ECE/CSC 506/406: Architecture of Parallel Computers Project-1

Accelerating Sorting in Graph Processing Adjacency Lists (OpenMP, MPI, and Hybrid versions)

Due: Monday, October 4th, 2021

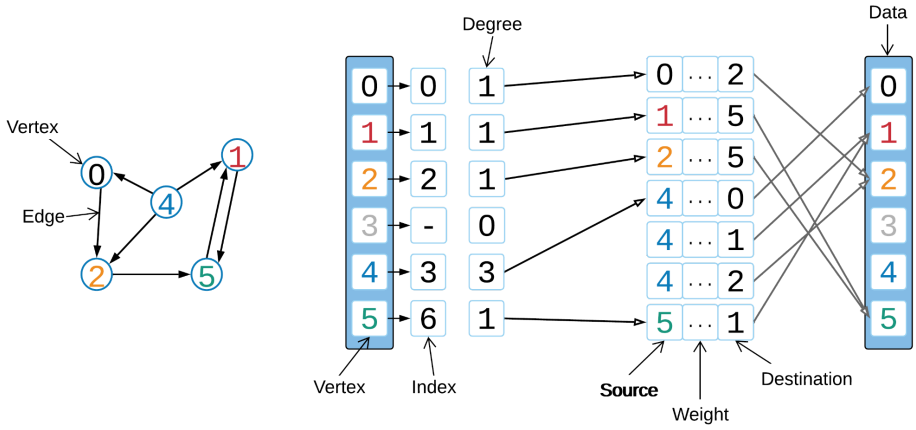

Figure :1 *Graphs are represented in memory as Compressed Sparse Row (CSR) structures, sorting is a fundamental step in representing compact graphs in memory*

Motivation

Graph processing is widely used to solve big data problems in numerous domains, such as social networks, protein interactions in bioinformatics, and web graph hierarchies. The cost of pre-processing (sorting) and building the graph Compressed Sparse Row (CSR) data structure often dominates the algorithm end-to-end execution time.

This calls into question the benefits of algorithmic optimization that depends on extensive pre-processing or CSR in general. For example, fast breadth-first search (BFS) algorithms, rely on a Compressed Sparse Row (CSR) representation of the graph. The processes of reading the graph list into a CSR representation and forming the adjacency list usually take up to 90% of the execution time!

In this project, you will work on three versions of parallelizing the pre-processing stage for building the graph Compressed Sparse Row (CSR) which is predominated by the sorting step. Utilizing Radix sort, an OpenMP, MPI, and Hybrid (OpenMP + MPI) versions will test our understanding of various concepts in shared memory parallelism,

message passing, and others. Provided with a framework that sorts and builds a CSR with a baseline model using count sort.

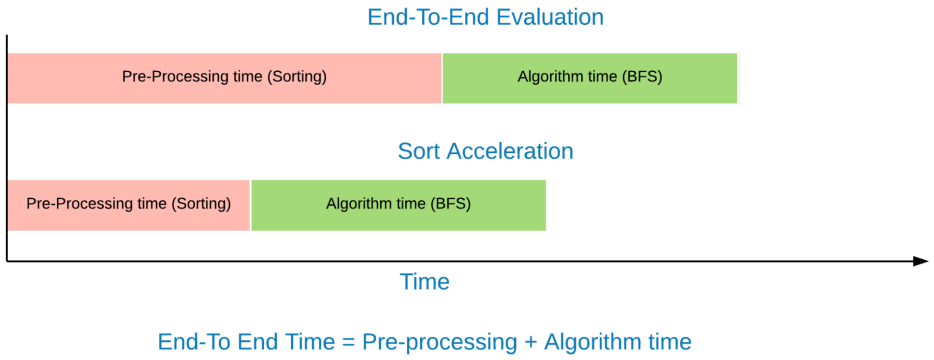

Figure:2 *Accelerating the pre-processing step, which is predominated by sorting time is the goal of Project-1*

To recap in this programming assignment, you will be focusing on parallelizing the pre-processing step (radix sorting) of Graph processing. i.e. you will be accelerating the building of the graph adjacency list as shown in Figure 1.

Memory-Efficient Adjacency List Implementation Technique With Count Sort (Baseline)

Count sort and radix sort will be explained in class and provided with extra slides.

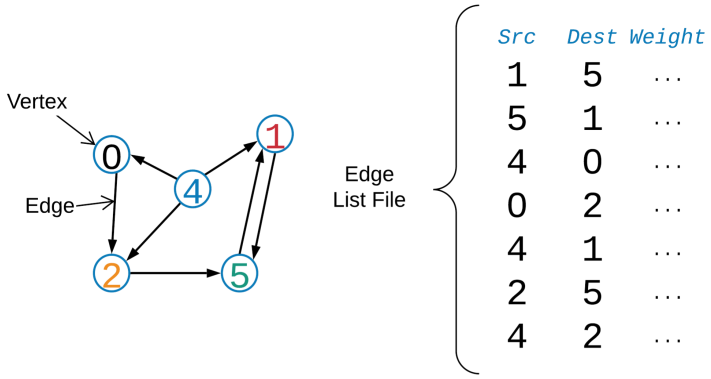

Figure:3 *Graphs are often stored as an unsorted edge list on file (SSD, HardDisk), this allows for simpler transfer and manipulation (adding more edges to the graph)*

Let us assume as shown in figure 4, that array edges are already populated with edges in memory, and the variable num_edges indicates the number of edges for the current instance of the problem.

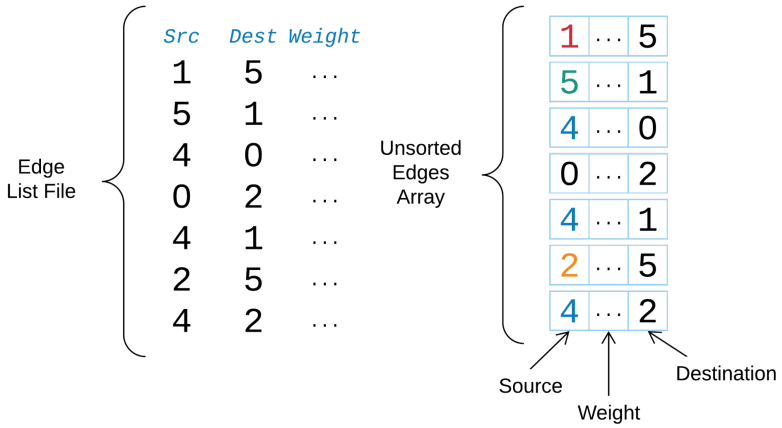

Figure: 4 *The unsorted edge list on file (SSD, HardDisk), is loaded to memory to construct the Graph CSR structure.*

Now, we can sort an array of edges by the identifier of a source vertex as shown in figure 5. It is possible to sort an array with O(E + V) runtime complexity using the Counting Sort/Radix sort.

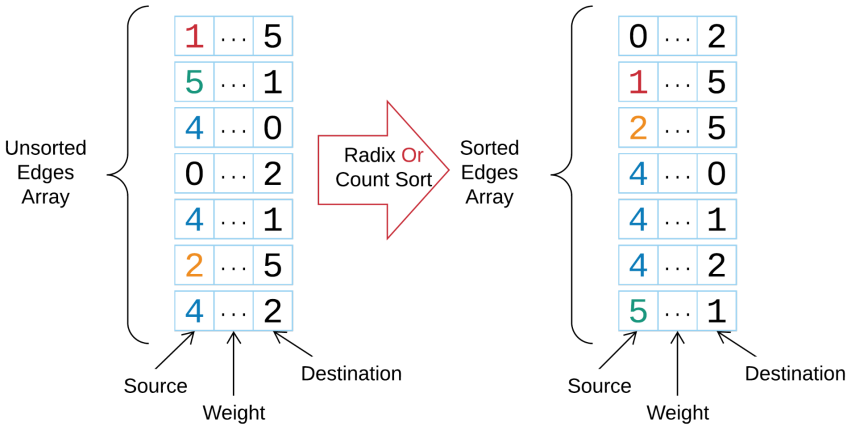

Figure: 5 *Sorting the edge list according to the source vertex would group the adjacent destination nodes, building a CSR representation of the graph.*

The described approach can be used to process many different graphs by reusing the same arrays which have been allocated only once - at the start of the program. Hence, the program operates within a constant amount of allocated memory.

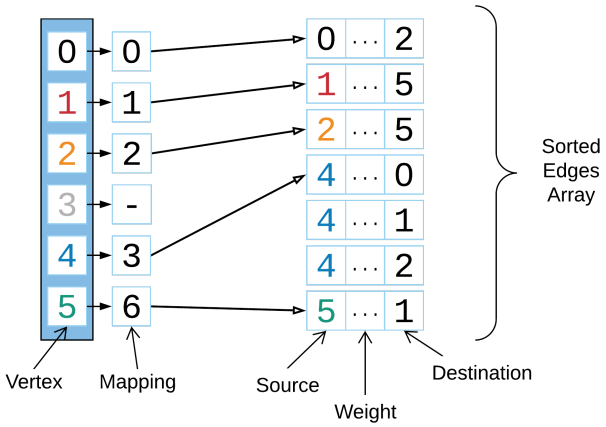

Figure: 6 *The final step of mapping the vertices with the starting index to point to the sorted edge list.*

Additional observation: each bucket of outgoing edges, which belong to the same vertex, can be sorted by the identifier of the destination vertex. In this case, we will be able to check the presence of an edge between any pair of vertices with O(log(E)) runtime complexity (using Binary Search).

Of course, the described approach is suitable for static graphs, as the addition of new edges to a packed array of sorted edges is costly -- requires rebuilding the CSR structure.

Memory-Efficient Adjacency List Implementation Technique With Radix Sort

Count sort and radix sort will be explained in class and provided with extra slides.

“Radix Sort is a non-comparative sorting algorithm with asymptotic complexity O(nd). It is one of the most efficient and fastest linear sorting algorithms. Radix sort was developed to sort large integers. As an integer is treated as a string of digits so we can also call it a string sorting algorithm” [1].

Radix sort orders the elements based on the last digit (least significant digit). Then the result is again sorted by the second digit, continue this process for all digits until we reach the most significant digit. We use counting sort to sort elements of every digit, so time complexity is O(nd).

- Radix Sort is a linear sorting algorithm.
- The time complexity of Radix Sort is O(nd), where n is the size of the array and d is the number of digits in the largest number.
- It is not an in-place sorting algorithm as it requires extra additional space.
- Radix Sort is a stable sort as the relative order of elements with equal values is maintained.
- Radix sort can be slower than other sorting algorithms like merge sort and quick sort if the operations are not efficient enough. These operations include insert and delete functions of the sub-list and the process of isolating the digits we want.
- Radix sort is less flexible than other sorts as it depends on the digits or letters. Radix sort needs to be rewritten if the type of data is changed.

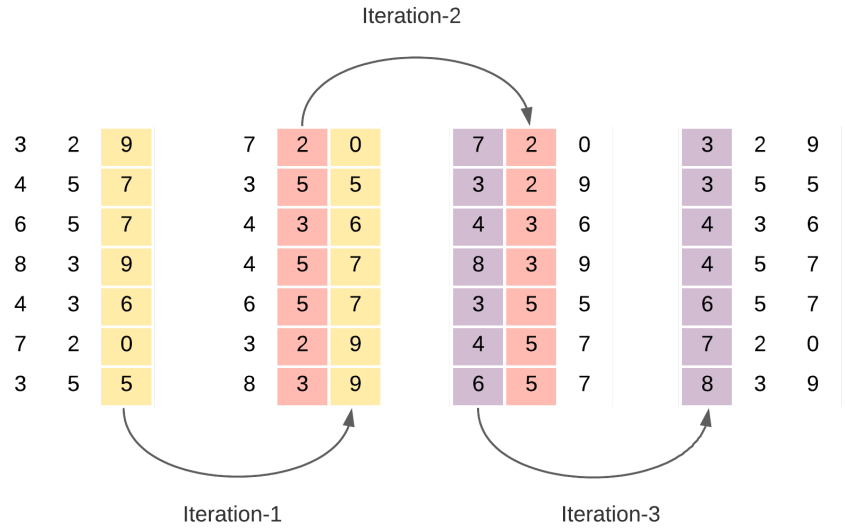

Figure: 7 *Radix sort uses multiple iterations of count sort*

In Figure 7:

- For 1st Iteration: Sort the array on basis of the least significant digit (1s place) using counting sort
- For 2nd Iteration: Sort the array on basis of the next digit (10s place) using counting sort
- For 3rd Iteration: Sort the array on basis of the most significant digit (100s place) using counting sort

In this algorithm running time depends on the intermediate sorting algorithm which is counting sort. If the range of digits is from 1 to k, then counting sort time complexity is O(n+k). There are d passes i.e counting sort is called d time, so total time complexity is O(nd+nk) =O(nd). As k=O(n) and d is constant, so radix sort runs in linear time.

To recap in this programming assignment, you will be focusing on parallelizing the radix sort. For each iteration, we will be using count sort as a sub iteration of radix sort.

Radix Sort Implementation Hints:

- Start with parallelizing count sort
- For each count sort iteration do not use the decimal value, use radix of 8-bit
- To calculate the radix assume the src is 32-bit integer and sorting by 8-bit radix (nbit) -- (32/8 =4) we need 4 iterations of count sort
- Use bitwise shifting to obtain 8 bit radix ((src >> (radix * nbit)) & 0xff)
- Use "#pragma omp parallel" to encapsulate different stages of parallel count sort
- Use "#pragma omp barrier" to synchronize between count sort parallel steps

Report

- (506/406) For the OpenMP part.
    - investigate the effect of varying the edge list benchmarks (links provided later by the TA) and the number of threads
    - Rationalized why radix sort is performing better than count sort for larger/smaller data sets (caching/coherence)
    - Include LDG/ITG analysis for each loop stage in count/radix sort
    - Speedup/Performance as num of processors is increasing
    - Speedup/Performance with different Samplings (4,8,16 bit)
- (506/406) For the MPI version
    - Sorting local edge list for each processor -- speedup
    - Processor communication overhead
- (506 only) Hybrid OpenMP/MPI
    - Sorting local edge list for each processor -- speedup
    - Processor communication overhead
- (506 only)
    - Comparison between all aforementioned approaches

Grading

20%: Your code compiles successfully

40%: Your output matches exactly for runs on all (provided 5 benchmarks)(points will be equally distributed).

40%: Report. Credit will be given on the statistics shown and the discussion presented.

Submission Format

In order to grade all submissions promptly, we have to ask you to follow the submission format.

Your final submission should be in a zip file named "unityid1_project1.zip". Your Unity ID is the one generated from your name, with alphabets and sometimes digits at the end. Do NOT use your campus card ID or your email alias.

Your zip file should only contain the following files:

- /include/sort.h ← your parallel/serial radix sort function header here
- / src /sort.c ← your parallel/serial radix sort body here
- / src /main.c ← your parallel/serial radix sort called here

Do not include the parent folder in your zip file "solved by cd into the directory". Also, no need to include the input and output files. This command should help you generate the zip file:

zip -r unityid1_project1.zip ./*

Project Code (GitHub):

In this project, you are provided a code with sample datasets to test the correctness of your sorting algorithm implementation. We will be providing bigger test benches for you to test the performance improvements.

Obtain the source code of the project from this link

- https://github.ncsu.edu/atmughra/ECE-506-406-Projects
- The code runs on any Linux machine (Used Ubuntu preferably)

To access NCSU GitHub each NCSU student has a github.ncsu.edu account that corresponds to their UnityID.

Benchmark Format:

- Edge lists are provided in textual format and they are formatted as source → destination separated by a tab. *Src → Dest*

Src	dest
30	3
6	11

Initial Setup:

- After cloning the repository. enter the directory
  - cd Project-1/code
- compile the code
  - make
- run the code
  - make run

Debugging

For debugging your output, you can use this function to print the edge array (small samples).

void printEdgeArray(struct Edge *edgeArray, int numOfEdges);

if you encounter a segmentation fault you can use "$make debug" and it will run your binary with gdb.

Organization:

- Project-1 - Project 1
  - code - base code for the problem
    - bin- binaries are generated in this folder
    - datasets - sample data sets for the code
    - include - [.h] header files to be included in the code
      - bfs.h - push/pull breadth-first search implementation for testing CSR correctness

        - graph.h - graph abstraction contains edge lists and inverse edge-lists with their mappings
        - bitmap.h - bitmap implementation not multi-thread safe
        - arrayQueue.h - simple array-based queue
        - edgelist.h
        - sort.h - count/radix sort implementation to be parallelized
        - timer.h
        - vertex.h
    - obj - [.o] object files generated here for linking
    - src - source code resides in this directory
        - bfs.c - push/pull breadth-first search implementation for testing CSR correctness
        - graph.c - graph abstraction contains edge-lists and inverse edge-lists with their mappings
        - bitmap.c - bitmap implementation not multi-thread safe
        - arrayQueue.c - simple array based queue not multi-thread safe
        - edgelist.c
        - sort.c - count/radix sort implementation to be parallelized
        - timer.c
        - vertex.c
        - main.c - parallel/serial count/radix sort called here
    - Makefile - a file that compiles the base code
- doc - Documents describes the Machine Problem

References

1. https://www.codingeek.com/algorithms/radix-sort-explanation-pseudocode-and-implementation/
2. https://www.geeksforgeeks.org/counting-sort/
3. http://lagodiuk.github.io/computer_science/2016/12/19/efficient_adjacency_lists_in_c.html#the-commonly-used-approaches
4. *2001-5 Erik D. Demaine and Charles E. Leiserson*
5. Everything you always wanted to know about multicore graph processing but were afraid to ask, USENIX Annual Technical Conference, Jul 12th -14th 2017 , Santa Clara, CA ,USA
6. Shin-Jae Lee, Minsoo Jeon, and Dongseung Kim, Partitioned Parallel Radix Sort, 2002: http://ebookbrowse.com/2002-09-partitioned-parallel-radix-sort-pdf-d54143025