Programming Assignment 3

Due Wednesday, October 22nd at 5:30 pm

The Floyd-Warshall Algorithm

Suppose we have a collection of cities and a map showing costs associated with travelling between cities — one cost for each pair of cities. How can we find out the cost of the least expensive trip between every pair of cities?

There are several algorithms for solving this problem. One of the simplest, and the one we'll be looking at in program 3 is sometimes called the Floyd-Warshall algorithm, or just Floyd's algorithm. It's named after Robert W. Floyd, who was a CS professor at Stanford, and Stephen Warshall, who was a computer scientist in industry. The basic idea of the algorithm is to look at every possible choice for an intermediate city, `int_city`. Then for each possible intermediate city, look at all possible pairs of cities, `city1` and `city2`. If we can make the cost of getting from `city1` to `city2` smaller by going first from `city1` to `int_city` and then from `int_city` to `city2`, we replace the cost of going from `city1` to `city2` with this lower cost. After checking all possible intermediate cities, a theorem says that we'll have the lowest cost for travelling between each pair of cities.

Of course, a map with routes and costs isn't a very convenient data structure. However, we can record the collection of costs in an *adjacency matrix.* Recollect that a matrix is just a two-dimensional array of numbers. So if there are n cities, first identify each city with a number between 0 and $n-1$. Then the cost of travelling from city i to city j is the entry in the ith row and jth column.

For example, suppose we have four cities and the following adjacency matrix.

Cities	0	1	2	3
0	0	2	2	1
1	5	0	3	∞
2	∞	∞	0	1
3	1	2	4	0

Then the cost of travelling from city 0 to city 2 is 2, while the cost of travelling from city 2 to city 0 is ∞. Note that the cost of travelling from any city to itself is 0, and all the other costs are strictly positive.

If we call this two-dimensional array `mat`, then we can write Floyd's algorithm as follows.

```
Read in n, the number of vertices.
Allocate storage for mat, the adjacency matrix.
Read in mat, the adjacency matrix.
for (int_city = 0; int_city < n; int_city++)
   for (city1 = 0; city1 < n; city1++)
      for (city2 = 0; city2 < n; city2++)
         mat[city1,city2] =
```

```
            min(mat[city1,city2],
                mat[city1,int_city] + mat[int_city,city2]);
Print out mat.
Free storage for mat.
```

If you look on the class website, you can find a C implementation of this in the file `floyd.c`. There's also a program, `gen_mat.c`. for generating matrices that can be used in the `floyd.c` program.

Programming Assignment 3

For program 3, you should parallelize Floyd's algorithm. There are several possible ways to parallelize this. The one you should use divides up the cities among the processes using a *block* partition: if there are n cities and p processes, then

```
Process 0 gets 0, 1, . . ., n/p - 1
Process 1 gets n/p, n/p+1, . . ., 2n/p - 1
   . . .
Process q gets qn/p, qn/p+1, . . ., (q+1)n/p - 1
   . . .
Process p-1 gets (p-1)n/p, (p-1)n/p+1, . . ., n - 1
```

The idea here is that process q is responsible for its assigned cities *and* all routes or *edges leaving* one of its assigned cities. So if we look at the adjacency matrix `mat`, process q is responsible for *rows*

```
qn/p, qn/p+1, . . ., (q+1)n/p - 1
```

Ignoring (for the moment) the input and output phases of the program, we see that the heart of Floyd's algorithm once we've parallelized it will look something like this.

```
for (int_city = 0; int_city < n; int_city++)
   for (city1 = my_first_city; city1 <= my_last_city; city1++)
      for (city2 = 0; city2 < n; city2++)
         mat[city1,city2] =
            min(mat[city1,city2],
               mat[city1,int_city] + mat[int_city,city2]);
```

Now process q "owns" all the rows in the range `my_first_city` to `my_last_city` So if

```
        my_first_city <= city1 <= my_last_city,
```

then process q will have direct access to both `mat[city1,city2]` and `mat[city1,int_city]` regardless of what `city2` and `int_city` are.

However, in general, *another* process will be assigned `mat[int_city,city2]`, since `int_city` can be any value between from 0 to $n-1$. As `city2` ranges from 0 to $n-1$, `mat[int_city,city2]` will range through *all* the values in row `int_city`. So in order for process q to execute the innermost loop, it needs all of row `int_city` This is true regardless of what `int_city` and q are. In other words, *every* process will need all of row `int_city` to execute the nested loops

```
for (city1 = my_first_city; city1 <= my_last_city ; city1++)
   for (city2 = 0; city2 < n; city2++)
      mat[city1,city2] =
         min(mat[city1,city2],
            mat[city1,int_city] + mat[int_city,city2]);
```

This suggests that we broadcast row `int_city` to all the processes at the start of the body of the `for int_city` loop. So our parallel algorithm should look something like this.

```
for (int_city = 0; int_city < n; int_city++) {
   Broadcast all of row int_city;
   for (city1 = my_first_row; city1 <= my_last_row; city1++)
      for (city2 = 0; city2 < n; city2++)
         mat[city1,city2] =
            min(mat[city1,city2],
               mat[city1,int_city] + mat[int_city,city2]);
}
```

Since each each process allocates storage for just its n/p rows, its row numbers are going to range from 0 to $n/p - 1$. So in the algorithm instead of having `city1` range from `my_first_row` to `my_last_row`, it should range from 0 to n/p :

```
for (int_city = 0; int_city < n; int_city++) {
   Broadcast all of row int_city;
   for (local_city1 = 0; local_city1 < n/p ; local_city1++)
      for (city2 = 0; city2 < n; city2++)
         mat[local_city1,city2] =
            min(mat[local_city1,city2],
               mat[local_city1,int_city] + mat[int_city,city2]);
}
```

The next section gives an example showing the difference between global and local numberings for the cities.

Details

Using C's two-dimensional arrays can be problematic. The reason is that we need to specify the number of columns in the array whenever we pass it to a function. Thus, we'll simply use a dynamically allocated one-dimensional array: after getting n, we allocate enough storage for n/p rows of n ints. Now, when we want to access `mat[city1][city2]`, we just access `mat[city1*n+city2]`. (This is, in fact, what C does "under the hood" when a two-dimensional array is being used.)

In order to broadcast row `int_city` we can use the MPI collective communication function `MPI_Bcast`. Here's a prototype

```
int MPI_Bcast(void* buf, int elt_count, MPI_Datatype type,
      int root, MPI_Comm comm);
```

Every process in `comm` must call this function — don't use `MPI_Recv`! The `root` process is the process that owns row `int_city`. In light of our assignment of rows to process, we can find the root by simply dividing:

```
root = int_city/(n/p);
```

(Note that this is *integer* division.) Before broadcasting, the `root` process should copy the row `int_city` into a buffer.

This brings up the question of which row is the row `int_city`: remember, that `int_city` is a *global* subscript, but each process stores *local* rows.

As an example, suppose that $n = 12$ and $p = 3$. Then global rows are assigned as follows:

Process	Global Rows	Local Rows
0	0, 1, 2, 3	0, 1, 2, 3
1	4, 5, 6, 7	0, 1, 2, 3
2	8, 9, 10, 11	0, 1, 2, 3

If you're mathematically inclined, you'll see that the local row number is the *remainder* when the global row number is divided by n/p. For example, global row number 9 is local row number 1 on process 2, and

```
1 = 9 % (12/3) = 9 % 4
```

You'll also see that the process that owns row q is given by the quotient after integer division. For example,

```
2 = 9 / (12/3) = 9 / 4
```

and process 2 is the owner of row 9.

So we can carry out the broadcast of global row `int_city` by executing the following code:

```
root = int_city/(n/p);  // Process that owns in_city
if (my_rank == root) {
   local_int_city = int_city % (n/p);
   for (j = 0; j < n; j++)
      row_int_city[j] = local_mat[local_int_city*n + j];
}
MPI_Bcast(row_int_city, n, MPI_INT, root, MPI_COMM_WORLD);
```

To summarize then, except for the input and output phases, we can parallelize Floyd's algorithm as follows.

```
for (int_city = 0; int_city < n; int_city++) {
   root = int_city/(n/p);
   if (my_rank == root) {
      local_int_city = int_city % (n/p);
      for (j = 0; j < n; j++)
         row_int_city[j] = local_mat[local_int_city*n + j];
```

```
   }
   MPI_Bcast(row_int_city, n, MPI_INT, root, MPI_COMM_WORLD);
   for (local_city1 = 0; local_city1 < n/p ; local_city1++)
      for (city2 = 0; city2 < n; city2++)
         local_mat[local_city1*n + city2] =
            min(local_mat[local_city1*n + city2],
               local_mat[local_city1*n + int_city]
                  + row_int_city[city2]);
}
```

Input and Output

Recall that when writing MPI programs for this class, we should, in general, adhere to the following rules for I/O.

1. In the solution that you submit for a programming assignment all I/O to `stdin` and `stdout` (i.e., calls to `printf` and `scanf`) will be carried out by Process 0.

2. During development, any process can print out data to `stdout` or `stderr` (i.e., calls to `printf` and `fprintf`), but only process 0 can read in data (calls to `scanf`).

3. During development, each process should identify itself when it prints data, e.g.,

   ```
   printf("Proc %d > This is a test\n", my_rank);
   ```

In the submitted solution we need to read in n and the adjacency matrix and we need to print out the matrix of minimum costs. Given the rules about I/O, there are basically two approaches to each of these: we can use a loop of sends or receives on process 0 or we can use MPI collective communications. We already know how to do the loops of sends and receives. So let's talk about the collectives.

We should read in n on process 0, and we can use `MPI_Bcast` to broadcast its value to all the processes:

```
if (my_rank == 0) {
   printf("Enter n\n");
   scanf("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

We can read in the matrix using a collective communication called `MPI_Scatter`. The idea here is that we read the *entire* matrix into temporary storage on process 0. (This, by the way, is a disadvantage of this approach as opposed to the loop of sends.) Then we can execute the following function call on all the processes:

```
MPI_Scatter(temp_mat, n*n/p, MPI_INT,
            local_mat, n*n/p, MPI_INT, 0, comm);
```

Note that the first "count" argument (`n*n/p`) should be the same as the second count.

Printing the matrix is very similar to reading it in: process 0 allocates storage for the entire matrix, and then the processes "gather" it onto process 0 by calling

```
MPI_Gather(local_mat, n*n/p, MPI_INT,
           temp_mat, n*n/p, MPI_INT, 0, comm);
```

Then process 0 can print out the matrix. (There are `man` pages for the MPI functions on CS department lab machines. For example, on a node of the cluster you can type

```
$ man MPI_Scatter
```

to see a description of `MPI_Scatter`.)

Specification

Input to the program will be a positive integer n and a square $n \times n$ matrix. The diagonal entries in the matrix ($i = j$) will be 0. All other entries will be positive integers. The constant `INFINITY` will be represented by 1000000. You can assume that the input will be correct. You can also assume that n will be evenly divisible by p, the number of MPI processes.

Output should be the matrix as updated by Floyd's algorithm. That is, the entry in the ith row and jth column will be the cost of the least-cost route from city i to city j.

You must use Floyd's algorithm to compute the least-cost paths, and you must distribute the adjacency matrix using a block-row distribution. Your program must be efficient in the implementation of Floyd's algorithm: a loop of sends in the `for int_city` loop is unacceptable.

Development

It's essential that you first get your I/O functions working. So start by writing the input and output functions. Once you're sure these are correct, you can start working on Floyd's algorithm. It is very important to start testing and debugging with just one MPI process: if your program doesn't work with one process, it's almost certainly not going to work with more processes.

You may find that when you're using more than one MPI process, debug output of a relatively large data structure can be corrupted by multiple processes attempting to simultaneously print. So I'll put a small function `Print_row` function on the class website. It converts a single row of `local_mat` to a string before printing it out. This tends to be more reliable than trying to print the elements of a row one at a time in a `for` loop.

There's a `gen_mat` program on the website which can be used to generate more or less random input that you can use for testing your program. However, it may be more helpful to use input for which you know the correct output. Two examples are

$$\begin{array}{ccccccc}
0 & 1 & \infty & \infty & \cdots & \infty & \infty \\
\infty & 0 & 1 & 0 & \cdots & \infty & \infty \\
\infty & \infty & 0 & 1 & \cdots & \infty & \infty \\
\infty & \infty & \infty & 0 & \cdots & \infty & \infty \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\infty & \infty & \infty & \infty & \cdots & 0 & 1 \\
\infty & \infty & \infty & \infty & \cdots & \infty & 0
\end{array}$$

and

$$\begin{matrix} 0 & 1 & \infty & \infty & \cdots & \infty & \infty \\ 1 & 0 & 1 & 0 & \cdots & \infty & \infty \\ \infty & 1 & 0 & 1 & \cdots & \infty & \infty \\ \infty & \infty & 1 & 0 & \cdots & \infty & \infty \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \infty & \infty & \infty & \infty & \cdots & 0 & 1 \\ \infty & \infty & \infty & \infty & \cdots & 1 & 0 \end{matrix}$$

(Draw a picture with small n!)

For most (if not all) of your testing you'll want to redirect input from a file:

```
mpiexec -n <number of processes> ./mpi_floyd < <matrix file>
```

As we've noted in class, typing `MPI_COMM_WORLD` can get to be a real chore: you can avoid this by including the following code in your main function:

```
int main(. . .) {
   . . .
   MPI_Comm comm;
   . . .

   MPI_Init(. . .);
   comm = MPI_COMM_WORLD;
   . . .
```

Now each time you want to use `MPI_COMM_WORLD`, you can use `comm` instead. A slight downside to this is that you'll probably need to pass `comm` into many of your functions.

Submission

Your final electronic copy should be committed to your CS SVN repository by 5:30 pm on Wednesday, October 22.

Grading

1. Correctness will be 65% of your grade. Does your program operate as required? Does it correctly read in and print out matrices. Does it find the least cost path between each pair of vertices in a variety of test cases. Is your code efficient in the sense described above?

2. Documentation will be 10% of your grade. Does your header documentation include the author's name, the purpose of the program, and a description of how to use the program? Are the identifiers meaningful? Are any obscure constructs clearly explained? Does the function header documentation explain the purpose of the function, its arguments and its return value?

3. Source format will be 10% of your grade. Is the indentation consistent? Have blank lines been used so that the program is easy to read? Is the use of capitalization consistent? Are there any lines of source that are longer than 80 characters (i.e., wrap around the screen)?

4. Quality of solution will be 15% of your grade. Are any of your functions more than 40 lines (not including blank lines, curly braces, or comments)? Are there multipurpose functions? Is your solution too clever — e.g., has the solution been condensed to the point where it's incomprehensible?

Academic Honesty

Remember that you can discuss the program with your classmates, but you cannot look at anyone else's source code. This includes source code that you might find on the internet. You also cannot show your code to another student. If you're unsure what's allowed, just ask me.