CS 426/525 Fall 2021 Project 2

Due 19/11/2021, 23:59

1 Introduction

In this project, we will focus on Single Source Shortest Path (SSSP) algorithm for graphs. First, you will implement a serial version of the algorithm, then design a parallel version of it using MPI library. As in the first project, you are going to use C or C++ programming languages. You will compare parallel and serial versions with various inputs, then write a short report of your findings. Submissions will be on Moodle.

2 SSSP as a Sparse Matrix-Vector Multiplication(SpMV) Problem

Finding the shortest paths for a given source can be mapped to an iterative matrix-vector multiplication operation. The intuition for mapping comes from the graph and matrix duality. We define graphs with set of vertices V and set of edges E as graph $G = (V, E)$. To represent a graph, we can use an adjacency matrix A of size $|V| \times |V|$. An element at row i and column j of matrix A, $A_{i,j}$, gives a non-zero weight value if edge $(i, j) \in E$ and ∞ otherwise. An example graph is given in Figure 1 and its adjacency matrix in Figure 2.

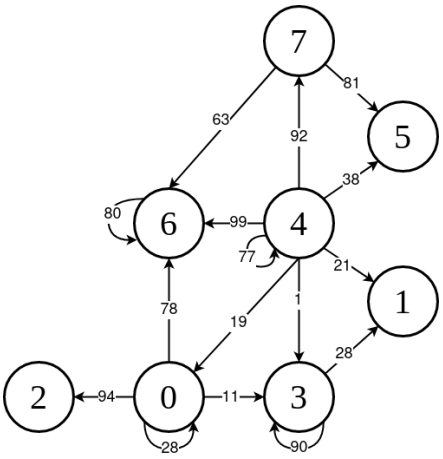

Figure 1: Sample graph with 8 vertices and 16 edges

$$\begin{pmatrix} 28 & \infty & 94 & 11 & \infty & \infty & 78 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 28 & \infty & 90 & \infty & \infty & \infty & \infty \\ 19 & 21 & \infty & 1 & 77 & 38 & 99 & 92 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 80 & \infty \\ \infty & \infty & \infty & \infty & \infty & 81 & 63 & \infty \end{pmatrix}$$

Figure 2: Adjacency matrix representation of sample graph

In SSSP problem, we seek the shortest distances and paths from a given source vertex to all other vertices. For simplicity, we will focus only on the distances. We can represent the distances as a vector D of size $|V|$, and initialize the vector as ∞ for all indices except the index of the source

vertex. The index of the source vertex in the vector should be 0. Now we take the transpose of adjacency matrix and multiply it with D. However, we don't execute the usual point-wise multiplication and addition operations of matrix-vector multiplication. Instead, we replace the point-wise multiplication operation with point-wise addition operation, and we replace the summation with minimum operation. That is to say, in normal matrix-vector multiplication we compute the product $R = M \cdot D$ where M is a $n \times n$ matrix, R and D are $n \times 1$ vectors. To find the product, we compute for $i = 1, 2, ..., n$

$R_{i,1} = \sum_{k=1}^{n} M_{i,k} \cdot D_{k,1}$

Now if we change multiplication by addition and summation by minimization operation, we get:

$R_{i,1} = \min_{k=1}^{n} (M_{i,k} + D_{k,1})$

If we apply this operation on transpose of adjacency matrix and initial distance vector, we do the following computation (assume the source vertex is 4):

$$
\begin{aligned}
R &= A^T \cdot D^{(0)} \\
&= \begin{pmatrix}
28 & \infty & \infty & \infty & 19 & \infty & \infty & \infty \\
\infty & \infty & \infty & 28 & 21 & \infty & \infty & \infty \\
94 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
11 & \infty & \infty & 90 & 1 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 77 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 38 & \infty & \infty & 81 \\
78 & \infty & \infty & \infty & 99 & \infty & 80 & 63 \\
\infty & \infty & \infty & \infty & 92 & \infty & \infty & \infty
\end{pmatrix}
\begin{pmatrix} \infty \\ \infty \\ \infty \\ \infty \\ 0 \\ \infty \\ \infty \\ \infty \end{pmatrix} \\
&= \begin{pmatrix} 19 \\ 21 \\ \infty \\ 1 \\ 77 \\ 38 \\ 99 \\ 92 \end{pmatrix}
\end{aligned}
$$

Notice that, the resulting vector now gives the shortest distances (within single edge) to source vertex 4 for vertices 0,1,3,5,6 and 7. Notice also that, we compute the distance to vertex 4 from itself as 77! In order to handle this situation, we refine the operation as below:

$R_{i,1} = \min\left(D_{i,1}, \min_{k=1}^{n} (M_{i,k} + D_{k,1})\right)$

Now, this operation will give the vector R as $\begin{pmatrix} 19 & 21 & \infty & 1 & 0 & 38 & 99 & 92 \end{pmatrix}^T$. If we repeat this operation by our new distances by assigning R to $D^{(1)}$, we get:

$$
\begin{aligned}
R &= A^T \cdot D^{(1)} \\
&= \begin{pmatrix}
28 & \infty & \infty & \infty & 19 & \infty & \infty & \infty \\
\infty & \infty & \infty & 28 & 21 & \infty & \infty & \infty \\
94 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
11 & \infty & \infty & 90 & 1 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 77 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 38 & \infty & \infty & 81 \\
78 & \infty & \infty & \infty & 99 & \infty & 80 & 63 \\
\infty & \infty & \infty & \infty & 92 & \infty & \infty & \infty
\end{pmatrix}
\begin{pmatrix} 19 \\ 21 \\ \infty \\ 1 \\ 0 \\ 38 \\ 99 \\ 92 \end{pmatrix} \\
&= \begin{pmatrix} 19 \\ 21 \\ 113 \\ 1 \\ 0 \\ 38 \\ 97 \\ 92 \end{pmatrix}
\end{aligned}
$$

Notice that, now we got the distance from vertex 4 to vertex 2 as 113 (19+94, through vertex 0) and we updated the distance from vertex 4 to vertex 0 as 97 instead of 99 once we use the path through vertex 6. If we do this iteration for $|V| - 1$ times, we are guaranteed to get the all shortest paths distances for given source vertex. This operation is very similar to Bellman-Ford algorithm to compute the shortest paths but just conveyed in a different way. For details on graph algorithms being interpreted as linear algebra operations, one can refer to [3] and Chapter 25 at [1]. One advantage of associating graph algorithms into linear algebra operations is to ease parallelization and that is what we are doing in this project.

3 Graph Representation

Adjacency matrix representation of graphs is good for visualization and understanding the concepts. Yet, this representation is not very efficient considering that real-life graphs are sparse (number of edges are much less than $|V| \times |V|$). Thus, Compressed Sparse Row (CSR), introduced in [2], is used in practice. In CSR representation, we have basically two arrays: *rows* for storing rows(vertices) and *columns* for storing non-zero column entries(edges) of a sparse matrix. An index i in *rows* array point to the location of its non-zero column entries in adjacency Matrix A until the next pointer in the *rows* array. CSR representation of the graph in Figure 1 is given in Figure 3.

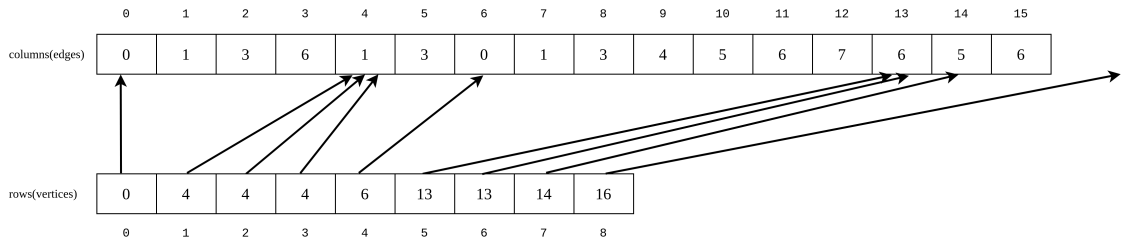

Figure 3: CSR Representation of the sample graph given in Figure 1

Let $off1 = rows[i]$ and $off2 = rows[i+1]$ for index i that represent a vertex i. The values in the range of $columns[off1]$ and $columns[off2]$ are the out-neighbors for the vertex i. For example, vertex 0 has $off1 = 0$ and $off2 = 4$ such that elements from $columns[0]$ to $columns[4]$ gives the out-neighbors of vertex 0, which are vertices 0,1,3, and 6. Notice that, if $off1 = off2$ as in the case for vertex 1, it means that there is no outgoing edge from vertex 1. Also notice that, although we have only 8 vertices(indexed from 0 to 7), we have one extra index in the *rows* array that points to the one plus last index of *columns* array. The symmetric of CSR is called Compressed Sparse Column(CSC) is used when we are concerned for incoming edges of vertices. A third array(*values* or *weights*) parallel to *columns* array is mostly used for non-zero values of the matrix A. The non-zero values are often referred as weights or edge attributes in graph context. CSC representation

along with *weights* array is given in Figure 4 for the same sample graph above. Moreover, a fourth array(*data*) parallel to *rows* is also often used for vertex attributes in the graph.

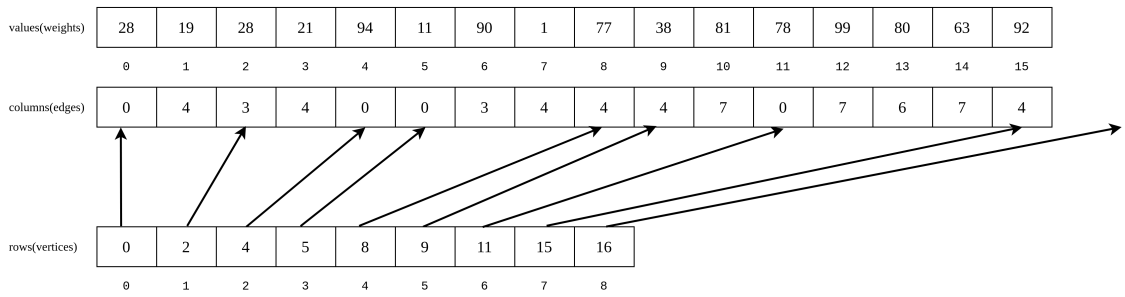

Figure 4: CSC Representation of the sample graph given in Figure 1

Notice that when we represent an adjacency matrix in CSR, its transpose is equivalent to CSC representation.

4 Serial Implementation

The first part of the project is to implement the iterative SSSP algorithm as described in Section 2. We are going to use CSC format to represent the graph, as what we need is transpose of adjacency matrix. An algorithm for SSSP with graph in CSC representation is given in Algorithm 1.

Algorithm 1: SSSP Algorithm as SpMV

Input : G=($rows$,$columns$,$weights$), sourceVertex
Output: D

Let V be the set of vertices in G
Let D and R be an array of size $|V|$, all values are initialized to ∞
$D[sourceVertex] \leftarrow 0$
repeat $|V| - 1$ **times**
  for *each vertex index i in rows* **do**
    $R[i] = \min\left(D[i], \min_{rows[i] \leq k < rows[i+1]} \left(weights[k] + D[columns[k]]\right)\right)$
  end
  $D \leftarrow R$
end

An optimization can be done to terminate the execution early. If none of the distance values change in an iteration, it means it won't change in the next iterations as well. Thus we don't need to repeat for $|V| - 1$ times. This optimization is very crucial, because that much of repetition is rarely needed in practice. You should add the early termination technique to your implementation.

Your program should take an input file name, source vertex index, and output file name as arguments. The shortest distances to the given source vertex should be written to the output file. The input file should be an ASCII text file, whose first line gives the number of vertices(row array size) m, the second line gives the number of edges(columns array size) n, the next $(m + 1)$ lines gives the rows values, the next n lines gives the columns values, and the very next n lines gives the weights values where each value separated by a new line. The output file will be an ASCII text file as well, consisting of m (number of vertices) lines, line i gives the shortest distance value from source vertex to vertex i. Sample inputs and sample outputs are given within this document for the examples above. The name of the program should be `sssp_serial` Following illustrates how your program should take the arguments for source vertex 4:

```
# ./sssp_serial ANY_INPUT_FILENAME.txt 4 ANY_OUTPUT_FILENAME.txt
```

For reading the input file and outputting the results, you may use the `util.h` and `util.c` files we provided.

5 Parallel Implementation

Parallel version of SSSP facilitates the SpMV structure of the algorithm using MPI library. We will use 1-D block distribution of vertices, edges, and weights given in CSC representation. In this partitioning scheme, master process reads the input file and then scatters the vertices, their incoming edges and weights to other processes. As a result, each processes will be responsible for a subset of

vertices and their in-neighbors. The number of vertices that each process responsible should be as equal as possible and master process itself is also responsible for a set of vertices. Figure 5 gives an illustration of this scatter operation among 4 processors for the sample graph given above. Notice that the number of edges that each processor gets might be different from each other; P0 and P1 gets 4 edges, P2 gets 3 edges, P3 gets 5 edges. With this partitioning, each process should have a partial CSC representation. Remember that *rows* array has an additional index that array points to one plus last index of *columns* array. By this logic, we should have an extra index for each partial *local_rows* arrays in the processors. In the example, we have 8 vertices and 4 processors and we should infer that each processor should be responsible of 2 vertices. Yet, we should add an extra index on each local *local_rows* arrays of processors that point to an index value outside their *local_columns* array. This last index value should be equal to $local\_rows[0] + length(local\_columns)$.

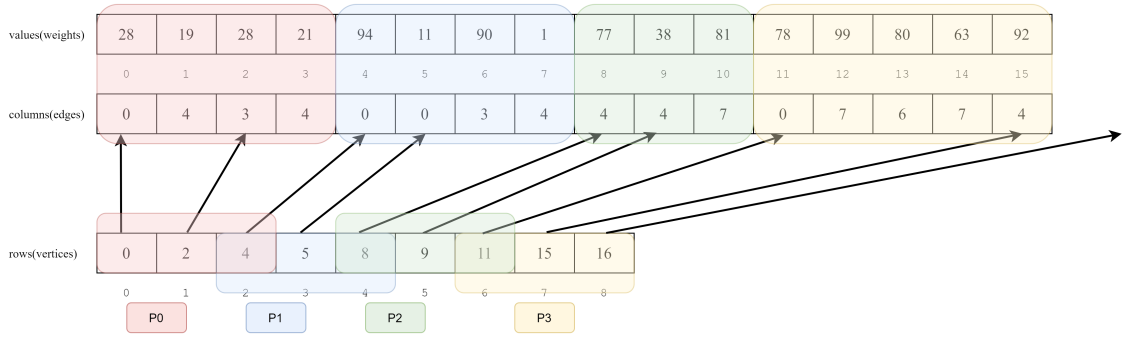

Figure 5: Partitioning the CSC Representation of the sample graph given in Figure 1

Once the 1-D partitioning is completed, we can advance to running the algorithm. Each processors will be responsible for iterative computation of the shortest distances in the set of vertices they have. In doing so, a global distance vector D of size $|V|$ should be initialized in each processor as in the serial algorithm where only $D[sourceVertex]$ equals to 0. For each iteration, each processor computes a local distance vector R of size $|V|/p$ where p is the number of processors. R gives the resulting distance values of the a processor's vertices at the current iteration. Before moving on to next iteration, we should use all-gather operation to update the global distance vector D. In this operation each processor sends their local R vectors to all other processors and receives the local R vectors of all other processors with placing them onto the global distance vector D. You should use `MPI_Allgather` operation to implement this[1]. **You will lose points if you don't utilize `MPI_Allgather` operation**.

You should also implement the early termination logic for the parallel part. Each processor decides whether their local distance values are changed or not. If at least one processor has different local distance values from the previous iteration, the execution should continue.

After the last iteration, the shortest values should be now computed and gathered on global distance vector of each processor. Master processor can print this resulting vector to a file.

Arguments of the parallel program should be the same as in the serial version. The input/output file formats are also the same. The name of the program should be `sssp_parallel`. Following illustrates how should your program take the arguments, for source vertex 4.

```
# ./sssp_parallel ANY_INPUT_FILENAME.txt 4 ANY_OUTPUT_FILENAME.txt
```

6 Assumptions

- You may assume that number of vertices in the graph are divisible by the number of processors with no remainder.
- You may assume that all weights in the graph have positive values, though the algorithm should also work with negative weight values if no negative cycle exists.
- Use INT32_MAX to represent ∞. If final shortest distance for a vertex is ∞, you should print INT32_MAX.
- We use directed graphs.

[1] See https://www.rookiehpc.com/mpi/docs/mpi_allgather.php for reference

7 Hints

- Be careful in adding an integer value to INT32_MAX, as it causes arithmetic overflows!
- Try to use collective operations in partitioning the graph for better performance in parallel version. Specifically, try to use `MPI_Scatter` and its variations such as `MPI_IScatter` and `MPI_Scatterv`.
- `MPI_Allreduce` operation might be useful for implementing the early termination logic in parallel version.
- You should start to see speed-ups for graphs having vertices more than 2^{20}.

8 Report

You should write a short report(around 4 pages). **Reports should be in .pdf format**, submissions with wrong format will get 0. Your report should include:

- *System Specifications.* Put the following at the start of your report:
  - Operating System:
  - CPU:
  - Number of Logical Cores[2]:
  - RAM:
  - HDD/SSD:
  - Compiler:
- *Implementation Details.* Briefly explain your implementations. Explain what data structures you used, which MPI calls you utilized etc.
- *Analysis.* Try to answers questions such as; What are your expected results? How much speed-up can we get asymptotically? What is the communication cost in this program?
- *Experiments.* Plot graphs for showing performance of your programs. Experiment with various the inputs. You should give the speed-up values for each experiment. Also try with sourceVertex values that reach more amount of vertices in the graph so that execution continues for a considerable number of iterations. For inputs, you may use real-life graphs or generated graphs. There is a nice list of real-life graphs at http://snap.stanford.edu/data/index.html. These graphs are mostly given in edge-list format so we shared a python script to convert those graphs into CSC format as well as adding some random weight values to these edges. For generating graphs, you may use the code at https://github.com/RapidsAtHKUST/Graph500KroneckerGraphGenerator. Try to experiment with as large data as possible.
- *Discussion.* Discuss your results that answers questions such as; Does parallel implementation improve the performance on any inputs? If not, what might be the reasons? You may have noticed that partitioning the graph like we did might cause some processors have much larger number of edges than other processors and causing load imbalance. This is related to the *Power Law Distribution* of real-life graphs. What are other methods of partitioning the graph and what might be alternative parallel implementations for SSSP. You may refer to research papers.

9 Grading

- Serial Implementation: 20 points
- Parallel Implementation: 50 points
- Report: 30 points

[2] `lscpu` command on Linux systems gives this information in the output at the line starting with "CPU(s):".

10 Submission

Send a single zip file **(yourname_lastname_p2.zip)** that includes:

- Your implementation with source files with following naming convention and any additional source files:
  - `sssp_serial.c` or `sssp_serial.cpp`
  - `sssp_parallel.c` or `sssp_parallel.cpp`
- A Makefile that generates the following 2 executables: [3]
  - `sssp_serial`
  - `sssp_parallel`
- Your report

Submit to the respective assignment on Moodle.
No Late Submission Allowed!

References

[1] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[2] Fred G. Gustavson. "Some Basic Techniques for Solving Sparse Systems of Linear Equations". In: *Sparse Matrices and their Applications: Proceedings of a Symposium on Sparse Matrices and Their Applications, held September 9–10, 1971, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, the National Science Foundation, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Donald J. Rose and Ralph A. Willoughby. Boston, MA: Springer US, 1972, pp. 41–52. ISBN: 978-1-4615-8675-3. DOI: 10.1007/978-1-4615-8675-3_4. URL: https://doi.org/10.1007/978-1-4615-8675-3_4.

[3] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Ed. by Jeremy Kepner and John Gilbert. Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/1.9780898719918. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719918. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898719918.

[3] We will execute the make command in the root directory of your project folder, the executables should be generated in this folder as well.