**GAZI UNIVERSITY**

**COMPUTER ENGINEERING DEPARTMENT**

**PARALLEL COMPUTER ARCHITECTURES AND PROGRAMMING**

**FINAL PROJECT**



*“Implementation of Dijkstra’s Shortest Path Algorithm via MPI”*

Melih GÖRGÜLÜ

161180032

June, 2020

**Table of Contents**

[**1.** **Algorithm of the Code** 3](#_Toc43476543)

[**2.** **Variables** 5](#_Toc43476544)

[**3.** **Used MPI Directives** 5](#_Toc43476545)

[**4.** **Functions** 6](#_Toc43476546)

[**5.** **Executing the Program** 8](#_Toc43476547)

[**6.** **References** 9](#_Toc43476548)

# **Algorithm of the Code**

Simply put: The matrix is partioned by columns so that each process gets n/p columns. In each iteration, each process finds its local vertex with the shortest distance from the source vertex 1. A global minimum vertex u of the found shortest distances is computed and then each process updates its local distance array if there's a shorter path that goes through u. As mentioned above, this algorithm assumes that the number of vertices can be precisely divided by the process count. If the number of processes given as input cannot be completely divided by the number of vertices, the program will fail. To explain the logic of the algorithm with an example:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 0 | 40 | 15 | Inf | Inf | Inf | 35 | Inf |
| 2 | 40 | 0 | 0 | 100 | Inf | Inf | 25 | Inf |
| 3 | 15 | 0 | 0 | 20 | 10 | 50 | Inf | 50 |
| 4 | Inf | 100 | 20 | 0 | 10 | Inf | 45 | Inf |
| 5 | Inf | Inf | 10 | 10 | 0 | 30 | 50 | 30 |
| 6 | Inf | Inf | 50 | Inf | 30 | 0 | Inf | 0 |
| 7 | 35 | 25 | Inf | 45 | 50 | Inf | 0 | 0 |
| 8 | Inf | Inf | 50 | Inf | 30 | 0 | 0 | 0 |

Table 1.1 Distance Matrix.

Suppose we have a distance matrix as in Table 1.1. In this case, the number of nodes in the graph will be n = 8. Assuming the number of processes to be used in the algorithm is 4, this distance matrix will be divided into 4 matrices with 2x8. For a process, the matrix to which it is assigned is its local matrix. The entire matrix is ​​called the global matrix. By dividing the matrix into processes, a share is formed:

* Process 0 : Assigned vertices are 1 and 2.
* Process 1 : Assigned vertices are 3 and 4.
* Process 2 : Assigned vertices are 5 and 6.
* Process 3: Assigned vertices are 7 and 8.

By doing a column partition of the matrix then each process gets responsible for all incoming edges to their assigned vertices. Each process finds the local minimum distance from the global source vertice to the vertice assigned to it. Now, one of the processes has found a new global minimum distance from the source vertex. This vertex will not be visited again. All processes update their local distance from the source to this new found global vertex by checking if the distance of the path from the source to the global vertex to the assigned vertex is shorter than the distance straight to the assigned vertex from the source. This procedure is repeated n – 1 times, therefore minimum distance from the source vertex to all the other vertices is now computed. To sample the first iteration in Table 1.1:

For example, for process 3, local matrix looks like this:

|  |  |
| --- | --- |
| **Local 0 (Global 7)** | **Local 1 (Global 8)** |
| 35 | Inf |
| 25 | Inf |
| Inf | 50 |
| 45 | Inf |
| 50 | 30 |
| Inf | 0 |
| 0 | 0 |
| 0 | 0 |

Table 1.2 Process 3 local matrix.

***Process 0:*** Finds its local minimum distance 40 from (local) 1 -> (global) 2

***Process 1:*** Can’t find path from local 1 (global vertice 3) to source (because value is infinity). But there is a local minumum path from local 0 (global vertice 3) to source (Value=15).

***Process 2:*** Finds its local minimum distance INF since there are no edge from 0 to any of its assigned vertices (source to global 5) and (source to global 6)

***Process 3:*** Can’t find path from local 1 (global vertice 8) to source (because value is infinity). But there is a local minumum path from local 0 (global vertice 7) to source (Its value is 35).

* Now each process checks if the distance to their assigned vertices can be updated.
* Process 2 notices that there is a new shorter path to its assigned (global) vertex 5 from the source 0. ( 5 -> 3 -> 1)
* This new distance is calculated by the adding the distance of the path to the global minimum and adding the cost of the path from that global minimum to the assigned vertex 5.
* The distance from the global minimum to the assigned vertex 5 is 10 (3 to 5 is 10), so the total distance from 1 to 5 ( 5->3->1) is now 10 + 15 = 25.
* The distance gets stored locally at process 2 and is going to be used in later findings of global minimum distances.

# **Variables**

**MPI\_Comm comm:** Communicator that describe a group of process (MPI\_COMM\_WORLD was used).

**MPI\_Datatype blk\_col\_mpi\_t:** to store MPI derived data type.

**n:** Number of vertices in the graph.

**loc\_n:** Assigned as n / nmb\_of\_process. Number cols in the block column. Eeach process’s local matrix size is : loc\_n X 8.

**loc\_mat:** Calling process's submatrix, Allocated submatrix size = n X number of columns (loc\_n)

**loc\_dist:** to store local shortest distances.

**loc\_known:** loc\_known[v] = 1 if vertex has been visited, 0 else.

**loc\_pred:** loc\_pred[v] = predecessor of v on a shortest path from source to v.

**blk\_col\_mpi\_t:** an MPI\_Datatype that represents a block column of a matrix.

**global\_dist:** Minumum distances found iny processes is saved in this variable.

**global\_distance\_matrix[8][8]:** To store input distance matrix.

**global\_pred[v]:** global\_pred[v] = predecessor of v on a shortest path from source to v (global)

**task\_id :** Process task id (0 for root process).

# **Used MPI Directives**

**MPI\_Datatype:** To create MPI derived data type.

**MPI\_Comm:** Communicator that describe a group of process. MPI\_COMM\_WORLD is used, so all processes that were started together by mpiexec. Also, each process assigned a unique integer rank between 0 and number of tasks – 1.

**MPI\_Comm\_rank:** To get the rank of the calling MPI process within the specified Communicator.

**MPI\_Comm\_size:** Returns the total number of MPI processes in the specified Communicator.

**MPI\_Aint:** Used for heterogeneous datatype creation.

**MPI\_Type\_contiguous:** To create contiguous datatype.

**MPI\_Type\_get\_extent:** Get the lower bound and extent for a Datatype

**MPI\_Type\_vector:** To create vector data type.

**MPI\_Type\_create\_resized:** Needed to get the right extent of the new datatype

**MPI\_Type\_commit**: To commit data type

**MPI\_Scatter:** Sends data from one process to all processes in a communicator. Used to scatter the distance matrix to the processes.

**MPI\_Allreduce:** Combines values from all processes and distributes the result back to all processes. Used to combine the output of processes on the local matrix.

**MPI\_Bcast:** To broadcast the number of vertices to the all proccess.

# **Functions**

**init\_number\_of\_vertices(int task\_id, MPI\_Comm comm):**  To initialize the vertice count.

***Arguments:***

***task\_id*** : Calling process rank, ***comm:*** Communicator containing all calling processes.

***Output:***Number of vertices in the graph.

**Build\_blk\_col\_type(int n, int loc\_n) :** Used to build an MIP\_Datatype that represents a block column of a matrix. This function uses the MPI directives mentioned in title 3.

***Arguments:***

***n:*** Number of vertices, ***loc\_n :*** calling process’s local matrix.

***Output:*** Builded MPI\_Datatype.

**init\_matrix(int loc\_mat[], int n, int loc\_n, MPI\_Datatype blk\_col\_mpi\_t, int task\_id, MPI\_Comm comm) :** Used to distrubute nXn matrix among the processes. In the fucntion, Firstly global\_matrix is readed. After that, the matrix is distrubuted among the processes via using MPI\_Scatter Directive.

***Arguments:***

***Task\_id:*** caller process’s id, loc\_mat: Caller process’s local matrix (receive buffer for Scatter Directive), ***n:*** number of vertices, ***loc\_n:*** row count (2 X loc\_n matrix), ***blk\_col\_mpi\_t:*** sended data type, ***comm:*** Communicator.

**Dijkstra\_Init(int loc\_mat[], int loc\_pred[], int loc\_dist[], int loc\_known[], int task\_id, int loc\_n) :** Used to initialize all the matrices so that Dijkstras Algorithm can be run. This function sets first vertice (source vertice) as visited and sets others vertices as not visited.

***Outputs:***

***loc\_mat:*** local matrix containing edge costs between vertices.

***loc\_dist:*** loc\_dist[v] = shortest distance from the source to each vertex v.

***loc\_pred:*** loc\_pred[v] = predecessor of v on a shortest path from source to v.

***loc\_known:*** loc\_known[v] = 1 if vertex has been visited, 0 else.

**Dijkstra(int loc\_mat[], int loc\_dist[], int loc\_pred[], int loc\_n, int n, MPI\_Comm comm)** Used to compute all the shortest paths from the source vertex 0 to all vertices v via Dijkstra Shortest Path Algorithm. This function calls the Dijsktra\_Init function. After that, run loop for n-1 time (n - 1 times since we already know the shortest path to global vertex 0 from global vertex 0). For each iterations, the following operations are done:

* Call the Find\_min\_dist function (To find minumum distances). Get the minimum distance found by the processes.
* Store the found min distance and global vertex in the glbl\_min via MPI\_ALLreduce directive.

***Arguments:***

***Loc\_mat:*** Calling process’s local matrix, ***loc\_dist*:** Calling process’s distance array, ***loc\_pred*:** loc\_pred[v] = predecessor of v on a shortest path from source to v, ***loc\_n:*** block column count, n: number of vertices

**Find\_min\_dist(int loc\_dist[], int loc\_known[], int loc\_n) :** This function finds the minimum distance between the edge assigned to the process that calls the method and the source vertice.

***Arguments:***

***loc\_dist:***  array with distances from source 0, ***loc\_known :*** array with values 1 if the vertex has been visited 0 if not, ***loc\_n :*** local number of vertices.

***Output:***

***loc\_u:*** the vertex with the smallest value in loc\_dist,-1 if all vertices are already known.

**print\_distance\_matrix(int global\_mat[]):** Used to pirnt contens of distance matrix.

**Print\_dists(int global\_dist[], int n) :** Used to print the length of the shortest path from 0 to each(uses the global distance)

# **Executing the Program**

**Mpiexec** directive is used to executing the program. The number of processes/tasks to be used in Dijkstra Shortest Path Algorithm must be given when running the program. The figure below shows how the program is executed and the program outputs.

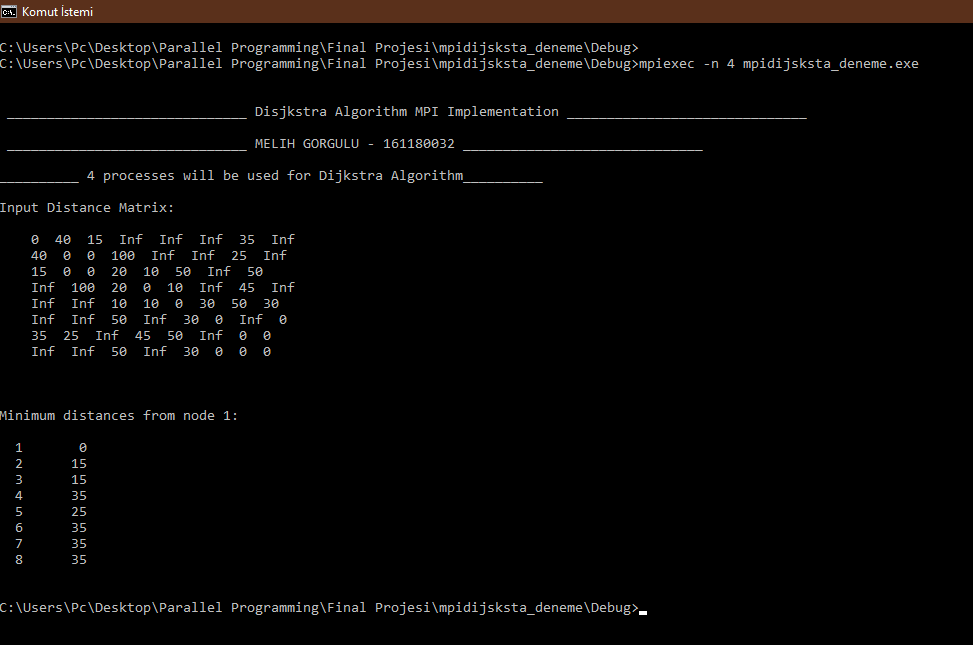


Figure 1.1 Executing the program.

Figure 1 shows the program run by assigning process number 4. As shown in figure, after running the program, the distance matrix and the minimum distances found are shown on the console.

# **References**

[1] *Blaise Barney, Message Passing Interface (MPI),* Internet: <https://computing.llnl.gov/tutorials/mpi/>,Last Access Date: 18.06.2020

[2] *MPICH (High-Performance Portable MPI),* Internet: <https://www.mpich.org/>, Last Access Date: 18.06.2020