Project Assignment I

All-Pairs Shortest Path Problem
Parallel Computing

Leonardo Hügens

November 2, 2023

1 Problem & Task

All-pairs shortest path problem

- Determining all shortest paths between every pair of nodes in a given weighted directed graph.
- Can be solved using `Repeated Squaring`, were regular matrix multiplication is replaced with `Min-plus` matrix multiplication, a.k.a `Distance Product`.

Fox Algorithm - Perform the `Min-plus Repeated Squaring` repeatedly until you obtain the matrix that corresponds to the size of the shortest path among all possible paths between nodes v_i and v_j, using the `Fox Algorithm` to use several processes in parallel, each performing operations on submatrices of the original matrix.

2 Implementation

2.1 Includes

These are the libraries used, with comments that specify which functions of each library I needed. I will omit the list of MPI functions used, as they are many, and will be presented in the rest of the report.

```
#include <mpi.h>        //mpi
#include <stdio.h>      //printf, scanf
#include <stdlib.h>     //malloc, free
#include <math.h>       //sqrt
```

2.2 Compile, Run

The compilation command for the source code file `fox.c`, producing the executable `fox`, is:

```
mpicc fox.c -o fox -lm
```

, where the `-lm` flag is used to link the `math.h` library to `fox.c`.
The run command is:

```
mpirun -np 4 --hostfile hostfile fox < input6
```

, where the executable `fox` is being run with `-np 4`, meaning 4 processes. Is is run in a computer cluster at `DCC`, the Computer Science department, and the `hostfile` is used to determine the machines and slots per machine that are used. The input matrix residing in the file `input6` is fed to the program using the input redirection operator `<`, which reads from the standard input (`stdin`).

2.3 Matrix input

In the source code, we read from `stdin` using `scanf`, and initialize a vector `mat` with the input matrix entries. We change it to a more useful form, by replacing the off-diagonal 0's by -1's, meaning that between nodes v_i and v_j there is no connection. We change back those -1's to 0's in the end of the program.

After taking the input matrix and converting it, the start time is recorded.

```
if(my_rank == ROOT){
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            scanf("%d", &mat[i * N + j]);

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if(i!=j && mat[i*N+j]==0){
                mat[i * N + j] = -1;
            }
}
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
```

We want to divide this matrix into submatrices and give them to the respective processes. We first broadcast (`BCast`) the full matrix to every process. Then, each process takes its respective submatrix, by using the coordinates of that process in the grid communicator `grid_comm`, multiplying them by `S`, the order of the square submatrices, and reading the relevant submatrix off of the input matrix. Initially, both buffers `submatA` and `submatB` hold copies of the same submatrix. In `grid_comm`, we define a `wrap_around` array with $[1, 1]$, meaning the coordinates are circular in both dimensions, although we only need them to be circular in the second dimension.

```
MPI_Bcast(mat, N*N, MPI_INT, ROOT, MPI_COMM_WORLD);
...
MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, wrap_around, reorder, &
    grid_comm);
MPI_Comm_rank(grid_comm, &my_grid_rank);
MPI_Cart_coords(grid_comm, my_grid_rank, 2, coordinates);

int i_init = coordinates[0]*S;
int j_init = coordinates[1]*S;

for(int i=0; i<S; i++){
    for(int j=0; j<S; j++){
        submatA[i * S + j] = mat[(i_init + i) * N + (j_init + j)];
        submatB[i * S + j] = submatA[i * S + j];
    }
}
```

Now each process has its own submatrices, were we need to make a distinction between `submatA` and `submatB` because we are going to multiply different matrices together and need to keep both in different buffers, to keep track of which came from which process.

2.4 Row and Column communicators

For the Fox algorithm, we are going to need to exchange matrices between processes by row and by column, so we establish the `row_comm` and `col_comm` communicators, which are created by calling `MPI_Cart_sub` with `grid_comm`. For `row_comm` we determine the second coordinate varies, meaning `varying_coords`$=[0, 1]$, and for `col_comm` we have `varying_coords`$=[1, 0]$. We also record what is the rank of the currect process in each of these communicators.

```
// create row communicators
MPI_Comm row_comm;
varying_coords[0] = 0; varying_coords[1] = 1;
MPI_Cart_sub(grid_comm, varying_coords, &row_comm);
MPI_Comm_rank(row_comm, &col_rank);
// create column communicators
MPI_Comm col_comm;
varying_coords[0] = 1; varying_coords[1] = 0;
MPI_Cart_sub(grid_comm, varying_coords, &col_comm);
MPI_Comm_rank(col_comm, &row_rank);
```

2.5 Min-Plus matrix multiplication

As the off-diagonal 0's in our input matrix are now -1's, we can perform the `Min-Plus` matrix multiplication, which instead of always taking the `min`, which theoretically works but practically we can't store ∞ (which happens when there is no connection between nodes v_i and v_j), we don't update our c when at least one element, row element x_i or column element y_i, is -1.

```
int special_vector_mult(int n, int x[], int y[]){
    int c = -1;
    for(int k=0; k<n; k++){
        if(x[k]!=-1 && y[k]!=-1){
            if(c != -1){
                c = min(c, x[k]+y[k]);
            } else {
                c = x[k]+y[k];
            }
        }
    }
    return c;
}
```

2.6 Fox Algorithm

The `Fox` algorithm happens in 4 major stages, identified in the code by comments. We need to perform it repeteadly until we obtain `Df`, the final matrix we desire. So the code structure is as follows:

```
    while(m<N-1){
        ...
        for(int step=0; step<Q; step++){
            // STAGE 1 //
            ...
            // STAGE 2 & STAGE 3 //
            ...
            // STAGE 4 //
            ...
        }
        m = m*2;
    }
    ...
}
```

2.6.1 Stage 1

The first stage is `1. Choose a submatrix of A for each row of processes`. For that, we choose the rank of the process in `row_comm` that is going to be the root of the `Bcast` for the other processes in that row.

```
int chosen_root = (row_rank + step) % Q;
```

2.6.2 Stage 2 & Stage 3

Next, we perform the `Bcast` of the chosen submatrix (stage 2), being careful to receive that submatrix in a different buffer for the processes that are receiving. We perform the `Min-plus` matrix multiplication, which I called `special_matrix_mult`.

```
if (chosen_root == col_rank) {
    MPI_Bcast(submatA, S*S, MPI_INT, chosen_root, row_comm);
    special_matrix_mult(S, submatA, submatB, submatC);
} else {
    MPI_Bcast(temp_submatA, S*S, MPI_INT, chosen_root, row_comm);
    special_matrix_mult(S, temp_submatA, submatB, submatC);
}
```

2.6.3 Stage 4

Next, we send the `submatB` to the process directly above (the processes in the first row sends it to the last row).

```
source_rank = (row_rank + 1) % Q;
dest_rank = (row_rank + Q - 1) % Q;
MPI_Sendrecv_replace(submatB, S*S, MPI_INT, dest_rank, TAG,
   source_rank, TAG, col_comm, &status);
```

2.6.4 Accumulate

Then, we accumulate the results in `submatACC`, by only keeping the `min` of the entries of `submatC`.

```
if(step == 0){
    for(int i=0; i<S; i++)
        for(int j=0; j<S; j++)
            submatACC[i * S + j] = submatC[i * S + j];
} else {
    special_matrix_min(S, submatC, submatACC, submatACC);
}
```

2.7 Output

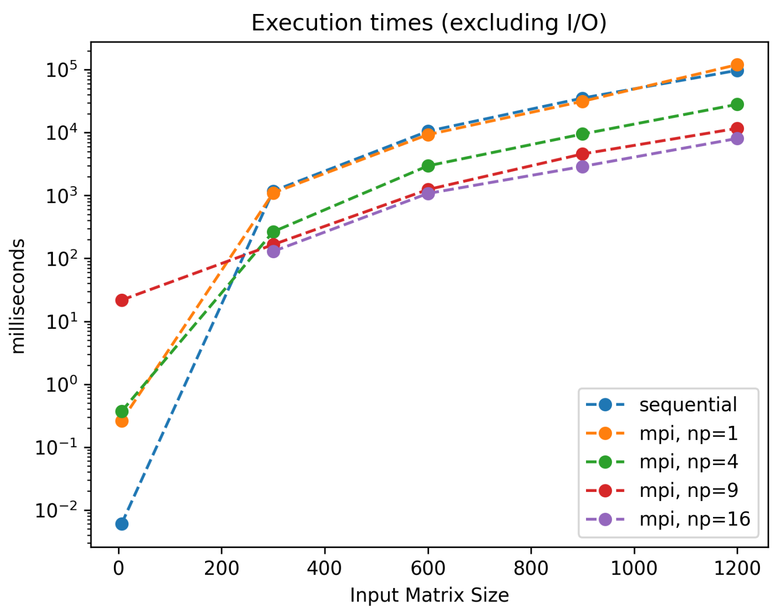
In order to print the resulting matrix orderly, we need to gather the results in `submatACC` on the `ROOT` process (process 0). To do this, each process sends it to `ROOT`, and it puts it back in the buffer of the input matrix.

```
MPI_Send(submatACC, S*S, MPI_INT, ROOT, TAG, MPI_COMM_WORLD);
...
if(my_rank==0){
for(int i=0; i<S; i++)
    for(int j=0; j<S; j++)
        mat[i * N + j] = submatACC[i * S + j];
```

```
for(int proc=1; proc<P; proc++){
    MPI_Recv(submatACC, S*S, MPI_INT, proc, TAG, MPI_COMM_WORLD, &
       status);
    ...
}
```

3 Performance Evaluation

This mplementation of the fox algorithm works perfectly for `input6`, but for matrices bigger than that (`input300, input600, input900, input1200`), it halts. This is due to the inefficiency with witch it gathers the final submatrix in each process into the root process. In order to still evaluate the performance (execution times), I commented out the code that gathers all those submatrices (in a new file `fox_time.c`), and evaluated the final time at that stage. This mean I am able to have in all processes the respective final submatrices, and I consider that a success for the `Fox` algorithm. In the next figure I plot the execution times, for a sequential version of the algorithm (`sequential.c`, which I compiled regularly with `GCC`, and for the MPI code with `n=1, n=4, n=9 and n=16` processes. I use a `log` scale in the execution time axis, in order to focus on the qualitative differences (bigger or smaller) between the curves.



We see that for `input6` (the first column of dots), the sequential algorithms has the smallest time, which makes sense since the input is of small size and it does not need to deal with anything other than performing the repeated `Min-Plus` squaring regularly. For the `MPI` versions, in general they take longer with more processes, due to the time spent with the communications.

The situation reverses for `input300` and bigger, now the sequential and MPI $n = 1$ are the slower ones, and the $n > 1$ are faster. This shows that now the time spent with the communications is worth the time spent actually doing operations related strictly with the repeated `Min-Plus` squaring.

Now for final remarks. Since in each process I have the wanted final submatrix at the end, I consider the algorithm successfully implemented. In a practical viewpoint, it should do the final operation, of gathering and printing all of them orderly, more efficiently, in order to see a final matrix printed in the end, for `input300` and bigger. I tried to implement a and `MPI struct` type and use it, but I didn't manage to use it successfully in this assignment. But at least for `input6` it works from start to finish, and it output the wanted final matrix.