Optimizing Edge List to Compressed Sparse Row Representation Conversion with Big Data Sets

Authors: Ulysses Butler, Joseph Li *{uab@truman.edu, josephli@wustl.edu}*
Advisors: Dr. Jeremy Buhler, Dr. Roger Chamberlain *{jbuhler@wustl.edu, roger@wustl.edu}*

This research is supported in part by NSF grants 1527510, 1619639, 1763503, and 1852343.

Introduction

Data preprocessing is becoming increasingly important as the demand for data handling rises. Many algorithms are designed to work with input in a specific format. Unfortunately, real-world data comes in all different shapes and sizes. For this reason, it can often be worthwhile to convert data from one format to another.

Graphs are commonly represented as adjacency matrices. This works well for small graphs, but as the number of vertices gets larger, the size of the matrix grows quadratically. For large, sparse graphs, other representations are often superior for conserving storage space. For this project, we set out to create an algorithm, that converts one such form to another. Given an array of tuples representing edges (and an optional weight array in parallel), how can we output the equivalent compressed sparse row representation of the graph?

For this project, we wanted to develop an algorithm that would follow the specifications of the Graph500 benchmarking suite. We designed it to work even when the data set is too large to fit into main memory, as is the case in the Graph500 specification. This means we can only deal with relatively small chunks of data at a time. In working towards this goal, we split the problem into stages summarized below.

```
Matrix:        Edgelist:      Presort:
0 1 0 6        <1, 0> 5       <0, 1> 5
5 0 0 0   ->   <3, 1> 3   ->  <1, 3> 3   ->
0 0 0 0        <0, 1> 1       <0, 1> 1
0 3 0 0        <0, 3> 6       <0, 3> 6

Sort:          Postsort:      CSR:
<0, 1> 1       <0, 1> 1       A:  {1, 6, 3}
<0, 1> 5  ->   <0, 3> 6  ->   IA: {0, 2, 3, 3, 3}
<0, 3> 6       <1, 3> 3       JA: {1, 3, 3}
<1, 3> 3
```

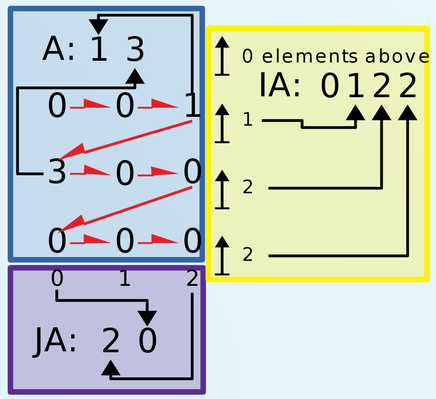
Compressed Sparse Row

Compressed sparse row is a general format for encoding information about matrices. In our case, we're storing the information normally encoded in an adjacency matrix. Instead of storing one 2D array, we use three 1D arrays.

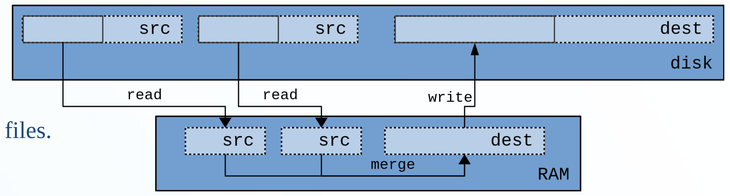
A stores the weight values as they are discovered in row-major order (shown with the red arrows).

IA stores the number of elements located above a given row.

JA stores the column numbers of the weights stored in **A**.



Sequential Algorithm

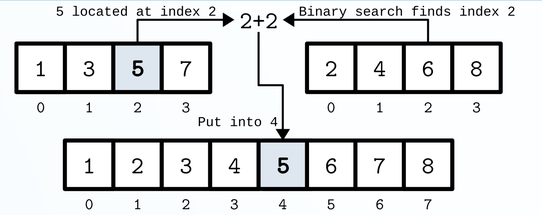


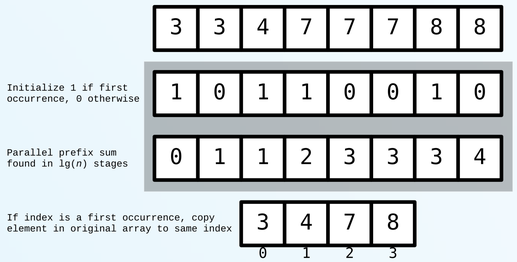
1. **Presort**
   - Since the graph is undirected, we list the smaller vertex first.
   - This ensures duplicate edges are adjacent to one another.
2. **Partition and Sort**
   - Since the edge list is too large to fit into memory, we split it into smaller files.
   - Each part is then then sorted using a library sort.
3. **Merge**
   - The sorted files are then merged (seen at right). Three buffers are created – one for each source and one for the destination.
   - These buffers are then merged. If a source buffer is depleted, new information is read. If the destination is full, it is written to the disk.
4. **Postsort**
   - Duplicates and self-loops are removed from the array.
5. **Convert to CSR**
   - This can be done in linear time. Weights are written into **A** as they're found, with their columns written into **JA**.
   - For **IA**, a counter tallies the number of edges considered. If the row of the current edge differs from the last, the value of the counter is written to **IA**.

Parallel Algorithm

The **presort** can trivially be done in parallel. Each thread handles a single element, allowing us to do many elements simultaneously.

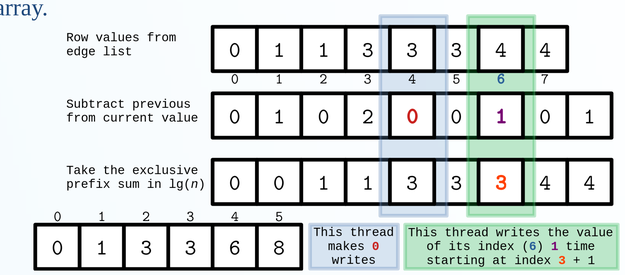
For the **merge**, if each thread is responsible for one element, each element can find its own place in the destination array (seen at right). The entire merge can be done using a binary search in lg(*n*) time.





The **postsort** can also be done in lg(*n*) time by taking advantage of the parallel prefix sum algorithm (seen at left).

The resulting array will be a parallel array to the original storing the value of the indexes for the new array. For example, since the first instance of "7" corresponds to "2" in the final array, that will be the index that "7" is placed at in the final array.



Finally, we have the **CSR conversion**. **A** and **JA** are trivially parallel. Each thread can write a weight and column into the output arrays at the same index at which they were found.

To find **IA**, we create a parallel array that is one element larger. This last element will always be "1", while every other element is computed as the previous value subtracted from the current one.

We then create a third array from an exclusive prefix sum of the previous array. Every thread then writes its index a number of times determined by the second array, starting at the index given by adding one to the value in the third array. This fills in all values except the initial 0, which can be added by the host.

Future Work

As GPU hardware improves, we hope to look into sorting these edge structures on the GPU as opposed to the CPU and to see whether the effort needed to perform the required data transfers is worthwhile. We are also interested in seeing how the parallel algorithm will work on other computing devices like FPGAs.

Acknowledgments

We would like to thank Dr. Ron Cytron, Steven Harris, Clayton Faber, and the rest of the Buhler and Chamberlain labs for their generous advice and support on this project.