Programming Assignment 1

Dijkstra's Algorithm

Edsger Dijkstra was a Dutch computer scientist. He is unquestionably one of the founding fathers of computer science. In addition to devising Dijkstra's algorithm, he was one of the originators of *structured programming* — a precursor to modern object-oriented programming. C, for example, is a structured programming language. He also made many contributions to the solution of process-coordination problems in parallel computing. His Wikipedia page lists more than fifty fundamental contributions to computer science.

Dijkstra's algorithm solves "the single-source shortest path problem" in a weighted, directed graph. The input consists of a weighted, directed graph and a specified vertex in the graph. A *directed graph* or *digraph* is a collection of *vertices* and *directed edges*, which join one vertex to another. A *weighted digrap*h has numerical weights assigned to the vertices and/or the edges.

The input graph for Dijkstra's algorithm has nonnegative weights assigned to the edges, and the algorithm finds a shortest path from the specified vertex, the *source* vertex, to every other vertex in the graph. The algorithm uses four main data structures:

1. The adjacency matrix mat of the graph: mat[v][w] is the cost or weight of the edge from vertex v to vertex w. If there is no edge joining the vertices, the cost or weight is ∞.
2. A list dist: dist[v] is the current estimate of the cost of the least cost path from the source to v.
3. A list pred: pred[v] is the "predecessor" of vertex v on the current path 0->v.
4. A set known consisting of those vertices for which the shortest path from the source to the vertex is known.

Our vertices will be 0,1,2, ... , n-1, and vertex 0 is the source.

The algorithm is divided into phases: initialization and path-finding. During initialization, dist[v] is set to mat[0][v] for each vertex v. So our initial estimate of the length of the shortest path 0->v is just the length of the edge from 0 to v. Of course, if there is no edge 0->v, then dist[v] will be ∞. Initially, then 0 is the "predecessor" on the path 0->v. So pred[v] should be set to 0 for each vertex v. We'll require that the distance from any vertex to itself is 0. So mat[v][v] is 0 for every vertex v. and after the initialization, 0 will have dist[0] = 0 and pred[0] = 0. Since edges have nonnegative weights, we know the shortest path to 0. So known = {0}.

After initialization, we iterate n-1 times, once for each remaining vertex. In each iteration, we first find a vertex u whose distance to 0 is as small as possible among the vertices not already in the set known. This vertex u is added to known, and the dist and pred entries are updated for the vertices still not in known: if there is a shorter path to v obtained by going through u, then dist and pred are updated accordingly.

This gives the following algorithm:

```
/* Dijkstra's Algorithm */
/*
 * Input:   n: the number of vertices in the digraph
 *           mat: the adjacency matrix of the digraph:
 *                mat[v][w] is the length of the edge from v to w or
 *               infinity if there is no edge from v to w.
 */
for (v = 0; v < n; v++) {
  dist[v] = mat[0][v];  /*  Length of edge 0->v or infinity */
  pred[v] = 0;  /*  0 precedes v on the current shortest path */
  known[v] = 0; /*  We don't know the shortest path to v */
}
known[0] = 1;   /* We do know the shortest path to 0 */

for (int i = 0; i < n; i++) {
  /* Among vertices not in  known, find vertex */
  /* u with smallest dist val   */
  u = Find_min_dist(dist, known, n);

  /* Add u to known */
  known = known + {u}.

  /* Among  vertices not in known, update Dist[v]   */
  /*   and  Pred[v] if there's a shorter path 0->u->v */
  for (v =  1; v < n;   v++)
    if (v is not in known) {
        new_dist = dist[u] + matu;
        if  (new_dist   < dist[v]) {
            dist[v] =   new_dist;
            pred[v] =   u;
        }
    }   /* if v */
}  /* for i */
```

Output:
Length of shortest path 0->v for each vertex v
Shortest path 0->v for each vertex v

Parallelizing Dijkstra's Algorithm

We can divide the work among several processes by assigning the responsibility for different vertices to different processes. If p, the number of processes evenly divides, n, the number of vertices, then we can assign responsibility for the vertices using a block partition:

Process 0:	Vertices 0, 1, ..., n/p - 1
Process 1:	Vertices n/p, n/p+1, ..., 2n/p - 1
...	
Process p-1:	Vertices (p-1)n/p, (p-1)n/p + 1, ... , pn/p-1

For example, if p = 3, and n = 6, then the vertices should be partitioned as follows:

Process 0:	Vertices 0, 1
Process 1:	Vertices 2, 3
Process 2:	Vertices 4, 5

Most of the computational work in the program is in the code that finds u, the vertex with minimum dist value, and the code that updates the dist and pred values after a new vertex has been added to known. In order to find u in the serial algorithm, we can do something like this:

```
u = -1;
min_dist = INFINITY;
for (v = 0; v < n; v++)
  if (v is not in known)
    if (dist[v] < min_dist) {
      u = v;
      min_dist = dist[v];
    }
```

So if dist and known are divided among the processes, this can be parallelized with essentially the same code:

```
loc_u = -1;
loc_min_dist = INFINITY;
for (loc_v = 0; loc_v < loc_n; loc_v++)
    if (loc_v is not in loc_known)
        if (loc_dist[v] < loc_min_dist) {
            loc_u = loc_v;
            loc_min_dist = loc_dist[v];
        }
```

Here, loc_n = n/p. Of course, now each process will have a candidate for u. So to get the "global" minimum vertex, we need to take a global minimum. This can be done fairly efficiently with tree-structured communication. However, it's easier to use the MPI function `MPI_Allreduce`. To do this each process should declare two two-element arrays of int:

```
int my_min[2], glbl_min[2];
```

The values in my_min should be

```
my_min[0] = loc_dist[loc_u];
my_min[1] = Global_vertex(loc_u);`
```

So the first element is the distance from 0 to loc_u. To understand the second, note that loc_u is a local vertex number: it ranges from 0 to loc_n = n/p. It needs to be a global vertex number. We can compute this by simply computing

```
global_u = loc_u + my_rank*loc_n;
```

Now we can call MPI_Allreduce:

```
MPI_Allreduce(my_min, glbl_min, 1, MPI_2INT, MPI_MINLOC, comm);
```

This will return the global min dist in glbl_min[0] and the global vertex number of the vertex with this min dist in glbl_min[1]. So

```
u = glbl_min[1].
```

A few comments on the call to MPI_Allreduce are in order: First we pass in both the minimum distance on each process together with the vertex where that minimum distance occurs. Our call to MPI_Allreduce will find the minimum of all the distances, and store that in glbl_min[0]. It will also find the vertex where the global minimum occurred and store that in glbl_min[1]. At this point we're really interested in u, the vertex where the global min occurs, but the distance is needed to find u. So we pass in a two-element array of ints. Hence, we use the special MPI datatype MPI_2INT, which can be used in communications for arrays storing two ints.

For the second part of the main loop, we need to look at all the vertices assigned to the current process that are not yet elements of known. For each such vertex loc_v, we need to compare its current distance from 0 to the distance obtained by looking at the path to loc_v that goes through u, the vertex just added to known. If each process has all the columns of mat that correspond its vertices, then this "submatrix" of columns can be used to calculate the possibly new distance:

```
new_dist = dist_u + loc_mat[u][loc_v];
```

Then if new_dist is smaller, we can update the current distance and the current predecessor:

```
if  (new_dist < loc_dist[loc_v]) {
  loc_dist[loc_v] = new_dist;
  loc_pred[loc_v] = u;
}
```

Program 1

For programming assignment 1, you should write an MPI program that parallelizes Dijkstra's algorithm using the approach discussed in the previous section. Input to the program will consist of the following.

1. The number n of vertices in the digraph. You can assume that n is evenly divisible by p; the number of processes.
2. The "adjacency matrix" mat of the digraph. This is a matrix with n rows and n columns. The entry in row v and column w, mat[v][w], is the distance from vertex v to vertex w. All the entries will be nonnegative ints, and the entries on the "diagonal" (row subscript = column subscript) will be 0.

The output of the program should consist of

1. A list of the lengths of the shortest paths from vertex 0 to each vertex v, and
2. The vertices on the shortest 0->v, for each vertex v.

Implementation Details

Working with arrays in distributed memory programs can be confusing, since subscripts may be different from the corresponding serial program. For example, suppose we have the following 4×4 matrix mat

0 3 1 2

5 0 9 1

6 1 0 4

4 2 8 3

in a serial program, but the matrix is divided among two processes in a parallel program:

Proc 0:	0	3	Proc 1:	1	2
	5	0		9	1
	6	1		0	4
	4	2		8	3

Then in the serial program, the element in row 2, column 3 is accessed with mat[2][3]. But if the process submatrices are called loc_mat, then this element is stored in row 2, column 1on process 1: loc_mat[2][1]. In this program row subscripts will be the same in both the parallel and serial programs, but column subscripts will be different.
To convert a "local" column or vertex to a "global" column or vertex, you can use the formula:

```
global vertex = local vertex + my_rank*loc_n
```

To go the other way

```
local vertex = global vertex % loc_n
```

To determine whether a process has been assigned a particulary global vertex, you can check whether

```
(global vertex)/loc_n == my_rank
```

Although C does have two-dimensional arrays, there are serious limitations on how they can be used. So any matrices used in your program (i.e., the input matrix mat and the blockcolumn submatrices loc_mat) should be stored as one-dimensional arrays, and elements should be accessed using arithmetic:

```
The entry in row v and column w of mat is mat[v*n + w].
The entry in row v and column loc_w of loc_mat
  is loc_mat[v*loc_n + loc_w].
```

Storage for these matrices can be allocated with malloc:

```
loc_mat = malloc(n*loc_nsizeof(int));
```

This will allocate storage for n*loc_n ints, and elements can be accessed using ordinary array subscripts. A similar approach can be used to allocate storage for dist, loc_dist, etc. Of course, storage allocated with malloc should be freed when it’s no longer needed:

```
free(loc_mat);
```

will free all the storage allocated with the call to malloc.

The global constant INFINITY will be input as 1000000. You can use some other value in your program, but your program must recognize 1000000 as INFINITY on input.

There some I/O functions on the class website in the file `mpi_io.c`. The most important of these is `Read_Matrix`. It uses a special MPI Datatype to distribute the matrix columns among the processes. You can treat this function and its supporting function `Build_blk_col_type` as a black box: you don't need to learn how they work.

The distributed arrays dist and pred can be collected onto process 0 for processing and/or output using the MPI collective communication function MPI_Gather. For example, if dist consists of n ints with a block of n/p = loc_n ints assigned to each process in loc_dist, then dist can be collected onto process 0 with the call

```
MPI_Gather(loc_dist, loc_n, MPI_INT, dist, loc_n, MPI_INT, 0, comm);
```

Note that the arg following dist, loc_n, is the number of elements received from each process. It's not the total number of elements received.

Development

Using conventional debuggers with distributed memory programs can be quite difficult: it will probably be easier to debug your code using printf/fflush. Remember that any process can print to stdout. So any process can print output. However, the order in which output is actually printed may be fairly random. In fact, the output from from one process can be interrupted by another process. The chances that this will happen are somewhat less, if you put a block of your output into a string, and then just print the string. For an example of this, see the function Print_local_matrix in the file mpi_io.c on the class website.

Submission

1. A full version of your source code and a brief documentation should be sent to TA's email (lc3018121@sjtu.edu.cn). You should compress them to one file and name it just like `<stu1_id>_<stu1_name>_<stu2_id>_<stu2_name>_proj1`, for example, `0000000001_小民_0000000002_小涛_proj1`
2. In the documentation, you should answer the following questions. How do you parallelize the serial implementation? Does your program work perfect? What's the speed-up of it comparing to the serial one?
3. We need several teams to do presentations in class. And volunteers or just chosen ones should prepare slides to show how you implement this project.

Academic Honesty

Remember that you can discuss the program with your classmates, but you cannot look at anyone else's pseudo-code or source code. (Of course, this includes code that you might find on the internet.)