A report on

PARALLEL SHORTEST PATH ALGORITHM USING MPI

Done by

Unmesh N Joshi
VUnet ID uji300

Department of Computer Science
Vrije University
Amsterdam, The Netherlands

Winter Semester

Abstract

There are many algorithms to find all pair shortest path of a road network. In this report, parallel version of Floyd Warshall's algorithm is discussed considering one dimensional row-wise decomposition of the adjacency matrix. The algorithm is implemented using MPI (C bindings). Experiments showed that parallel version is effective for large graphs sizes.

Contents

List of Figures

Chapter 1

Objective

The goal of this assignment is to write parallel implementation of shortest path algorithm for a real road networks using MPI and C. Calculation of total road distance and diameter of a road network are other requirements.

1.1 About Tool

A Readme file is provided with the source code for general use. Here is some extra information:

- The tool assumes that value of infinity is 10000.
- Buffer for printing path for one vertex to other is 4K.
- Output for distances only is adjacency matrix which is printed entirely in the standard format
- Output for paths is given in the recommended format (requires optional argument p).

Chapter 2

Work Done

2.1 Algorithm

Several algorithms are studied for solving the assignment problem of all-pair shortest path. Johnson's algorithm and Floyd Warshall's algorithm were two choices. Johnson's algorithm internally uses Dijkstra's algorithm and Bellman Ford algorithm. [1] is a noteworthy and interesting read on the real road network algorithms. Pallottino's two queue algorithm is ranked first in this paper of Three fastest algorithms on real road networks. It is surprising and somewhat disappointing that there is no mention of Floyd Warshall's algorithm in the paper. However, for this practical assignment Floyd Warshall's algorithm is chosen because of its simplicity. It also works for directed edges. The simple parallel Floyd algorithm is based on a one-dimensional, rowwise domain decomposition of the intermediate matrix I and the output matrix S . Notice that this means the algorithm can use at most N processors. Each task has one or more adjacent rows of I and is responsible for performing computation on those rows. [4]

The basic idea is shown in the figure below:

```
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if local row then
                Broadcast it using TAG
            else
                Receive row with TAG
                Calculate minimum
            end if
        end for
    end for
end for
```

Figure 2.1: Parallel ASP Algorithm

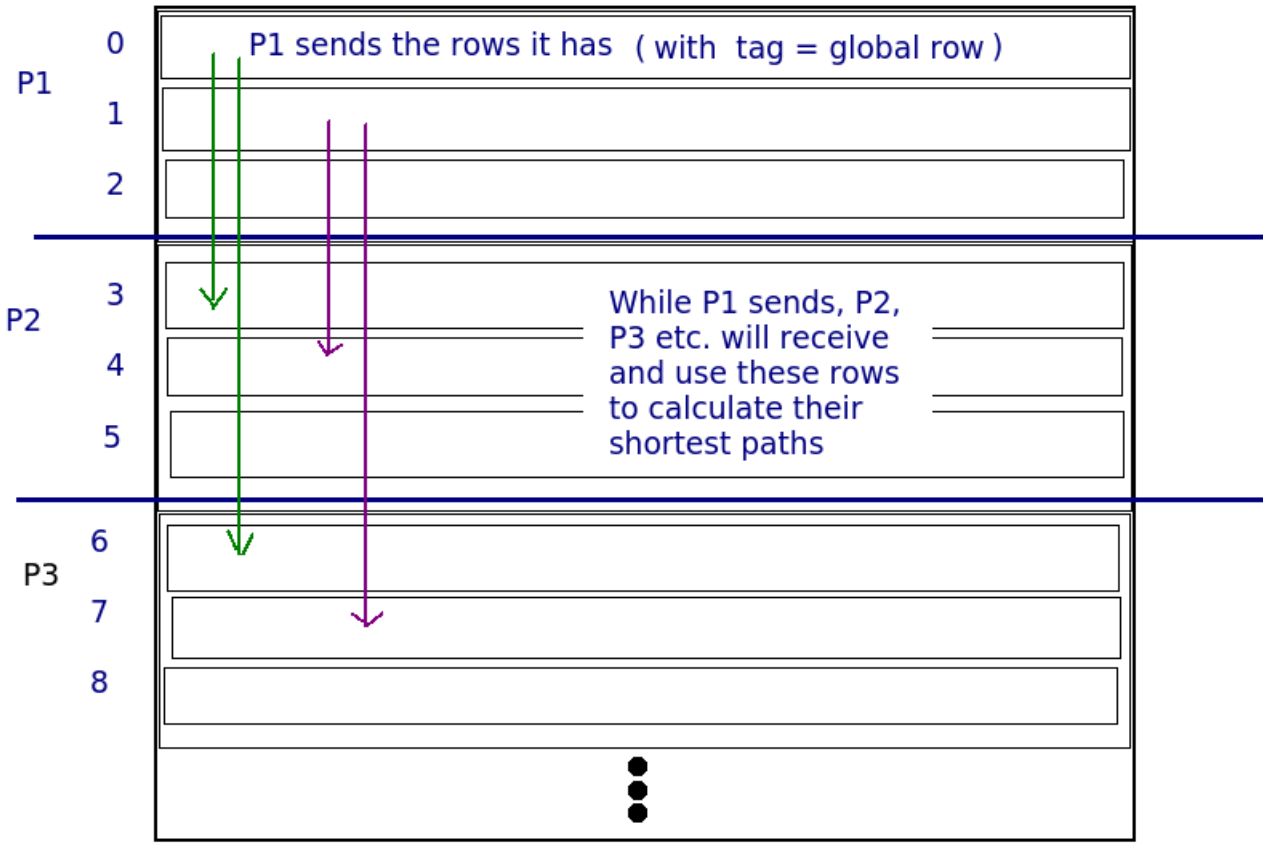

Figure 2.2: ASP logic

2.1.1 Path construction

To learn about entire path from one place to another, an easy modification in algorithm is suggested [3]. We build a so called *Next* matrix which contains *Next*[i][j] represents the highest index vertex one must travel through if one intends to take the shortest path from i to j. Note that this does not incur time for distributing the work. Each node will locally allocate and process the next matrix. However, master node must at the end receive the next matrix in order to make it useful for printing.

2.2 Implementation Highlights

Rank 0 works as master process. It distributes (almost evenly) the rows among available processors. If two processors are available, the job is ultimately done by one processor. This simplifies the code; however it makes master process idle until all processors finish their job. At the end master process will collect the rows of output matrix and *Next* matrix from these processors. The algorithm is generalized so that it works for both directed and undirected edges. No extra optimizations are applied so that undirected graphs are processed faster than directed ones.
This imposes following restriction:

- Symmetric nature of adjacency matrix of undirected graph cannot be exploited. It incurs redundant work for certain vertices.
- Entire matrix has to be printed (as opposed to triangular matrix)
- Not suitable for *partial updates* as explained later

Some benefits of structure of adjacency matrix are:

- Simplicity
- Better load balancing in case of directed graphs

For undirected graphs, following figure shows the possible optimization. This can work well, because implementation runs in three nested for loops and if the bounds of any of them can be lessened, it will result in good performance.

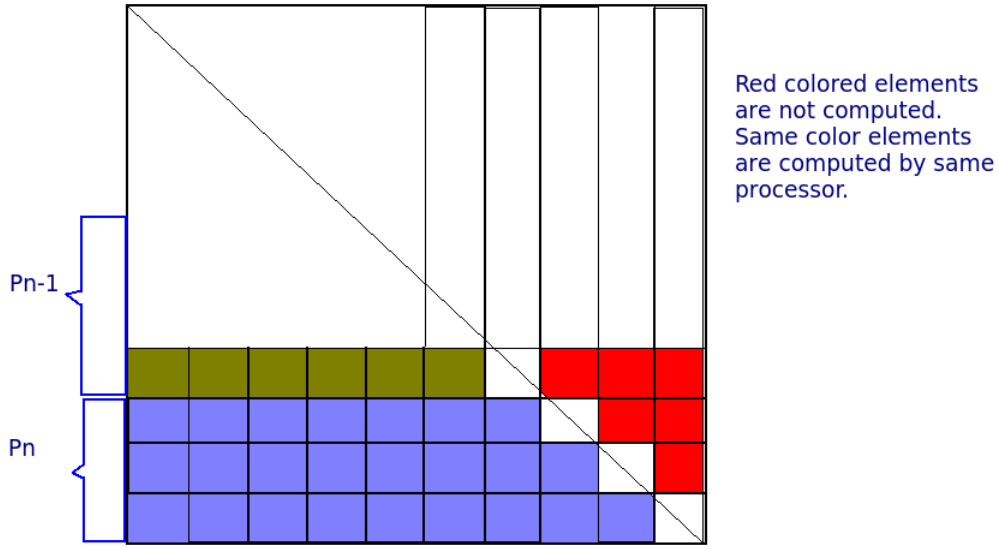

Figure 2.3: Triangular optimization for undirected graph

2.3 Communication and Computation

Computation for every node will be $O(N^3)$. For every iteration, each CPU sends P-1 messages with N integers and does N/P X N comparisons. Thus, communication/computation ratio will be

$$Ratio = \frac{P(P-1)}{N^2} \tag{2.1}$$

If N >>P, then this ration can be smaller. For any parallel algorithm, we are doing good, if we are doing more computation than communication. Performance break-down for implemented algorithm can be found in the experiments section.

2.3.1 Improving communication

When processor with rank 0 distributes work, it sends block of rows (rectangular matrix) to each processor available. Note that road network graphs are generally sparse in nature. Each row will have more *infinity* values or *zero* values than non-zero edge weights. Thus it makes sense to send only the edges which have non-zero eight weights. This will save initial start up

time. However, at the end, when processor with rank 0, collects the results, it must receive entire block of rows, because then it would be a dense matrix.

Rank 0 sends edge-list to all worker processors Each processor builds local adjacency sub-matrix to work with Do parallel algorithm Each processor sends its adjacency sub-matrix to Rank 0

2.3.2 Tree structured communication

[4] says that communication for Floyd algorithm can be performed in log (P) steps using tree structure.

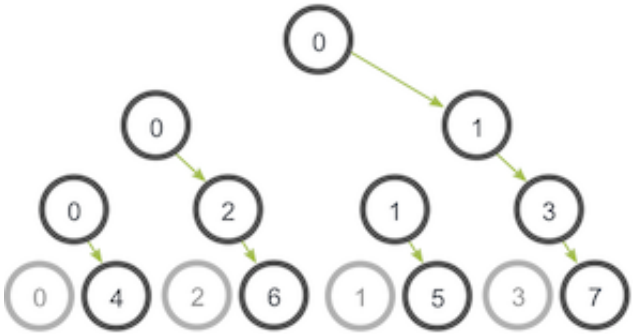

Figure 2.4: Tree structure of communication

2.4 MPI features and some tweaks

For total distance calculation and diameter calculation global reduction function called MPI_ Reduce() was particularly useful. However, for general communication, MPI imposes some limitations. One notable is that it cannot handle sending and receiving of multi-dimensional arrays. It was instructive exercise to write allocation and deallocation routines for matrix structure which is used to solve the algorithm. It helps to bind multiple rows into one message. Thus we can send *more* data in *less* number of messages.

2.5 Performance testing and Experiments

To test the parallel implementation graphs given for benchmarking were used. In addition some interesting experiments are conducted on graphs having more than one thousand vertices. This section contains a choice selection from the conducted experiments:

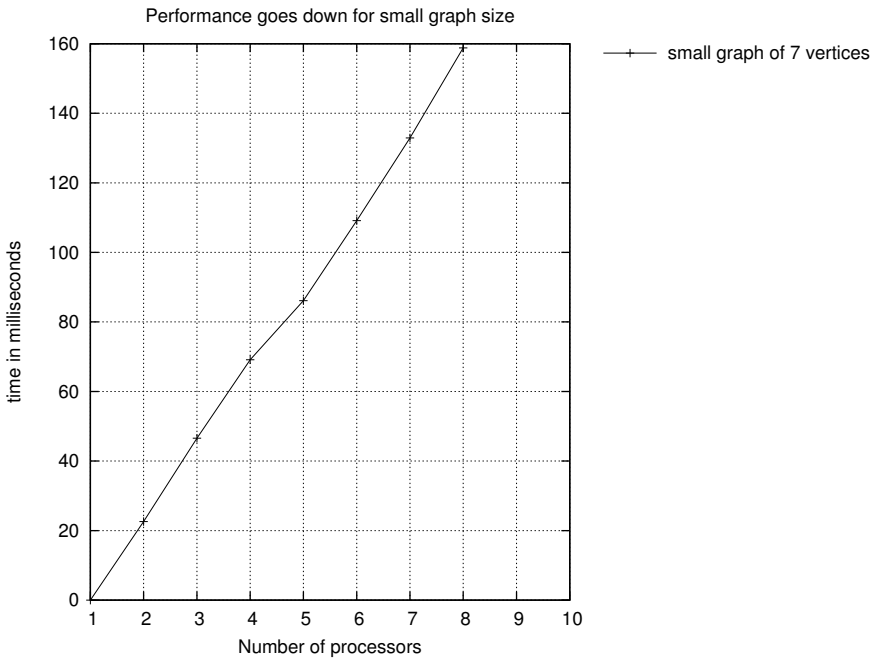

Figure 2.5: Performance for Small graph

This experiment showed that parallel algorithms should be used to solve large problems. Following analogy holds here: There is group of ten friends. They are in Amsterdam for study-tour. One of them can remember the map of VU; but remembering the map of Amsterdam can be distributed among all of them.

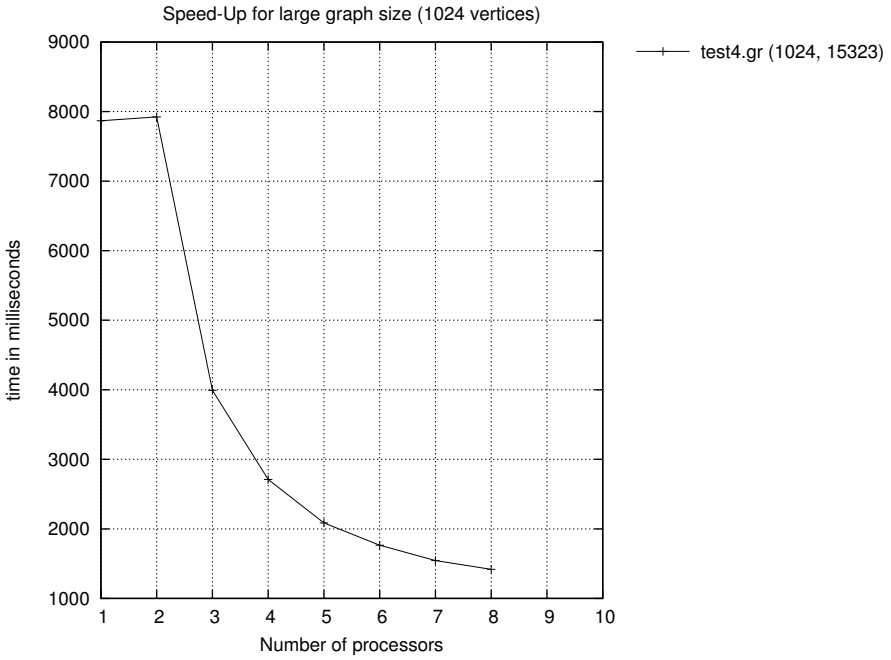

Figure 2.6: Performance for large graph

This graph shows how parallel algorithm gives good timing results when number of processors are increased. However, observe that at the right end of graph, difference is not logarithmic as it was towards the left end. This is because, as we have more number of processors, they have more messages to exchange. Thus performance will be dictated by communication.

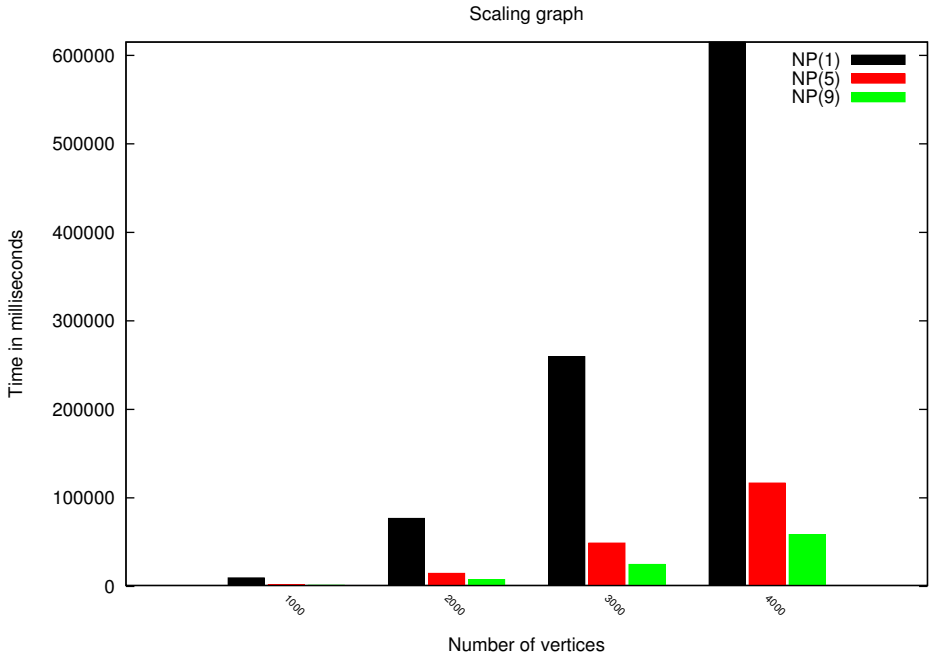

Figure 2.7: Comparison with large graphs

Graphs with thousands of vertices were tested on parallel implementation and it was observed that they gain respectable speed-up as compared to sequential version. The histogram tells it all.

To break down performance, use of MPI_ Reduce() function is made. Each processor calculates its own communication and computation time. At the end, we take maximum of all of them. Adding the time would not be sensible, because processors are working in parallel environment.

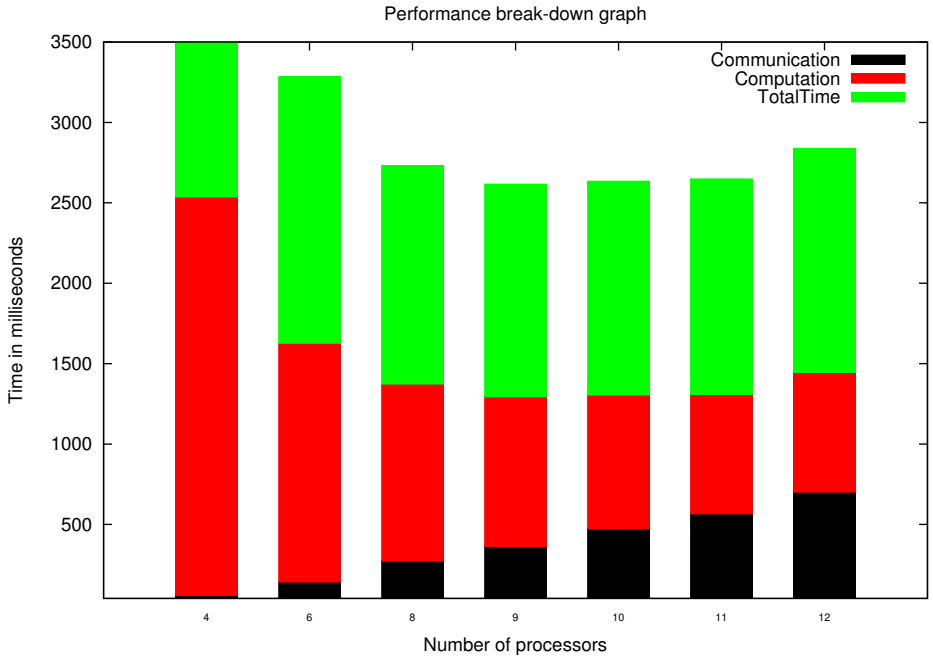

Figure 2.8: Performance break-down for graph of 1000 vertices

This graph corroborates what we have observed in figure 2.6.

2.6 Conclusions

In parallel implementation of Floyd Warshall's algorithm bottleneck occurs as we increase number of processors, as communication time dominates the computation time.

Chapter 3

Future Work

There are lot of things which are *on paper* for this assignment. It was interesting enough topic to be a good food for thought. Following is the *can be done* list of things:

Building shortest path trees for every vertex at each processor. For example, a processor responsible for N vertices, will store N shortest path trees. This tree will be n-ary tree and can be logically implemented in form of arc lists and node lists.

3.1 Suggested solution for Partial Updates

It is assumed that partial updates will be given as input to the program by means of file. The file will contain list of updated edges *only*. In order to update fast, we must store output of graph on a stable storage (a file). Partial update file must contain identifier for previous graph or we will calculate the update on most recent graph. Here we digress for a moment.
If we are given a map of city which is one year old, it is possible some places have different names, but it is not possible that some places are entirely gone from the city. Of course, there may be several alternative ways available to reach the same place. But important point to note is that the place just cannot be erased from the city-map.
When we are given a partial update. Edges will be deleted and not the vertices. In this case, adjacency matrix structure has got serious limitations. Also *Next* matrix stores only the next vertex and not the edges. It is difficult to determine how many vertices have to be changed in the *Next* matrix and in the adjacency matrix as well.
One possible solution is to update the given edges with their new weights and send only those rows to all other processors. Then wait for any possible

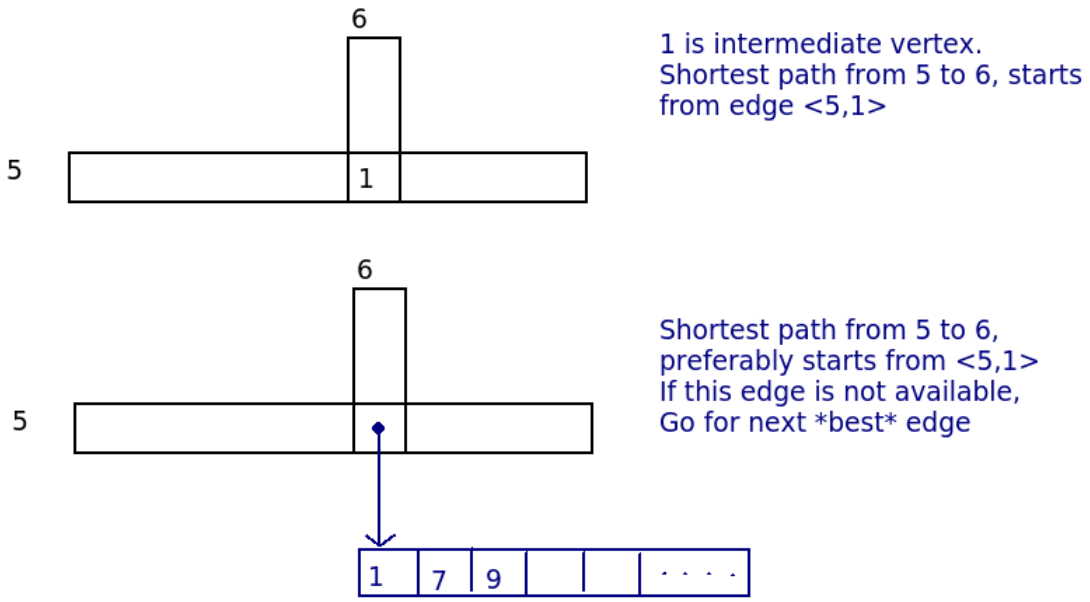

Figure 3.1: Priority list of intermediate vertices in Next Matrix

updates from other processors. The pseudo-code might look like following:

Send all affected rows
while not done **do** MPI_ Iprobe(MPI_ ANY_ SOURCE,...& flag) ▷ Probe for messages
while flag **do** MPI_ Receive() ▷ Receive the updated row Process row Probe again
end while
end while

However, a better approach is to maintain another dimension for *Next* matrix and store priority-wise, the list of intermediate vertices as shown in the figure below:

When edge weights are increased, this list will remain unchanged. However, if edge weight is decreased, then we have to recompute this list, to allow shorter route.

References

[1] F. Benjamin Zhan, Three fastest algorithms for real road networks: Data structures and procedures, Journal of Geographic information and Decision Analysis, vol.1, no.1, pp.70-82,

[2] MPI: A Message Passing Interface Standard, Programming manual,

[3] `http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`

[4] I. Foster, Designing and Building Parallel Programs , Addison-Wesley, 1995