleta da função `allgather` de x.

9. A função `MPI_Type_contiguous` pode ser usada para construir um tipo de dados derivado de uma coleção de elementos contíguos em uma matriz. Sua sintaxe é

```
int MPI_Type_contiguous(
    int count /* in */,
    MPI Datatype old_mpi_t /* in */,
    MPI Datatype* new_mpi_t_p /* out *);
```

   Modifique as funções `Read_vector` e `Print_vector` (`mpi_vector_add.c`) para que elas usem um tipo de dados MPI criado por uma chamada para `MPI_Type_contiguous` e um argumento de contagem de 1 nas chamadas para `MPI_Scatter` e `MPI_Gather`.

10. A função `MPI_Type_indexed` pode ser usada para construir um tipo de dados derivado de elementos arbitrários de um vetor. Sua sintaxe é

```
int MPI_Type_indexed(
    int count /* in */,
    int array_of_blocklengths[] /* in */,
    int array_of_displacements[] /* in */,
    MPI_Datatype old_mpi_t /* in */,
    MPI_Datatype* new_mpi_t_p /* out */);
```

    Ao contrário da função `MPI_Type_create_struct`, os deslocamentos são medidos em unidades de `old_mpi_t` - não em bytes. Use a função `MPI_Type_indexed` para criar um tipo de dados derivado que corresponda à parte triangular superior de uma matriz quadrada. Por exemplo, na matriz 4 x 4

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

    a parte triangular superior são os elementos 0, 1, 2, 3, 5, 6, 7, 10, 11, 15. O processo 0 deve ler uma matriz $n \times n$ como um vetor unidimensional, criar o tipo de dados derivado e enviar a parte triangular superior com uma única chamada de `MPI_Send`. O processo 1 deve receber a parte triangular superior com uma única chamada ao `MPI_Recv` e depois imprimir os dados recebidos.

11. As funções `MPI_Pack` e `MPI_Unpack` fornecem uma alternativa aos tipos de dados derivados para agrupar dados. O `MPI_Pack` copia os dados a serem enviados, um bloco por vez, em um *buffer* fornecido pelo usuário. O *buffer* pode então ser enviado e recebido. Após o recebimento dos dados, `MPI_Unpack` pode ser usado para descompactá-los do *buffer* de recebimento. A sintaxe do `MPI_Pack` é

```
int MPI_Pack(
    void* in_buf /* in */,
    int in_buf_count /* in */,
    MPI_Datatype datatype /* in */,
    void* pack_buf /* out */,
    int pack_buf_sz /* in */,
    int* position_p /* in/out */,
    MPI_Comm comm /* in */);
```

Poderíamos, portanto, empacotar os dados de entrada para o programa da regra dos trapézios com o seguinte código:

```
char pack_buf[100];
int position = 0;
MPI_Pack(&a, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&b, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&n, 1, MPI_INT, pack_buf, 100, &position, comm);
```

A chave é o argumento da `position`. Quando `MPI_Pack` é chamado, a posição deve referir-se ao primeiro slot disponível no `pack_buf`. Quando `MPI_Pack` retorna, ele se refere ao primeiro slot disponível após os dados que acabaram de ser compactados, portanto, após o processo 0 executar este código, todos os processos podem chamar `MPI_Bcast`:

```
MPI_Bcast(pack_buf, 100, MPI_PACKED, 0, comm);
```

Observe que o tipo de dados MPI para um *buffer* compactado é `MPI_PACKED`. Agora os outros processos podem descompactar os dados usando: `MPI_Unpack`:

```
int MPI Unpack(
    void* pack_buf /* in */,
    int pack_buf_sz /* in */,
    int* position_p /* in/out */,
    void* out_buf /* out */,
    int out_buf_count /* in */,
    MPI_Datatype datatype /* in */,
    MPI_Comm comm /* in */);
```

`MPI_Unpack` pode ser usado "invertendo"as etapas do `MPI_Pack`, ou seja, os dados são descompactados um bloco por vez, começando em $position = 0$.

Escreva outra função `Get_input` para o programa da regra dos trapézios. Este deve usar `MPI_Pack` no processo 0 e `MPI_Unpack` nos demais processos.

12. Cronometre a implementação do livro da regra dos trapézios que usa `MPI_Reduce` para diferentes números de trapézios e processos, n e p, respectivamente.

    (a) Qual critério você utilizou para escolher n?

(b) Como os tempos mínimos se comparam aos tempos médios e medianos?

(c) Quais são os *speedups*?

(d) Quais são as eficiências?

(e) Com base nos dados que você coletou, você diria que a regra dos trapézios é escalável?

13. Encontre os *speedups* e as eficiências da ordenação ímpar-par paralela (`mpi_odd_even.c`).

    (a) O programa obtém *speedups* lineares?

    (b) É escalável?

    (c) É fortemente escalável?

    (d) É fracamente escalável?

14. Modifique a ordenação ímpar-par paralela (`mpi_odd_even.c`) para que as funções `Merge` simplesmente troquem os ponteiros do vetor após encontrar os elementos menores ou maiores. Que efeito essa mudança tem no tempo de execução geral? Fixe uma quantidade de processos (ex.: 8 processos) e analise a influência do tamanho dos vetores no tempo de execução.

Referência

- PACHECO, Peter S. An introduction to parallel programming. Amsterdam Boston: Morgan Kaufmann, c2011. xix, 370 p. ISBN: 9780123742605.