Par-a-graph: Parallelising PageRank

Aditi Ahuja
PES University
Bangalore, India
aditiahuja@pesu.pes.edu

Pranav Kesavarapu
PES University
Bangalore, India
pranavkesavarapu@gmail.com

Samyak S. Sarnayak
PES University
Bangalore, India
samyak201@gmail.com

Project Guide
Dr. T.S.B Sudarshan
PES University
sudarshan@pes.edu

***Abstract*—Graph algorithms are widely used as the backbone of many modern systems spanning various domains. Some of the most popular applications include graph databases, social networks and search engines. With graphs now far more vast than earlier, efficient implementations of commonly used graph algorithms are increasingly relevant.**

This work details a parallel PageRank algorithm on an efficient data structure, implemented using Golang. We observe speedups ranging from 1.58x to 4.23x on various graphs of different characteristics. The results are also explained by studying the cache characteristics of our algorithm on these graphs.

***Index Terms*—algorithms, graphs, pagerank, parallelization**

I. Introduction and Use Cases

Pagerank was originally developed [1] to rank web-pages on the internet, based on their relative importance. Though, the algorithm is a general graph algorithm that can be used to identify the most important nodes in any graph network. Graphs can be seen in a variety of real use cases such as social networks, research paper citations, chemical structure of molecules, road networks and even call graphs of a large code base. Subsequently, pagerank has been applied to these areas to identify top users to follow, ranking/credibility of researchers, assess hydrogen bonding, predict traffic flow and to identify the most important functions respectively [2].

Our implementation of pagerank is not tied to any of these applications. The Golang code base released alongside this paper can be integrated into other projects that make use of graphs, such as *graph databases*.

II. Implementation

A. Data Structures

Two different representations of graphs were tested in our implementations of PageRank. The first was a standard **Adjacency array** where each node u is mapped to an array of outgoing nodes O_u. This was implemented using a Golang `map` (a `map[int][]int` to be precise). We found some drawbacks when implementing pagerank using this data structure. The pagerank algorithm we implemented requires a list of incoming nodes I_u to get and update the pagerank values from. Therefore, an additional adjacency array needs to be created at the beginning of the algorithm which maps each node u to its array of incoming nodes I_u. This requires up to $O(N^2)$ extra space and an additional traversal of the whole graph, in the worst case. Since the algorithm requires the outdegree (cardinality of the set of outgoing nodes for a given node, $|O_u|$) of every node in the inner loop, they have to be pre-computed and stored in a separate array. This leads to $O(N)$ extra space and computation.

We used edge lists to store the graphs on disk. An edge list is a list with each entry in the form u, v that denotes an edge of the graph - from node u to v. To be useful in the algorithm, edge lists have to be converted to an adjacency array of either incoming or outgoing nodes. Using these in the algorithm directly is not feasible, hence we do not compare them.

An alternative representation of graphs that we test is the **Compressed Sparse Row (CSR)** representation [3]. The representation consists of three different arrays - a *vertex array*, an *edge array* and an *outdegree array*. The vertex array stores cumulative in-degrees (starting from 0), the edge array stores edges but in the reverse order (destination to source mapping) and the out degree array stores out degree of every node. The edge sub-array corresponding to each node i.e., `edgeArray[vertexArray[i]:vertexArray[i+1]]` is sorted to increase data locality and reduce cache misses.

B. Algorithm

The pagerank algorithm used was as follows:

1) Initialise the pagerank of each of the N vertices to $\frac{1}{N}$.
2) Each iteration for a vertex u involves the following computation (where PR_u denotes the pagerank of vertex u, I_u denotes the set of incoming nodes to u, O_v denotes the set of outgoing nodes from v and α is the damping factor which is usually set to 0.85):

```
$sum_u \leftarrow 0$
for $v \in I_u$ do
    $sum_u \leftarrow sum_u + \frac{PR_v}{|O_v|}$
end for
$PR_u \leftarrow \frac{1-\alpha}{N} + \alpha \cdot sum_u$
```

3) This is continued till the difference between the page rank of all vertices between two consecutive iterations is less than a set error threshold.

The first approach to parallelization we used was as follows. A new goroutine [1] is launched to process each node's iteration (given in the algorithm in step 2. above). This introduced a massive overhead as N goroutines were launched in every iteration, leading to an approximately 20x slowdown in performance, compared to the serial implementation. Note that Golang's `sync.WaitGroup` is used to synchronize the completion of the computation of all goroutines.

The next approach used the *Fanout pattern* of concurrency on Golang. A fixed number of goroutines were launched at

[1]A process in Golang's Communicating Sequential Processes (CSP) based concurrency model.

beginning of the algorithm. All of the goroutines used a shared *channel* to receive the next node to process and perform the computation for that node. In every iteration, all of the nodes are sent over the buffered channel and the work is divided at runtime based on which goroutine is free to receive on the channel. This means there are N number of sends and receives over the channel in each iteration with multiple goroutines contesting for the lock on the channel. Hence, there still exists a large overhead. It can be observed here that the same set of nodes are sent over the channel in every iteration.

The final approach builds upon this idea and statically assigns a chunk of nodes to each goroutine. Let P be the number of goroutines used and V be the set of nodes (vertices). V is divided into equally-sized (except for the last chunk), P number of chunks and each goroutine only performs computations on its chunk of nodes. An array of P channels of `struct{}` are used to signal the start of an iteration to the goroutines. Note that only P sends and receives are performed in total, with each of them being on separate channels i.e., there is no contestation for locks.

Each goroutine is also given a separate δ (error term) which are then summed up at the end of the iteration to find the total error. Hence the δ computation does not involve any locks and only requires a summation over a P-sized array. A similar optimization was implemented for the leak computation too. The algorithm for the main loop of the final implementation is as follows:

```
$sig \leftarrow (ch_1\ ch_2\ \ldots\ ch_P)$
$\delta s \leftarrow (0.0\ 0.0\ \ldots\ 0.0)$
$leaks \leftarrow (0.0\ 0.0\ \ldots\ 0.0)$
while true do
  $\delta \leftarrow 0.0$
  for $i \leftarrow 0$ to $P-1$ do
    $sig_i \leftarrow$ struct{}
  end for
  wait for all goroutines to finish computation
  $leak \leftarrow 0.0$
  for $i \leftarrow 0$ to $P-1$ do
    $leak \leftarrow leak + leaks_i$
    $leaks_i \leftarrow 0.0$
    $\delta \leftarrow \delta + \delta s_i$
    $\delta s_i \leftarrow 0.0$
  end for
  $leak \leftarrow leak \cdot \alpha$
  swap($x$, $x_{new}$)
  if $\delta < \epsilon$ then
    break
  end if
end while
```

III. Comparison with Existing Works

In comparison with existing pagerank implementations, our approach provides the following benefits:

- The CSR representation provides a large increase in performance for all variations of the pagerank algorithm tests. It is more cache friendly and provides better data locality as even edges of neighbouring nodes (in terms of their ordering, not edges) are stored contiguously and the edges themselves are sorted. While this leads to better cache locality when accessing the neighbours, there still exist cache misses when accessing pageranks of these neighbours.
- Our final implementation leverages data parallelism where similar computations are performed on each section in a separate goroutine.
- We provide an open source implementation [2] of the algorithm and the experiments. Our implementation is readable as well as practical since we make use of the Go programming language.

IV. Results

A. Testing and Benchmarking Methodology

The tests were developed using Go's `testing` package. For verifying accuracy, the results of our implementation were compared with a reference, serial implementation in Go. [3]

Benchmarking too used the `testing` package. The pagerank computation was performed multiple times to get more accurate timings. The benchmarks were repeated with multiple error thresholds: 10^{-6}, 10^{-9}, 10^{-11}.

B. Datasets Used

We have tested our algorithms on the following datasets - Wikipedia vote network [4] (named *WikiVote*), Quora Question Pairs [4] (named *Quora*), Stanford web graph [5] (named *Stanford*) and the English Wikipedia graph [5] (named *Large*).

Graph	Edges	Vertices	Max. Density
WikiVote	103689	7115	0.0021
Quora	404291	537935	$2.8e^{-6}$
Stanford	2312497	281903	$5.82e^{-5}$
Large	46092177	2080370	10^{-5}

C. Performance Results

Prior to the final implementation, we observed the following on the Large graph:

- PageRank with adjacency lists and N goroutines - **1.03x** maximum speedup (serial - 35.03s, parallel - 33.97s with $\epsilon = 10^{-6}$).
- Fixed number of goroutines with fanout pattern - **1.36x** maximum speedup (serial - 136.47s, parallel - 100.04s with $\epsilon = 10^{-11}$).
- Node partitions/chunks, parallel initializations, leak and delta calculations - **2.71x** maximum speedup (serial - 221.45s, parallel - 81.61s with $\epsilon = 10^{-11}$)

The following results were obtained with the final implementation involving CSR representation and node partitioning.

[2]https://github.com/metonymic-smokey/par-a-graph
[3]https://www.github.com/dcadenas/pagerank
[4]https://www.kaggle.com/c/quora-question-pairs/data
[5]https://cfinder.orgwiki/?n=Main.Data

- Large Graph:

Error (ϵ)	Serial (s)	Parallel (s)	Speedup
E6	24.17	15.30	1.58
E9	43.75	29.07	1.504
E11	70.13	44.32	**1.582**

- Quora Graph:

Error (ϵ)	Serial (s)	Parallel (s)	Speedup
E6	0.46	0.18	2.62
E9	1.20	0.45	**2.64**
E11	1.66	0.68	2.43

- Stanford Graph:

Error (ϵ)	Serial (s)	Parallel (s)	Speedup
E6	0.91	0.251	3.62
E9	1.93	0.0.46	**4.23**
E11	2.36	0.63	3.73

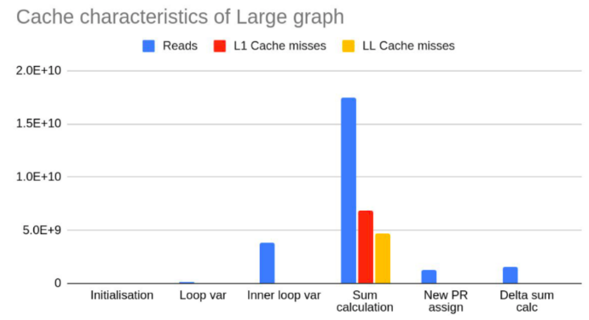
- WikiVote Graph:

Error (ϵ)	Serial (s)	Parallel (s)	Speedup
E6	0.0073	0.0037	1.96
E9	0.012	0.063	**2.02**
E11	0.016	0.087	1.83

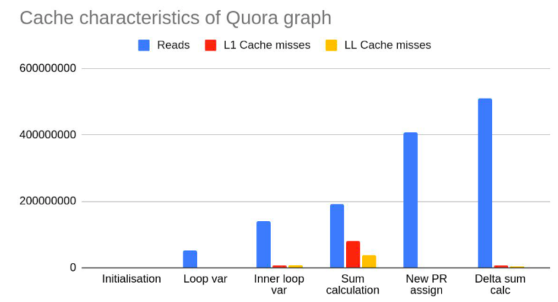
D. *Cache characteristics*

Using Cachegrind, each line of code was profiled for number of data reads, number of L1 cache misses and number of L3 cache misses across all the datasets. The experiments were performed on a machine with an L1 cache size of 128KB and an L3 cache size of 6MB.

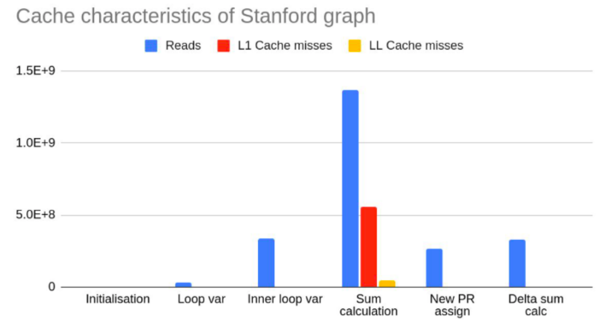
1) Large graph: For the Large graph, we notice a large number of cache misses. This is because each partition has a page rank vector size of 1MB. All 16 partitions do not fit inside the L3 cache.



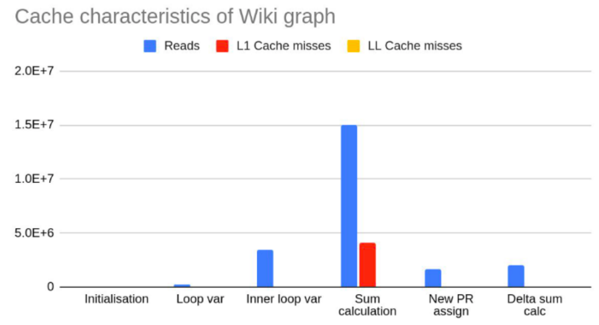
2) Quora graph: The Quora graph has a Pagerank vector size of 263 KB per partition. All partitions fit into the L3 cache of the machine used, but may be evicted due to other data. Since this graph is extremely sparse, most new reads are observed in new partition assignment and delta sum.



3) Stanford graph: The Stanford graph has a Pagerank vector size of 138 KB per partition. All partitions fit into the L3 cache, but shows a small number of misses due to other vectors being accessed.



4) WikiVote graph: Smallest PageRank vector size with no L3 cache misses. The sum calculations shows the most L1 cache misses, as expected, due to random accesses.



V. YouTube Video

This is the link to our YouTube video: https://youtu.be/Xl2a8j3zats

Acknowledgment

We would like to thank our guide Dr. T.S.B Sudarshan for his guidance during the Heterogrenous Parallelism course, during which the project was developed. We would like to thank him and the teaching assistants, Siddarth Karki and Nikhil Khatri, for their guidance from the ideation to the final development phases of the project.

References

[1] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Computer Networks*, vol. 30, pp. 107–117, 1998.
[2] D. F. Gleich, “Pagerank beyond the web,” *CoRR*, vol. abs/1407.5107, 2014.
[3] S. Morishima and H. Matsutani, “Performance evaluations of graph database using cuda and openmp compatible libraries,” *SIGARCH Comput. Archit. News*, vol. 42, p. 75–80, Dec. 2014.
[4] J. Leskovec, D. Huttenlocher, and J. Kleinberg, “Signed networks in social media,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’10, (New York, NY, USA), p. 1361–1370, Association for Computing Machinery, 2010.
[5] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, “Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters,” *CoRR*, vol. abs/0810.1355, 2008.
[6] K. Lakhotia, R. Kannan, and V. Prasanna, “Accelerating pagerank using partition-centric processing,” in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC ’18, (USA), p. 427–440, USENIX Association, 2018.