Grupo 4 - Tiago Koji Castro Shibata (8988730), Adilson Torres Gregório de Souza (8988254)

Sistemas de Computação de Alto Desempenho
Aula 11

Relatório:
MPI - Comunicação Coletiva e Não Bloqueante

- **Introdução**

Em aula, estudamos o uso de MPI para sistemas distribuídos com foco na comunicação entre processos, em comunicação bloqueante e não-bloqueante utilizando funções de comunicação coletiva.

- **Configuração do sistema**

Testes foram realizados em um processador Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz, com 4 núcleos. O sistema utilizado é Linux e os processos foram executados com *niceness* mínimo para que outros processos do sistema interferissem menos nos resultados.

Todos os programas foram compilados com *-O3 -DNDEBUG -Wall -Wextra -pedantic*.

- **Programas**

No **exercício 1**, implementamos uma versão com MPI da busca da primeira ocorrência de um número inteiro (de 0 a 19) em 4 vetores (A, B, C, D) de tamanho 100000 cada. Para isso foi utilizado 4 nós, cada um populando o vetor com números aleatórios entre 0 a 19, onde o nó 0 é o que envia o número a ser encontrado para os outros nós.

Implementação com MPI:

```
#include <mpi.h>
#include <stdio.h>


main(argc, argv)

int  argc;
char *argv[];


{
    int n, n_nos, rank, i, valor, vet_ocorrencias[4], indice;
    int vetor[100000], vet_ocorrencias[10000];

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &n_nos);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    n=100000;
```

```
    for (i=0; i<n; i++){
        vetor[i] = rand() % 20;
    }

    if  (rank == 0) {
        printf("Valor a ser encontrado (entre 0 e 19): ");
        fflush(stdout);
        scanf("%d", &valor);
    }
    MPI_Bcast(&valor,1,MPI_INT,0,MPI_COMM_WORLD);

    for (i=0; i<n; i++){
        if (vetor[i] == valor) {
            indice = i;
            break;
        }
    }

    MPI_Gather(&indice,1,MPI_INT,vet_ocorrencias,1,MPI_INT,0,MPI_COMM_WORLD);

    if (rank == 0){
        for (i=0; i< n_nos; i++){
            printf("Ocorrência no nó %d com índice %d\n", i, vet_ocorrencias[i]);
            fflush(stdout);
        }
    }
    MPI_Finalize();
    return(0);
}
```

Na solução foi utilizado o *MPI_Bcast* para comunicar a todos os nós o valor a ser encontrado nos vetores, e foi utilizado também *MPI_Gather* para os valores de ocorrência serem recebidos pelo nó 0.

Ao executar utilizando 4 nós obtemos:

```
$ mpirun --oversubscribe -np 4 01-search-occurrence
Valor a ser encontrado (entre 0 e 19): 4
Ocorrência no nó 0 com índice 44
Ocorrência no nó 1 com índice 44
Ocorrência no nó 2 com índice 44
Ocorrência no nó 3 com índice 44
```

Os índices foram iguais pelo fato da função *rand()* de cada nó teve a mesma semente geradora. Se utilizador sementes diferentes em cada nó obtemos valores diferentes nos vetores.

Ao executar com inicialização da semente por cada nó, *srand()*, obtemos:

```
$ mpirun --oversubscribe -np 4 01-search-occurrence
Valor a ser encontrado (entre 0 e 19): 2
Ocorrência no nó 0 com índice 20
Ocorrência no nó 1 com índice 19
```

```
Ocorrência no nó 2 com índice 10
Ocorrência no nó 3 com índice 34
```

No **exercício 2**, implementamos uma versão com MPI para encontrar o maior elemento de uma matriz, tamanho 1000 x 1000.

Implementação com MPI:

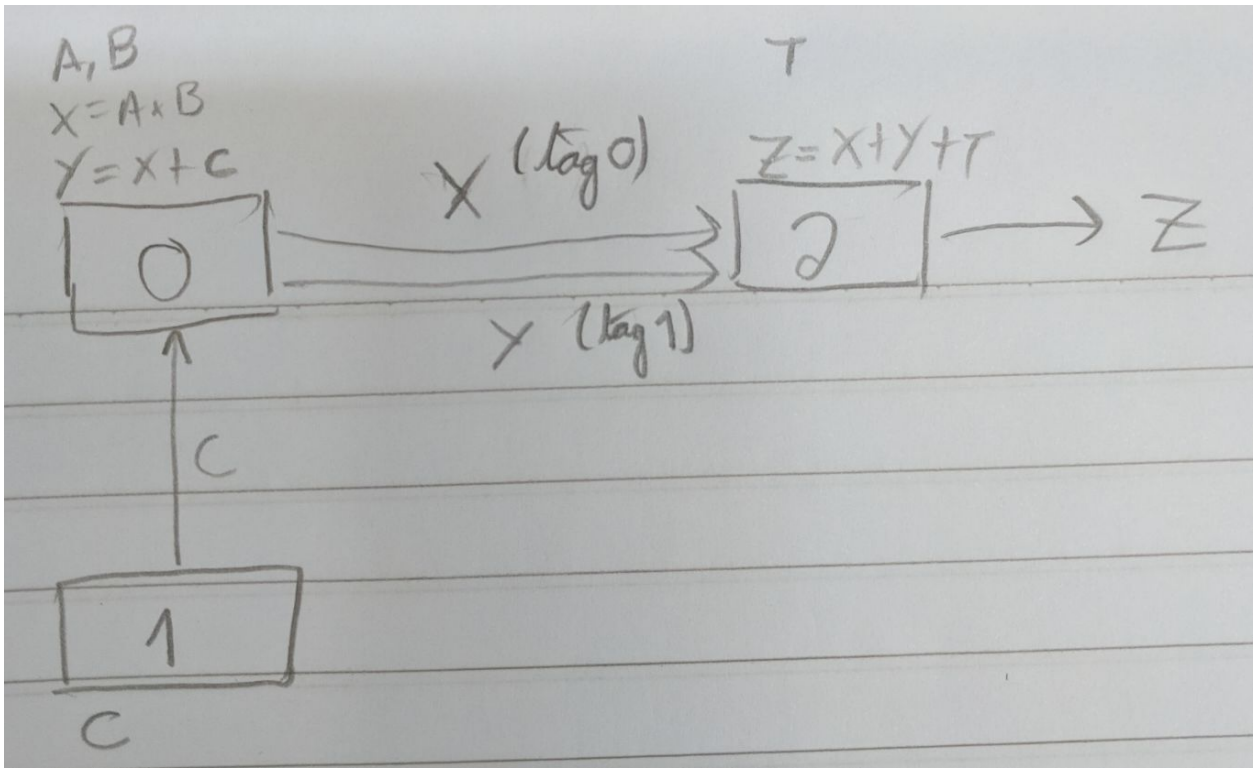
```c
#include <limits.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define SIZE    1000
#define max(a, b)   ((a) > (b) ? (a) : (b))

int A[SIZE][SIZE];

void initialize_data() {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            A[i][j] = rand();
        }
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int comm_size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (!rank) {
        initialize_data();
    }

    int job_size = SIZE * SIZE / comm_size;
    int job[job_size];
    MPI_Scatter(A, job_size, MPI_INT, job, job_size, MPI_INT, 0, MPI_COMM_WORLD);

    int line_max = INT_MIN;
    for (int i = 0; i < SIZE; i++) {
        line_max = max(line_max, job[i]);
    }

    int global_max;
    MPI_Reduce(&line_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    if (!rank) {
        printf("Max value: %d\n", global_max);
    }

    MPI_Finalize();
    return 0;
}
```

Na solução foi utilizado o *MPI_Scatter* para distribuir as linhas da matriz, de modo que cada nó calcula seu máximo local e foi utilizado o *MPI_Reduce* para calcular o máximo total a partir dos máximos locais.

Ao executar utilizando 4 nós obtemos:

```
$ mpirun --oversubscribe -np 4 search-in-matrix
Max value: 2147469841
```

No **exercício 3**, implementamos uma versão de *pipeline* de processamento com MPI. O seguinte diagrama indica os 3 nós realizando processamento:



$$Z=X+Y+T$$

$$X=A \times B, \quad Y=X+C$$

$$A[i][j]=i+j, \quad B[i][j]=2*i-j, \quad C[i][j]=2*i+j, \quad T[i][j]=i+j$$

$$T, X, Y, Z, A, B, C: \text{ matrizes double } 500 \times 500$$

O nó 1 gera a matriz C. O nó 0 gera A e B e calcula X e Y. O nó 2 gera T e calcula Z.

```c
#include <limits.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define SIZE    500
#define max(a, b)   ((a) > (b) ? (a) : (b))

int B[SIZE][SIZE], Z[SIZE][SIZE];

void node_0() {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            B[i][j] = 2 * i - j;
        }
    }

    MPI_Request request_Y = NULL;
    for (int i = 0; i < SIZE; i++) {
        int A_line[SIZE];
        for (int j = 0; j < SIZE; j++) {
            A_line[j] = i + j;
        }
        int C_line[SIZE];
        MPI_Request request_C;
        MPI_Irecv(C_line, SIZE, MPI_INT, 1, 0, MPI_COMM_WORLD, &request_C);

        int X_line[SIZE];
        for (int j = 0; j < SIZE; j++) {
            X_line[j] = 0;
            for (int k = 0; k < SIZE; k++) {
                X_line[j] += A_line[k] * B[k][j];
            }
        }
        MPI_Request request_X;
        MPI_Isend(X_line, SIZE, MPI_INT, 2, 0, MPI_COMM_WORLD, &request_X);

        MPI_Wait(&request_C, MPI_STATUS_IGNORE);
        if (request_Y) {
            MPI_Wait(&request_Y, MPI_STATUS_IGNORE);
        }
        int Y_line[SIZE];
        for (int i = 0; i < SIZE; i++) {
            Y_line[i] = X_line[i] + C_line[i];
        }
        MPI_Isend(Y_line, SIZE, MPI_INT, 2, 1, MPI_COMM_WORLD, &request_Y);

        MPI_Wait(&request_X, MPI_STATUS_IGNORE);
    }
    MPI_Wait(&request_Y, MPI_STATUS_IGNORE);
}

void node_1() {
    MPI_Request request_C = NULL;
    for (int i = 0; i < SIZE; i++) {
        int C_line[SIZE];
        for (int j = 0; j < SIZE; j++) {
            if (request_C) {
                MPI_Wait(&request_C, MPI_STATUS_IGNORE);
```

```
            }
            C_line[j] = 2 * i + j;
        }
        MPI_Isend(C_line, SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &request_C);
    }
    MPI_Wait(&request_C, MPI_STATUS_IGNORE);
}

void node_2() {
    for (int i = 0; i < SIZE; i++) {
        MPI_Request request_X;
        MPI_Request request_Y;
        int X_line[SIZE];
        int Y_line[SIZE];
        MPI_Irecv(X_line, SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &request_X);
        MPI_Irecv(Y_line, SIZE, MPI_INT, 0, 1, MPI_COMM_WORLD, &request_Y);
        for (int j = 0; j < SIZE; j++) {
            Z[i][j] = i + j;
        }
        MPI_Wait(&request_X, MPI_STATUS_IGNORE);
        for (int j = 0; j < SIZE; j++) {
            Z[i][j] += X_line[j];
        }
        MPI_Wait(&request_Y, MPI_STATUS_IGNORE);
        for (int j = 0; j < SIZE; j++) {
            Z[i][j] += Y_line[j];
        }
    }
    printf("Z[0][0] = %d, Z[200][100] = %d, Z[499][499] = %d\n", Z[0][0],
Z[200][100], Z[499][499]);
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    switch (rank) {
        case 0:
        node_0();
        break;

        case 1:
        node_1();
        break;

        case 2:
        node_2();
        break;
    }

    MPI_Finalize();
    return 0;
}
```

O nó 1 gera cada linha de C e faz envio assíncrono. Antes de reiniciar escritas na linha, *MPI_Wait* é chamado para não sobrescrever dados em envio.

O nó 0 calcula B inteiramente (suas colunas serão lidas iterativamente na multiplicação de matrizes para gerar X, portanto pré-calculamos todos os seus valores). Depois, para cada linha, a linha de A é gerada dinamicamente, X é calculado pelo produto matricial, e Y pela soma de X e C. *MPI_Wait* é chamado antes de sobrescrever cada vetor.

O nó 2 inicia cada posição de Z com *i* + *j* (valor inicial de T), depois soma *X* e *Y*.

Testamos comparando os resultados com implementação serial, fornecida pelos professores:

```
$ mpirun -np 3 --oversubscribe ./pipeline
Z[0][0] = 166167000, Z[200][100] = 221017800, Z[499][499] = 41668995
$ aula11-exerc3-seq
Z[0][0]=166167000  Z[200][100]=221017800  Z[499][499]=41668995
```