Project Report:

Parallelizing Graph Algorithms

Samyak S Sarnayak - PES1201801565
Aditi Ahuja - PES1201800165
Pranav Kesavarapu - PES1201800299

Overview

- Parallelized Pagerank on CPU and GPU
- Using Go - parallelising using goroutines, channels and similar Go constructs.
- Using compressed sparse row (CSR) representation of graphs
- Benchmark-based optimization to iteratively get better results
- Compared cache characteristics to explain results

Datasets

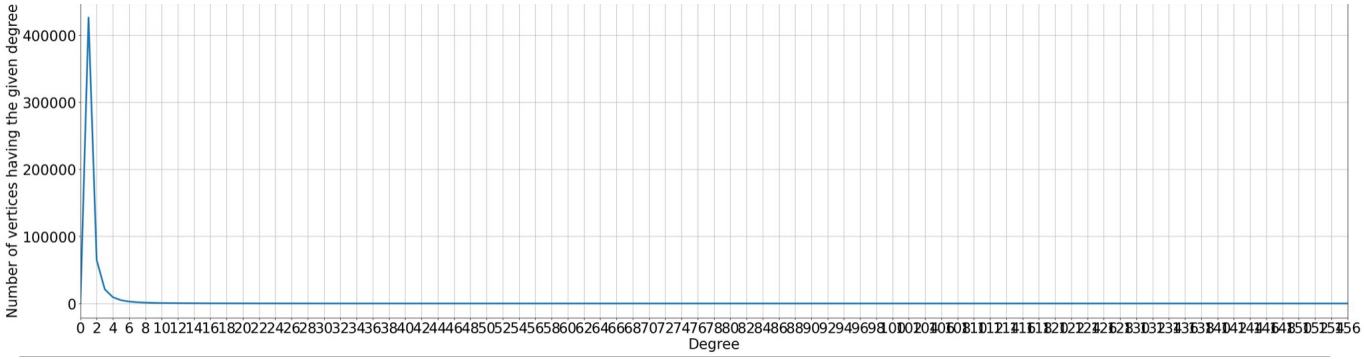
- Large Graph
- Quora
- Wikipedia Vote
- Stanford Web

Large Graph Dataset

- Edges: 46092177
- Vertices: 2080370
- Maximal density: 10^-5

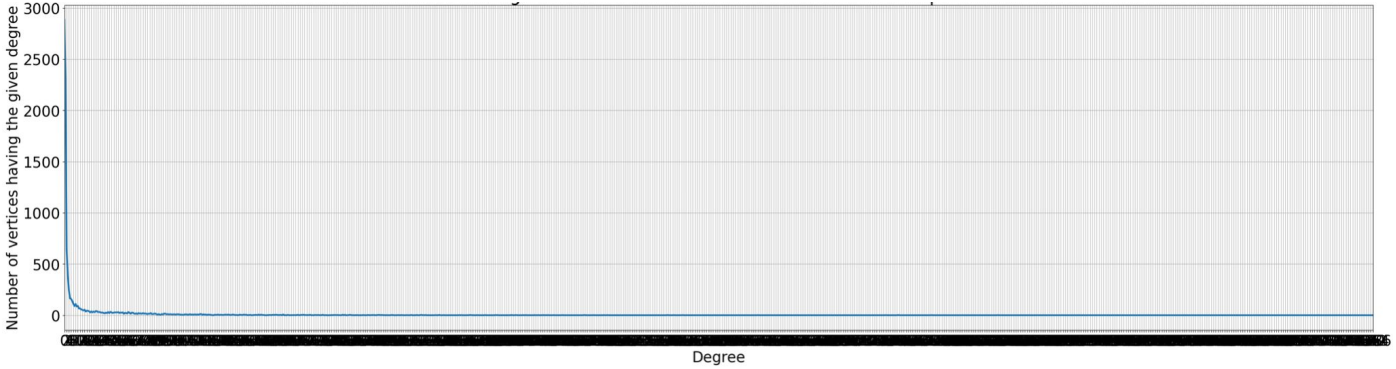
Quora Question Dataset

- Edges: 404291
- Vertices: 537935
- Maximal density: 2.8e-06
- Average degree: 1.5
- Maximum degree: 157
- 426153 vertices have degree 1



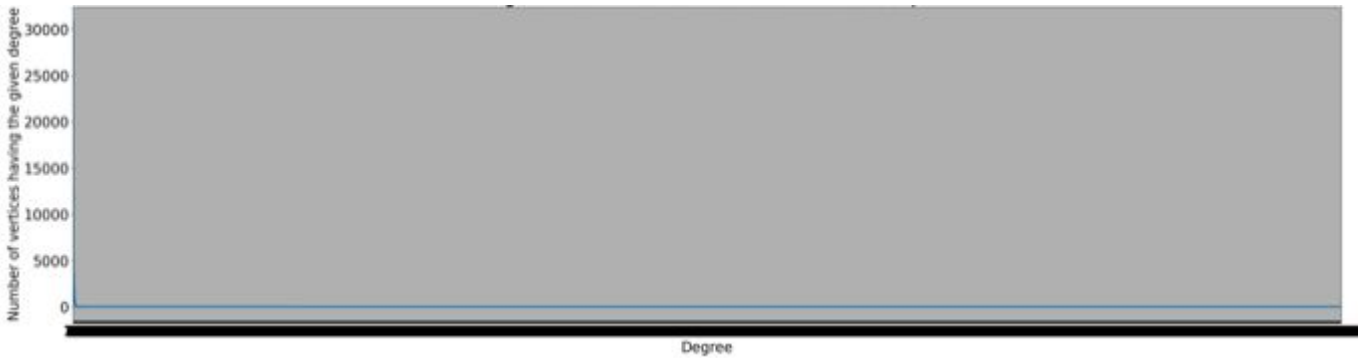
Wikipedia Vote Dataset

- Edges: 103689
- Vertices: 7115
- Maximal density: 0.0021
- Average degree: 20.7378
- Maximum degree: 1167
- 2885 vertices have degree 0



Stanford Web Dataset

- Edges: 2312497
- Vertices: 281903
- Maximal Density: 5.82e-05
- Average degree: 16.4
- Maximum degree: 38626
- 30870 vertices have the degree 7.



PageRank - Iterations

- Standard pagerank with adjacency arrays formed from edge lists.
- First attempt at parallelizing gave ~20x slowdown
- Progressed to topological pagerank, which we parallelised.
- Implemented partitioning based parallelisation.

PageRank

- The formula we used for pagerank computation is as follows.

$$P_i = \frac{1-\alpha}{N} + \alpha \sum_{j \in I_i} \frac{P_j}{\sum_{k \in O_j} 1} + \alpha \frac{\sum_{j \in D} P_j}{N}$$

where P is the pagerank vector, O_j is the set of out-neighbours of j and I_i is the set of in-neighbours of i, D is the set of vertices with no out-links (dangling vertices). α is the damping factor and N is the number of vertices.

- We used an iterative, benchmark driven procedure to optimize our parallel implementation of pagerank

PageRank - Topological

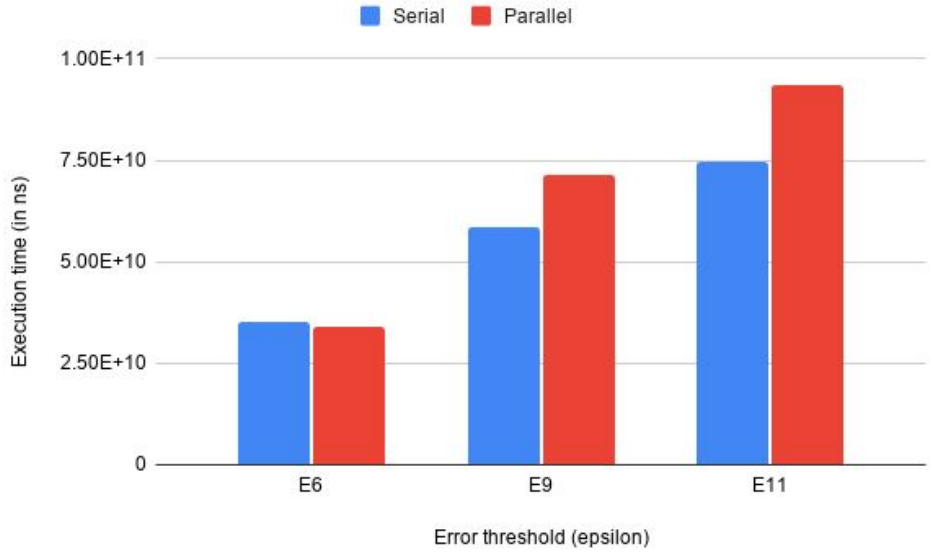
- Topo pagerank processes all vertices of a graph in one iteration.
- Vectorized version of the standard pagerank.
- Potential issues include random DRAM accesses when accessing the neighbours of a vertex, since the neighbours may be located anywhere in the DRAM.
- This leads to poor locality of the data being accessed.

PageRank Results 1 - Topological

Initial optimized implementation with adjacency lists, a new goroutine is launched for every node's calculation

Max Speedup:

1.03x

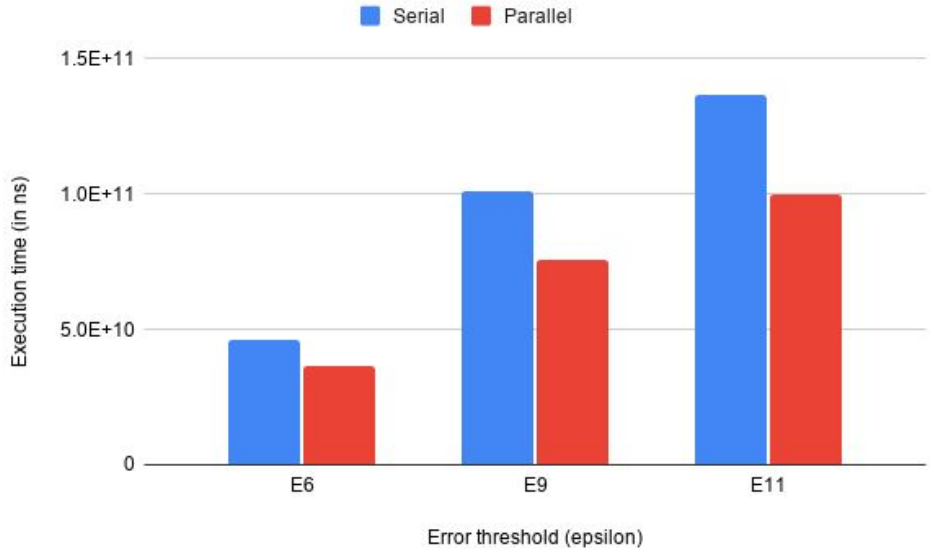


PageRank Results 2

Fixed number of goroutines launched initially, data sent via a channel to them (fanout pattern)

Max Speedup:

1.36x



PageRank - Partitioning

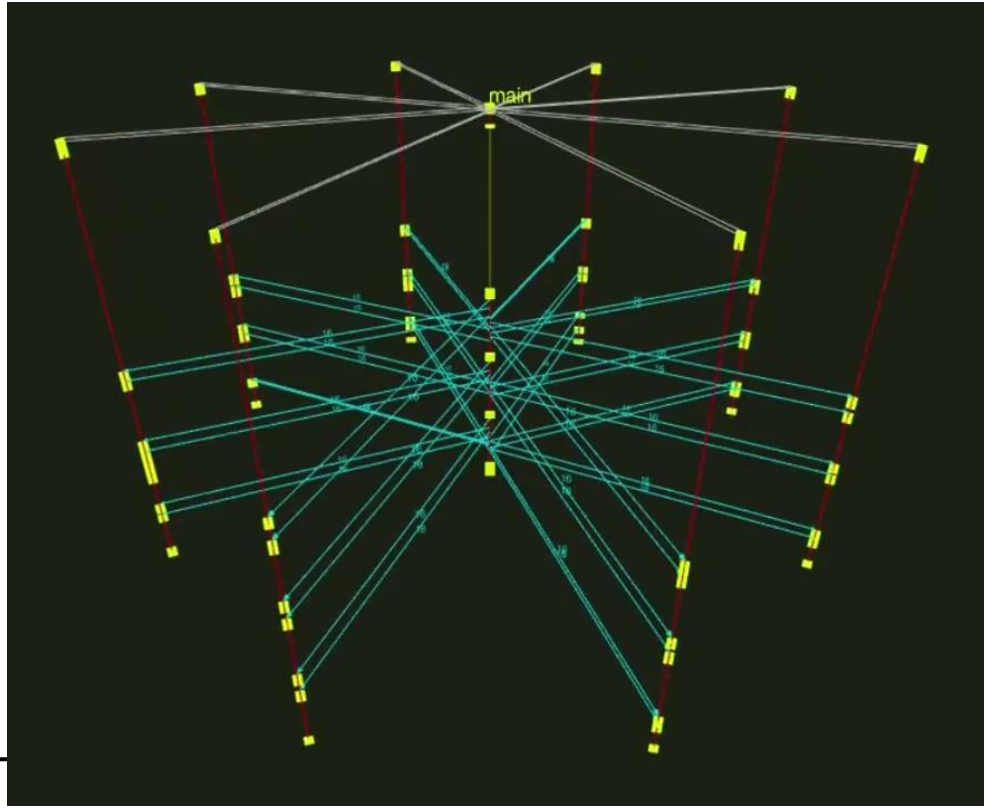
- Since the data sent via channel was fixed in every iteration, we saw an opportunity for optimization here
- Partitioning splits each block of nodes and allocates it to a specific goroutine.
- Computations are performed separately for each slice and then combined at the end.

PageRank - Visualisation

- A visualisation of the goroutines
- Yellow indicates parts where the CPU is computing/busy
- Blue arrows are channel sends/receives

- The fan-out pattern is seen here
- Synchronization between iterations can also be seen

Link to video: https://drive.google.com/file/d/1rM5dmKO19X6jmMyLS7-QVWVAn28xnACi/view?resourcekey

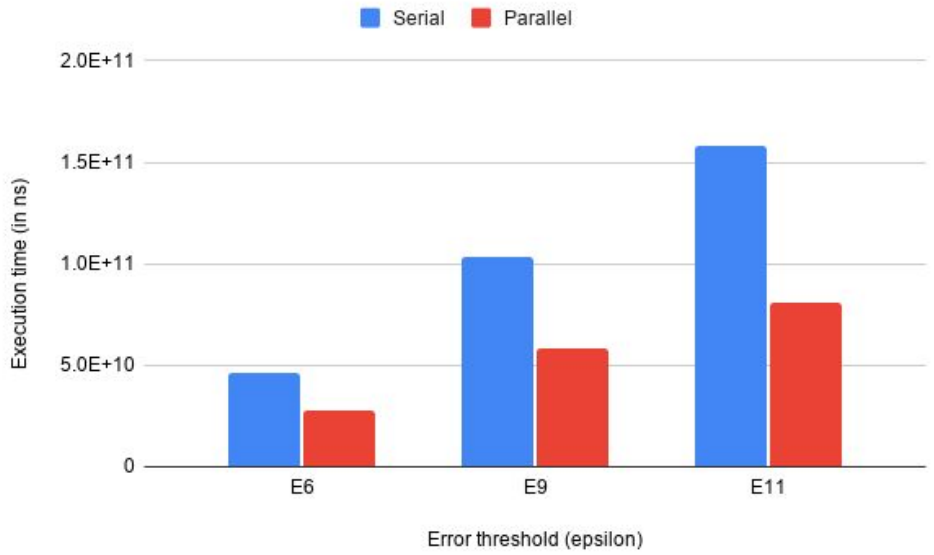


PageRank Results 3 - Partitioning

Fixed number of goroutines launched initially, each goroutine is assigned a fixed partition of nodes, use signal channels for communication

Max Speedup:

`1.95x`

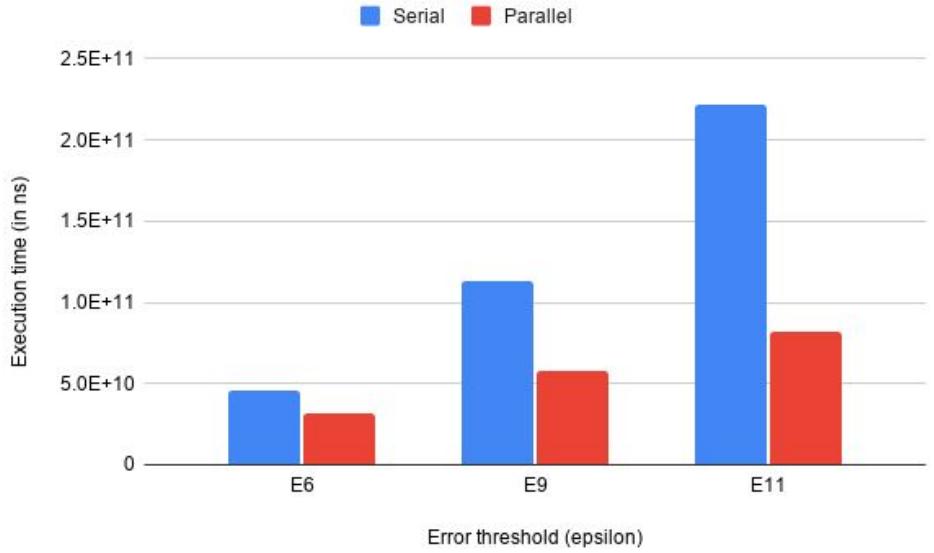


PageRank Results 4

Parallelized initialisations, leak calculations, delta calculations

Max Speedup:

2.71x

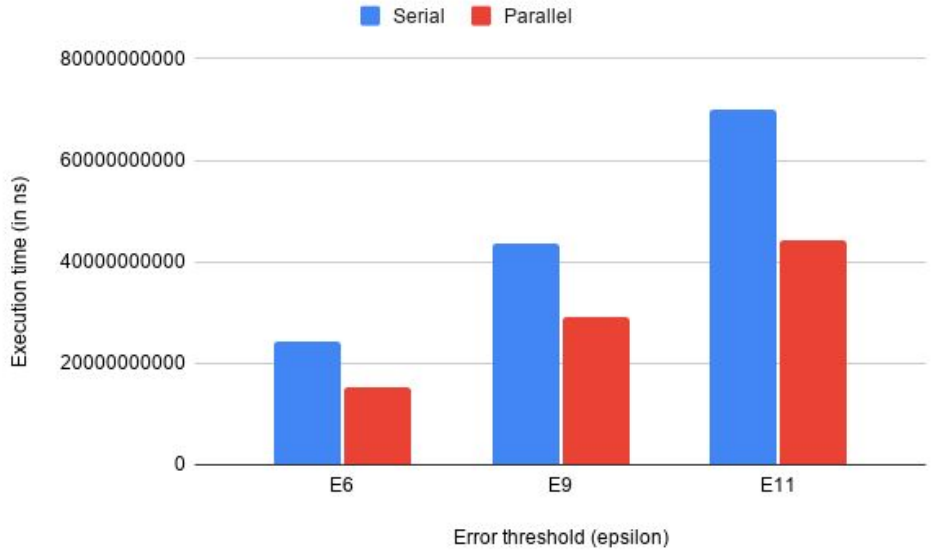


PageRank Results 5

Used compressed sparse row (CSR) representation of graph. This also gave a boost to serial implementation.

Max Speedup:

1.58x

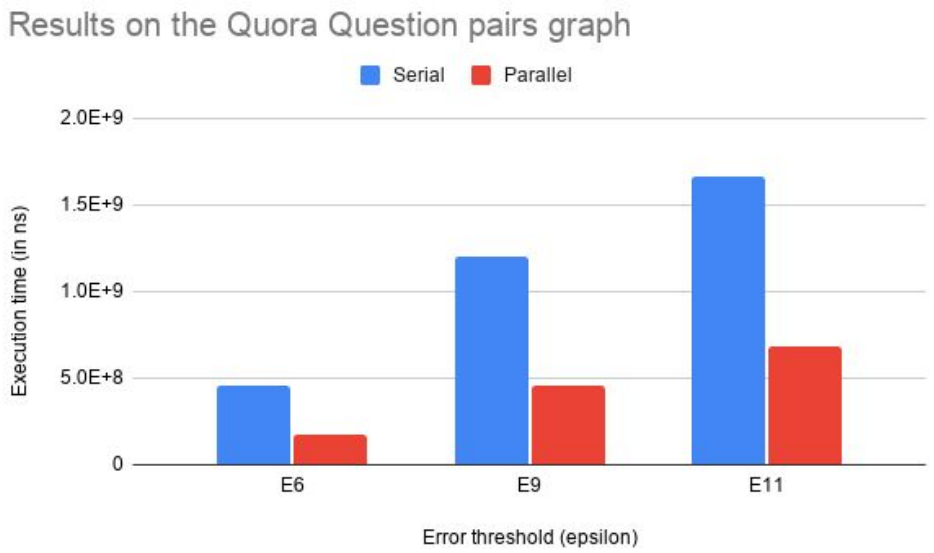


PageRank Results 6

Same as previous, tested on the Quora Question pairs dataset.

Max Speedup:

2.64x

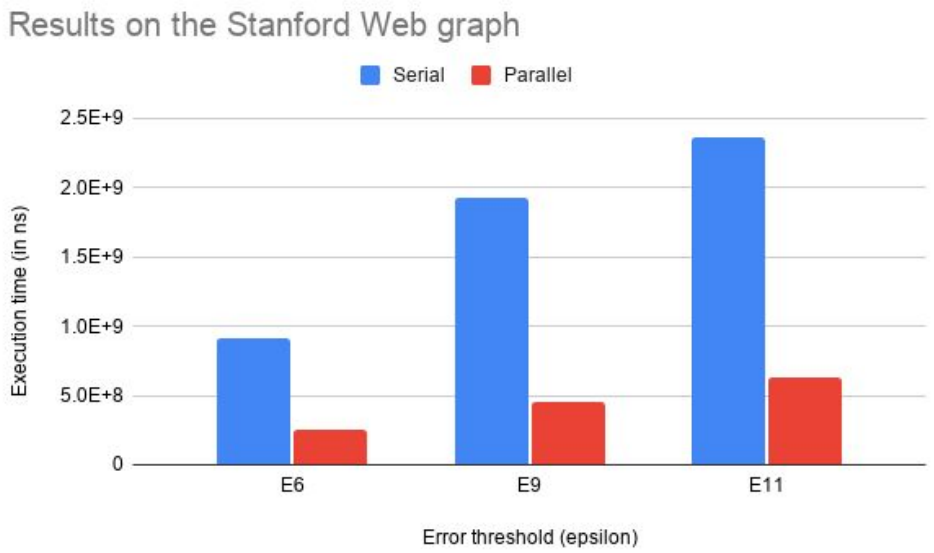


PageRank Results 7

Same as previous, tested on the Stanford Web graph

Max Speedup:

4.23x



CUDA

CUDA Implementation of Pagerank Algorithm

Cuda tested on

- Stanford Web Graph
- Large Graph

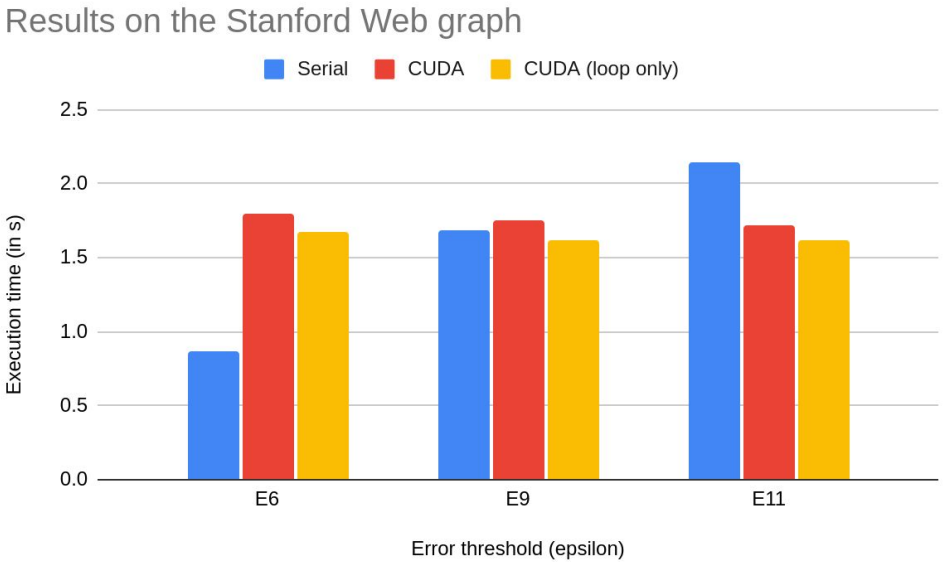
CUDA CGO Implementation

- Taking the serial CSR implementation of Pagerank, we implemented a CUDA version of it.
- Initially we tried to use CGO binding to interface between Golang and CUDA.
- CGO is an interface for interacting with C code.
- The CGO CUDA implementation gave anywhere between 20 to 200 times slowdown.
- Then we switched to another package called CU, which offers go native interface with CUDA.

CUDA CU Implementation

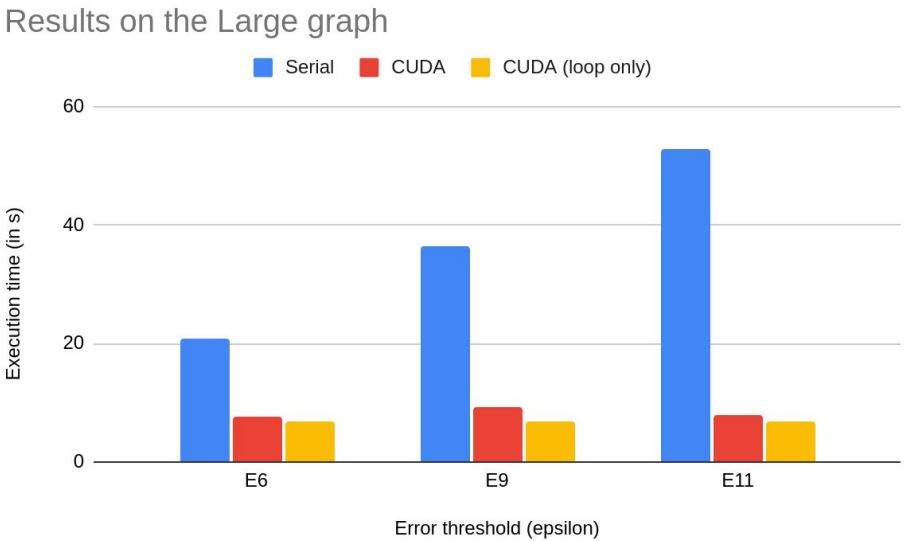
- With the CU Implementation we were still getting a slow down with the smaller data sets.
- This is due to the overhead of memory transfer between device and host.
- After testing on larger datasets, the CU Implementation started to show signs of small speed up.
- After optimizing the memory transfer and adding an extra kernel, we have now achieved a significant speed up.

CUDA on Stanford Result



- As the dataset size increases, we start to see speedups
- Timing only the inner loop shows more speedups as memory transfers take up much of the time

CUDA on Large Result



- We see much better speedups with the large graph as memory transfer times are smaller compared to the computation loop

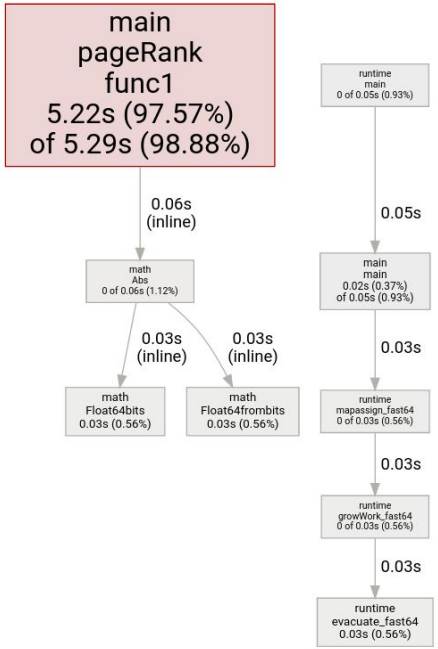
Profiling

Tools used for profiling:

- igraph - python library for graph characteristics
- Cachegrind (a part of valgrind) - for cache misses
- pprof - profiling tool built into Go. Includes both a CPU and memory profiler. Useful for finding which functions take most CPU time
- trace - tracing tool built into Go. This shows us a breakdown of all goroutines and threads.

Pagerank efficiency

Profiling the pageRank function with pprof shows that it takes most of the execution time with almost no runtime overhead

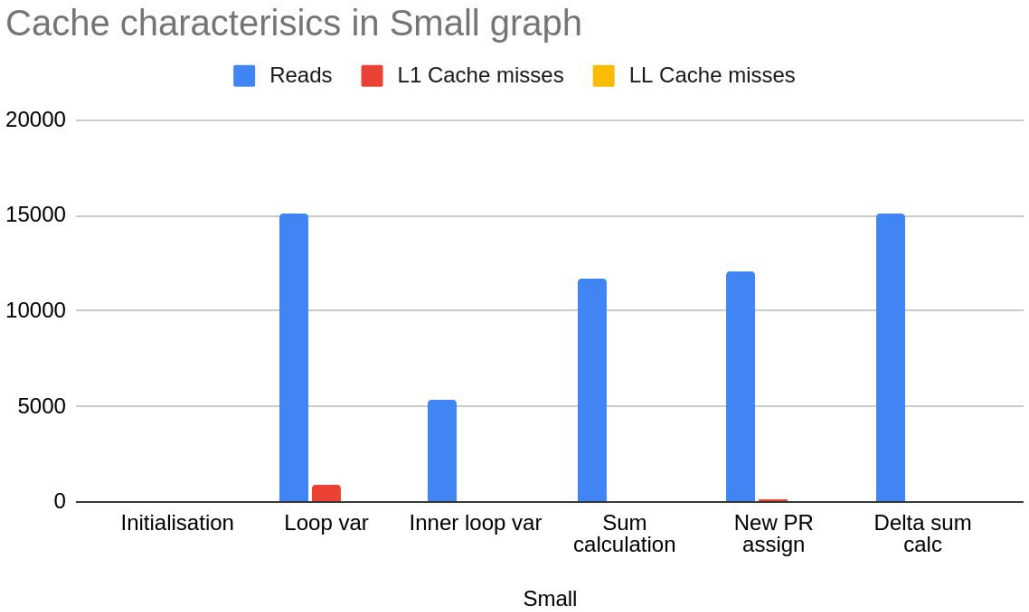



```
(pprof) top10
Showing nodes accounting for 5.33s, 99.63% of 5.35s total
Dropped 11 nodes (cum <= 0.03s)
      flat  flat%   sum%        cum   cum%
     5.22s 97.57% 97.57%      5.29s 98.88%  main.pageRank.func1
     0.03s  0.56% 98.13%      0.03s  0.56%  math.Float64bits (inline)
     0.03s  0.56% 98.69%      0.03s  0.56%  math.Float64frombits (inline)
     0.03s  0.56% 99.25%      0.03s  0.56%  runtime.evacuate_fast64
     0.02s  0.37% 99.63%      0.05s  0.93%  main.main
          0     0% 99.63%      0.06s  1.12%  math.Abs (inline)
          0     0% 99.63%      0.03s  0.56%  runtime.growWork_fast64
          0     0% 99.63%      0.05s  0.93%  runtime.main
          0     0% 99.63%      0.03s  0.56%  runtime.mapassign_fast64
```

Cache characteristics

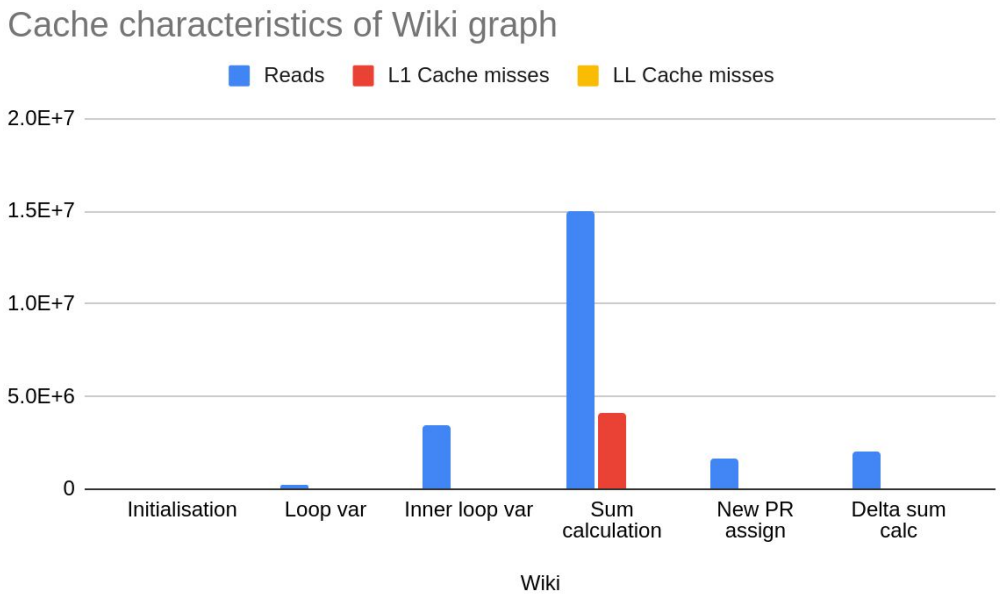
- Using cachegrind we found
  - Number of data reads
  - Number of misses in L1 cache
  - Number of misses in LL cache (L3 cache)
- At each line in the code
- We compare these characteristics across different graphs

Cache characteristics - Small



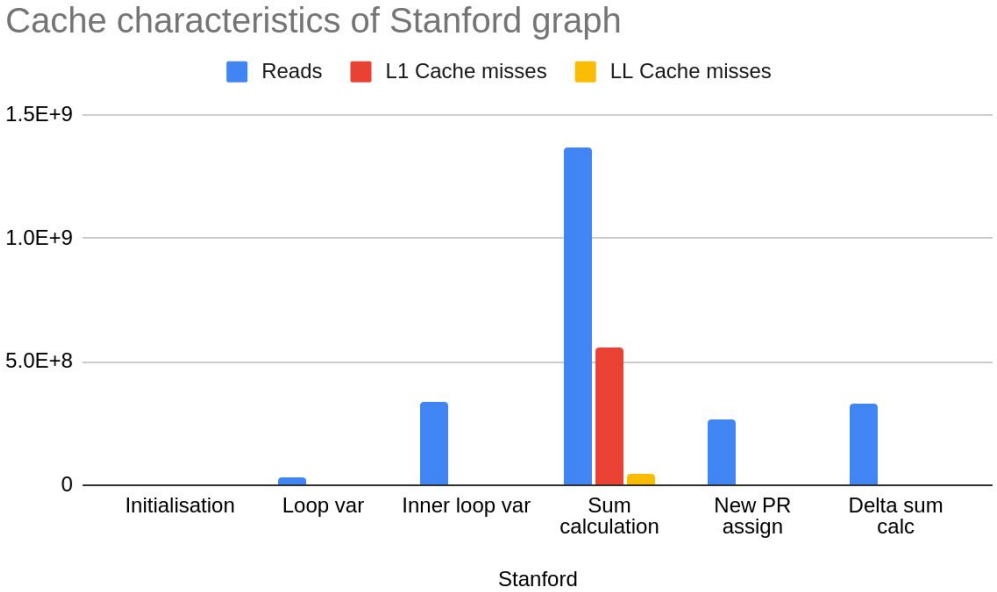
- The small graph fits easily into both the caches
- No significant misses

Cache characteristics - Wiki



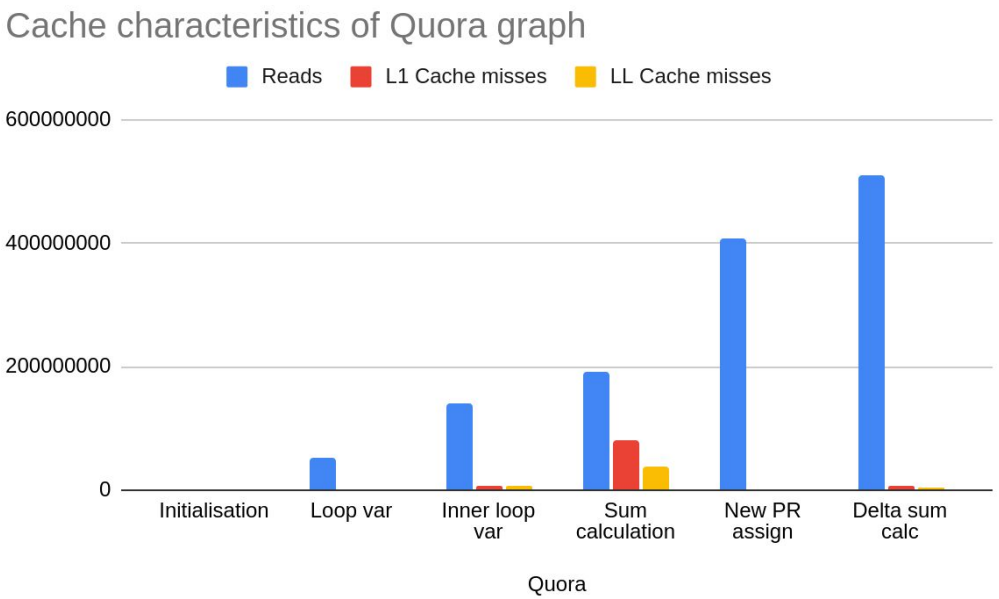
- The Wiki graph has a Pagerank vector size of 3 kB per partition
- All 16 partitions fit into the L3 cache (6MB)
- Thus, there are no L3 cache misses
- The sum calculation shows most L1 cache misses as we predicted
    - Due to random accesses

Cache characteristics - Stanford



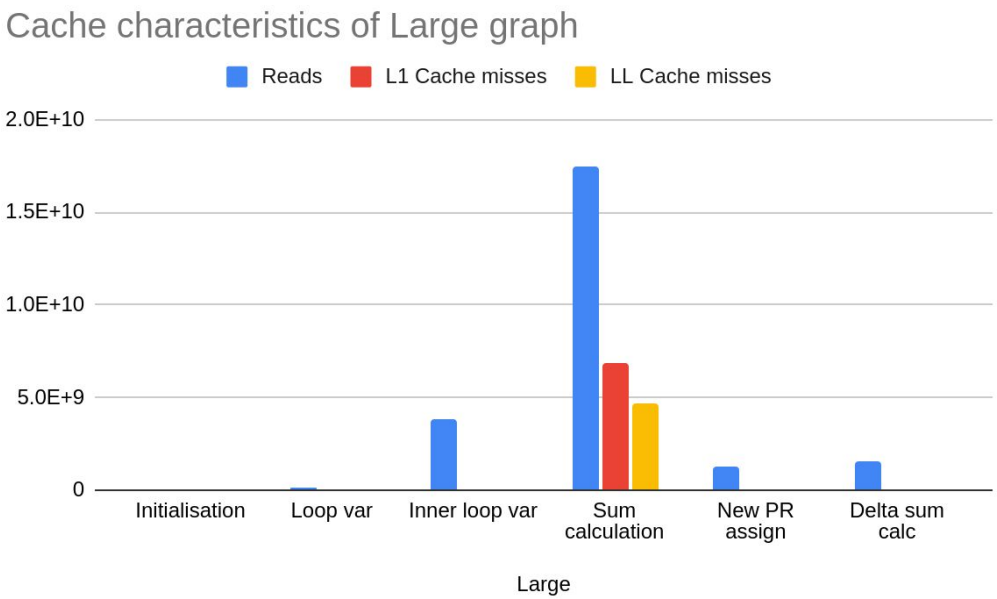
- The Stanford graph has a Pagerank vector size of 138 kB per partition
- Fits into L3 cache (6MB), but due to other vectors also being accessed it shows some misses
- Sum calculation shows most misses as predicted

Cache characteristics - Quora



- The Quora graph has a Pagerank vector size of 263 kB per partition
- Fits into L3 cache but might be evicted due to other data
- Due to unusual characteristics of this graph - extremely sparse - we see most reads in new PR assignment and delta sum
- Sum calculation still shows the highest proportion and number of misses

Cache characteristics - Large



- The Large graph has a Pagerank vector size of 1MB per partition
- All 16 partitions do not fit into the L3 cache of 6MB
- Therefore the number of cache misses is higher.

Future Work

Risk 1

- Yet to fully understand graph characteristics impacting the performance of the algorithm. Have attempted to understand based on density and degree distribution.

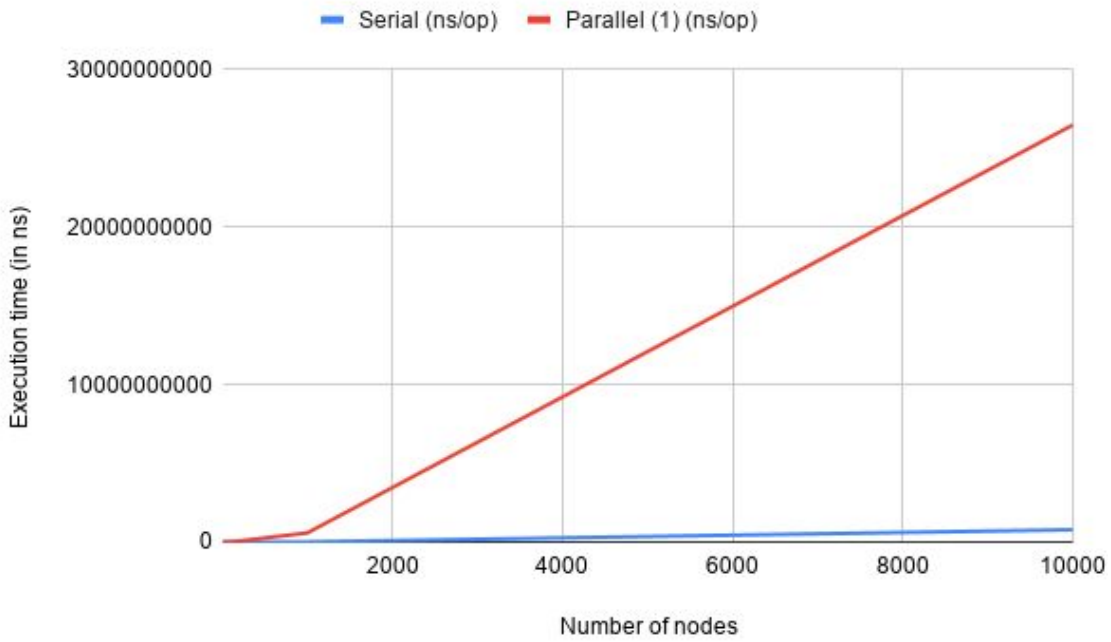
Risk 2

- Go was built for concurrency and not parallelism. OpenMP or MPI levels of parallelism is not possible

Dijkstra's SSSP

- Dijkstra's Single Source Shortest path (SSSP) algorithm is used to find the minimum cost path between given source and destination vertices
- On a higher level, the algorithm consists of two nested loops:
  - In the outer loop, the vertex with minimum cost is taken from the priority queue and the inner loop is run
  - In the inner loop, the neighbours of the min node are checked and their distances are updated
- We first tried to parallelize the inner loop.

Dijkstra's SSSP - inner loop



- The results were bad, the parallel version was an order of magnitude slower
- Speculated to be because of
  - Runtime overhead of creating many (= number of nodes) goroutines every loop
  - The inner loop computation was too small

Dijkstra's SSSP

- We did not explore this further as we concentrated on pagerank

Teamwork Distribution

Member	Areas worked on	Hours
Aditi Ahuja	Topological pagerank and parallelising it, initial ordinary pagerank implementation,profiling with igraph and cachegrind	25 hours
Pranav Kesavarapu	Fixing data races in pagerank, entire CUDA version of pagerank, infrastructure management	25 hours
Samyak S. Sarnayak	Dijkstra's algorithm, Partitioning pagerank and its parallelisation, profiling with pprof and cachegrind.	25 hours

References

1. https://neo4j.com/blog/graph-algorithms-neo4j-15-different-graph-algorithms-and-what-they-do/
2. https://github.com/purtroppo/PageRank
3. https://arxiv.org/abs/1903.12085
4. https://www.semanticscholar.org/paper/A-Parallelization-of-Dijkstra's-Shortest-Path-Crauser-Mehlhorn/97c9d8113fe0ac0ab1058e13d3501e1ea8fe9485