ECE/CSC 506: Architecture of Parallel Computers Machine Problem (MP-1)

“Stage one Serial, OpenMP”

Accelerating Graph “memory-efficient” Adjacency Lists (Serial, OpenMP)

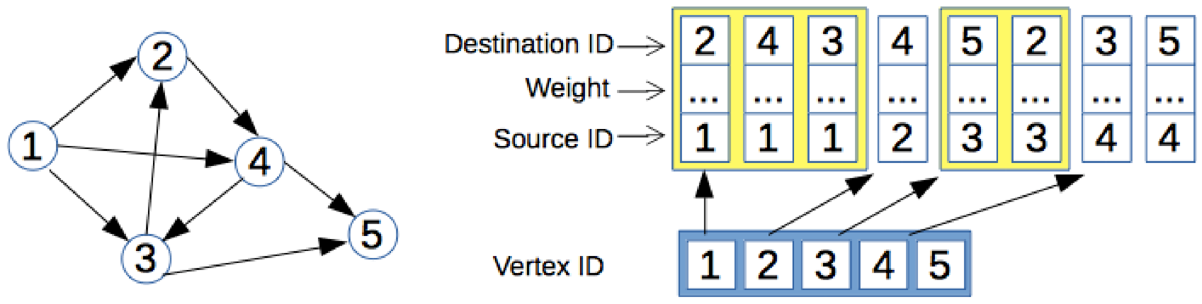


Figure 1: optimal graph representation using Compressed Sparse Row (CSR) data structure

Motivation

Graph processing is widely used to solve big data problems in multiple domains, such as social networks, protein interactions in bioinformatics and web graph hierarchies. The cost of pre-processing (building the graph data structure) in many cases dominates the algorithm execution step. This calls into question the benefit of proposed algorithmic optimization that relies on extensive preprocessing. For example, fast breadth-first search (BFS) algorithms, rely on 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 takes up to 90% of the execution time!

In this Project you will work on two versions of this preprocessing stage for building Compressed Sparse Row (CSR), using Radix sort, a serial version, and an OpenMP version. Provided with a framework that sorts and builds a CSR with a baseline model using count sort.

Many graph problems have special limitations, which are known in advance:

- Graph is sparse, making it difficult and memory inefficient to use matrix representation for processing it.
- Number of Vertices does not exceed *|V|*
- Amount of Edges does not exceed *|E|*
- Graph is static (consists of a fixed sequence of nodes and edges)

The need of efficient implementation usually leads to additional limitations, such as:

- Avoiding dynamic memory manipulations
- Allocation of all required memory at once (on startup of the program)

- Limited memory for loading large graph, partitioning the graph.

For recap in this programming assignment you will be focusing on the preprocessing step of Graph processing optimization. i.e. you will be building an adjacency list using OpenMP as shown in figure 1. named CSR.

The commonly used approaches

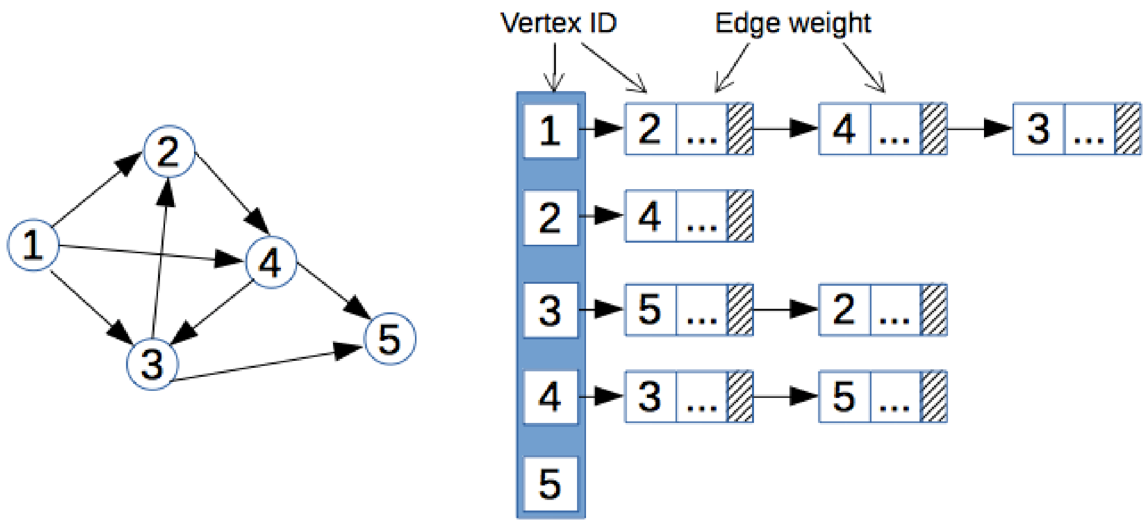


Figure 2: Using linked lists to build a graph is slow and memory inefficient

The quick analysis of various sources (e.g.: Wikipedia, Stackoverflow, CLRS (the book of: Cormen, Leiserson, Rivest, Stein))showed that it is usually advised to use the linked list based implementation of adjacency lists. So usually, the most straightforward implementation involves the usage of pointers.

However, the problem with choosing such implementation is traversing linked lists is not efficient in terms of performance. Furthermore, the memory requirements for such implementation is usually larger than compact arrays.

The proposed approach

Without loss of generality let's assume that each vertex is associated with some unique integer identifier from the interval: [0, |V| - 1]. As far as it is known in advance, that the maximal amount of edges is |E| - at the startup of the program, we can allocate an array of size |E|, which can handle all edges of the graph.

Let's define a structure, which represents edges of the graph:

```
struct Edge{
   int src;            // id of a source vertex
   int dest;    // id of a destination vertex
};
```

```
struct Vertex{
   int edges_idx; // index of the first outgoing edge
};
```

In our main program we will preload the edge list with data from a graph list file:

```
// get |v| |e| count do we can allocate our edge array and vertex array
   loadEdgeArrayInfo(fname, &numOfVertices, &numOfEdges);
   printf("Edges : %d Vertices: %d\n", numOfEdges, numOfVertices);

   // allocate our edge array and vertex array
   struct Edge * edgeArray = newEdgeArray(numOfEdges);
   struct Edge * sortedEdgeArray = newEdgeArray(numOfEdges);
   struct Vertex * vertexArray =  newVertexArray(numOfVertices);

   // populate the edge array from file
   loadEdgeArray(fname, edgeArray);
```

Then we will sort the edge list as represented in figure.1 and figure .3.

```
   Start(timer);
   countSortEdgesBySource(sortedEdgeArray, edgeArray, numOfVertices, numOfEdges);
   Stop(timer);
   printMessageWithtime("Time Sorting (Seconds)",Seconds(timer));
```

Memory-efficient adjacency list implementation technique with *counting sort*

Let's assume, that an array of edges is already populated , and the variable numOfEdges indicates the amount of edges for the current instance of the problem. Now, we can sort the array by source vertex with O(E + V) runtime complexity using the Counting Sort.

The described approach can be used for processing of many different graphs by reusing the same arrays which been allocated only once - at the start of the program. Hence, we can be sure that the program operates within the constant amount of allocated memory.

Additional observation: each bucket of outgoing edges, which belong to the same vertex, can be sorted by 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, described approach is well suitable for problems, which require the computations on static graphs, because the addition of new edges to the packed array of sorted edges is costly.

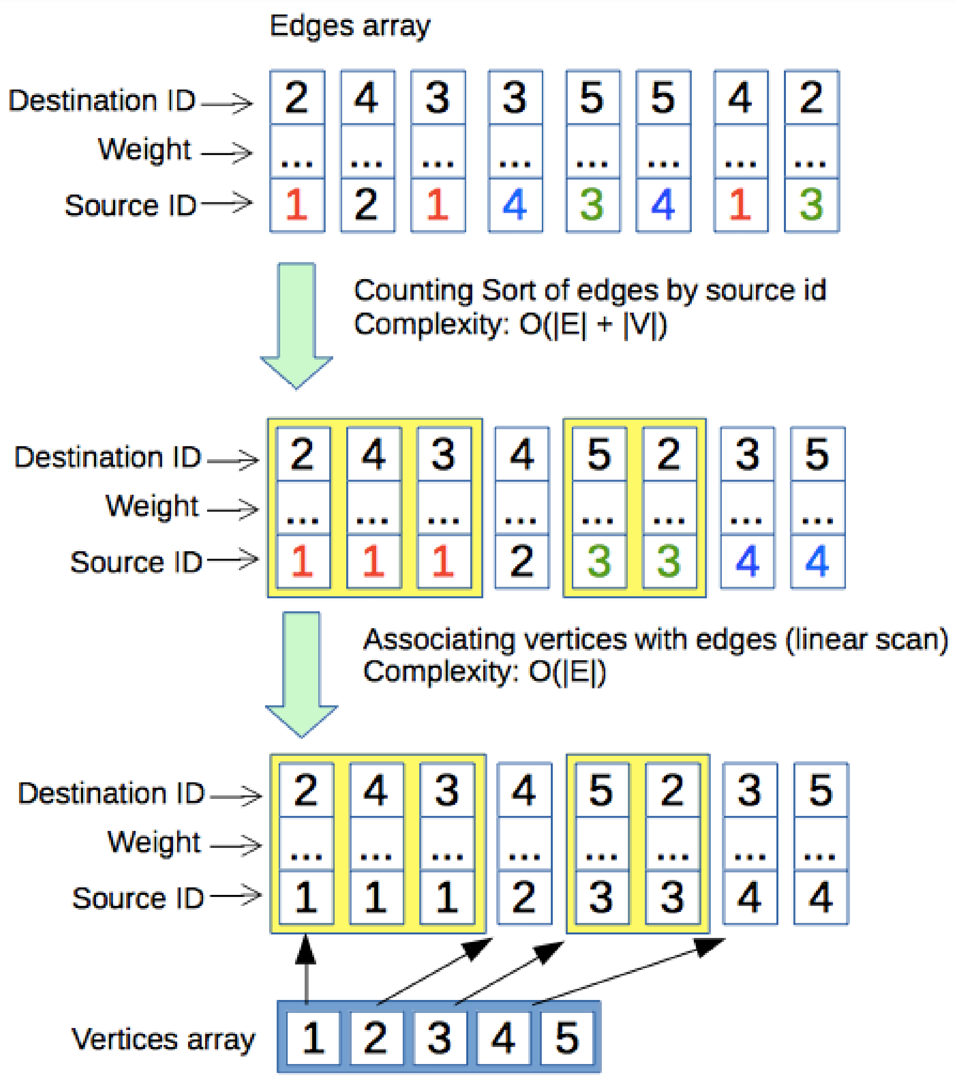


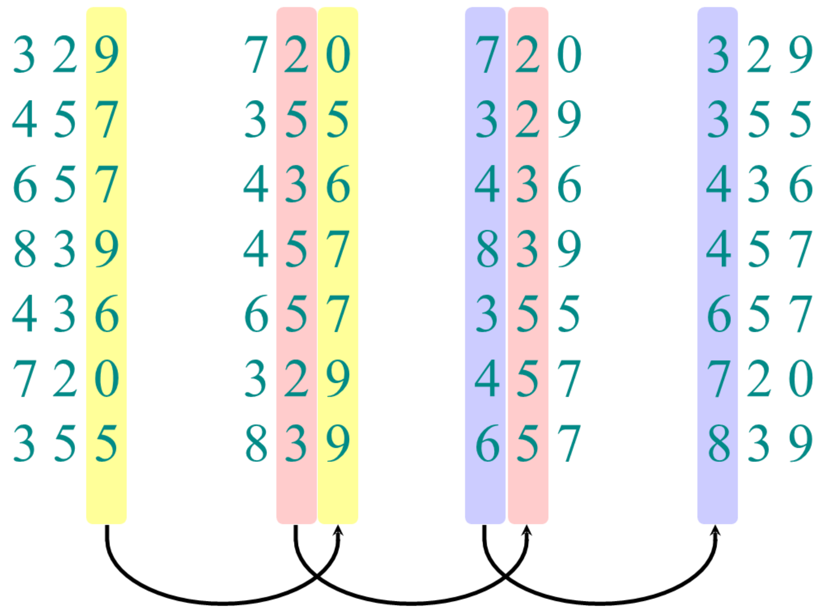
Figure 3: Three steps of build CSR representation of a graph, read edge list, sort, then map vertices to the sortedd edge list.

Memory-efficient adjacency list implementation technique with *Radix Sort*

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 integer is treated as a string of digits so we can also call it as string sorting algorithm.

In radix sort, we first sort the elements based on last digit (least significant digit). Then the result is again sorted by second digit, continue this process for all digits until we reach 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.
- Time complexity of Radix Sort is O(nd), where n is the size of 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 stable sort as 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 letter. Radix sort needs to be rewritten if the type of data is changed.



In the above example:

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

In this algorithm running time depends on 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.

Note: radix sort parameters should be the same as count sort when implemented in the framework

```
Start(timer);

radixSortEdgesBySource(sortedEdgeArray, edgeArray, numOfVertices, numOfEdges);

Stop(timer);

printMessageWithtime("Time Sorting (Seconds)",Seconds(timer));
```

PSEUDOCODE OF RADIX SORT

```
adix-Sort(A, d)
//It works same as counting sort for d number of passes.
//Each key in A[1..n] is a d-digit integer.
//(Digits are numbered 1 to d from right to left.)
    for j = 1 to d do
        //A[]-- Initial Array to Sort
        int count[10] = {0};
        //Store the count of "keys" in count[]
        //key- it is number at digit place j
        for i = 0 to n do
         count[key of(A[i]) in pass j]++

        for k = 1 to 10 do
         count[k] = count[k] + count[k-1]

        //Build the resulting array by checking
        //new position of A[i] from count[k]
        for i = n-1 downto 0 do
         result[ count[key of(A[i])] ] = A[j]
         count[key of(A[i])]--

        //Now main array A[] contains sorted numbers
        //according to current digit place
        for i=0 to n do
          A[i] = result[i]

    end for(j)
end func
```

Running the base code

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

Benchmark format

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

Src	dest
30	3
6	11

Note: To read the benchmark just change the fname variable to point to the file you want to sort.

```
// const char * fname = "./datasets/test/test.txt";

// const char * fname = "./datasets/wiki-vote/wiki-Vote.txt";

const char * fname = "./datasets/facebook/facebook_combined.txt";
```

The framework skeleton

- /bin *hold the executable
- /datasets *holds the edgelist benchmarks you need to sort
- /include *hold the *.h files
    - /include/bfs.h *a simple breadth first search implementation
    - /include/edgelist.h
    - /include/sort.h
    - /include/timer.h
    - /include/vertex.h
- /obj *holds the *.o files you make
- /src *holds the main with the source files you need
    - / src /bfs.c
    - / src /edgelist.c
    - / src /sort.c
    - / src /timer.c
    - / src /vertex.c
    - / src /main.c

First run

After downloading the files on your machine.

Unzip the folder and access it, notice the steps as illustrated on the command prompt Figure.4.

You will get three outputs:

- first is the time your sorting algorithm took (0.000418s),
- second of discovered nodes (3490) using the generated CSR.
- third the time BFS (0.000694) took.

"Don't be deceived by this benchmark, it is small and doesn't show the sorting as a bottleneck in fact count sort performs better than radix sort when it comes to small range of numbers. However, we will see how both scale with larger benchmarks and how well they perform when parallelized."

For debugging purposes, the first serial implementation for count sort is provided, it is considered you golden reference.

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.

```
~/ECE506-P1$ make
making main <- main.o
making main <- edgelist.o
making main <- sort.o
making main <- vertex.o
making main <- timer.o
making main <- bfs.o
linking main <- main.o edgelist.o sort.o
~/ECE506-P1$ make run
linking main <- main.o edgelist.o sort.o
./bin/main
Edges : 88234 Vertices: 4039
 -----------------------------------------------------
| Time Sorting (Seconds)                              |
 -----------------------------------------------------
| 0.000418                                            |
 -----------------------------------------------------
 -----------------------------------------------------
| 3490                                                |
 -----------------------------------------------------
 -----------------------------------------------------
| Time BFS (Seconds)                                  |
 -----------------------------------------------------
| 0.000694                                            |
 -----------------------------------------------------
```

Figure 4:First run

Report

For the OpenMP version, investigate the effect of varying the edge list benchmarks (links provided later by the TA) and the number of threads. Graph the results and explain what you see.

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 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_unityid2_program2.zip”.if you are not working as a team, just name the file “unityid_program2.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:

The framework skeleton

- /bin *hold the executable
- /datasets *holds the edgelist benchmarks you need to sort
- /include *hold the *.h files
    - /include/bfs.h *a simple breadth first search implementation
    - /include/edgelist.h
    - /include/sort.h ← your parallel/serial radix sort function header here
    - /include/timer.h
    - /include/vertex.h
- /obj *holds the *.o files you make
- /src *holds the main with the source files you need
    - / src /bfs.c
    - / src /edgelist.c
    - / src /sort.c ← your parallel/serial radix sort body definition here
    - / src /timer.c
    - / src /vertex.c
    - / 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_unityid2_program2.zip ./*
```

Not following the submission format property will result in a maximum of 5 points penalty.

Suggestions

- Read the main and superclasses carefully, and understand how the program works
- Most of the code given to you is well encapsulated, so you do not have to modify most of the existing functions. You just need to define the incomplete functions Commented

- Understand how the different OpenCL APIs handle memory-allocation errors and out-of-bounds references.
- Make sure there are no memory leaks in your program by de-allocating memory after you are done with it.
- Most of the code given to you is well encapsulated, so you do not have to modify most of the existing functions. You just need to complete the definition of the incomplete functions.
- There will be occasional downtime with ARC. Please start working on the assignment at the earliest and not wait until the last minute.

References

- https://www.codingeek.com/algorithms/radix-sort-explanation-pseudocode-and-implementation/
- https://www.geeksforgeeks.org/counting-sort/
- http://lagodiuk.github.io/computer_science/2016/12/19/efficient_adjacency_lists_in_c.html#the-commonly-used-approaches
- *2001-5 Erik D. Demaine and Charles E. Leiserson*
- 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