

AN EFFICIENT CONNECTED COMPONENTS ALGORITHM
FOR MASSIVELY-PARALLEL DEVICES

by

Jayadharini Jaiganesh

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2017

Committee Members:

Martin Burtcher, Chair

Apan Qasem

Vangelis Metsis

COPYRIGHT

by

Jayadharini Jaiganesh

2017

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgement. Use of this material for financial gain without the author's express written permission is not allowed.

Duplication Permission

As the copyright holder of this work I, Jayadharini Jaiganesh, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

DEDICATION

To my mentor, thank you for your guidance and motivation. To my loving husband for his unwavering support.

ACKNOWLEDGEMENTS

This project is supported by the National Science Foundation (NSF grant #1406304) and equipment donations from Nvidia.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	v
LIST OF FIGURES	viii
LIST OF ABBREVIATIONS	x
ABSTRACT	xi
CHAPTER	
1. INTRODUCTION	1
1.1 Connected Components	1
1.2 General Serial Connected Components Algorithm	2
1.3 General Parallel Connected Components Algorithm	4
1.4 Contributions.....	4
1.5 Outline	5
2. BACKGROUND STUDY	7
2.1 Parallel Implementations of CC.....	7
3. RELATED WORK.....	15
4. ECL ALGORITHM AND IMPLEMENTATIONS	17
4.1 ECL Base Algorithm	17
4.2 ECL - GPU Implementation	20
4.3 ECL - Parallel CPU Implementation	23
4.4 ECL - Serial CPU Implementation	23
4.5 ECL _{af} - Atomic Free CC Implementation	23
5. EVALUATION METHODOLOGY	26
5.1 GPU and CPU Machines	26
5.2 Input Graphs Format and Specifications	27

5.3 Compiler Information	30
6. RESULTS	31
6.1 Configurations.....	31
6.2 Comparison with Parallel GPU Benchmarks	31
6.3 Comparison with Parallel CPU Benchmarks.....	34
6.4 Comparison with Serial CPU Benchmarks.....	36
6.5 Performance Comparison across different Configurations.....	38
6.6 Performance Comparison - Atomic-Free Implementations.....	41
7. SUMMARY	45
7.1 Summary.....	45
7.2 Future Work.....	46
APPENDIX SECTION.....	47
LITERATURE CITED	54

LIST OF FIGURES

Figure	Page
1.1. Connected Components in a Graph	2
2.1. Graph before Hooking	7
2.2. Graph after Hooking	8
2.3. Graph after Pointer Jumping	8
4.1. Double-sided worklist	21
5.1. Graph	29
5.2. Compressed Adjacency List Representation	30
6.1. Parallel GPU Benchmarks Slowdowns relative to ECL on the Titan X.....	32
6.2. Parallel GPU Benchmarks Slowdowns relative to ECL on the K40	33
6.3. Parallel CPU Benchmarks Slowdowns relative to ECL on the Zurich	34
6.4. Parallel CPU Benchmarks Slowdowns relative to ECL on the Denver	35
6.5. Serial CPU Benchmarks Slowdowns relative to ECL on the Zurich	36
6.6. Serial CPU Benchmarks Slowdowns relative to ECL on the Denver	37
6.7. Geometric-Mean Slowdowns of CC Implementations on different systems.....	38
6.8. Throughput (Edges/s) of various CC Implementations on different systems.....	39
6.9. Throughput (Nodes/s) of various CC Implementations on different systems	40
6.10. Parallel GPU Benchmarks Slowdowns relative to ECL _{af} on the Titan X	41
6.11. Parallel GPU Benchmarks Slowdowns relative to ECL _{af} on the K40	42
6.12. Parallel CPU Benchmarks Slowdowns relative to ECL _{af} on the Zurich	43

6.13. Parallel CPU Benchmarks Slowdowns relative to ECL_{af} on the Denver	44
--	----

LIST OF ABBREVIATIONS

Abbreviation	Description
BFS	Breadth First Search
BFSCC	Breadth First Search for Connected Components
CC	Connected Components
DFS	Depth First Search
GPU	Graphic Processing Unit
LSG	Lonestar GPU
LS	Lonestar CPU

ABSTRACT

Massively-parallel devices such as GPUs are best suited for accelerating regular algorithms. Since the memory access patterns and control flow of irregular algorithms are data dependent, such programs are more difficult to parallelize in general and a direct parallelization may not yield good performance, on GPUs in particular. However, by carefully studying the underlying problem, it may be possible to derive new algorithms that are more suitable for massively-parallel accelerators. This thesis involves studying and analyzing such an irregular algorithm, called Connected Components, and proposes an efficient algorithm, called ECL, which is faster than the existing CC algorithms on most tested inputs. Though atomic operations are fast, they can represent a bottleneck as these operations run serially and might hinder performance in the future parallel devices. This thesis also proposes a synchronous and atomic-free algorithm, called ECL_{af}, whose performance is comparable to the fastest existing CC algorithms.

CHAPTER 1

INTRODUCTION

Finding the Connected Components (CC) is a key preprocessing step in many graph algorithms. It is used in real-world applications such as navigation, image segmentation, and in the medical field. Thus, a faster connected components algorithm and implementation has the potential to improve many important graph processing codes.

1.1 Connected Components

For an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a connected component C is a subset of V such that all the vertices belonging to C are reachable from any vertex in C , and there are no edges between vertices belonging to different components. The connected components problem is to find the number of such components present in a given graph, assign a unique ID to each component, and label each vertex in the graph with its component ID. Figure 1.1 shows a graph with multiple connected components.

There are several variants of connected components. A strongly Connected Component of a directed graph is a maximal set of vertices such that every pair of vertices in the set is reachable from each other. Bi-Connected Components are components in which removing any vertex still results in a connected component. A directed graph is Weakly Connected [7] if replacing all its directed edges with undirected edges produces a connected (undirected) graph.

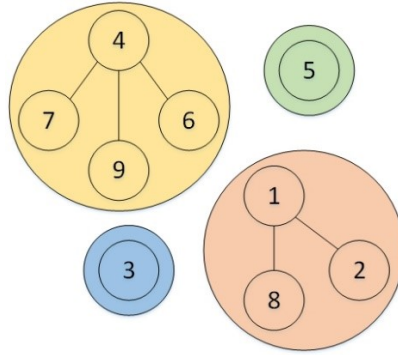


Figure 1.1 Connected Components in a Graph

1.2 General Serial Connected Components Algorithm

There are two main serial approaches for Connected Components - Depth First Search [15] and the Union-Find algorithm [5] using disjoint data structure.

1.2.1 Union-Find Algorithm

This algorithm initially iterates over all the vertices and places each of them in a separate disjoint set and uses the vertex ID as the representative (label) of the set. Then, it loops over each edge (u, v) combining the sets of vertex u and vertex v . Thus, the algorithm returns a collection of sets where each set represents a connected component in the graph. Algorithm 1 depicts the union-find algorithm.

Algorithm 1. Union-Find algorithm

1. **procedure:** Union-Find CC (V, E)
2. for each vertex v in V
3. Add vertex to a disjoint set S_v
4. Assign vertex v as the representative of the set S_v

5. end for
6. for each edge (u, v) in E
7. if $\text{Representative}(S_u) \neq \text{Representative}(S_v)$
8. Merge sets S_u and S_v
9. Choose Representative of S_u or S_v as new representative
10. end for

1.2.2 Depth-First Search

Hopcroft and Tarjan [15] proposed the approach of using depth-first search to compute connected components which is shown in Algorithm 2. It involves iterating over all the vertices and performing depth-first search recursively to find the connected components. It maintains a Boolean array “visited”, which denotes whether a vertex has been visited or not. The algorithm loops over each vertex and performs DFS only on those vertices that have not yet been visited. During the depth-first search, it visits the neighbors of that vertex and sets their visited array element to true. In this way, all the vertices that are connected will be visited in a single recursive search. As this algorithm visits each vertex once, it takes $O(|V|)$ time, i.e., $O(n)$, where $n = |V|$ denotes the number of vertices in the graph.

Algorithm 2. DFS for CC

1. **procedure:** DFS-CC (V, E)
2. for each vertex v in V
3. if $\text{visited}[v]$ is false
4. $\text{visited}[v] \leftarrow \text{true}$

5. increment number of connected components
6. Dfs(v)
7. end if
8. end for

1.3 General Parallel Connected Components Algorithm

The general approach used to determine connected components is as follows. Each vertex has a label to hold the component ID to which it belongs. Initially, this label is set to the vertex ID, that is, each vertex is considered a separate component. Then each vertex and their neighbors are iteratively processed, in parallel, to find the connected components. In each step, the labels are updated until all the vertices in a connected component have the same label. Often, the ID of the minimum vertex in each component is chosen as the component ID to guarantee uniqueness. This general approach is common to several algorithms, which is discussed below.

1.4 Contributions

This thesis proposes ECL, an efficient Connected Components algorithm. I implemented this algorithm in parallel for GPUs and CPUs using CUDA and OpenMP, respectively, as well as serial C code. In my algorithm, each vertex asynchronously processes its neighbors to find the ID of the minimum vertex in that component. All vertices follow a path through their neighbors, neighbors' neighbors, and so on, until the vertex with the lowest ID is found, which is then used to label all vertices on the path. My implementation is lock-free and uses a novel termination criterion to stop the computation at a vertex as soon as possible. Moreover, to improve the load balance, a

worklist is maintained for high-degree vertices to delay their processing.

Atomic operations play a significant role in parallel codes as they help prevent data race conditions. Though atomic operations are faster, it is a bottleneck to parallel code as they serialize the threads and affects the scalability. This thesis also proposes an atomic-free implementation of CC algorithm, ECL_{af} , in CUDA and OpenMP which is as fast as the fastest existing approaches.

I tested all the implementations on 18 real-world and synthetic graphs of varying sizes, including road maps, RMAT and Kronecker graphs, Internet topology graphs, citation graphs, web-link graphs, etc. I compared both of my CUDA implementations with the best pre-existing algorithms on two different GPUs - K40 and Titan X. On average, ECL is about 1.7x faster than the existing fastest GPU algorithm.

I implemented the parallel and serial CPU codes using OpenMP and C, respectively, and compared them with corresponding programs from the literature. The parallel CPU implementation of ECL is about 1.6x faster than the existing fastest Parallel CPU algorithm. I implemented the atomic free algorithm ECL_{af} in OpenMP as well and it is 1.2x faster than the fastest parallel CPU algorithm from the literature. The serial implementation of ECL is 5x faster than the fastest preexisting serial CC algorithm.

1.5 Outline

The rest of this thesis is organized as follows: Chapter 2 discusses various implementations of connected components. Chapter 3 includes related work on connected components. Chapter 4 describes my CC implementation under various configurations as

well as an atomic-free CUDA implementation of CC. Chapter 5 explains the various environments on which CC is tested, the inputs, and the evaluation methods. Chapter 6 presents a comparison of my CC implementation with other benchmarks and analyzes the performance. Chapter 7 concludes with a summary and future work.

CHAPTER 2

BACKGROUND STUDY

2.1 Parallel Implementations of CC

A straightforward approach to implement CC is to mark each vertex with a unique label and propagating vertex labels through neighboring vertices until all the vertices in the same component are labelled with a unique ID. This is called Label propagation.

Shiloach and Vishkin's approach computes connected components by two major steps - Hooking and Pointer Jumping.

The Hooking operation works on edges. The vertices on either side of an edge belong to a same component. For a given an edge (u, v) , the Hooking operation checks if the vertices are labelled with the same component ID. If not, the higher of the two vertex IDs is "overwritten" with the lower ID. This is achieved by making the higher representative point to the lower ID. Figure 2.1 shows an input graph where vertices 8 and 14 have different component ID. After hooking, vertices 8 and 14 are assigned the same component ID as shown in Figure 2.2.

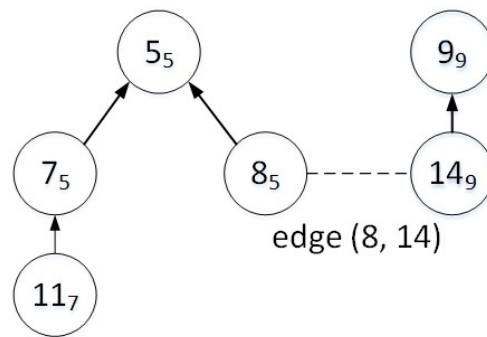


Figure 2.1 Graph before Hooking

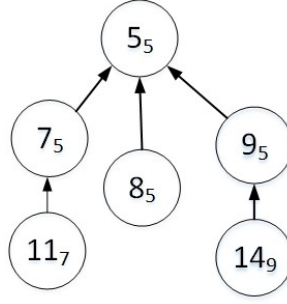


Figure 2.2 Graph after Hooking

Given a vertex u , Pointer Jumping reduces the depth of the tree (that u belongs to) by one by replacing its ID with its parent's ID. Figure 2.2 shows a tree before Pointer Jumping. After the algorithm terminates, the tree in Figure 2.2 has been reduced to a single-level tree shown in Figure 2.3. It represents that all the vertices belong to the same component with vertex 4 as their label.

Initially, the algorithm considers each vertex as a separate tree (or component labelled by its own ID). During each iteration, it performs Hooking and Pointer Jumping until all the multi-level trees are reduced to one-level trees (stars). Researchers have published different parallel algorithms for CC based on this approach, which use different names for these steps. This algorithm does not use any additional memory space for its computation.

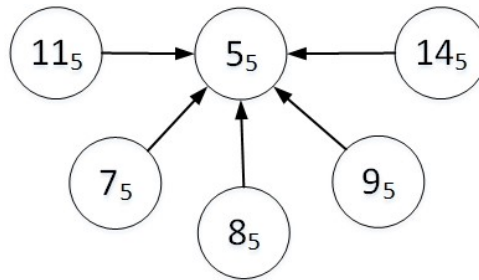


Figure 2.3 Graph after Pointer Jumping

Soman et al. [2] proposed a variant of Shiloach-Vishkin's [1] approach by introducing Multiple Pointer Jumping. The algorithm defines an array "Label" to hold the component ID and initializes it with the vertex ID. Then, it iterates over the hook kernel until all the vertices in the same component are connected. It iteratively performs Pointer Jumping to convert the multi-level tree to a single-level tree (star), thus setting all the vertices in the same component to the same ID. Algorithm 3 shows this approach. Algorithms 4 and 5 show the Hook and Multiple Pointer Jumping kernels.

Algorithm 3. Overall Steps – Soman's Algorithm

1. **procedure:** Soman's CC (V, E)
2. for each vertex v in V
3. Initialize $\text{Label}[v] \leftarrow v$
4. repeat
5. for each edge (u, v)
6. Hook (u, v)
7. until Hook performs no change in Label []
8. repeat
9. multiple-pointer jump ()
10. until Jump performs no change in Label []

Algorithm 4. Hook Kernel in Soman's Algorithm

1. **procedure:** Soman's Hook (u, v)
2. if (edge is unmarked && $\text{Label}[u] \neq \text{Label}[v]$)
3. $\min \leftarrow \min (\text{Label}[u], \text{Label}[v])$

4. $\max \leftarrow \max (\text{Label}[u], \text{Label}[v])$
5. if iteration is even
6. $\text{Label}[\min] = \max$
7. else
8. $\text{Label}[\max] = \min$
9. end if
10. else
11. Mark edge (u, v) // edge hiding
12. end if

Algorithm 5. Multiple Pointer Jumping Kernel in Soman’s Algorithm

1. **procedure:** Multiple Pointer Jumping $(u, \text{Label} [])$
2. repeat
3. $\text{Label}[u] \leftarrow \text{Label} [\text{Label} [u]]$
4. $u \leftarrow \text{Label}[u]$
5. until u is not a root

Sutton et al. [7] proposed a work-efficient connected components algorithm based on Soman’s work. The algorithm, presented in Algorithm 6, splits the graph into $2|E|/|V|$ edge-list segments where $|V|$ denotes the number of vertices and $|E|$ denotes the number of edges in the graph and iteratively performs Hook and pointer jumping operations on each of the segments. It performs atomic Hook and a single multi pointer jumping for each vertex.

Algorithm 6. Sutton et al's CC Algorithm

1. **procedure:** Sutton's CC (V, E)
2. for each vertex v in V
3. Initialize $\text{Label}[v] \leftarrow v$
4. for each segment in the edge-list
5. repeat
6. for all edges (u, v) in the segment
7. AtomicHook (u, v)
8. end for
9. until Hook performs no change in Label []
10. for all v in V
11. multi-pointer jump ()
12. end for
13. end for

Gunrock's [3] connected components algorithm is again a variant of Soman's approach. It involves iterating over Hooking and pointer jumping until all the vertices in the same component have the same component ID. However, instead of processing all the vertices and edges in each iteration, this approach tries to reduce the workload by using a filter operator. After Hooking, the filter operator removes an edge if both the end vertices have the same component ID. Similarly, after pointer jumping, it removes a vertex whose own vertex ID is equal to the component ID. In this way, it reduces the workload after each iteration.

Ligra [4] is a graph processing framework for shared-memory parallel/multicore machines. Ligra provides two implementations for CC - Components and BFSCC. For an undirected graph $G = (V, E)$, the Components algorithm, outlined in Algorithm 7, maintains two arrays “ID” and “prevID” of size $|V|$ initialized such that $ID[i] = i$ and $prevID[i] = i$. It iterates over the vertices, updates its ID with the minimum ID of its neighbors until all the vertices in the same component have the same ID (minimum ID). It reduces the workload in each iteration by only processing those vertices whose ID has changed in the previous iteration. The algorithm tracks a vertex’s ID by comparing $prevID$ with ID .

Algorithm 7. Components - Ligra Algorithm

1. **procedure:** Ligra Component’s CC (V, E)
2. $frontier = \{0, \dots, |V|-1\}$
3. for each vertex v in V
4. $ID[v] \leftarrow v$
5. $prevID[v] \leftarrow v$
6. end for
7. repeat
8. for each vertex v in V
9. $prevID[v] \leftarrow ID[v]$
10. end for
11. for each vertex v in V
12. $ID[v] \leftarrow \min ID \text{ of its neighbors}$
13. if ($ID[v] == prevID[v]$)

14. remove v from Frontier
15. end if
16. end for
17. until Frontier is empty

Ligra's BFSCC algorithm uses Breadth-First Search in parallel to compute CC. It maintains an array "Parent" of size $|V|$ to hold the component ID of that vertex and initializes it with -1 to denote that a vertex has not yet been processed. The algorithm iterates over all the vertices and in each iteration, it maintains a worklist that contains the vertices to be processed. It carries out Breadth-first search on the vertices in the worklist and all the vertices reachable through the search are again pushed back onto the worklist. This process continues until there are no more vertices reachable and all the vertices are processed. It labels all the vertices processed in the same iteration with the same ID. In this way, the algorithm computes connected components. Ligra uses two simple routines -VertexMap and EdgeMap, for mapping over vertices and edges, respectively.

Algorithm 8. BFSCC - Ligra Algorithm

1. **procedure:** Ligra BFSCC's CC (V, E)
2. Worklist = $\{\}$
3. for each vertex v in V
4. Parent[v] \leftarrow -1
5. for each vertex v in V
6. if (Parent[v] \neq -1)
7. push v to Worklist

8. repeat
9. $v = \text{pop.Worklist}$
10. $\text{BFS}(v)$
11. push all vertices reachable from v to Worklist
12. until Worklist is empty
13. end if
14. end for

Ligra+ [5] is a variant of Ligra that is based on a compressed graph representation. It uses an encoder program to compress the input graphs with a difference encoding scheme. This difference encoding scheme targets the adjacency list of a vertex and encodes the difference between the edges of each vertex using variable length codes. The encoder compresses each integer into k -bit blocks and each block has a continue bit to indicate if the next block is also used to compress the integer x . An integer x is simply written in binary representation in a block. If a block cannot fit all the bits, then the encoder continues it in the following blocks by setting the continue bit. Decoding works the same way by converting the binary representation to the original form and including blocks with the continue bit set.

CRONO [12] is a benchmark suite for graph algorithm in shared-memory multicore machines. CRONO's CC algorithm is implemented using pthreads and it maintains an array to store each vertex's label (Component ID). Then, it loops over all the vertices iteratively updating their label based on the connectivity.

CHAPTER 3

RELATED WORK

Hopcroft and Tarjan’s [15] approach was one of the first algorithms devised to compute connected components in serial. It is simple and stated as “well known” at that point of time. It is linear in time as it visits each vertex once. Later algorithms were devised based on this work. Various graph processing libraries such as Boost [9], Lemon [14], igraph [13], etc. include serial code to compute CC.

Shiloach-Vishkin’s [1] proposed a different approach involving Hooking and pointer jumping to compute CC in parallel. Their algorithm is suitable for GPUs. Soman et al. [2] adapted this algorithm by modifying their pointer jumping to multiple pointer jumping. Though the number of iterations taken by their approach is the same as in Shiloach-Vishkin’s algorithm, multiple pointer jumping drastically reduces the number of reads and writes to the memory, thus improving the performance.

Sutton et al. [7] proposed a variant of Soman’s algorithm by introducing Atomic Hook and single Multi pointer jump. By using atomics, it locks one of the vertices in the edge until the hook operation succeeds, which eliminates the CPU-side convergence loop. Instead of multiple calls to the pointer jumping kernel, it works with a single call to the multi pointer jump kernel, thus reducing the GPU-CPU communication overhead. Further, this algorithm splits the input graph into edge-list segments, which minimizes the number of atomic operation in the next Hook.

Ligra's connected components algorithm is simple and its BFSCC algorithm is based on its own BFS algorithm. The authors claim that it is efficient and scalable. Ligra+ is an optimized version of Ligra. As it works on compressed graphs, it requires a smaller memory footprint, making it possible to fit larger graphs into the available memory. It is faster than Ligra when using the fast compression scheme.

CRONO [12] does not scale well to large inputs. It allocates four arrays of size $|V|$ and two two-dimensional arrays of size $(|V| * \text{Degree}_{\max})$, where Degree_{\max} indicates the graph's maximum vertex degree. As a result, it runs out of memory for larger graphs. In fact, the graphs that they used are relatively small. Their largest graph is about 15 times smaller than my largest input graph.

CHAPTER 4

ECL ALGORITHM AND IMPLEMENTATIONS

This section discusses my ECL algorithm, various optimizations to improve its performance and its implementation on various platforms (GPU, Parallel and Serial CPU).

4.1 ECL Base Algorithm

The ECL CC Algorithm is simple and straightforward. It uses three main functions (kernels in CUDA) - “init”, “compute” and “flatten”. Algorithm 9 shows the overall layout. Each vertex has a label to hold the component ID to which it belongs. The algorithm chooses the ID of the minimum vertex in each component as the component ID to guarantee uniqueness of the labels. Most of the existing CC algorithms initialize the label of each vertex with the vertex ID. The ECL code optimizes the init function by initializing the label of each vertex with the first smaller neighbor ID it encounters. This improves performance as it moves the computation towards the minimum vertex ID in each component faster.

The compute function visits each vertex and processes each edge of that vertex so that both ends of the edge has the same ID. It uses a representative function on each vertex and its neighbors. The representative function follows the path from a vertex’s ID, to that vertex’s ID and so on until it finds the end of the chain, i.e., a vertex whose ID points to itself, which is then assigned to the starting vertex. This is a variant of pointer jumping as it updates the label of each of the vertices with a better label (smaller value).

It has a special “if” condition to make sure that each edge is considered only once, i.e., in one direction but not the other.

After the compute function, all the vertices are labelled either directly or indirectly with the ID of the lowest vertex in each of the components. The flatten function, a form of pointer jumping, visits all vertices and updates the label so that it represents the component ID directly.

Since CC is an irregular graph application, iteratively processing all vertices by assigning them to separate threads would result in load imbalance as the vertex degrees vary to a great extent. The CUDA implementation of CC uses a double-sided worklist to efficiently distribute work to each thread.

Algorithm 9. ECL CC Overall Algorithm

1. **procedure:** ECL CC (V, E)
2. Init (V, nstat)
3. Compute (V, E, nstat)
4. Flatten (V, nstat)

Algorithm 10. ECL CC Algorithm - Init Function

1. **procedure:** Init (V, nstat)
2. $\text{nstat} = \{0, \dots, |V|-1\}$
3. for each vertex v in V
4. $\text{nstat}[v] \leftarrow$ First neighbor smaller than v .

Algorithm 11. ECL CC Algorithm - Compute Function

1. **procedure:** Compute (V, E, nstat)

2. for each v in V
3. $vstat \leftarrow \text{representative}(v, nstat)$
4. for each edge (u, v) in E
5. if $(v > u)$
6. $ostat \leftarrow \text{representative}(u, nstat)$
7. if $(vstat < ostate)$
8. $nstat[ostat] \leftarrow vstat$
9. else
10. $nstat[vstat] \leftarrow ostate$
11. end if
12. end if
13. end for
14. end for

Algorithm 12. ECL CC Algorithm - Flatten Function

1. **procedure:** Flatten ($V, nstat$)
2. for each vertex v in V
3. $vstat \leftarrow nstat[v]$
4. while $(vstat > nstat[vstat])$
5. $vstat \leftarrow nstat[vstat]$
6. end while
7. end for

Algorithm 13. ECL CC Algorithm - Representative Function

1. **procedure:** Representative (v , $nstat$)
2. $curr \leftarrow nstat[v]$
3. if ($curr \neq v$)
4. $prev \leftarrow v$
5. $next \leftarrow nstat[curr]$
6. while ($curr > next$)
7. $nstat[prev] \leftarrow next$
8. $prev \leftarrow curr$
9. $curr \leftarrow next$
10. end while
11. end if

In addition to the two arrays (neighborlist & neighborindex) used for representing the graph, the algorithm maintains an array ‘nstat’ of size $|V|$, which stores the component ID of each of the vertices. The CUDA implementations maintain another array “worklist” of size $|V|$ to store the vertices.

4.2 ECL - GPU Implementation

I implemented the ECL CC Algorithm in CUDA for GPUs devices. It contains GPU-specific optimizations to improve the performance. It consists of three main kernels - init, compute1, and flatten. It also has two more compute kernels (compute2 & compute3) to process the vertices in the worklist.

As described above, the init kernel initializes the vertex labels with the first neighbor, if any, whose ID is smaller than the vertex ID. The algorithm divides the processing of vertices among the three compute kernels. It maintains a double-sided worklist and the compute1 kernel fills the worklist with vertices on both the sides based on their edge degree. The compute1 kernel populates the double-sided worklist as shown in Figure 4.1. Two threshold values (16 and 352) are identified by trial and error method. The compute1 kernel pushes vertices with degree greater than 16 and less than or equal to 352 to the front of the worklist and vertices with degree greater than 352 to the rear of the worklist, which are later read by the other compute kernels. The compute1 kernel processes the vertices with degree less than 16, i.e., all the threads in the compute1 kernel process low-degree vertices and no thread has to wait long for other threads to finish their computation, thus resulting in better load balancing. Each thread processes the assigned vertex and loops over each neighbor, updating the vertex labels, until there is no change in the vertex's label or its neighbor's labels.

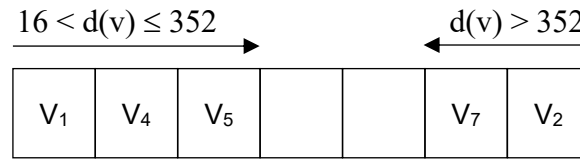


Figure 4.1 Double-sided worklist

The compute2 kernel reads vertices from the front of the worklist and utilizes warp-level parallelism in the GPU, i.e., each warp processes a single vertex. A warp in a GPU represents a set of 32-contiguous threads. All the threads within a warp execute the same instruction in the same clock cycle and they can exchange data with each other using shuffle machine instructions without explicit synchronization. In the compute2

kernel, each warp reads from the front of the worklist and all the threads within a warp share the work and process that vertex's neighbors using a cyclic assignment.

The compute3 kernel reads vertices from the rear of the worklist and utilizes block-level parallelism in the GPU. A block is a group of threads. Each block can hold up to 1024 threads. The programmer decides the actual number of threads per block. I use 256 threads per block. All the threads in a block have access to shared memory, which enables fast data exchange. It requires explicit synchronization, which can be achieved by the CUDA instruction `__syncthreads()`. The compute3 kernel uses block-level parallelism as it processes vertices with large numbers of neighbors. It assigns each thread block to process a single vertex and all the threads within a block share the work of processing the neighbors and make sure that the two ends of each vertex have the same component ID.

After the compute kernels are done, all the vertex labels directly or indirectly point to the ID of the lowest vertex in each connected component. The flatten kernel updates all the vertex labels so that they directly refer to the component ID.

ECL uses atomic operations to update the vertex labels in the compute kernels as more than one thread might be updating a vertex's label. Moreover, it uses atomic operations to insert vertices in the worklist and to read them from the worklists.

In terms of space complexity, ECL uses two arrays "neighborlist" and "neighborindex" of size $|V|$ and $|E|$, respectively, to represent the input graph. It uses a double-sided worklist of size $|V|$ for load balancing. Overall, ECL's space complexity is $O(|V|+|E|)$.

4.3 ECL - Parallel CPU Implementation

I implemented the ECL CC algorithm in OpenMP for multicore CPUs. It has the same three functions as mentioned in Algorithm 9. The init and compute functions use a “guided” schedule to allocate vertices to threads. The algorithm assigns each thread with a vertex. Each thread processes the vertex and loops over each neighbor, until there is no change in their labels. To avoid data races, this algorithm uses a GCC-specific atomic-compare-and-swap operation (`__sync_val_compare_and_swap`) to update the vertex labels.

4.4 ECL - Serial CPU Implementation

I implemented the ECL CC algorithm in C for serial CPU devices. It is straightforward and does not involve any worklists or atomic operations as there are no load balancing issues or data races in the serial code.

4.5 ECL_{af} – Atomic-Free CC Implementation

The ECL algorithm uses atomic operations to prevent data races as there are situations when multiple threads try to access the same memory location. These operations prevent data races by ensuring that no other thread can access a given memory location until the operation is done. However, it is a significant bottleneck as the atomic operations are performed serially and any thread that tries to access that memory location must wait until the atomic operation is done. In fact, future systems may be so widely parallel that implementing fast atomics will be difficult.

I propose an atomic free CC algorithm, ECL_{af} for parallel codes, which loops over

the compute kernel(s) to avoid atomic operations. Eliminating atomic operations would result in data race conditions which in turn would result in incorrect computation. To avoid such data races, this algorithm repeatedly calls the compute kernels. A variable ‘reiterate’ decides whether to loop over the compute kernel or not. Algorithm 14 and 15 show the pseudocode of the general ECL_{af} algorithm and its compute function, respectively. Similar to the previous GPU implementation, ECL_{af}’s GPU implementation groups vertices based on their edge degree and uses three compute kernels for load balancing. Each compute kernel has its own ‘reiterate’ variable and it is set whenever a vertex’s label or its neighbor’s label is updated. The algorithm reiterates compute kernels until all the vertex labels and all of its neighbor’s labels remain unchanged.

Algorithm 14. General ECL_{af} - Overall Algorithm

1. **procedure:** ECL_{af}CC (V, E)
2. Init (V, nstat)
3. reiterate \leftarrow 1
4. do
5. if reiterate
6. Compute (V, E, nstat, &reiterate)
7. end if
8. while (!reiterate)
9. Flatten (V, nstat)

Algorithm 15. ECL_{af} CC Algorithm - Compute Function

1. **procedure:** Compute (V, E, nstat, *reiterate)
2. for each vertex v in V

```

3. vstat  $\leftarrow$  representative (v, nstat)
2.   for each edge (u, v) in E
3.     if (v > u)
4.       ostat  $\leftarrow$  representative (u, nstat)
5.       *reiterate  $\leftarrow$  1
6.       if (vstat < ostat)
7.         nstat[ostat]  $\leftarrow$  vstat
8.       else
9.         nstat [vstat]  $\leftarrow$  ostat
10.      end if
11.    end if
12.  end for
13. end for

```

I implemented ECL_{af} in OpenMP for parallel CPU devices. As discussed above, by repeatedly calling the compute function, it eliminates the CPU-specific atomic operation (`__sync_val_compare_and_swap`). It uses a “guided” schedule to dynamically allocate work to each thread and the chunk size reduces as the program runs to minimize load imbalance. Since CPU does not face significant load balancing issues, this implementation uses only one compute function. Its own variable ‘reiterate’ decides whether to loop over compute function or not.

CHAPTER 5

EVALUATION METHODOLOGY

5.1 GPU and CPU Machines

In this thesis, I have implemented my CC algorithm in parallel both for GPUs and CPUs in CUDA and OpenMP, respectively, and as a serial C code. I tested the parallel CUDA implementations and related benchmarks on two GPUs, a Titan X and a K40.

The first is a GeForce GTX Titan X, which is based on the Maxwell architecture. The second is a Tesla K40c, which is based on the Kepler architecture. The Titan X has 3072 processing elements distributed over 24 multiprocessors that can hold the contexts of 49,152 threads. Each multiprocessor has 96 kB of shared memory and 48 kB of L1/texture cache. The 24 multiprocessors share a 2 MB L2 cache as well as 12 GB of global memory with a theoretical peak bandwidth of 336 GB/s. I use the default clock frequencies of 1.1 GHz for the processing elements and 3.5 GHz for the GDDR5 memory.

The K40 has 2880 processing elements distributed over 15 multiprocessors that can hold the contexts of 30,720 threads. Each multiprocessor has 48 kB of texture memory as well as 64 kB of cache that is split between the shared memory and the L1 data cache. The 15 multiprocessors share a 1.5 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 288 GB/s. I disabled ECC protection of the main memory and used the default clock frequencies of 745 MHz for the processing elements and 3 GHz for the GDDR5 memory. Both GPUs are plugged into 16x PCIe 3.0 slots in

the same system, which has dual 10-core Xeon E5-2687W v3 CPUs running at 3.1 GHz. The host memory size is 128 GB and the operating system is Fedora release 23.

I test the parallel CPU implementations on two machines. Machine 1 uses an Intel(R) Xeon(R) CPU E5-2687W based x86_64 architecture, clocked at 3.10 GHz. It has 2 sockets and 10 cores per socket. As it supports hyper threading, it can run 40 threads simultaneously. It has a 32 kB L1 data cache and a 256 kB L2 cache. Machine 2 uses an Intel(R) Xeon(R) CPU X5690 based x86_64 architecture, clocked at 3.47 GHz. It has 2 sockets, 6 cores per socket and does not support hyper threading, meaning it can run 12 threads simultaneously. It also has a 32 kB L1 data cache and a 256 kB L2 cache.

5.2 Input Graphs Format and Specifications

I use 18 graphs as inputs, which include road maps (europe_osm , USA-road-d.NY and USA-road-d.USA), a grid (2d- 2e20.sym), a random graph (r4-2e23.sym), RMAT graphs (rmat16.sym and rmat22.sym), a synthetic Kronecker graph from the Graph500 (kron_g500-logn21), a product co-purchasing graph (amazon0601), Internet topology graphs (as-skitter and internet), publication citation graphs (citationCiteseer and coPapersDBLP), a patent citation graph (cit-Patents), a Delaunay triangulation (delaunay_n24), an online journal maintenance community graph (soc-livejournal) and web-links graphs (in-2004 and uk-2002). The Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome [16], the Galois framework [17], the Stanford Network Analysis Platform [18], and the University of Florida Sparse Matrix Collection [19] provided these graphs. The graph sizes vary from 65K vertices and 387K edges for the smallest graph to 18M vertices and 523M edges for the largest

graph. Table 1 shows the graph sizes and other pertinent information.

Table 5.1 Input Graph Information

S. No	Graph Name	No. of Edges	No. of Vertices	Vertex degree			No. of CC
				min	max	avg	
1	2d-2e20	1,048,576	4,190,208	2	4	3	1
2	amazon0601	403,394	4,886,816	1	2752	12	7
3	as-skitter	1,696,415	2,219,059	1	35455	1	756
4	citationCiteseer	268,495	2,313,294	1	1318	8	1
5	cit-Patents	3,774,768	33,037,894	1	793	8	3,627
6	coPapersDBLP	540,486	30,491,458	1	3299	56	1
7	delaunay_n24	16,777,216	100,663,202	3	26	5	1
8	europa_osm	50,912,018	108,109,320	1	13	2	1
9	in-2004	1,382,908	27,182,946	0	21869	19	134
10	internet	124,651	387,240	1	151	3	1
11	kron_g500-logn21	2,097,152	182,081,864	0	213904	86	553,159
12	r4-2e23	8,388,608	67,108,846	2	26	7	1
13	rmat16	65,536	967,866	0	569	14	3,900
14	rmat22	4,194,304	65,660,814	0	3687	15	428,640
15	soc-livejournal	4,847,571	85,702,474	0	20333	17	1,876
16	uk-2002	18,520,486	523,574,516	0	194955	28	38,359
17	USA-NY	264,346	730,100	1	8	2	1
18	USA-USA	23,947,347	57,708,624	1	9	2	1

I use the Compressed Adjacency List format to represent all the graphs, which is quite space efficient yet still allows to identify the neighbors of a vertex quickly. It is based on two arrays: “neighborlist” of size $|E|$ is simply the concatenation of all the adjacency lists, and “neighborindex” of size $|V| + 1$ stores the starting point of each adjacency list. It needs one extra element to indicate the end of the last adjacency list. “neighborindex” helps to find the number of edges of each vertex. “neighborlist” indexed by “neighborindex” can be used to iterate over the edges. For example, Figure 5.1 shows a graph with 4 vertices and 7 edges. Figure 5.2 shows the corresponding Compressed Adjacency List format.

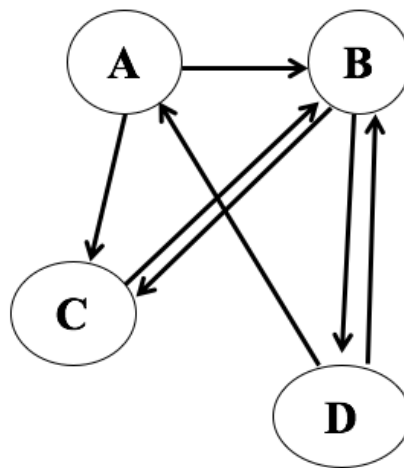


Figure 5.1 Graph

Adjacency Lists

A: B, C

B: C, D

C: B

D: A, B

Combined Adjacency list, $L = B, C, C, D, B, A, B$

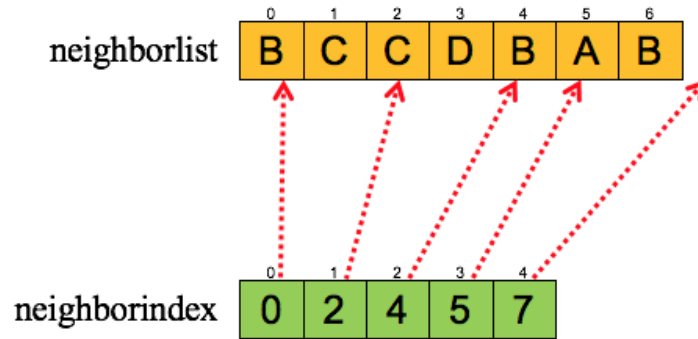


Figure 5.2 Compressed Adjacency List Representation

5.3 Compiler Information

I use nvcc 8.0 for compiling my CUDA CC implementation and for the other CUDA-based benchmarks. I compile all the CPU codes using g++ 5.3.1. In all cases, I use the -O3 optimization flag.

CHAPTER 6

RESULTS

6.1 Configurations

I evaluate my CC CUDA implementations (including the atomic free version) on two different GPUs. I run all the GPU benchmarks on both GPUs using the same compiler flags. I compare performance in terms of slowdown with respect to my CC implementation. I follow a similar procedure for performance comparison in the parallel and serial CPU codes.

6.2 Comparison with Parallel GPU Benchmarks

This subsection compares my CC implementation (ECL) with other GPU benchmarks on the Titan X and K40 GPUs. I run all codes on all 18 input graphs and normalize their runtimes. I fix the ECL values as 1.0 and calculate slowdowns for the other benchmarks with respect to ECL. Figure 6.1 shows the slowdowns of all GPU benchmarks on Titan X. Figure 6.2 shows the slowdowns on K40. Bars in the chart that are higher than 1.0 are slower than ECL and there is a reference line in the charts for easier comparison. Average (geometric mean) slowdowns are present for all benchmarks in the charts.

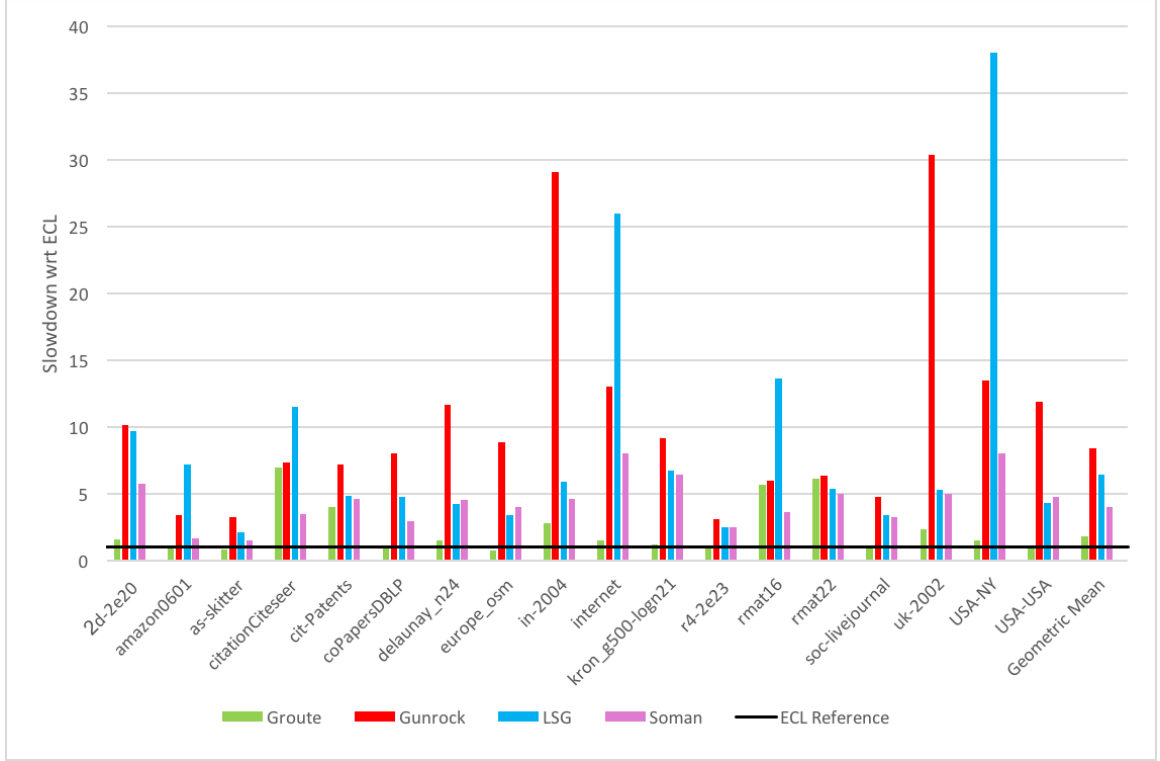


Figure 6.1 Parallel GPU Benchmarks Slowdowns relative to ECL on the Titan X

Figure 6.1 shows the slowdowns in GPU benchmarks with respect to ECL on Titan X. ECL is the fastest on 16 input graphs and Groute is 1.3x faster than ECL on the remaining two input graphs. On average, ECL is 1.8x faster than Groute, 4x faster than Soman's, 6.4x faster than LSG and 8.4x faster than Gunrock.

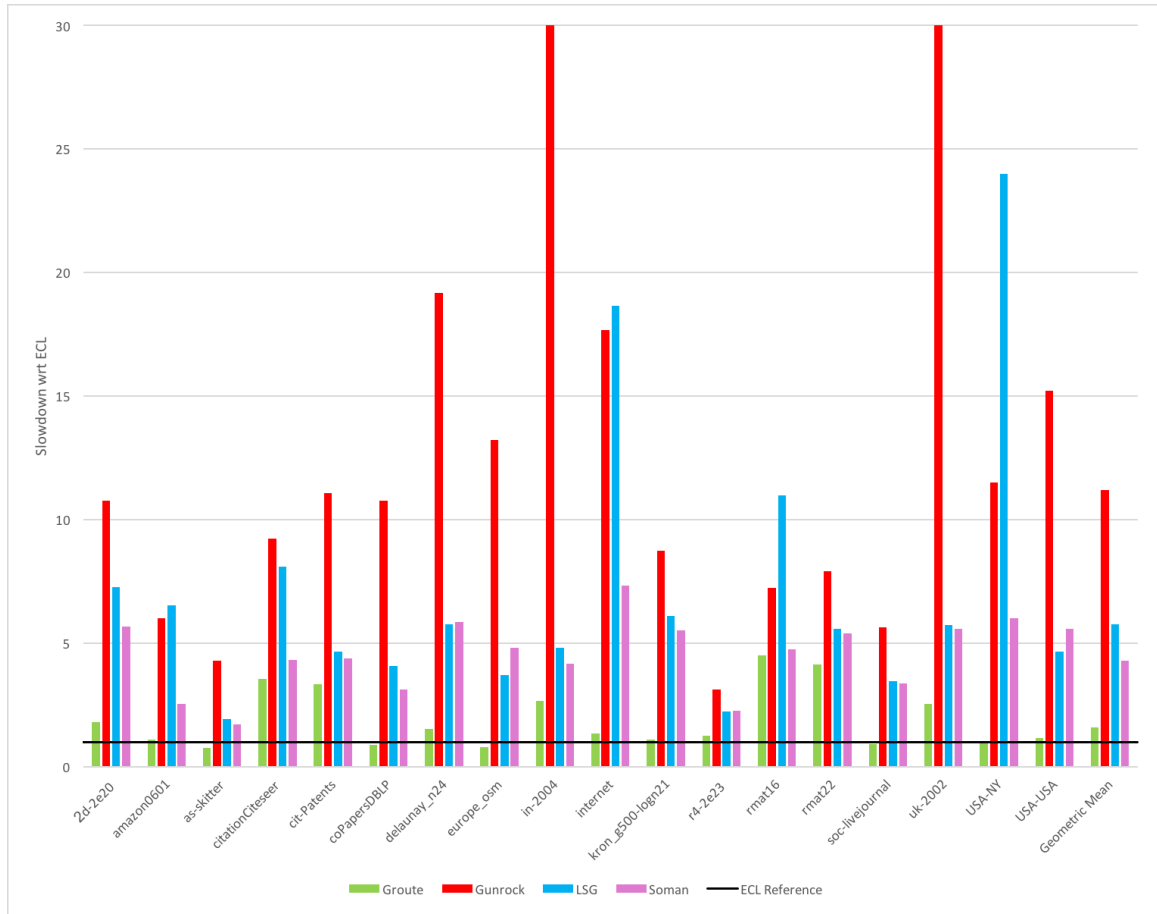


Figure 6.2 Parallel GPU Benchmarks Slowdowns relative to ECL on the K40

Note: Bars exceeding a height of 30 are cut off.

Figure 6.2 shows the slowdowns in GPU benchmarks with respect to ECL on K40. ECL is the fastest on 14 input graphs and Groute is 1.4x faster than ECL on the remaining four input graphs. On average, ECL is 1.6x faster than Groute, 4.3x faster than Soman's, 5.8x faster than LSG and 9x faster than Gunrock.

6.3 Comparison with Parallel CPU Benchmarks

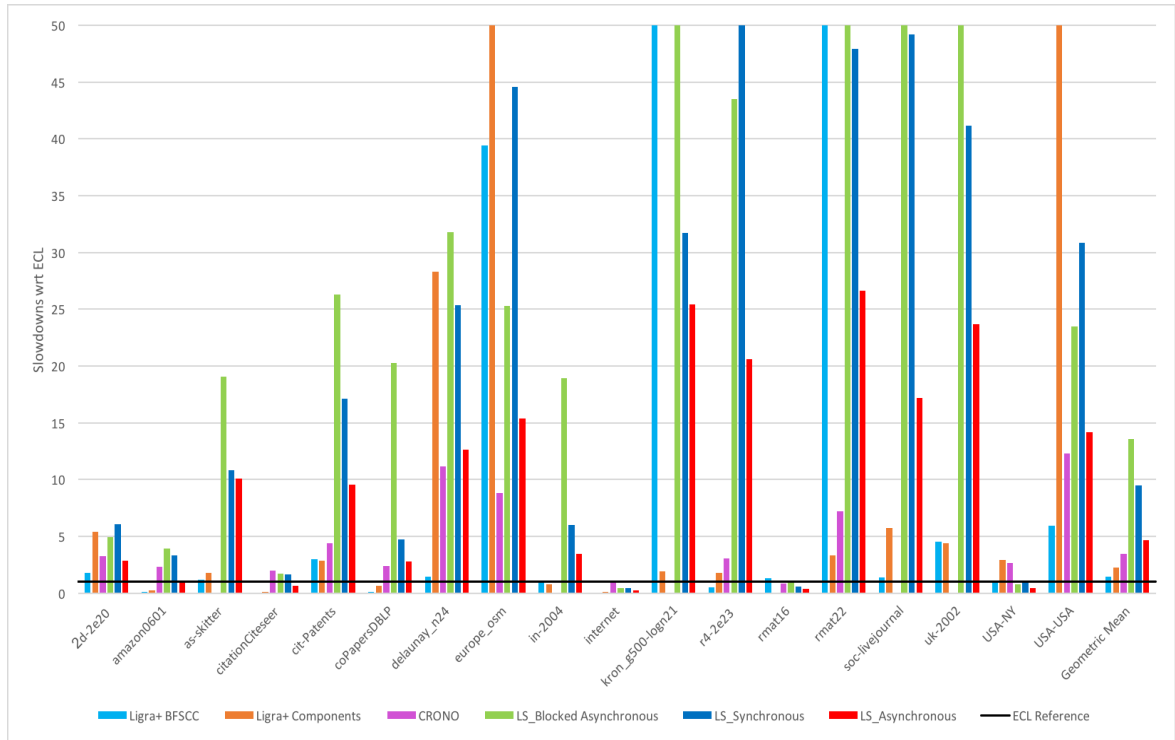


Figure 6.3 Parallel CPU Benchmarks Slowdowns relative to ECL on the Zurich

Note: Bars exceeding a height of 50 are cut off.

Figure 6.3 shows the slowdowns in Parallel CPU benchmarks with respect to ECL on Zurich. ECL is 4.2x faster than the existing fastest benchmark, Ligra+ BFSCC, on more than half of the input graphs and it beats the other benchmarks on overall performance. On average, ECL is 1.4x faster than Ligra+'s BFSCC, 2.2x faster than Ligra+'s Components, 3.4x faster than CRONO, 4.7x faster than Asynchronous LS, 9.5x faster than Synchronous LS and 13.6x faster than Blocked Asynchronous LS. CRONO does not support 5 input graphs (as-skitter, in-2004, kron_g500-logn21, soc-livejournal & uk-2002) as it runs out of memory.

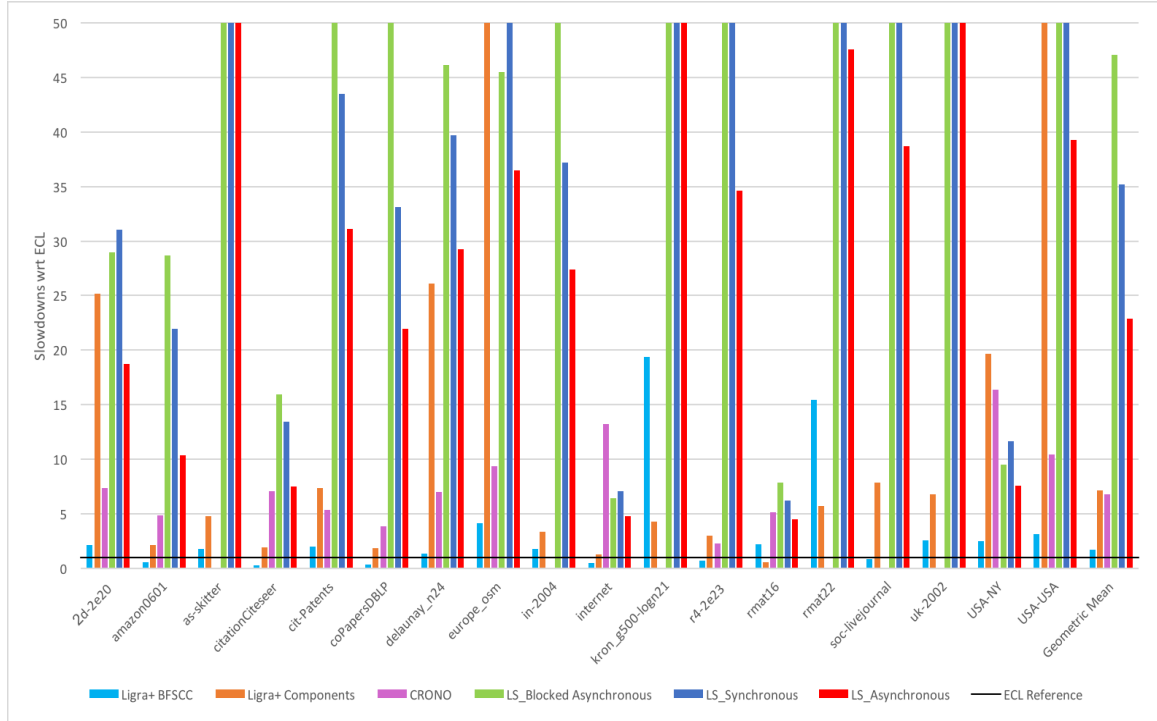


Figure 6.4 Parallel CPU Benchmarks Slowdowns relative to ECL on the Denver

Note: Bars exceeding a height of 50 are cut off.

Figure 6.4 shows the slowdowns in Parallel CPU benchmarks with respect to ECL on Denver. ECL is faster than the existing fastest benchmark, Ligra+ BFSCC, on more than half of the input graphs by 3.2x and it beats the other benchmarks on overall performance. On average, ECL is 1.7x faster than Ligra+'s BFSCC, 7.1x faster than Ligra+'s Components, 6.8x faster than CRONO, 22.8x faster than Asynchronous LS, 35.2x faster than Synchronous LS and 47.1x faster than Blocked Asynchronous LS.

6.4 Comparison with Serial CPU Benchmarks

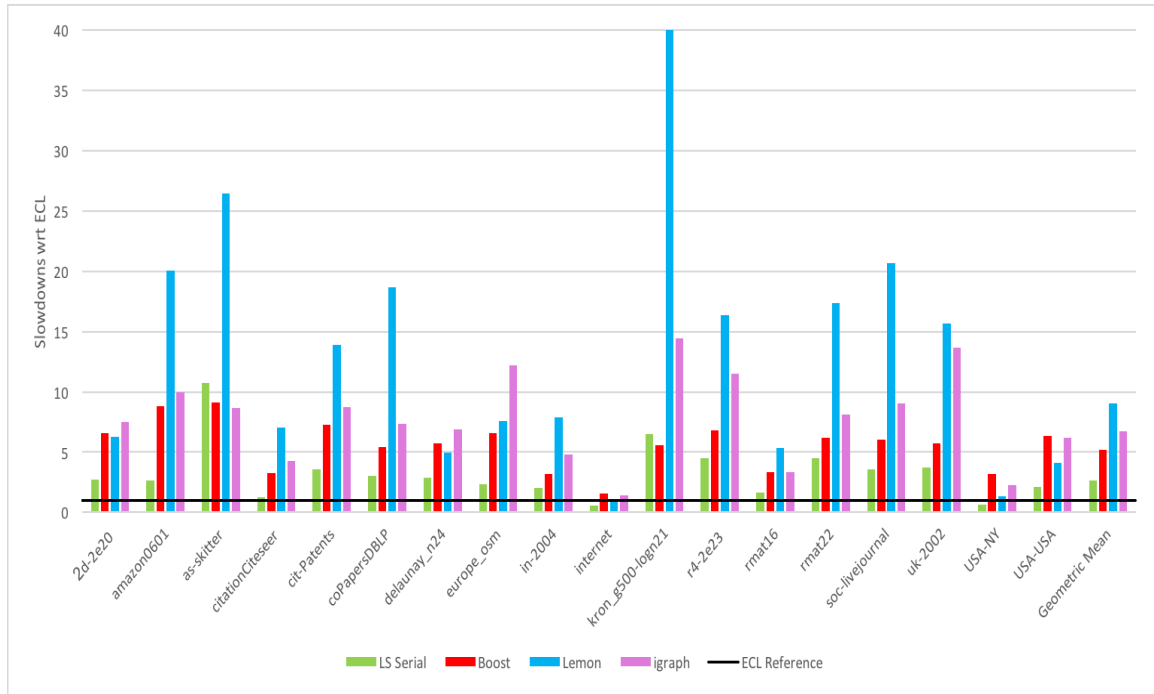


Figure 6.5 Serial CPU Benchmarks Slowdowns relative to ECL on the Zurich

Figure 6.5 shows the slowdowns in Serial benchmarks with respect to ECL on Zurich. ECL is 3.1x faster than LS Serial, on 16 input graphs. On overall comparison, ECL is 2.6x faster than LS Serial, 5.2x faster than Boost, 6.7x faster than igraph and 9.1x faster than Lemon.

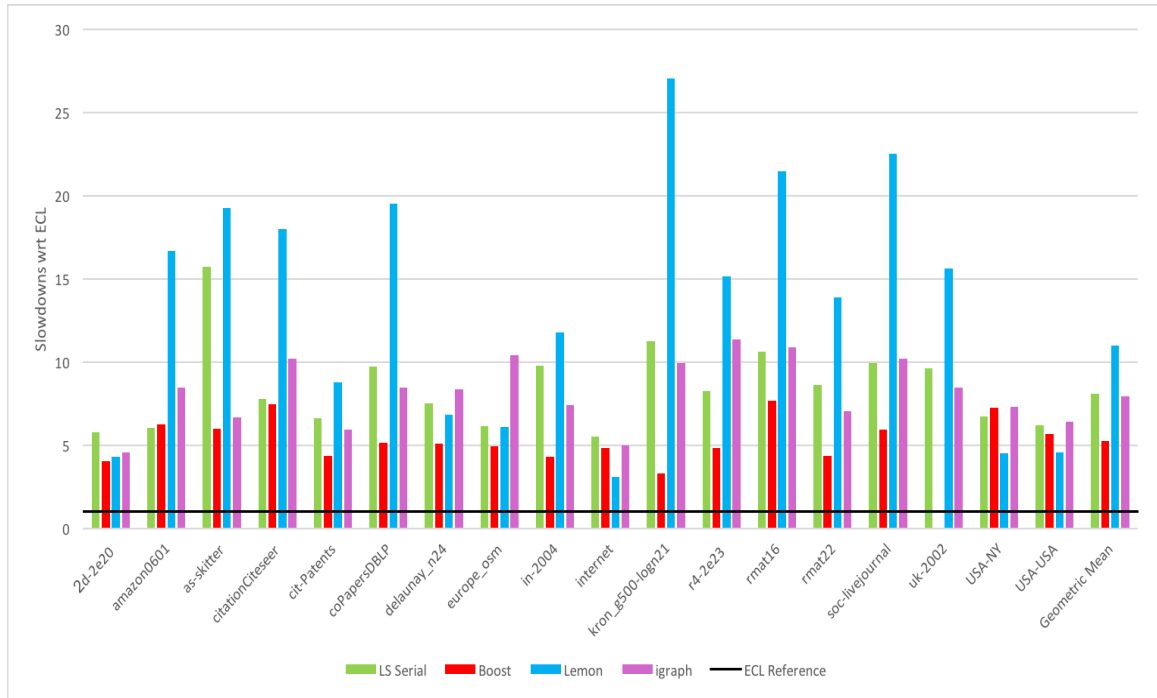


Figure 6.6 Serial CPU Benchmarks Slowdowns relative to ECL on the Denver

Figure 6.6 shows the slowdowns in Serial benchmarks with respect to ECL on Denver. ECL is the fastest on all the input graphs. Overall, ECL is 5.3x faster than Boost, 7.9x faster than igraph, 8.1x faster than LS Serial and 11x faster than Lemon.

6.5 Performance Comparison across different Configurations

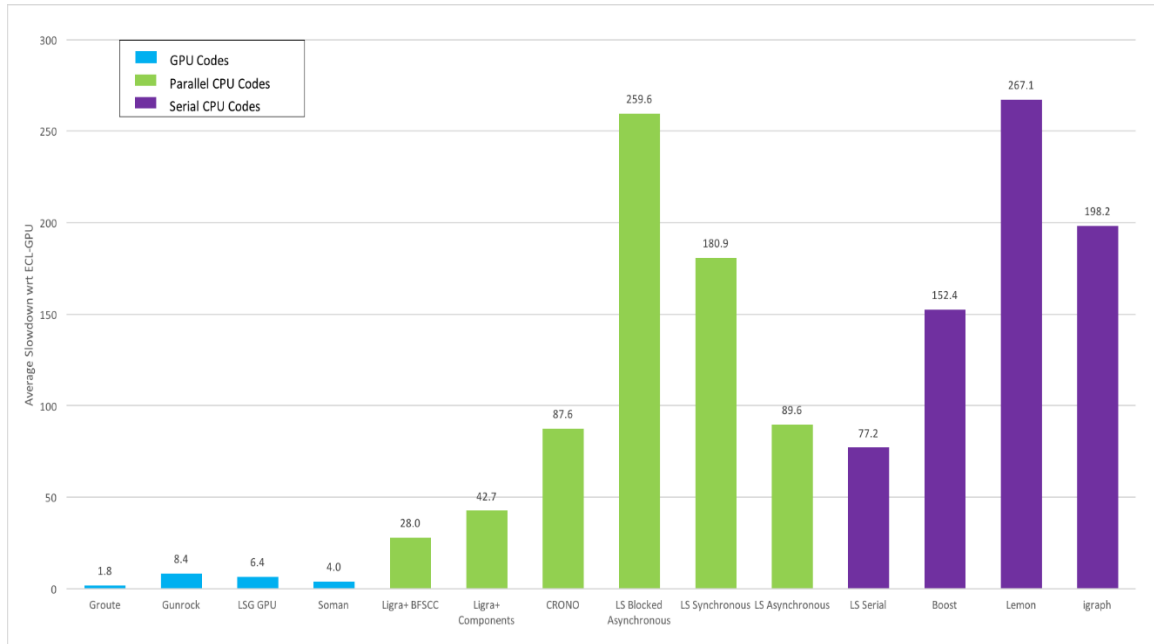


Figure 6.7 Geometric-Mean Slowdowns of CC Implementations on different systems

Figure 6.7 compares the average slowdowns for all the benchmarks across different configurations - GPU, Parallel and Serial CPU. I calculate the slowdowns for all the benchmarks with respect to the GPU ECL runtimes on the Titan X. The chart considers the runtimes of the parallel and serial CPU benchmarks run on Zurich. As the chart shows, ECL GPU is the fastest of all the tested benchmarks. As expected, the GPU benchmarks are faster than the CPU codes. Interestingly, some of the parallel CPU codes are slower than the serial codes. In particular, all the parallel LS codes are slower than their respective serial implementations.

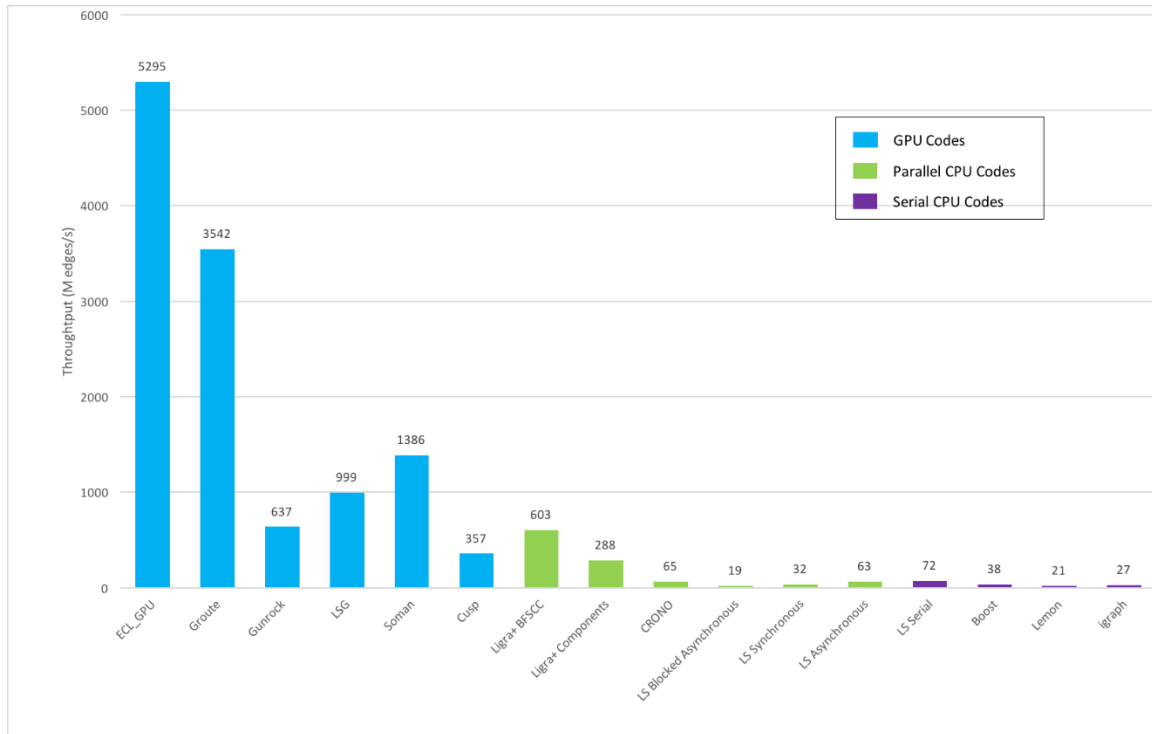


Figure 6.8 Throughput (Edges/s) of various CC Implementations on different systems

Figure 6.8 shows the average throughput for all the benchmarks across different configurations - Titan X GPU, Parallel and Serial CPU on Zurich. I calculate throughput in terms of number of edges processed per second for all the benchmarks. As seen in the above chart, ECL GPU code has the highest throughput of 5.295 billion edges per second. Most of the GPU benchmarks have higher throughput than the others. It is interesting to see that some of the serial codes have higher throughput than the parallel CPU codes.

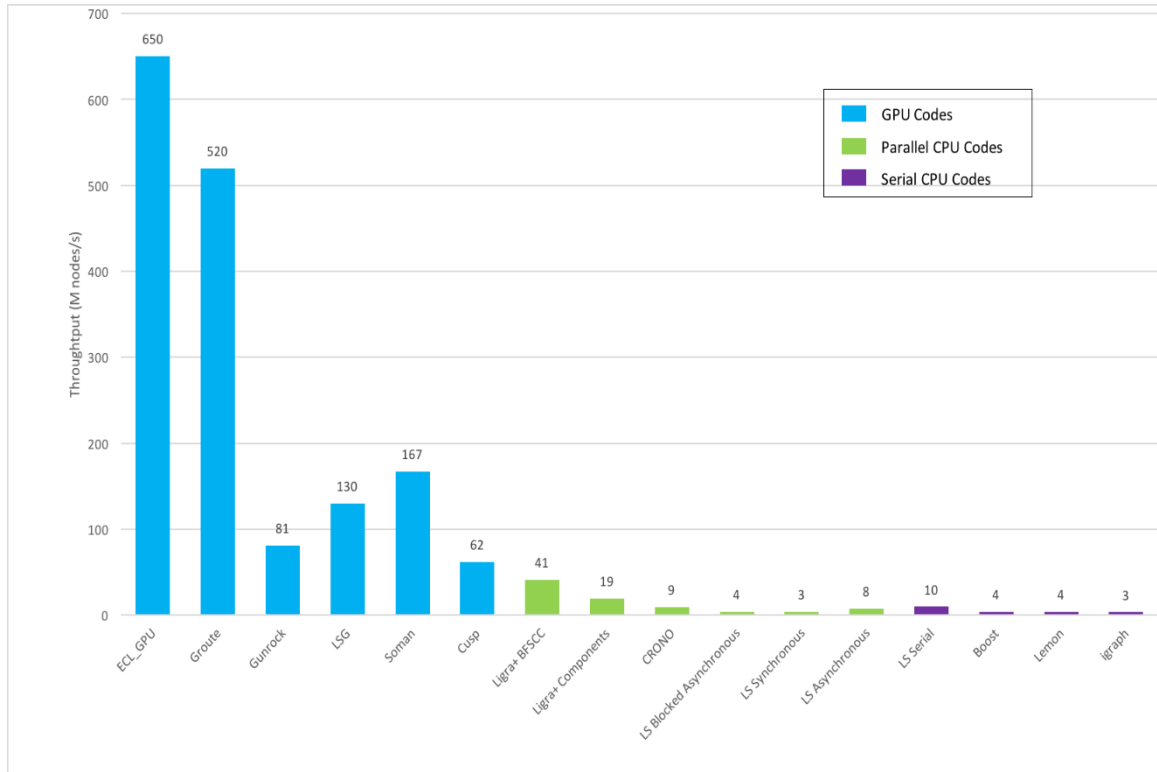


Figure 6.9 Throughput (Nodes/s) of various CC Implementations on different systems

Figure 6.9 shows the average throughput, calculated in terms of nodes processed per second, for all the benchmarks across different configurations - Titan X GPU, Parallel and Serial CPU on Zurich. As seen in the above chart, ECL GPU code has the highest throughput of 650 million nodes per second. Most of the GPU benchmarks have higher throughput than the others CPU and serial codes.

6.6 Performance Comparison - Atomic-Free Implementations

I implemented ECL_{af} , the atomic free CC algorithm, both in CUDA and OpenMP and compared its performance with parallel GPU and CPU benchmarks.

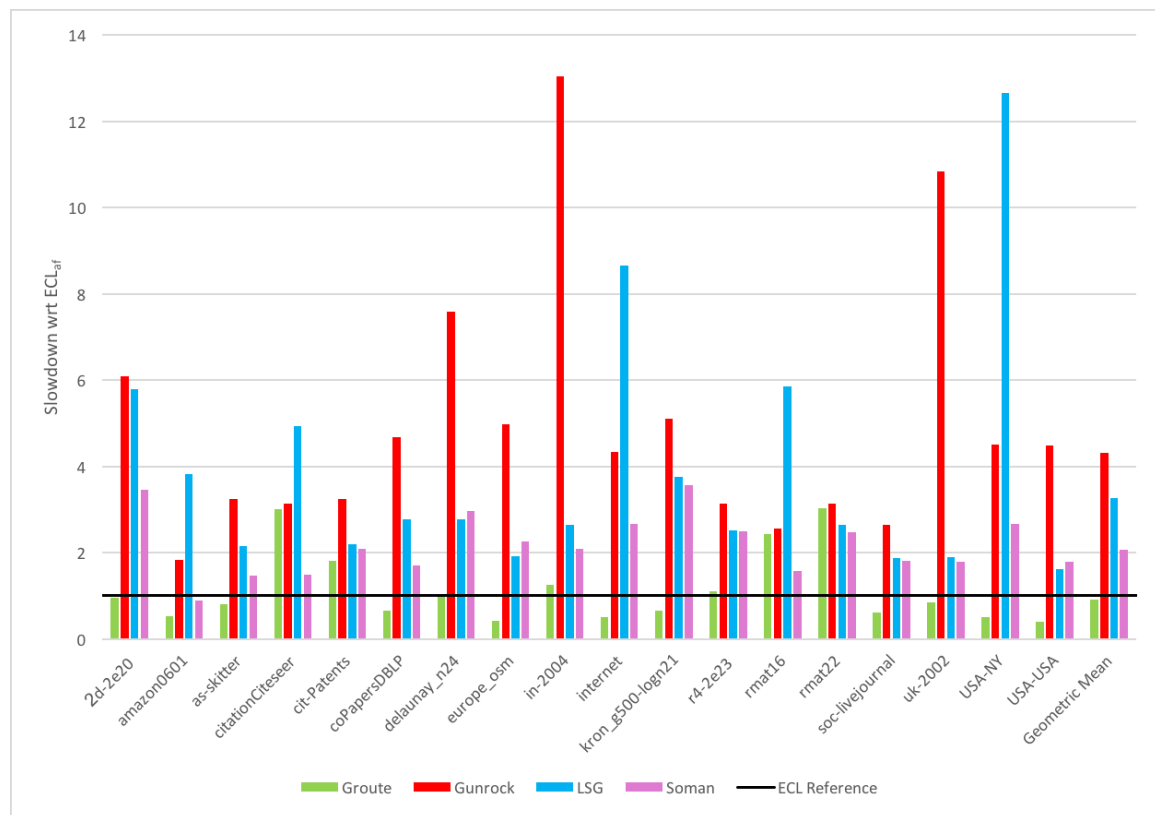


Figure 6.10 Parallel GPU Benchmarks Slowdowns relative to ECL_{af} on the Titan X

Figure 6.10 shows the slowdowns in GPU benchmarks with respect to ECL_{af} on Titan X. On Titan X, Groute is 1.1x faster than ECL_{af} on average and is faster on more than half of the input graphs. On the remaining graphs, ECL_{af} is 1.9x faster than Groute. On overall comparison, it is 2x faster than Soman's, 3.2x faster than LSG and 4.3x faster than Gunrock.

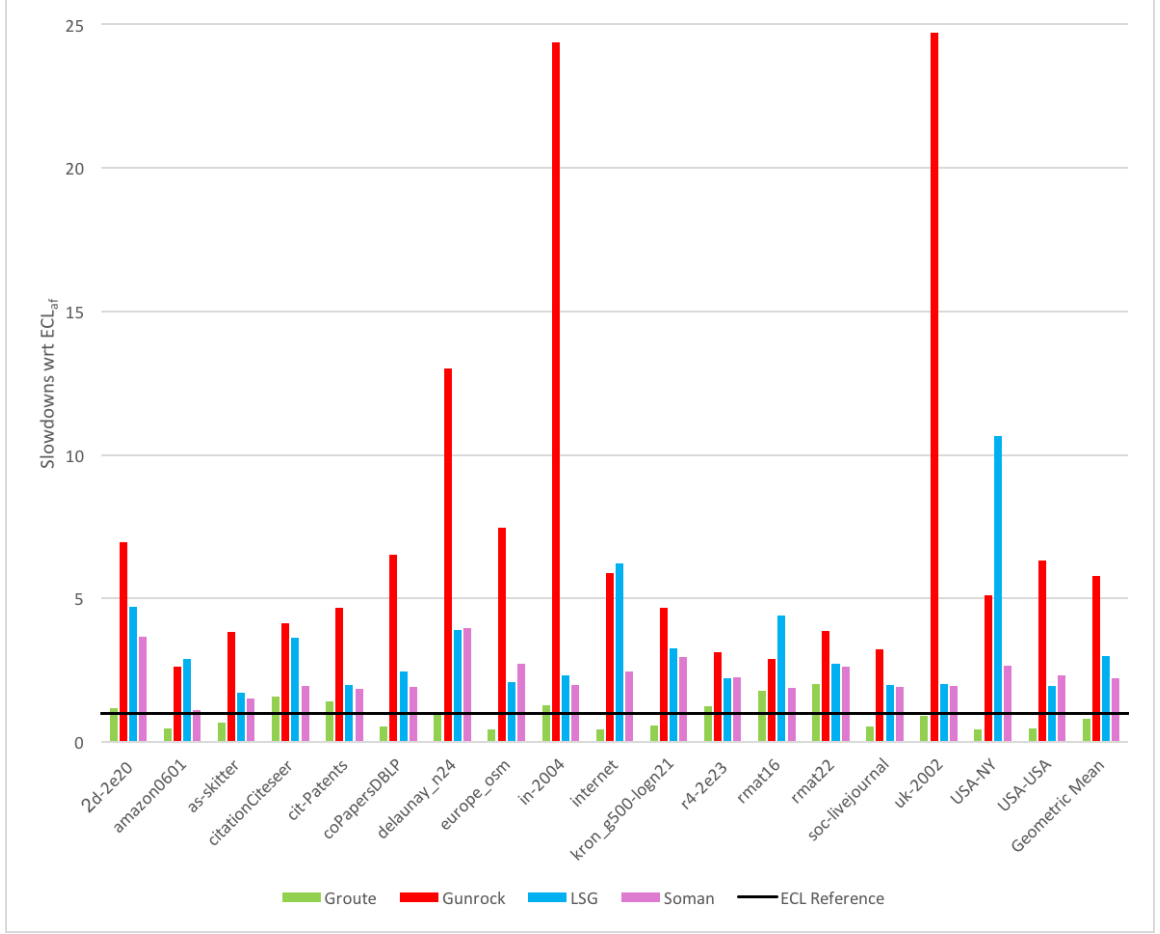


Figure 6.11 Parallel GPU Benchmarks Slowdowns relative to ECL_{af} on the K40

Figure 6.11 shows the slowdowns in GPU benchmarks with respect to ECL_{af} on K40.

Though Groute is 1.2x faster than ECL_{af} in overall performance, ECL_{af} beats Groute on half of the input graphs by an average of 1.4x. On overall comparison, ECL_{af} is 2.2x faster than Soman's, 2.9x faster than LSG and 5.7x faster than Gunrock.

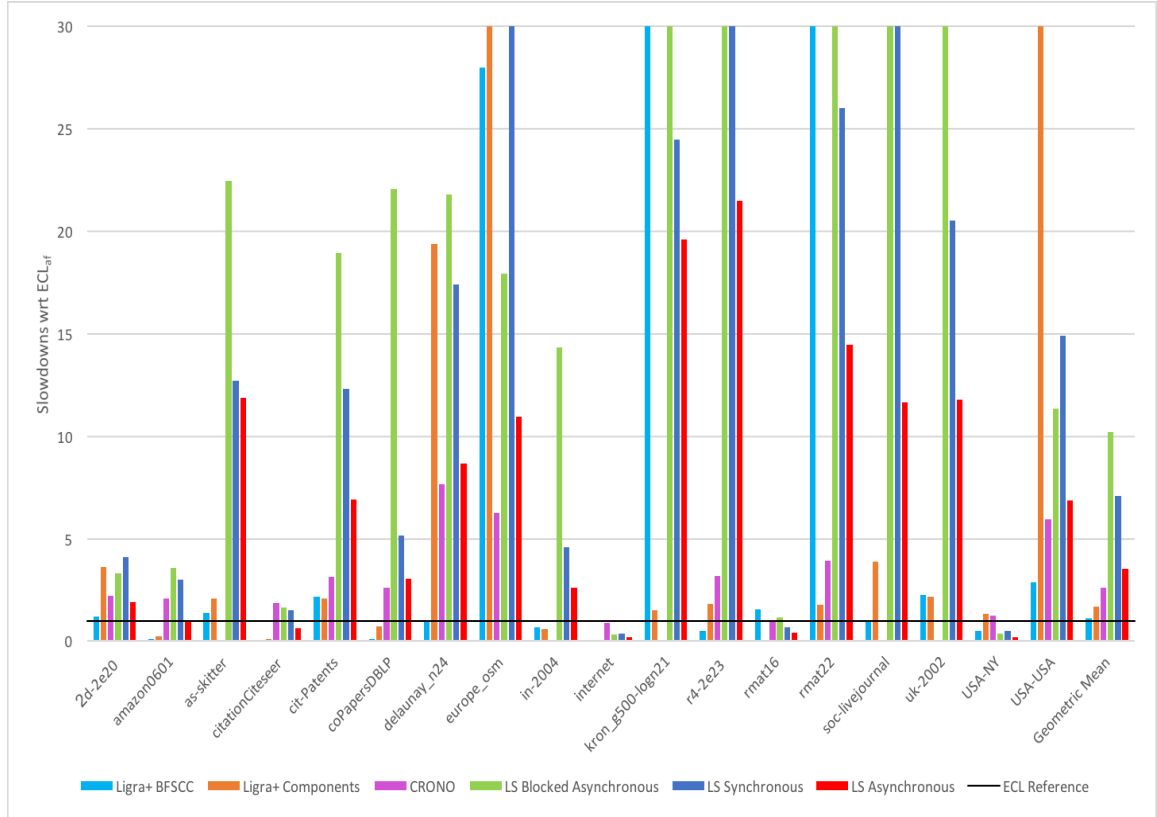


Figure 6.12 Parallel CPU Benchmarks Slowdowns relative to ECL_{af} on the Zurich
Note: Bars exceeding a height of 30 are cut off.

Figure 6.12 shows the slowdowns in Parallel CPU benchmarks with respect to ECL_{af} on Zurich. ECL_{af} is faster than Ligra+ BFSCC on half of the input graphs by 4.2x. On overall comparison, it is 1.1x faster than Ligra+ BFSCC, 1.7x faster than Ligra+ Components, 2.62x faster than CRONO, 3.5x faster than Asynchronous LS, 7.1x faster than Synchronous LS and 10.2x faster than Blocked Asynchronous LS.

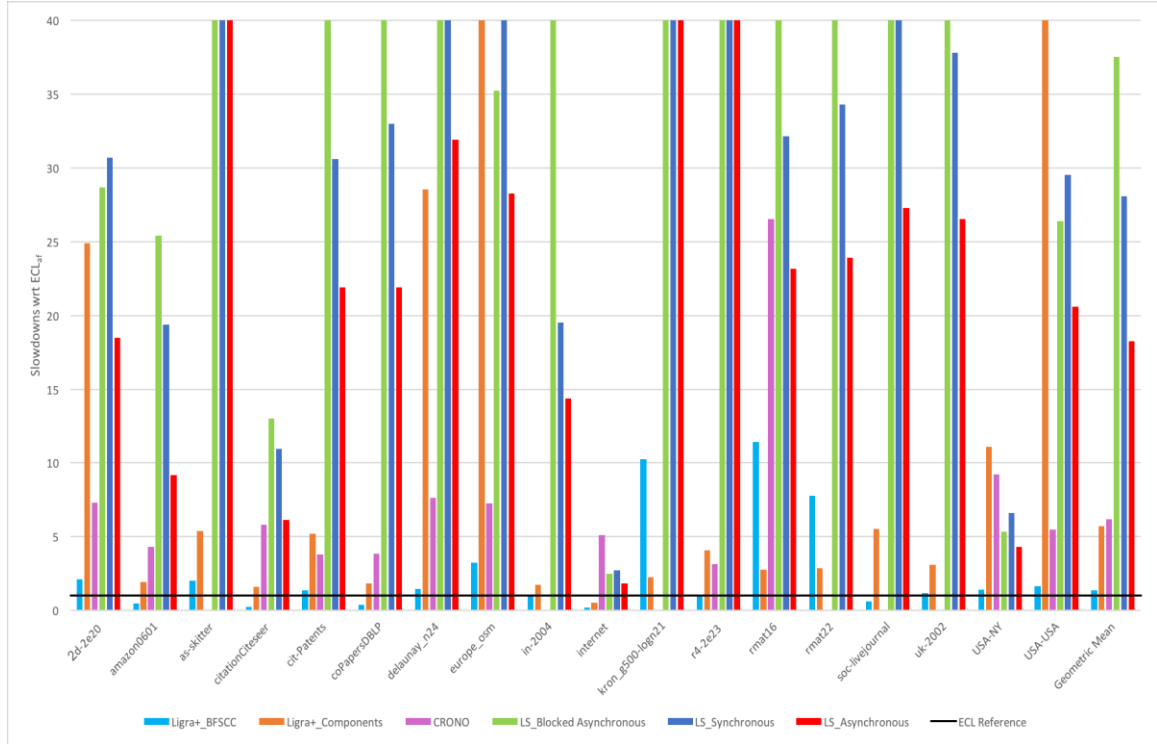


Figure 6.13 Parallel CPU Benchmarks Slowdowns relative to ECL_{af} on the Denver

Note: Bars exceeding a height of 40 are cut off.

Figure 6.13 shows the slowdowns in Parallel CPU benchmarks with respect to ECL_{af} in Denver. ECL_{af} is faster than the existing fastest CPU Benchmark, Ligra+ BFSCC, on more than half of the input graphs by 2.7x and it beats the other benchmarks on overall performance. On average, ECL_{af} is 1.4x faster than Ligra+'s BFSCC, 5.7x faster than Ligra+'s Components, 6.1x faster than CRONO, 18.2x faster than Asynchronous LS, 28.1x faster than Synchronous LS and 37.6x faster than Blocked Asynchronous LS.

CHAPTER 7

SUMMARY

7.1 Summary

Computing the connected components is an important graph application used in various fields. This thesis proposes two efficient algorithms for connected components - ECL and ECL_{af}. ECL is an asynchronous algorithm suitable for massively parallel devices. My CUDA implementation of ECL includes optimizations to improve load-balance and uses a variant of pointer jumping to speed up the calculation of the connected components. I also ported the ECL algorithm to parallel and serial CPU devices, where I implemented it in OpenMP and C, respectively.

Atomic operations play a significant role in many algorithms specific to parallel devices, as they prevent data races. However, these operations incur overhead as they briefly serialize the threads that execute atomic operations. ECL_{af} is an atomic-free synchronous implementation, which reiterates over kernel(s) to solve the connected components problem. I implemented it for GPUs and parallel CPU devices. I tested both algorithms on 18 input graphs and compared their performance with corresponding programs from the literature. ECL shows a significant improvement in performance and, on average, it outperforms the existing fastest GPU algorithm by 1.7x. On CPUs, it is 1.6x faster than the fastest parallel CPU algorithm from the literature. The serial implementation is 5x faster than the fastest preexisting serial CC algorithm. Though ECL_{af} is atomic-free and therefore needs to perform multiple rounds of computation, its CUDA implementation shows comparable performance with the fastest GPU algorithm

and its OpenMP-implementation is 1.2x faster than the fastest parallel CPU algorithm.

7.2 Future Work

In the future, researchers could evaluate the proposed connected components algorithms and optimize them for other architectures such as AMD CPUs and GPUs as well as ARM CPUs. Researchers could also extend these algorithms to multi-GPU and/or Xeon-Phi-based systems. It might also be interesting to study the energy efficiency in addition to the runtime and adding optimizations to improve that aspect.

APPENDIX SECTION

This section contains the runtimes for all the ECL codes and their corresponding Benchmarks.

Table A.1 Parallel GPU Code Runtimes on the Titan X

	ECL	ECL_{af}	Groute	Gunrock	LSG	Soman	Cusp
2d-2e20	0.0012	0.0020	0.0019	0.0122	0.0116	0.0069	0.0278
amazon0601	0.0009	0.0017	0.0009	0.0031	0.0065	0.0015	0.0513
as-skitter	0.0025	0.0025	0.0020	0.0081	0.0054	0.0037	4.4829
citationCiteseer	0.0006	0.0014	0.0042	0.0044	0.0069	0.0021	0.0110
cit-Patents	0.0146	0.0322	0.0586	0.1046	0.0709	0.0676	21.5052
coPapersDBLP	0.0025	0.0043	0.0028	0.0201	0.0119	0.0073	0.0182
delaunay_n24	0.0146	0.0225	0.0221	0.1706	0.0622	0.0667	0.0656
europe_osm	0.0290	0.0516	0.0217	0.2569	0.0995	0.1168	0.2388
in-2004	0.0026	0.0058	0.0073	0.0757	0.0154	0.0121	0.8626
internet	0.0002	0.0006	0.0003	0.0026	0.0052	0.0016	0.0091
kron_g500-logn21	0.0209	0.0376	0.0246	0.1918	0.1410	0.1345	2164.2301
r4-2e23	0.0216	0.0216	0.0238	0.0676	0.0546	0.0538	0.0503
rmat16	0.0003	0.0007	0.0017	0.0018	0.0041	0.0011	19.5460
rmat22	0.0205	0.0415	0.1261	0.1299	0.1095	0.1024	1700.5245
soc-livejournal	0.0127	0.0229	0.0140	0.0604	0.0431	0.0413	12.5244
uk-2002	0.0458	0.1282	0.1085	1.3891	0.2438	0.2297	494.1125
USA-NY	0.0002	0.0006	0.0003	0.0027	0.0076	0.0016	0.0133
USA-USA	0.0141	0.0374	0.0153	0.1680	0.0604	0.0671	0.0697

Table A.2 Parallel GPU Code Runtimes on the K40

	ECL	ECL_{af}	Groute	Gunrock	LSG	Soman	Cusp
2d-2e20	0.0022	0.0034	0.0040	0.0237	0.0160	0.0125	0.0329
amazon0601	0.0011	0.0025	0.0012	0.0066	0.0072	0.0028	0.0377
as-skitter	0.0041	0.0046	0.0031	0.0176	0.0079	0.0070	5.0367
citationCiteseer	0.0009	0.0020	0.0032	0.0083	0.0073	0.0039	0.0105
cit-Patents	0.0194	0.0458	0.0647	0.2150	0.0904	0.0848	23.8706
coPapersDBLP	0.0045	0.0074	0.0040	0.0484	0.0183	0.0141	0.0261
delaunay_n24	0.0210	0.0309	0.0324	0.4026	0.1210	0.1230	0.1184
europa_osm	0.0442	0.0783	0.0349	0.5846	0.1645	0.2134	0.3267
in-2004	0.0046	0.0096	0.0123	0.2339	0.0222	0.0192	0.9114
internet	0.0003	0.0009	0.0004	0.0053	0.0056	0.0022	0.0081
kron_g500- logn21	0.0401	0.0749	0.0436	0.3511	0.2452	0.2216	2743.4945
r4-2e23	0.0298	0.0298	0.0375	0.0931	0.0667	0.0672	0.0731
rmat16	0.0004	0.0010	0.0018	0.0029	0.0044	0.0019	21.5828
rmat22	0.0270	0.0553	0.1122	0.2141	0.1505	0.1457	1941.3319
soc-livejournal	0.0201	0.0350	0.0186	0.1134	0.0698	0.0678	13.9202
uk-2002	0.0835	0.2376	0.2129	5.8692	0.4801	0.4668	580.6068
USA-NY	0.0004	0.0009	0.0004	0.0046	0.0096	0.0024	0.0146
USA-USA	0.0238	0.0573	0.0275	0.3624	0.1111	0.1328	0.1151

Table A.3 Parallel CPU Code Runtimes on Zurich

	ECL	Ligra+ 1	Ligra+ 2	CRONO	LS_1	LS_2	LS_3
2d-2e20	0.0489	0.0888	0.2628	0.1607	0.2407	0.2967	0.2967
amazon0601	0.0477	0.0053	0.0116	0.1100	0.1874	0.1573	0.1573
as-skitter	0.0637	0.0756	0.1126	NA	1.2143	0.6879	0.6879
Citation Citeseer	0.0550	0.0026	0.0064	0.1097	0.0963	0.0901	0.0901
cit-Patents	0.1108	0.3334	0.3191	0.4839	2.9170	1.8954	1.8954
coPapersDBLP	0.0731	0.0062	0.0471	0.1752	1.4819	0.3461	0.3461
delaunay_n24	0.1373	0.2021	3.8830	1.5323	4.3657	3.4851	3.4851
europe_osm	0.1787	7.0460	26.5400	1.5742	4.5203	7.9685	7.9685
in-2004	0.0525	0.0465	0.0410	NA	0.9930	0.3160	0.3160
internet	0.0377	0.0024	0.0030	0.0431	0.0158	0.0172	0.0172
kron_g500- logn21	0.1173	6.1170	0.2280	NA	14.5817	3.7193	3.7193
r4-2e23	0.1198	0.0590	0.2113	0.3631	5.2150	7.9287	7.9287
rmat16	0.0387	0.0512	0.0018	0.0341	0.0383	0.0232	0.0232
rmat22	0.0830	6.3520	0.2745	0.5992	6.9970	3.9790	3.9790
soc-livejournal	0.0890	0.1231	0.5110	NA	6.6913	4.3803	4.3803
uk-2002	0.1655	0.7545	0.7275	NA	29.7646	6.8136	6.8136
USA-NY	0.0278	0.0292	0.0812	0.0741	0.0216	0.0305	0.0305
USA-USA	0.1170	0.6957	43.1600	1.4374	2.7470	3.6081	3.6081

Ligra+ 1 - Ligra+ BFSCC
 Ligra+ 2 - Ligra+ Components
 LS_1 - LS Blocked Asynchronous
 LS_2 - LS Synchronous
 LS_3 - LS Asynchronous

Table A.4 Parallel CPU Code Runtimes on Denver

	ECL	Ligra+ 1	Ligra+ 2	CRONO	LS_1	LS_2	LS_3
2d-2e20	0.0273	0.0576	0.6800	0.1993	0.7829	0.8382	0.5053
amazon0601	0.0182	0.0088	0.0349	0.0780	0.4623	0.3530	0.1667
as-skitter	0.0262	0.0529	0.1410	NA	2.5565	1.6440	1.4935
Citation Citeseer	0.0175	0.0043	0.0277	0.1015	0.2275	0.1917	0.1074
cit-Patents	0.1564	0.2160	0.8130	0.5934	6.3997	4.7877	3.4271
coPapersDBLP	0.0329	0.0117	0.0597	0.1262	3.2563	1.0859	0.7203
delaunay_n24	0.1545	0.2260	4.4100	1.1776	7.7839	6.7026	4.9345
europe_osm	0.2874	0.9260	30.8000	2.0876	10.1316	15.8009	8.1270
in-2004	0.0471	0.0442	0.0826	NA	1.9132	0.9191	0.6771
internet	0.0132	0.0026	0.0067	0.0676	0.0329	0.0360	0.0243
kron_g500- logn21	0.2179	2.2300	0.4920	NA	29.1799	10.8242	9.6110
r4-2e23	0.1340	0.1310	0.5460	0.4207	9.9513	13.8737	6.3252
rmat16	0.0017	0.0194	0.0047	0.0451	0.0694	0.0547	0.0394
rmat22	0.2718	2.1100	0.7790	NA	15.4968	9.3222	6.5073
soc-livejournal	0.1756	0.1070	0.9720	NA	14.3023	9.3904	4.7901
uk-2002	0.4532	0.5270	1.4100	NA	55.7538	17.1402	12.0359
USA-NY	0.0101	0.0141	0.1120	0.0932	0.0541	0.0666	0.0433
USA-USA	0.2399	0.3980	70.2000	1.3166	6.3369	7.0886	4.9382

Ligra+ 1 - Ligra+ BFSCC
 Ligra+ 2 - Ligra+ Components
 LS_1 - LS Blocked Asynchronous
 LS_2 - LS Synchronous
 LS_3 - LS Asynchronous

Table A.5 Serial CPU Code Runtimes on Zurich

	ECL	LS Serial	Boost	Lemon	igraph
2d-2e20	0.0427	0.1179	0.2813	0.2687	0.3192
amazon0601	0.0175	0.0469	0.1546	0.3517	0.1741
as-skitter	0.0562	0.6048	0.5116	1.4873	0.4849
citationCiteseer	0.0244	0.0315	0.0795	0.1722	0.1040
cit-Patents	0.2688	0.9639	1.9611	3.7353	2.3449
coPapersDBLP	0.0666	0.2025	0.3595	1.2438	0.4880
delaunay_n24	0.5100	1.4646	2.9137	2.5452	3.5161
europe_osm	0.8898	2.0601	5.8685	6.7573	10.8726
in-2004	0.0818	0.1675	0.2610	0.6426	0.3924
internet	0.0098	0.0057	0.0155	0.0087	0.0140
kron_g500-logn21	0.4472	2.9119	2.4865	17.9984	6.4585
r4-2e23	0.5136	2.3232	3.4911	8.3909	5.9166
rmat16	0.0072	0.0120	0.0241	0.0385	0.0243
rmat22	0.4585	2.0729	2.8357	7.9761	3.7217
soc-livejournal	0.4050	1.4531	2.4589	8.3611	3.6557
uk-2002	1.0040	3.7731	5.7324	15.7312	13.7288
USA-NY	0.0133	0.0082	0.0422	0.0176	0.0303
USA-USA	0.5993	1.2499	3.8238	2.4520	3.7154

Table A.6 Serial CPU Code Runtimes on Denver

	ECL	LS Serial	Boost	Lemon	igraph
2d-2e20	0.0762	0.1141	0.2789	0.3010	0.3137
amazon0601	0.0312	0.0465	0.1285	0.3588	0.1760
as-skitter	0.0804	0.5945	0.4813	1.5809	0.4887
citationCiteseer	0.0191	0.0317	0.0857	0.1829	0.1043
cit-Patents	0.2095	0.9381	1.9496	3.8537	2.3879
coPapersDBLP	0.0659	0.2021	0.3675	1.2395	0.4811
delaunay_n24	0.5203	1.5231	2.7283	3.1990	3.4651
europe_osm	0.8984	2.0191	5.5534	8.1286	10.8305
in-2004	0.0550	0.1682	0.3324	0.6387	0.3887
internet	0.0100	0.0057	0.0149	0.0132	0.0224
kron_g500-logn21	0.4738	2.8016	2.5766	17.8190	6.7094
r4-2e23	0.4019	2.2554	3.4077	8.8993	6.0508
rmat16	0.0028	0.0121	0.0218	0.0689	0.0304
rmat22	0.2987	2.0697	2.8030	8.2363	3.9400
soc-livejournal	0.2449	1.4759	2.4893	8.2123	3.6784
uk-2002	1.0063	3.5618	5.4292	15.9495	8.5397
USA-NY	0.0118	0.0082	0.0420	0.0290	0.0305
USA-USA	0.5832	1.3121	3.6517	3.2654	3.8020

LITERATURE CITED

- [1] Shiloach, Yossi, and Uzi Vishkin. "An $O(\log n)$ parallel connectivity algorithm." *Journal of Algorithms* 3.1 (1982): 57-67.
- [2] Soman, Kishore, and Narayanan. "A fast GPU algorithm for graph connectivity." (2010).
- [3] Wang, Yangzihao, et al. "Gunrock: A high-performance graph processing library on the GPU." *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016.
- [4] Shun, Julian, and Guy E. Blelloch. "Ligra: a lightweight graph processing framework for shared memory." *ACM SIGPLAN Notices*. Vol. 48. No. 8. ACM, 2013.
- [5] Shun, Julian, Laxman Dhulipala, and Guy E. Blelloch. "Smaller and faster: Parallel processing of compressed graphs with Ligra+." *Data Compression Conference (DCC)*, 2015. IEEE, 2015.
- [6] Cormen, Thomas H. *Introduction to algorithms*. MIT press, 2009.
- [7] Connectivity - [https://en.wikipedia.org/wiki/Connectivity_\(graph_theory\)](https://en.wikipedia.org/wiki/Connectivity_(graph_theory))
- [8] Sutton, Michael, et al. "Adaptive Work-Efficient Connected Components on the GPU." *arXiv preprint arXiv:1612.01178* (2016).
- [9] Boost: <http://www.boost.org/>
- [10] CUSP: <http://cusplibrary.github.io/index.html>
- [11] Burtcher, Martin, Rupesh Nasre, and Keshav Pingali. "A quantitative study of irregular programs on GPUs." *Workload Characterization (IISWC)*, 2012 IEEE International Symposium on. IEEE, 2012.

- [12] Ahmad, Masab, et al. "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores." Workload Characterization (IISWC), 2015 IEEE International Symposium on. IEEE, 2015.
- [13] igraph: <https://github.com/igraph>
- [14] Lemon: <http://lemon.cs.elte.hu/pub/tutorial/index.html>