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

Sistemas de Computação de Alto Desempenho

Aula 12

Relatório:
MPI e OpenMP

- **Introdução**

Em aula, estudamos o uso de MPI para sistemas distribuídos com foco na criação dinâmica de processos, uso de barreira com MPI e utilização de MPI com OpenMP.

- **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`*.

- **Programa**

No **exercício 1**, implementamos uma versão de *pipeline* de processamento com MPI e OpenMP com 3 processos. OpenMP foi usado para processamento de matrizes localmente, e MPI para criar processos distribuídos e enviar dados entre eles.

Matrizes 1000 x 1000 com valores *double* foram usadas. O processo 0 inicializa as matrizes A e B e realiza a multiplicação delas na matriz X. O processo 1 realiza as mesmas operações do processo 1 para as matrizes C e D, o resultado da multiplicação é a matriz Y. O processo 2 recebe as matrizes X e Y, realizando a soma delas na matriz Z.

Seguindo o enunciado, processamento com OpenMP foi realizado com 2 threads.

Implementação:

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

#define SIZE    1000

typedef double matrix_t[SIZE][SIZE];

void initialize(matrix_t m) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
```

```
            m[i][j] = rand();
        }
    }
}

void node_0_1(int rank) {
    static matrix_t a, b;
    initialize(a);
    initialize(b);

    for (int i = 0; i < SIZE; i++) {
        double result[SIZE];
        #pragma omp parallel for num_threads(2)
        for (int j = 0; j < SIZE; j++) {
            result[j] = 0;
            for (int k = 0; k < SIZE; k++) {
                result[j] += a[i][k] * b[k][j];
            }
        }
        MPI_Send(result, SIZE, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);
        if (i == 0 || i == SIZE - 1) {
            printf("Rank %d: [%d][%d] = %f\n", rank, i, i, result[i]);
        }
    }
}

void node_2() {
    matrix_t result;

    for (int i = 0; i < SIZE; i++) {
        MPI_Recv(result[i], SIZE, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        double y_line[SIZE];
        MPI_Recv(y_line, SIZE, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        #pragma omp parallel for num_threads(2)
        for (int j = 0; j < SIZE; j++) {
            result[i][j] += y_line[j];
        }
    }
    printf("Rank 2: [0][0] = %f, [999][999] = %f\n", result[0][0],
result[999][999]);
}

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

    srand(rank << 4);
    switch (rank) {
        case 0:
        case 1:
        node_0_1(rank);
        break;

        case 2:
        node_2();
        break;
    }
```

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

Resultados:

```
$ mpirun -np 3 --oversubscribe ./pipeline
Rank 0: [0][0] = 1122307270902069198848.000000
Rank 1: [0][0] = 1140682408211081330688.000000
Rank 2: [0][0] = 2262989679113150660608.000000, [999][999] = 2257963230117456773120.000000
Rank 0: [999][999] = 1109973334172160622592.000000
Rank 1: [999][999] = 1147989895945296150528.000000
```

Vídeo