Project Assignment I

All-Pairs Shortest Path Problem

Parallel Computing

Leonardo Hügens

October 31, 2023

1 Problem

All-pairs shortest path problem

- Determining all shortest paths between pairs 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`.

2 Task

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 vi and vj, using the `Fox Algorithm` to use several processes in parallel, each performing operations on submatrices of the original matrix.

3 Implementation

3.1 MPI software

I use the `mpicc` compiler my source code file `fox.c`, and `mpirun` to run the resulting executable file `fox`.

3.2 Includes

These are the libraries I used, with comments that specify with functions of each library I needed. I will omit the list of MPI functions used now, they 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
```

3.3 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 there for the use of `math.h`.
The run command is:

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

, where the executable `fox` is being run with `-np 4`, meaning 4 processes, a hostfile is used to determine the machines and slots per machine that are going to be used, and fed the matrix residing in the file `input6` thorought the `stdin` to the executable.

3.4 Matrix input

The input matrix is fed to the program by `stdin`, using `scanf`, and initialized into a vector `mat`, that represented the matrix. We change it to a more useful form, by replacing the off-diagonal 0's by -1, meaning between nodes vi and vj there is no connection.

After that, we are finished with input, and start the clock.

```
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 broaccast (`BCast` the full matrix to every process. Then, each process takes ifs respective submatrix, by using the coordinates of that process in the `grid_comm` grid comunicator, multiplying them by `S`, the order of the square submatrices, and reading that part of the matrix.

```
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 that 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.

3.5 Row and Column communicators

For the Fox algorithm, we are going to need to exchange matrices between processes by row and by column, so I establish the `row_comm` and `col_comm` communicators:

```
// create row communicators
MPI_Comm row_comm;
...
MPI_Cart_sub(grid_comm, varying_coords, &row_comm);
...
// create column communicators
MPI_Comm col_comm;
...
MPI_Cart_sub(grid_comm, varying_coords, &col_comm);
...
```

3.6 Fox Algorithm

The `Fox` algorithm happens in 4 major steps, identifies in the code by comments. We need to perform it repeteadly until we reach `Df`, the final matrix we want. So the code structure is as follows:

```
    while(m<N-1){
        ...

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

3.6.1 Stage 1

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

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

3.6.2 Stage 2 & Stage 3

Next, we perform the Bcast of the chosen submatrix, being careful to receive that submatrix in a different buffer for the processes different than the root process. We perform the special 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);
}
```

3.6.3 Stage 4

Next, we send the submatB to the process directly above (the processes in the first row send them to the last row). When we established the grid_comm grid communicator, we specified a wrap_around array, that allows us to accommodate this subtlety about the first 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);
```

3.6.4 Accumulate

Then, we accumulate the results, 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);
}
```

3.7 Output

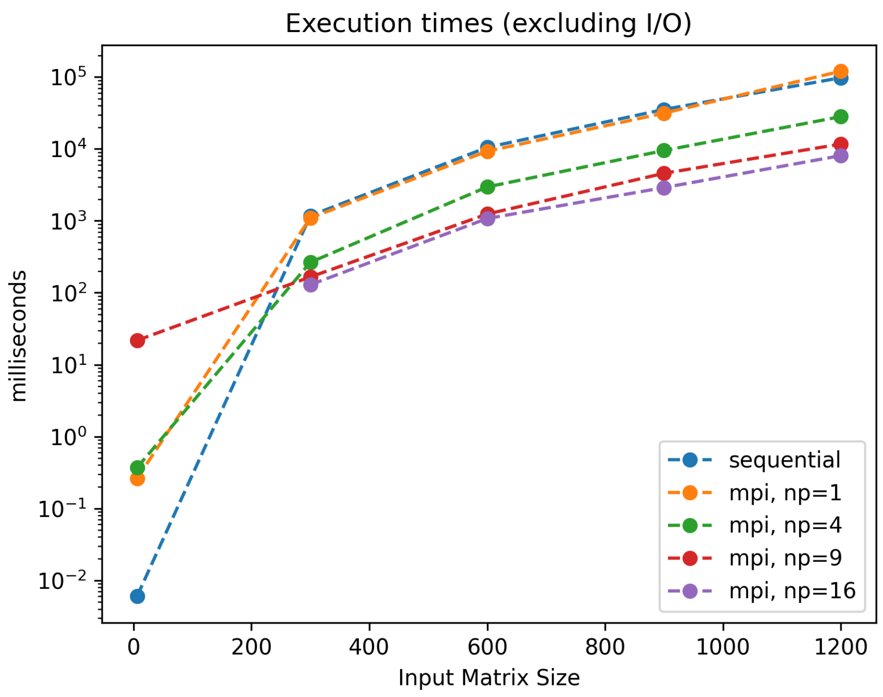
In order to print the resulting matrix orderly, we need to gather the results of the accumulated submatACC on the ROOT process (process 0). To do this, we send them to it, like so:

```
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);
    ...
}
```

4 Performance Evaluation

My implementation of the fox algorithm works perfectly for `input6`, but for matrices bigger than that (`input300, input600, input900, input1200`) halts. This is due to the inefficiency with witch I gather the final submatrix in each process into the root process. In order to still evaluate the performance (execution times), I commented the code gathers all those submatrices, 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, which I compiled regularly with `GCC`, and with 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 it is a 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 is always the slower one, and with the `MPI` versions, the more processes, the less time needed. This shows that now the time spent with the communications is worth the time spent actually doing things related to the repeated `Min-Plus` squaring.

5 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 step, 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.