Assignment – 2

Group-13 Report

Ayyappa Reddy Maramreddy – R11888944

Jason Heinrich – R11913725

Rishabh Kumar – R11785189

Description:

The provided code implements an integer and two-dimensional array which represents a *nxn* matrix with non-negative elements (with zeros on the diagonal). The main goal of this function is to speed up this process by parallelizing the two for loops indicated in the code at the same time.

Our code implements the parallelization of the HW2 function using MPI. Initially, the main function initializes MPI and sets up some variables. It uses “MPI_Init ()” to start the MPI, “MPI_Comm_rank ()” to get the rank of the current process and “MPI_Comm_size ()” to determine the total number of processes. Memory allocation is done for a 4x4 matrix (“data_matrix”) and an array (“output”). The “data_matrix” is initialized with specific values.

The HW2 function is the core of the code. It takes an integer “n”, a 2D array “matrix”, and an array ‘output’ as parameters. This function implements an algorithm for solving a specific problem, which seems to be involved in finding the shortest path through a given “matrix”. The function initializes some variables and dynamically allocates memory for an array “done” to keep track of processed rows.

Inside the HW2 function, there’s a loop that iterates until all rows of the matrix are processed. Within the loop, each process identified by its rank computes a portion of the matrix independently. This parallelization is achieved by dividing the work among the processes based on their ranks. Each process communicates with others using MPI to share local minimum positions and values and then receives the global minimum from process 0.

Once the shortest path is determined, the “output” array is updated accordingly. After all rows are processed, the memory allocated for the “done” array is free. Finally, back in the “main ()” function the results are printed if the rank is 0. This includes the output arrays obtained after the processing the input matrices. The code demonstrates how MPI can be used to distribute computation processes and collaborate to solve a problem in parallel, highlighting parallelization strategies and communication patterns essential for efficient parallel programming.

Overall, Our MPI program code demonstrates the shortest path problem, distributing computation efficiently across processes. Communication and coordination among processes, especially those managed by Rank=0, are crucial for achieving parallel efficiency and obtaining the final results.

Code:

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


void HW2(int n, int **matrix, int *output);
int min(int a, int b);

MPI_Status status;
int rank, process_count;

void main()
{
    //Initializing MPI
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);
    int n = 4;

    int **data_matrix = (int **)malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++) {
        data_matrix[i] = (int *)malloc(n * sizeof(int));
    }
    int  *output = (int *)calloc(n, sizeof(int));
```

Fig.1 Initializing MPI.

```c
    //Matrix with zeros at diagonal positions - 4x4
    data_matrix[0][0] = 0;
    data_matrix[0][1] = 1;
    data_matrix[0][2] = 6;
    data_matrix[0][3] = 0;

    data_matrix[1][0] = 1;
    data_matrix[1][1] = 0;
    data_matrix[1][2] = 3;
    data_matrix[1][3] = 0;

    data_matrix[2][0] = 6;
    data_matrix[2][1] = 3;
    data_matrix[2][2] = 0;
    data_matrix[2][3] = 1;

    data_matrix[3][0] = 0;
    data_matrix[3][1] = 0;
    data_matrix[3][2] = 1;
    data_matrix[3][3] = 0;

    //Call to main function - all processes will enter
    HW2(n, data_matrix, output);

    //
    MPI_Barrier(MPI_COMM_WORLD);
```

Fig2. Matrix with Zero at diagonal positions.

```
    if (rank == 0) {
        printf("rank: %d outputs with zero: \n", rank);
        for (int i = 0; i < n; i++) {
            printf("Output %d: %d\n", i, output[i]);
        }
    }

    //Matrix with zero (except node zero) turned 999 - 4x4
    data_matrix[0][0] = 0;
    data_matrix[0][1] = 1;
    data_matrix[0][2] = 6;
    data_matrix[0][3] = 999;

    data_matrix[1][0] = 1;
    data_matrix[1][1] = 999;
    data_matrix[1][2] = 3;
    data_matrix[1][3] = 999;

    data_matrix[2][0] = 6;
    data_matrix[2][1] = 3;
    data_matrix[2][2] = 999;
    data_matrix[2][3] = 1;

    data_matrix[3][0] = 999;
    data_matrix[3][1] = 999;
    data_matrix[3][2] = 1;
    data_matrix[3][3] = 999;
```

Fig3. Print Results and Data Matrix with zero turned 999.

```
int min(int a, int b){
    if (a<b) {
        return a;
    }
    return b;
}



void HW2(int n, int **matrix, int *output) {
    int i, j, count, tmp, leastVal, leastPos, *done, chunk_leastVal, chunk_leastPos;

    int chunk_size, p_used;
    if(process_count <= n)
        { chunk_size = n / process_count;
        p_used = process_count;
        }
    else {
        chunk_size = 1;
        p_used = n;
    }
    done = (int *) calloc( n, sizeof(int) );
    //printf("chunk size is %d", chunk_size);

    for(i=0; i<n; i++) {
        done[i]= 0;
        output[i] = matrix[0][i];
    }
```

Fig4. Min Function.

```
}
done[0] = 1 ;
count = 1 ;

 if(rank >= n) {
    return;
 }

//Outer loop is not parallelized since it is sequentially dependent.
while( count < n ) {
    leastVal = 987654321 ;
    chunk_leastVal = leastVal;

    //Starting and ending points of the segments for parallelization
    int start = rank * chunk_size;
    int stop = (rank +1) * chunk_size;

    //This loop can be run in parallel since segments of any row can be analyzed in parallel and then merged.
    //We simply divided the number of iterations by the number of processes used and gave them a segment based on their rank
    for(i=start; i<stop && i < n; i++) { // <-- parallelize this loop
        tmp = output[i] ;
        if( (!done[i]) && (tmp < leastVal) ) {
            chunk_leastVal = tmp ;
            chunk_leastPos = i ;
        }
    }
```

Fig5. Parallelizing the first for loop.

```
    // Send data from non-zero processes to process zero
    if (rank != 0) { //Non-zero processes block
        // Send the smallest position and value found by each non-zero process to the zero process
        MPI_Send(&chunk_leastPos, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        MPI_Send(&chunk_leastVal, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
        MPI_Recv(&leastPos, 1, MPI_INT, 0, 3, MPI_COMM_WORLD, &status);
        MPI_Recv(&leastVal, 1, MPI_INT, 0, 4, MPI_COMM_WORLD, &status);

    } else { //Process-zero block
        // Process zero's lowest value and corresponding position
        leastPos = chunk_leastPos;
        leastVal = chunk_leastVal;

        // Process zero takes in the lowes values and corresponding posiions form other non-zero processes
        for (int i = 1; i < p_used; i++) {
            MPI_Recv(&chunk_leastPos, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
            MPI_Recv(&chunk_leastVal, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
            // As values are taken in they are compared with exiisting lowest value and only saved if lower than encumbant
            if (chunk_leastVal < leastVal) {
                leastVal = chunk_leastVal;
                leastPos = chunk_leastPos;
            }
        }
```

Fig6. Sending data from non-zero processes to zero process.

```
    // Send the global lowest value and corresponding position to all non-zero processes
    for (int i = 1; i < p_used; i++) {
        MPI_Send(&leastPos, 1, MPI_INT, i, 3, MPI_COMM_WORLD);
        MPI_Send(&leastVal, 1, MPI_INT, i, 4, MPI_COMM_WORLD);

    }
}
// Make sure each process finishes calc before moving on.
MPI_Barrier(MPI_COMM_WORLD);

//Remove row of leastPos from candidates with done flag and increment count
done[leastPos] = 1;
count++ ;

//int chunk_output[n];
for(i=start; i<stop && i < n; i++) { // <-- parallelize this loop
    if( !(done[i]) )
        output[i] = min(output[i], leastVal + matrix[leastPos][i]);
}
```

Fig7. Parallelizing the second for loop.

```
    // Send output updates to process 0 for compilation
    if (rank != 0) {
        MPI_Send(output + start, chunk_size, MPI_INT, 0, rank, MPI_COMM_WORLD);
    } else {
        for (int src = 1; src < process_count; src++) {
            int loop_start = src * chunk_size;
            MPI_Recv(output + loop_start, chunk_size, MPI_INT, src, src, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }


} /*** End of while ***/
free(done) ;
}
```

Fig8. Send output updates to process 0 for compilation.

Result:

```
PS F:\My Programs\c_not_plusplus_programs\pp_assignments_2-6\assignment_2\group_solution_2> mpiexec -n 4 group_solution_2
rank: 0 outputs with zero:
Output 0: 0
Output 1: 0
Output 2: 1
Output 3: 0
rank: 0 outputs with zero turned 999:
Output 0: 0
Output 1: 1
Output 2: 4
Output 3: 5
```

Fig9. Output