PARALLEL PAGERANK

Swarali Patil (smpatil), Aditya Bhawkar (abhawkar)

I. SUMMARY

Our project implements the Page Rank Algorithm for classifying the rank of various interconnected web pages using graph data structures. We have achieved speedup up to 6x in computation through parallelism using various techniques, specifically through OpenMP, SIMD operations and Message Passing Interface (MPI). The results have been obtained on the GHC machines (8 core) as well as on the PSC Bridges Machine (128 core). We provide a comparative analysis of our incremental approach to the final versions of our various implementations giving the best possible performance, including our unsuccessful attempts at various methods. As our bonus goal, we were also able to explore the GraphLab (TuriCreate) library taught in class.

II. BACKGROUND

The Page Rank Algorithm was first invented by the founders of Google, Larry Page and Sergey Brin in 1998 to assign a degree of order to web pages when there were a few million of them. In terms of application and scale today, it could be argued that every google search, inherently utilizes a core of the algorithm we have implemented. The applications today extend beyond webpages to any directed graphs. The algorithm addresses a complex problem and elegantly provides a simple human-readable solution. In addition the scale of the problem provides significant scope to parallelize for computing massive graphs.

The web can be represented as a directed graph where nodes represent the web pages and the edges form links between them. This is represented as an edge list in our project using a “from index” and “to index”. Quoting from the original Google paper, PageRank is defined like this:

We assume page A has pages T1...Tn which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. There are more details about d in the next section. Also C(A) is defined as the number of links going out of page A. The PageRank of a page A is given as follows:

$$Pr(A) = (1-d) + d(\frac{Pr(T_1)}{C(T_1)} + ... + \frac{Pr(T_n)}{C(T_n)})$$

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one.

PageRank or PR(A) can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the web.

To elaborate on their algorithm, PR represents the importance of each page as compared to every other page in the graph. Each page takes into account a summation of all the incoming nodes (pages), averaged out by the number of their own outgoing links C(Tn). Additionally, the PageRank theory holds that an imaginary surfer who is randomly clicking on links will eventually stop clicking. The probability, at any step, that the person will continue is a damping factor d. Various studies have tested different damping factors, but it is generally assumed that the damping factor will be set around 0.85. In simpler terms, the damping factor is primarily to ensure that webpages that have no outbound links, do not adversely affect the computations for the other pages.

Our algorithm follows the following approach across implementations using the data structures specified:

- Read in the graph as an edge list into our data structure
- Initialize pageranks for each page in a C++ Vector
- Initialize number of outgoing links for each page in a C++ Vector
- Initialize indices of dangling pages (no outgoing links) in a C++ Vector
- Read in additional data into Page data structure
- The Page data structure maintains a count of incoming pages, the indices of incoming pages as a C++ list
- Compute preliminary page rank in function for each page by iterating over all pages, getting the list of incoming nodes per page, then dereferencing their out going counts

$$Pr(A) = \frac{Pr(T_1)}{C(T_1)} + ... + \frac{Pr(T_n)}{C(T_n)}$$

To do workload analysis and explore parallelism avenues, we need to separate out different sections of the code. We have provided the pseudocode for the same in Fig. 1.

```
for i in number of pages:
   sum = 0;
   if not dangling:
     For  j in incoming_indices
          in_id = pages[i].incoming_ids[j];
          sum += pr[in_id] / out_links[in_id];

     new page rank [i] = sum;
```

Fig. 1. The pseudocode for calculating sum of all weights at each node.

The most computationally expensive part of the code is the simple sum calculation of pagerank of incoming id divided by number of outlinks at incoming id. This is done for all incoming ids for all pages in the graph. The main dependency here is the cumulative sum calculated for all incoming ids. Other than that, the pagerank calculation for each page is independent of the pagerank calculation for other pages. This is the main parallelizable part of the code.

Next we calculate the effect of dangling pages in a separate function and add their value to the previously calculated page ranks. As seen from the pseudo code, each loop is dependent on the sum calculated in the previous loop. This introduces some serial task dependency. However we can see that the code is mostly data parallel and we can try different parallelization approaches.

```
for page in dangling_pages:
     sum += page_rank[page]

for pr in page_rank:
     pr += sum / num_of_pages
```

Fig. 2. The pseudo-code for calculating effect of dangling nodes.

```
For pr in page_rank:
    Pr = pr * damping_factor
        + (1-damping_factor) / num_of_pages
```

Fig. 3. The pseudocode for the random surfer

The loops in Fig. 2 and Fig. 3 are definitely parallelizable using SIMD instructions. However the main computation part has multiple random accesses and indirections based on the incoming ids which is not great for SIMD execution.

III. APPROACH

We explored different approaches to map our serial version to parallel hardware. In the process we went through multiple iterations of optimizations using techniques taught in class like OpenMP, SIMD, MPI and other optimization methods like loop unrolling. At a higher level, our main data structure holding information of all pages is split across the parallel resources for the algorithmic segment that is computationally intensive. For the OpenMP version, each thread does the computation of page ranks for a chunk of the pages. This computation is split similarly for the MPI version. We also have some serial sections which we speed up using SIMD programming. Here we exploit the data parallel properties of our loop computations to map segments of the page rank array to SIMD vectors and perform operations on them in parallel.

A. Original Implementation:

The original serial algorithm was written using a C++ STL Map data structure which was the most intuitive way to implement the algorithm. It was simple in terms of code to dereference web pages by their index that are mapped to our “Page” data structure. As graph indices were not numbered in increasing order, it was tricky to loop over them initially. While reading in the graph, we stored the index in a “lookup” map.

B. Optimizing the original serial code:

In the main loop, we would dereference the index and then use it to in turn get the Page data structure from the “pages” map. It was simple to verify for correctness on small graphs for few iterations. However we quickly realised that the map data structure is inefficient as maps (ordered and unordered) store their elements in non-contiguous memory locations, either using trees or hash maps. On profiling certain sections of the map based approach, we saw that significant time was spent accessing elements which was bad for performance. We attribute this to the fact that the map graph structure is very irregular and added indirections resulted in random memory accesses making it difficult to get good performance.
The first optimization we attempted was to change our data structures to vectors to explore the fact that they store data in contiguous memory locations (push_back()). Since all the pages and their metadata were fixed in increasing order, we could immediately think of ways to extract data parallelism. The baseline vector code itself was significantly faster than the map version primarily aided by O(1) dereferences and spatial memory locality.

C. OpenMP:

The next approach we tried was to identify data parallelism in our implementation, aided by the contiguous memory allocation using our vectors. As each chunk of the page rank array is independent of the other, individual threads could process them in place in the loops. This parallelism is exploited using OpenMP.
The overall computation time can be divided into sections - first section which computes the sum of all weights for the out links at each node, next we compute the page rank for all dangling pages and the final part which adds the damping factor.
On profiling the code to identify our first major bottlenecks, we realized that a significant part of the computation time was spent in summing up the weights at each node. On average across 3 datasets, the summing of pageranks contributed to 90% of overall computation time.
The first step was to use openMP to parallelize the outer loop that computes the sum of the weights vector at each node. We attempted different scheduling policies (static, dynamic) and

have presented the results in the next section. Each OpenMP thread now computes the intermediate sum of pageranks for a chunk of the total number of pages.

D. Loop unrolling and other small optimizations:

Next, we targeted further opportunities in parallelism in our incremental exploratory approach to further optimize our bottlenecks. We optimized the main pagerank summation loop by reducing the number of function calls from within the loop, precomputing some of the fixed values like reciprocal of the number of out links for each node and unrolling the loop. Loop unrolling helped us to expose a few more independent operations within the openmp parallelized loop. A few benefits of loop unrolling that would be inherently used by the processor would be reducing the overhead of incrementing and testing the loop counter. Reducing jumps back to a loop's entry might improve the pipeline behavior. Unrolling makes instruction-level parallelism in loops explicit and thus potentially enables compiler optimizations.
Given time constraints we decided to focus on the sequential bottleneck which we were observing with the other sections of the overall computation time as well as and other parallel techniques. A drawback of excessive loop unrolling would be increase in the size of code and debug time too.

E. SIMD:

We observed that even with OpenMP speedup we were limited by certain serial sections of the code. For instance on running our OpenMP version for 8 threads we observed a speedup of 6.78 for the main computation but there was still 0.49s of other compute time for dangling pages and damping factor which would limit the overall speedup. Based on what we learnt about Amdahl's Law, we knew that our speedup would be limited by the serial parts of the code and we had to focus on achieving speedup for that section
We chose to parallelize the loops that computed pagerank for the dangling pages and added the damping factor using SIMD approach given some of the data parallelism we could observe. On initial attempt, simply using openmp simd pragmas did not help much. The assembly code did not show the desired simd translation which we expected with using that pragma. So we rewrote the loops with explicit simd instructions like _mm256_loadu_ps, _mm256_add_ps, _mm256_fmadd_ps and _mm256_storeu_ps.This helped us bring down the serial section time from 0.49s to 0.146s.

F. OpenMP with SIMD:

The next step was to combine both our simd and openmp code to achieve maximum speedup. We used OpenMP for our compute intensive section followed by SIMD optimization for our serial sections to maximize the gains from both our parallelization techniques.

G. MPI:

The last iteration of our optimization was to use the Message Passing Interface library to make use of several nodes to process the page rank in parallel. As we earlier identified our bottleneck to be the intermediate pagerank computation where we add the values of neighbouring node, we explored the speedup in parallelizing this portion. At the cost of trading memory for better interprocessor communication, we initialize a local copy of the data structures at each node, and following a similar approach to that of the Wire Routing programming assignment, we divide the workload statically, between the processors. To account for remainder nodes, which is very crucial as they may be the highest ranked pages, we process them at root. Unlike the ocean or galaxy dataset, the web graphs do not change over time in our use case, so a static partition of the workload seemed like an ideal approach.

In our implementation, we broadcast the initial equal pageranks to all nodes to their page rank vectors, following which each node individually computes the partition it has been assigned. We do an AllGather and collect the partitions computed across the nodes.

However some remaining pages still need to be computed at the root node. Additionally we also compute the dangling pages part and damping factor calculation at the root node. At the end of this iteration, we broadcast the final computed pagerank across all nodes. However, despite attempting to get speedup comparable to our OpenMP/SIMD version, we weren't able to achieve it due to high cost of interprocessor communication while using dense graphs.

H. GraphLab:

As taught in lecture, as a bonus approach we explored and experimented with the functionality of GraphLab as well. However, since the inception of the library, it has undergone several makeovers namely being rebranded as Turi, and has finally been bought over by Apple. It is still available open source as a python library called TuriCreate. Most of the older API's have wrappers over them which can be run in one line of python code. As we were unsuccessful in installing the library locally on GHC and ECE machines due to several dependencies and access issues toward the end of the project, we resorted to testing it on Google Collaboratory.
Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education. More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, while providing free access to computing resources. However, they only provide a 2 core Intel Xeon 2.2 Ghz processor. As this would not be a fair comparison, we decided to use it for sanity testing.

```
g = turicreate.load_sgraph("/content/web-Google.txt", "snap")
pr = turicreate.pagerank.create(g, reset_probability=0.15,
    threshold=0.000000001, max_iterations=80,
    _distributed='auto')
pr_out = pr['pagerank']
```

Fig 4. Python code for TuriCreate (GraphLab)

The library also takes 11 seconds to read in the same graph and set up it's internal data structures. It is unclear from the documentation as to how the distributed = "auto" flag actually distributes the work. Regardless, the project gave us the opportunity to explore yet another tool, that does the job slower but makes programming significantly easier, as all things Python.

IV. RESULTS

We measured our performance based on two parameters - the actual wall clock time taken for compute and the speedup. The wall-clock time was simply used to compare the original baseline code with our faster implementations as the time difference between them differed by an order of magnitude so reporting speedup on that would not be as inferential.

The following figure shows the time taken comparisons for the original code and our optimizations (run on multithreaded CPU core with num_threads = 8)

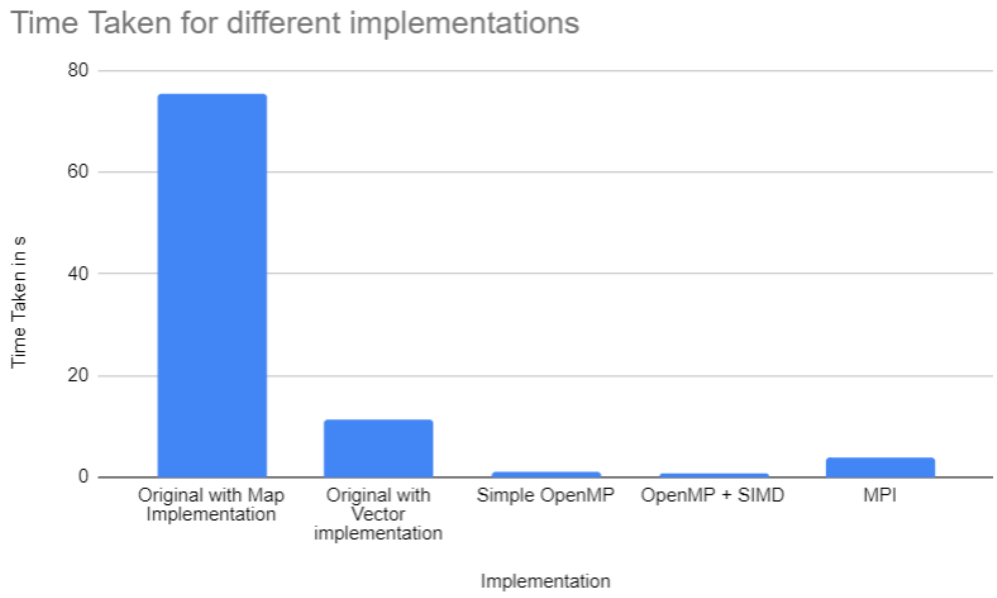

Fig 5. Comparison of various implementations' wall clock time

For other results we chose to use our optimized code with vector implementations of the graph data structure as the baseline to report final speedup comparison values.

The baseline configuration was run on a single-threaded CPU core. The inputs to the pagerank code was pulled from the Stanford Large Network Dataset Collection.

We worked with three different sized graphs to perform our analysis. For our various iterations, we chose to analyse the code trajectory for the web_google dataset. The measurements were first taken on the GHC Machines, and then run on the PSC machines subsequently to utilize more cores to exploit parallelism.

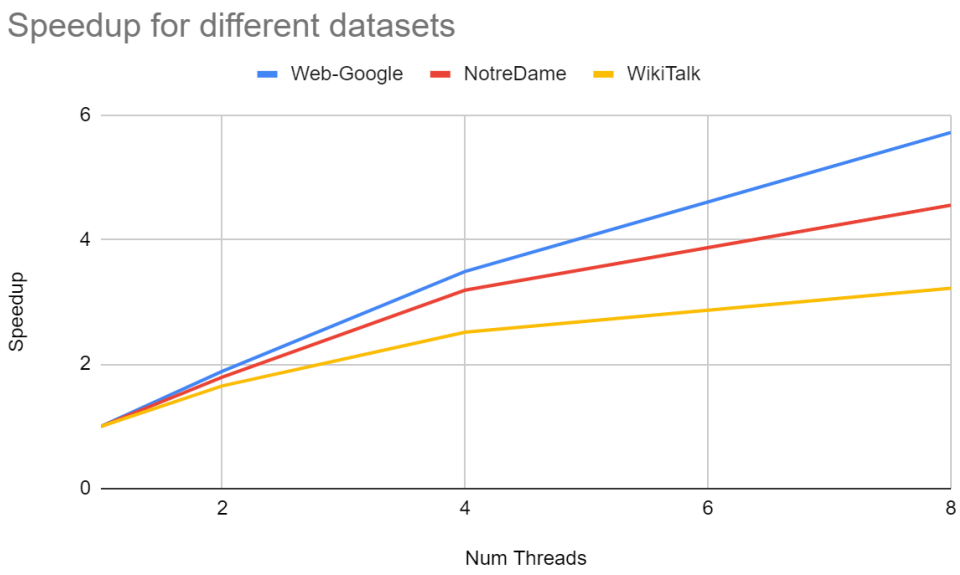

Fig 6. Graph depicting speedup for different datasets on GHC machines for our best beforming version

Experimental Setup and Inputs:

Our experimental setup consisted of the GHC machine, which has an eight-core, 3.0 GHz Intel Core i7 processor (although dynamic frequency scaling can take them upto 4.7 Ghz). Each core can execute AVX2 vector instructions, supporting simultaneous execution of the same operation on multiple data values. Most of the development was done on these machines, with higher core count being provided from the PSC Bridges Machine. This machine has 2 AMD EPYC 7742 CPUs 64 cores per CPU, 128 cores per node with a clock speed ranging from 2.25-3.40 GHz.

The inputs were graphs of varying sizes taken from the Stanford Connected Large Datasets resource. Characterized by the number of pages, the google graph was 875713, Wiki talk having 2394385 and Notre Dame with 325729. These datasets were stored as edgelists having a “from index” and “to index” in plaintext files which were read into our data structures as specified earlier.

The problem size had some impact on the performance. The general trend we observed was that speedup plateaued faster for a smaller dataset compared to a larger dataset. We had a smaller graph input to measure correctness of our implementation but we did not see speedup beyond parallelising beyond 2-4 cores. This can be justified by our code profiling which showed that there was always a minimum overhead time which would dominate any benefit we got from splitting up the work across more cores. Another observation was that some of the graph inputs had more sequential part (like the wiki-talk dataset). As a result the dangling nodes computation would take longer and although we did speedup a bit using SIMD instructions here it limits performance as there is always a constant compute time involved which does not scale as we increase the number of threads.
The following graphs show the speedup across different optimization iterations across the different graph inputs.

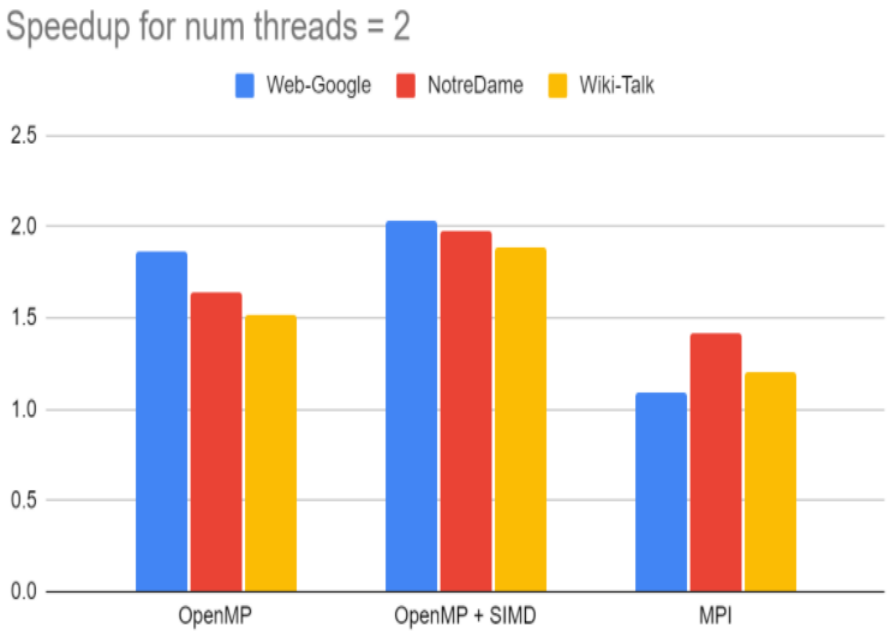

Fig 7. Comparison across implementations and datasets for 2 threads

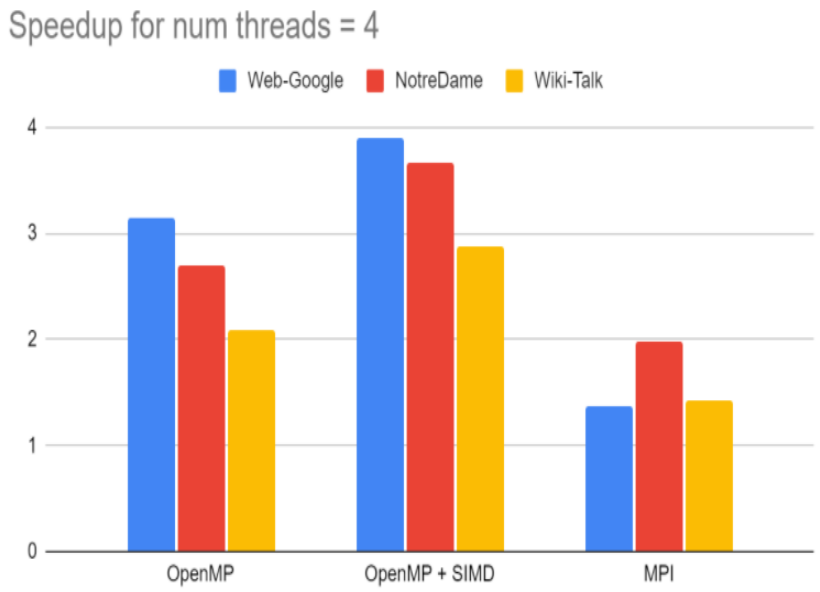

Fig 8. Comparison across implementations and datasets for 4 threads

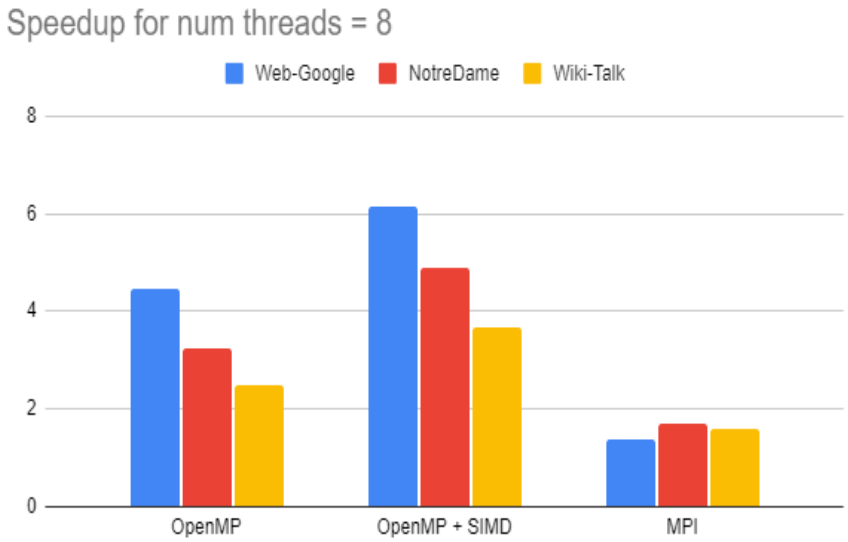

Fig 8. Comparison across implementations and datasets for 8 threads (Google dataset)

Analysis of different optimization iterations:

As we have explained before, our computation time has a very compute intensive loop and another sequential part which deals with dangling nodes and damping factor calculations. We initially added OpenMP pragmas in our compute intensive loop but we were not able to observe linear speedup proportional to num threads. We then checked time taken in each of the loops and realised that the sequential part of the computation was always taking a constant time as seen from the bar graph on the left. As a result, even if we have linear speed up from parallelising the

most compute intensive part of our code, the sequential code adds a constant time to the overall computation time which limits speedup as we increase the number of threads.

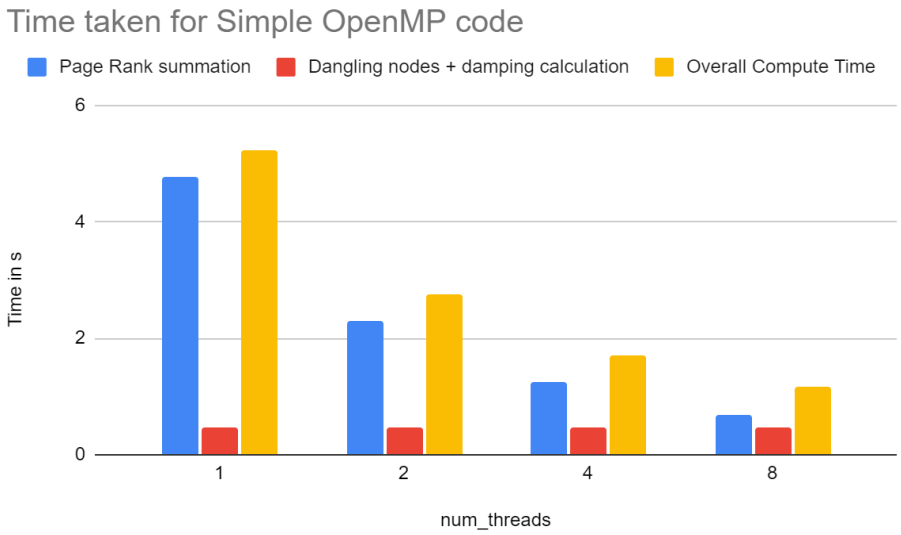

Fig 9. OpenMP threads to profile compute time of different sections (Google dataset)

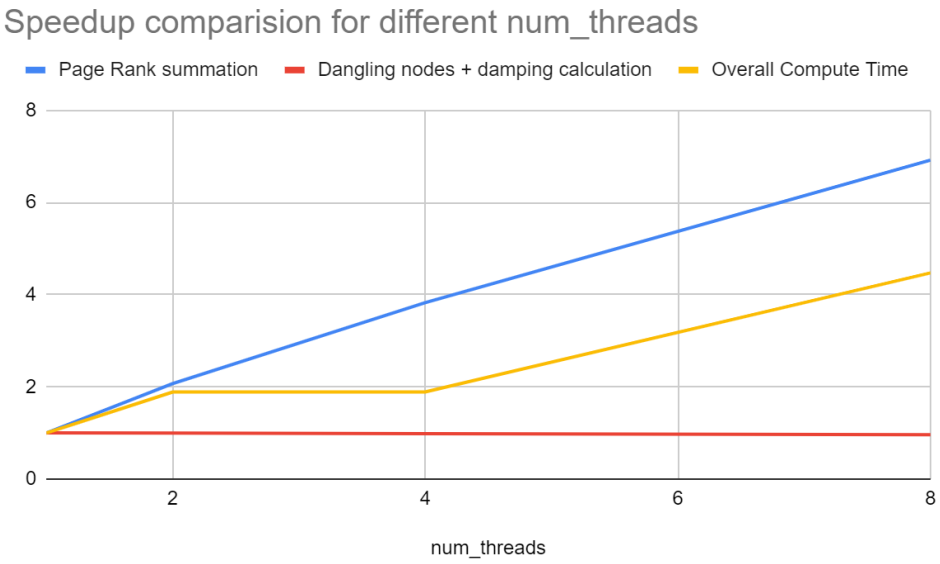

Fig 10. Comparison of speedup for different sections of compute time for the basic OpenMP code for web-Google dataset

The following graph shows the benefit we get from replacing some of the loops in the sequential part with SIMD instructions.

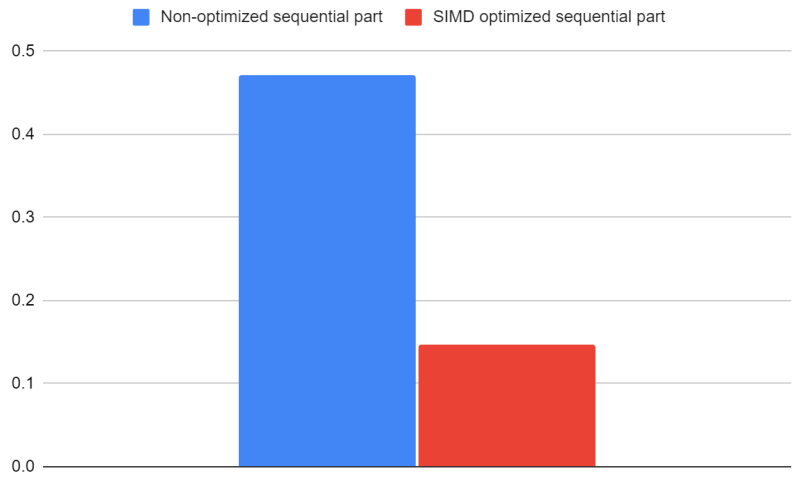

Fig 10. Comparing SIMD optimization

The speedup achieved here was 3.2. This is one of the reasons why we don't see perfect speedup. There is still one loop in this section of code which we did not get to parallelize. The main reason for this is it loops over all the old page rank values for each of the dangling pages. Although this is a simple sum calculation, because we don't index contiguously into the old pagerank array, adding simd vectors to load the values and compute the sum was very inefficient. So we chose to leave it as it is.

We then combined the two approaches and took the readings on PSC machines. As seen in the figure below, the overall speedup tapers off as we increase the number of threads for the simple OpenMP version because of the dominating sequential part. But with the OpenMP for the compute intensive part and SIMD for the remaining section, we still see increasing speedup as we increase num_threads. We assume that as the core count increases beyond 64, the overhead of scheduling threads dominates which negates the speedup we have achieved up till 64.

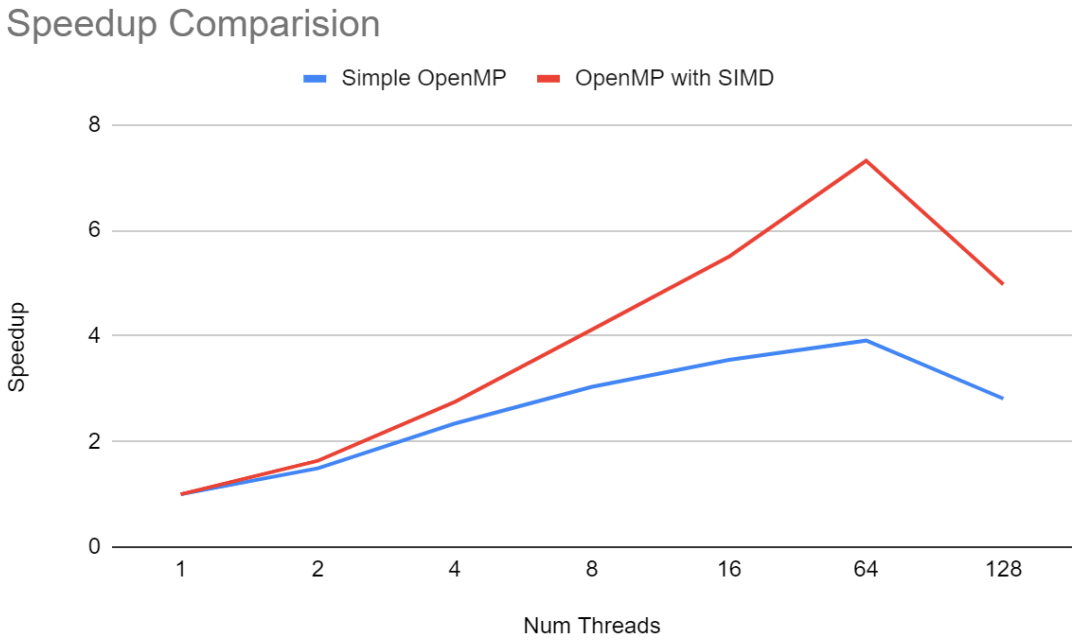

Fig 11. Speedup comparison with and without SIMD on PSC machines

MPI Analysis

We observed speedup with our MPI version as well, as structurally, it follows a similar pattern with static work scheduling that is distributed among the nodes. Ideally, we would expect more nodes to give us faster overall compute performance on our bottleneck. However, we did not observe better speedup with our MPI version as compared to the OpenMp (with SIMD) versions. On profiling the time taken by the Broadcast and All_Gather MPI functions, we do see a considerable portion of time taken. Our timing code showed that the collective communication time increased linearly as we increased the number of nodes. This was pretty consistent with what we had learnt in lectures and in our assignment 4. With an increasing number of nodes, the inherent memory barrier due to the broadcasting and gathering makes a difference as all nodes have to wait for more webpage metadata to be communicated. Hence we are bound by the overhead of communication which is not seen in the other versions as computations are updated in place in our page rank vectors.

We did perform an analysis of the execution time of our algorithm. On average 90% of the time was spent in the function which calculated the sum of the weights for each node. Rest of the time was spent in the pagerank calculation for dangling nodes and the loop that scaled pageranks with the damping factor.

Scope for improvement

A major constraint of our problem that we weren't able to completely address was being memory bound. Despite having parallelism through many cores, per iteration, large chunks of graph data have to be brought into local cache. Despite improving performance using contiguous memory allocation and improving spatial locality, a possible optimization could've been pre-fetching.
We went through the assembly code to analyse some of the SIMD optimizations we added. We use only one of the fma units there and based on the architecture we could have squeezed in some more speedup there by utilizing more available resources.
We also considered using a sparse matrix representation, in the iterative power method algorithm, but due to time constraints and significant code refactoring, we decided to stick to our custom data structures.

GraphLab Inference

We were able to use GraphLab to run basic inference tests on our datasets, which verified the algorithm and its results by simply seeing the connections between the top few ranked pages.

In the google-web graph with 875173 nodes,

ID	In degree	ID	Out Degree	ID	PageRank
537039	6326	506742	456	597621	564.270617
597621	5354	203748	372	41909	562.68622
751384	5182	305229	372	163075	552.224045
32163	5097	768091	330	384666	480.684524

Table 1: Table of top nodes mapped to maximum incoming links, outgoing links and Page Rank

As is visible from the results, the order of nodes based on incoming and outgoing links, compared to the final pagerank is different. This brings into context the essence of the algorithm, where the rank of a page could be very well determined by the page it is connected to as well. Just because a page has several incoming links, if they are coming from sinks (pages with no more outgoing links), that directly represents a human surfer reaching the end of his web surfing session. A page with fewer links, but connected a popular page, will have greater chances of being visited by a user.

V. CONCLUSION

We show that we can speed up the pagerank computation process significantly with a parallel machine. Through the process of optimizing our code, we were able to apply various techniques taught in the course like OpenMP, SIMD and MPI. This gave us a better understanding of these libraries in addition to the programming assignments. In addition, this was also a great opportunity to gain in depth knowledge and get into the details of one of the most popular algorithms that is in use today. We also realized that it can be extended not only to rank pages but any directed graph, whether it be social media analysis, tweet/post impact, electrical grid analysis etc. One of our main takeaways was that having more resources does not always guarantee speedup. The sequential portion of the code would always be a bottleneck and we had to spend equal effort trying to optimize the serial parts as well. A significant chunk of our time was spent in thinking about the design in order to refactor code from one version to the other. Writing high performance C++ code close to the machine will always be a great challenge for programmers.

VI. DISTRIBUTION OF WORK

Majority of the work on the project was done together. So we propose a 50%-50% split of the grade.

VII. REFERENCES

1. Page, Lawrence, et al. The PageRank citation ranking: Bringing order to the web. Stanford InfoLab, 1999.
2. Rabenseifner, Rolf, et al. "Hybrid MPI and OpenMP parallel programming." PVM/MPI. 2006.
3. Stanford Datasets (http://snap.stanford.edu/data/#web.)
4. TuriCreate repository (https://apple.github.io/turicreate/docs/api/turicreate.toolkits.graph_analytics.html)
5. Google Collaboratory (https://colab.research.google.com)
6. Hou, Guanhao, et al. "Massively parallel algorithms for Personalized PageRank." Proceedings of the VLDB Endowment 14.9 (2021): 1668-1680.
7. Whang, Joyce Jiyoung, et al. "Scalable data-driven pagerank: Algorithms, system issues, and lessons learned." European Conference on Parallel Processing. Springer, Berlin, Heidelberg, 2015.