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

Sistemas de Computação de Alto Desempenho

Aula 10

Relatório:
MPI - Comunicação

- **Introdução**

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

- **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 de um serviço mestre-escravo. O mestre atribui a cada escravo a função a ser executada, recebe cada resultado retornado pelo escravo e imprime, ao todo foram 10 funções. Cada escravo executa a função atribuída e retorna o resultado ao mestre e solicita uma nova função, finalizando ao receber *DIETAG*.

Implementação com MPI:

```c
/*
 Master-slave implementation on mpi.
 */

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

#define WORKTAG 1
#define DIETAG 2
#define NUM_WORKS_REQS 10

static void master();
static void slave();

void (*tab_func[10])();

void func0(){
    int result=0;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

```c
void func1(){
    int result=10;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

void func2(){
    int result=20;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

void func3(){
    int result=30;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

void func4(){
    int result=40;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

void func5(){
    int result=50;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

void func6(){
    int result=60;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

void func7(){
    int result=70;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

void func8(){
    int result=80;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

void func9(){
    int result=90;
    MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

void master(){
    int ntasks, rank, work;
    int result;

    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    work=NUM_WORKS_REQS - 1;

    for (rank=1; rank<ntasks; ++rank){
        MPI_Send(&work, 1, MPI_INT, rank, WORKTAG, MPI_COMM_WORLD);
        work--;
    }

    while (work > -1){
```

```
        MPI_Recv(&result, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
        printf("Result task from slave %d = %d\n", status.MPI_SOURCE, result);
        fflush(stdout);
        MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE, WORKTAG, MPI_COMM_WORLD);
        work--;
    }

    for (rank=1;rank<ntasks;++rank){
        MPI_Recv(&result, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
        printf("Result task from slave %d = %d\n", status.MPI_SOURCE, result);
        fflush(stdout);
        MPI_Send(0, 0, MPI_INT, status.MPI_SOURCE, DIETAG, MPI_COMM_WORLD);
    }
}

void slave(){
    int work;

    MPI_Status status;
    tab_func[0]=func0;
    tab_func[1]=func1;
    tab_func[2]=func2;
    tab_func[3]=func3;
    tab_func[4]=func4;
    tab_func[5]=func5;
    tab_func[6]=func6;
    tab_func[7]=func7;
    tab_func[8]=func8;
    tab_func[9]=func9;

    for(;;) {
        MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if (status.MPI_TAG==DIETAG)
            break;
        (*tab_func[work])();
    }
}

int main(int argc, char *argv[]){
    int myrank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        master();
    } else {
        slave();
    }

    MPI_Finalize();
    return(0);
}
```

Considerando o número de processos igual a 2, obtemos o seguinte resultado:

```
$ mpirun --oversubscribe -np 2 mpi-master-slave
Result task from slave 1 = 90
Result task from slave 1 = 80
Result task from slave 1 = 70
Result task from slave 1 = 60
Result task from slave 1 = 50
Result task from slave 1 = 40
Result task from slave 1 = 30
Result task from slave 1 = 20
Result task from slave 1 = 10
Result task from slave 1 = 0
```

Ou seja, todas as tarefas foram feitas pelo único escravo disponível. Se usarmos mais processos, 4 por exemplo, teremos uma distribuição das tarefas entre os processos:

```
$ mpirun --oversubscribe -np 4 mpi-master-slave
Result task from slave 1 = 90
Result task from slave 2 = 80
Result task from slave 2 = 50
Result task from slave 1 = 60
Result task from slave 2 = 40
Result task from slave 1 = 30
Result task from slave 2 = 20
Result task from slave 1 = 10
Result task from slave 2 = 0
Result task from slave 3 = 70
```

No **exercício 2**, utilizamos comunicação *bufferized* (não bloqueante) entre processos. No programa, 3 processos produtores geram linhas de uma matriz e enviam, linha a linha, com a função *MPI_Bsend* para um consumidor, que soma as 3 matrizes.

Os produtores chamam *MPI_Buffer_attach* para anexar um buffer ao processo atual. Essa função fornece ao MPI um buffer onde dados pendentes podem ser copiados. Depois, *MPI_Bsend* é chamado para os envios. O consumidor recebe as linhas

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

#define SIZE    500

void producer_0() {
    int m[SIZE][SIZE];
    int buffer[sizeof *m + MPI_BSEND_OVERHEAD];
    MPI_Buffer_attach(buffer, sizeof(buffer));

    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
```

```
            m[i][j] = i * 2 - j;
    for (int i = 0; i < SIZE - 1; i++) {
        for (int j = 1; j < SIZE - 1; j++)
            m[i][j] = m[i + 1][j - 1] * 3 - m[i + 1][j + 1];
        MPI_Bsend(m[i], SIZE, MPI_INT, 3, 0, MPI_COMM_WORLD);
    }
    MPI_Bsend(m[SIZE - 1], SIZE, MPI_INT, 3, 0, MPI_COMM_WORLD);
    int size = sizeof(buffer);
    MPI_Buffer_detach(buffer, &size);
}

void producer_1() {
    int m[SIZE][SIZE];
    int buffer[sizeof *m + MPI_BSEND_OVERHEAD];
    MPI_Buffer_attach(buffer, sizeof(buffer));

    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
            m[i][j] = i * 3 - j;
    for (int i = 0; i < SIZE - 1; i++) {
        for (int j = 1; j < SIZE - 1; j++)
            m[i][j] = m[i + 1][j - 1] * 4 - m[i + 1][j + 1];
        MPI_Bsend(m[i], SIZE, MPI_INT, 3, 0, MPI_COMM_WORLD);
    }
    MPI_Bsend(m[SIZE - 1], SIZE, MPI_INT, 3, 0, MPI_COMM_WORLD);
    int size = sizeof(buffer);
    MPI_Buffer_detach(buffer, &size);
}

void producer_2() {
    int m[SIZE][SIZE];
    int buffer[sizeof *m + MPI_BSEND_OVERHEAD];
    MPI_Buffer_attach(buffer, sizeof(buffer));

    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
            m[i][j] = i * 2 + j;
    for (int i = 0; i < SIZE - 1; i++) {
        for (int j = 1; j < SIZE - 1; j++)
            m[i][j] = m[i + 1][j - 1] * 2 - m[i + 1][j + 1];
        MPI_Bsend(m[i], SIZE, MPI_INT, 3, 0, MPI_COMM_WORLD);
    }
    MPI_Bsend(m[SIZE - 1], SIZE, MPI_INT, 3, 0, MPI_COMM_WORLD);
    int size = sizeof(buffer);
    MPI_Buffer_detach(buffer, &size);
}

void consumer() {
    int result[SIZE][SIZE];
    for (int i = 0; i < SIZE; i++) {
        int row[SIZE];
        MPI_Recv(result[i], SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        for (int source = 1; source < 3; source++) {
            MPI_Recv(row, SIZE, MPI_INT, source, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            for (int j = 0; j < SIZE; j++)
                result[i][j] += row[j];
        }
    }
    printf("consumer:\n\tW[0][0] = %d\n\tW[300][400]=%d\n\tW[499][499]=%d\n",
```

```
result[0][0], result[300][400], result[499][499]);
}

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

    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    if (comm_size != 4) {
        fprintf(stderr, "Must be run with 4 MPI processes\n");
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    switch (rank) {
        case 0:
        producer_0();
        break;

        case 1:
        producer_1();
        break;

        case 2:
        producer_2();
        break;

        case 3:
        consumer();
        break;
    }

    MPI_Finalize();
    return 0;
}
```

Ao executar *mpirun -np 4 --oversubscribe ./ex2*, recebemos:

```
consumer:
        W[0][0] = 0
        W[300][400]=2921
        W[499][499]=2994
```

Que é o mesmo resultado da versão serial do programa.