Parallelize and compare PageRank on OpenMP & Pthreads

Utkarsh Kumar
New York University
New York, NY, USA
uk2012@nyu.edu

Shiv Ratan Sinha
New York University
New York, NY, USA
srs9969@nyu.edu

Mohamed Zahran
New York University
New York, NY, USA
mzahran@nyu.edu

***Abstract*—Pagerank is a method to measure the quality of web pages based on the formation of the hyperlink graph. Even the shared memory graph processing suffers costly atomic updates and poor locality of data on a single-machine. Many parallel implementations of large and real-world graphs suffer from the issue of random DRAM accesses. In this paper, we mainly focus on the speedup and efficient utilization of the cache for the PageRank algorithm by reducing the cache misses. We showcase an efficient, parallel version of PageRank using the edge-centric model for optimizing the performance of the memory without changing the algorithm implementation and the data structures used. The ordered list of the incoming edges helps in the sequential access of neighbours of a vertex thus retrieving the incoming neighbours in an efficient manner. Using this, we see a significant reduction in the execution time as well as reads from the DRAM. This optimized pre-processing of huge graphs helps in the efficient implementation of other graph-based algorithms in a parallel manner. We evaluate the optimized code on the NYU HPC crunchy nodes with 64 AMD Opteron 6272 processors. We use 2 large real-world datasets. We use different scheduling: static, dynamic, guided and auto with combinations of different chunk sizes in OpenMP. We see that the implementation achieves up to 1-1.5x better speedup and 3x lesser cache misses.**

We also present a pThreads based parallel implementation of PageRank to contrast it with the performance of OpenMP implementation.

***Index Terms*—openmp, pagerank, pthreads**

I. Introduction

Graph processing is one domain that has been extensively studied in previous years. Such graph-based algorithms play an important role in many of the world's applications such as task scheduling, social networks, Google Maps routing, genome analysis, etc. Due to the highly complex, massive size of the datasets, the processing of these graphs poses a challenge. Similarly, with the web-search algorithm, PageRank, touted by Google. This novel web search ranking algorithm was introduced by Michigan alum Larry Page and fellow Stanford Ph.D. student Sergey Brin in 1998. It became the foundation of the Google search engine. This was much needed to provide users, like us to be able to navigate through important and useful content on the web where nowadays we have a plethora of web pages, around 50 billion. These web pages form part of over a billion websites out of which around 20% are active and the remaining are not active. And it is obvious, that majority of these large number of web pages do not provide much quality content and most might be even outdated. Thus, PageRank helps in ranking the pages in their order of importance.

Even though the importance of such a graph algorithm is crucial, the processing of massive real-world graphs for various analyses is cumbersome. Such graph applications are highly notorious for their poor cache locality. Even though an unoptimized parallel implementation of PageRank performs better than a sequential version, it is limited by the random external memory accesses.

In this paper, our evaluation of the optimized edge-centric model observes an improved speedup compared to other parallel implementations of the PageRank algorithm.

II. Background and Literature Survey

A. PageRank

Whenever someone performs a search for something on the web, more than 85% of people choose to go with Google's search engine to further their exploration. The phrase 'Google it' has become a part of our society and much of its success can be attributed to the algorithm that measures the relative importance of web pages by calculating a rank for every web page. Thus, we can imagine the complete web as a directed graph where the web pages are nodes and the hyperlinks between them as directed edges. The value assigned to a vertex after the PageRank algorithm runs is the representation of the frequencies and the orientation of the vertex's outgoing links. Therefore, we can say that when a web page is referred to by many different vertices or web pages then the page being referred to becomes important and thus will have a higher score allotted to it by the PageRank algorithm. Moreover, when we have a web page that has a comparatively high score allotted to it and it is referring to other web pages then the importance of the other web pages increases.

Algorithm: For the algorithm to begin, it is assumed, in several research papers as well that the early distribution is distributed evenly between all the web pages pr vertices in the graph at the starting of the computational process. The algorithm then requires many passes or iterations through the whole graph to calibrate the PageRank score to more closely resemble the conceptual value.

The PageRank algorithm models a theoretical web surfer. Assuming that the surfer is currently exploring a given web-page and then they will click with equal probability on any

of the outgoing links and reach another page. Therefore, the PageRank score of a webpage is distributed equally between each of its linked web-pages thus raising each the neighbour's value. Thus, the theory can be boiled down to a simple formula for computing a particular web-page's PageRank:

$$\text{PageRank of site} = \sum \frac{\text{PageRank of inbound link}}{\text{Number of links on that page}}$$

$$PR_{k+1}(u) = \sum_{v \in Adj(u)} \frac{PR_k(v)}{totalOutgoing(v)}$$

Here, *Adj(u)* represents the set of nodes that link to *u*, and *totalOutgoing(v)* is the out-degree of node *v*. But this does not suffice the case when we encounter a sink web page while traversing the graph. A sink is a page that has no outgoing links: which in our surfer's terminology represents that the surfer is stuck and is not departing from the web page. To overcome this, we assume that the surfer will restart their browsing session on a random node and therefore, the sink has to contribute a portion of its PageRank score to every web page.

$$PR_{k+1}(u) = \sum_{v \in Adj(u)} \frac{PR_k(v)}{totalOutgoing(v)} + \sum_{\text{sink w}} \frac{PR_k(w)}{N}$$

Here, N is the total number of pages in the graph. Finally, we introduce a teleportation factor, known as the *damping factor d*. This adds a probability that the random surfer will visit any page at random, thus damping the current web pages score by *d* and then the remaining probability 1-*d* is divided between all the web-pages so that each node receives *(1-d)/N* as their share.

$$PR_{k+1}(u) = \frac{1-d}{N} + d\Big(\sum_{v \in Adj(u)} \frac{PR_k(v)}{tO(v)} + \sum_{\text{sink w}} \frac{PR_k(w)}{N} \Big)$$

Above, *totalOutgoing* has been shortened to *tO* for readability purpose. The above formula is applied iteratively to compute the results. The algorithm converges if we do not have cycles in the graph.

In this paper we will be focusing only on the static graphs. The other graphs are dynamic which means that the analysis has to be done on streaming data.

The basic algorithm for in-memory topology driven PageRank is shown in Fig 2.

```
1  PageRank(Partition p) {
2   Delta =(1- dFactor)/vCount;
3   parallel for v in p.vertices {
4     sum = 0;
5     for (nbr in p.incomingNbrs) {
6       sum = sum + nbr.rank;
7     }
8     v.rank = delta + dFactor * sum;
9   }
10 }
```

Fig. 1. PageRank algorithm

In Fig 1, the PageRank Algorithm, suppose we have p partitions since we are discussing the parallel implementation of the algorithm, each partition consists of some sets of nodes. Then in each partition *p* we iterate through the incoming set of vertices for each node in the partition and update the PageRank value of that node. The *dFactor* in the figure is the damping factor *d* that we have discussed already.

In the algorithm, the performance is highly impacted by the retrieval of the neighbour nodes for a given node in a partition. Such retrievals lead to random accesses of the DRAM, thus resulting in the poor spatial locality of data access, rendering the cache ineffective. The temporal locality of the accesses is inefficient too as the neighbours retrieved once in a certain iteration will be needed again in the subsequent iteration of the algorithm.

B. Edge-centric ordering

The congestion point of this algorithm is that the process to access the neighbours of a particular vertex mostly leads to random accesses in the memory. This is because the neighbours can be placed anywhere in the DRAM. Thus, this leads to the poor spatial locality of accessing the data, rendering the cache to be highly inefficient. Here, most of the time is spent getting the data from the DRAM. Moreover, the temporal locality of accessing the data is inefficient too, since a node's adjacency list once accessed will only be accessed in the next iteration.

One solution that can be helpful here is to retrieve the edges for a vertex in an ordered manner such that the data accesses are sequential.

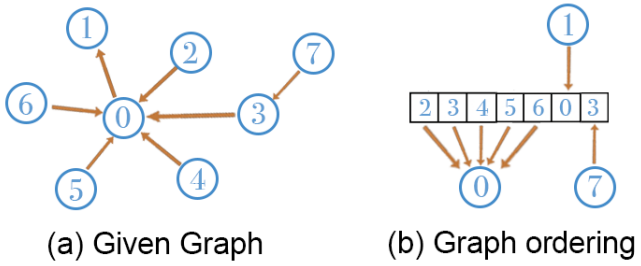

Fig. 2. Example of graph edge reordering

C. Scheduling policies in OpenMP

OpenMP provides different scheduling policies that can be used to distribute iterations to different threads in *for* loop. This can be done by leveraging the *schedule* clause in OpenMP and help optimize the way loop iterations are distributed.

1) Static scheduling: By default, the threads are assigned loop iterations statically. Each thread gets one chunk of totalNodes/totalThreads loop iterations, regardless of the number of threads. If the same computation cost is involved in each iteration, static scheduling is appropriate. Thus, it is non-optimal for the distribution of iterations taking different amounts of time.

2) Dynamic Scheduling: Suppose we have totalNodes as the total number of web pages in the graph and totalThreads as the number of threads to be spawned in OpenMP, then the loop iterations are broken up into fixed-size chunks k that each thread will execute. When the thread finishes its chunk of iterations, it will be assigned a new chunk that has not been executed yet. This is useful when we do not know about the correlation between the number of iterations and the threads and depends on the runtime. Even though it provides good workability, there is a lot of computation and management involved which presents an overhead at runtime. Therefore, the dynamic scheduling type is suitable when the iterations among the threads need different computational costs.

3) Guided Scheduling: This scheduling involves dynamic chunk size. Here, for totalNodes as the total number of web pages in the graph and totalThreads as the number of threads, the first thread is assigned a fixed size of chunk k of total iterations where k is totalNodes/totalThreads. Thereafter, the second thread is allotted a chunk size that is proportional to the yet to be assigned iterations divided by the totalThreads. In this manner, the chunk size keeps on reducing till a threshold set by the user, with the possibility that the last remaining set of iterations might be smaller than the threshold. This type of scheduling is useful when we have an imbalance in the iterations amongst the threads. Since it has to make lesser decisions at runtime than the dynamic scheduling it is slightly better than the dynamic one. On comparing it with static, guided is significantly better as it can handle the imbalanced load by using different chunk sizes from start to end.

4) Auto Scheduling: This type of scheduling involves transferring the whole control to the compiler to decide the distribution of the chunk sizes of iterations amongst the threads. This scheduling can also involve the decision of the runtime system.

D. Related Work

As observed in [1] a lot of time is unnecessarily spent on CPU cache latency in the retrieval of data. Apart from the time spent by the algorithm, this data access overhead is a real performance killer. Ailamaki et al. show that the latency of the CPU cache miss takes half of the execution time in databases. For tree algorithms, Chen et al. propose prefetching the B+ tree which leverages cache prefetch while loading the nodes of the tree to reduce the latency [2]. Similar studies related to tree algorithms has been done in [[3], [4], [5], [6]]. However, these works can not be effectively used to deal with all different graph algorithms involving massive and complex graphs. In [7], Beamer et al. propose the propagation blocking approach. In this, they initially store the messages in cached bins and then those are gathered before being written into the DRAM. Even though the number of memory accesses is lessened, eventually there are additional memory requirements as there is no optimization of the data layout.

Some existing works improve the computations on the graph by ordering the graph and node clustering. Banerjee et al. [8] improve the efficiency of DAG traversal by proposing the children depth first traversal method.

Our implementation: Here we explored a general methodology to reduce the CPU cache misses for the parallel implementation of the algorithm PageRank by taking inspiration from [9]. We implement a solution by graph ordering, which is to formulate a combination among all the edges in the given graph by keeping the edges that will be frequently accessed together for a certain node, thus minimizing the CPU cache miss ratio.

III. Implementation and Optimization

A. Graph ordering

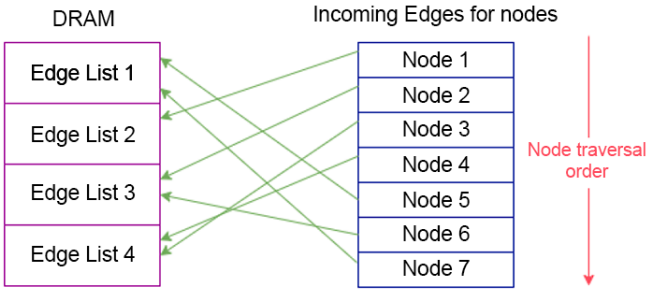

Fig. 3. Incoming edge list

Here we model a graph as directed graph G =(V,E), where G.V represents the vertices and the G.E is for the set o edges. The number of nodes are denoted as $n = |G.V|$ and total edges as $m = |G.E|$. We represent the incoming edges to a node u as $N_{in}(u)$ as the in-neighbour set such that $N_{in}(u) = v|(v, u) \in G.E$.

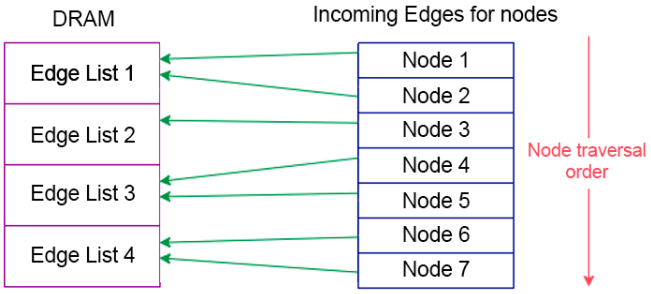

Fig. 4. Sorted Incoming edge list

Our main implementation for CPU speedup for the PageRank algorithm is to represent a graph in such a manner that can reduce the CPU cache miss ratio. This general approach can be easily used for many other graph-based algorithms. Here, what is meant by reducing the CPU cache miss ratio is to reduce the number of times the data is copied from the main

memory to the cache, or from L3 cache to L2 cache or L2 cache to L1 cache. To pull it off we keep the edges frequently accessed together stored adjacently in the main memory so that these edges are loaded into the cache in conjunction with a single transfer of the cache line.

B. OpenMP Implementation

In this implementation, we used the graph layout optimization to run the algorithm in parallel on threads using OpenMP [10]. We divided the total set of nodes amongst certain threads. The types of OpenMP thread scheduling used were static, dynamic, guided and auto. This helped in analyzing and comparing the different scheduling for the algorithm and also how the chunk size mattered in different schedulings.

C. pthreads Implementation

pthreads, POSIX Threads [11] is a parallel execution model. It helps in controlling multiple different flows of a task that usually overlap in time. It gives much granular control over the threads and their lifecycle by providing POSIX Threads API to make calls to the threads. We worked on this implementation to compare the efforts required to parallelize a program in OpenMP versus pthreads. We know that this implementation is not using the same graph memory layout optimization but it helped us in the desired comparison tasks. During the whole algorithm we create and joined three times - first for initialization of the partitions, second for running the main parallel PageRank algorithm, and thirdly for updating parameters and checking if the algorithm is complete or needs another iteration.

IV. Experimental setup and Dataset

A. Experimental Setup

We have carried out the experiments on NYU HPC cluster which provides a platform for executing CPU-intensive jobs. The experiments done were on the Linux server with four AMD Opteron 6272 processors with a total of 64 cores and each core is 2.1 GHz running CentOS Linux 7 operating system. The total memory being 256 GB. The per-core L1 cache size is 16kB and L2 cache size is 2048 kB and the shared L3 cache size is 6144 kB. The code is in the language C++ and GCC9.2 along with optimization was used for compilation purposes. We initially profiled the code using *gprof* which helped in analyzing the time-intensive portions of the code which were later parallelized. Secondly, we also used the *perf* tool to analyze and compare the cache and memory statistics.

B. Metrics for comparison

For the performance analysis, we have mainly focused on the execution time. We run for a maximum of 20 iterations to compare it with the sequential benchmark of the PageRank algorithm [13]. Then we used a baseline parallel implementation of the algorithm [14].

Apart from the four different scheduling policies in OpeMP, we used a different number of threads - 1, 2, 3, 8, 16, 32 and 64 to compare the performance. Moreover, different chunk sizes of 1,16 and 32 were used to make further comparisons. For pthreads implementation, we experimented with 1, 2, 4, 8 and 16 threads.

C. Datasets

We have used two large-scale and synthetic datasets for our evaluation purposes of Baseline and the optimized algorithm. We used web-Google:*Web graph from Google* and web-Stanford:*Web graph of Stanford* synthetic graphs downloaded from [12]. Some important statistics can be found in the table Table 1 below.

TABLE I
GRAPH DATASETS USED

Dataset	Vertices	Edges	Average degree	Type
web-Google	0.88M	5.1M	5.8	Directed
web-Stanford	0.28M	2.3M	8.2	Directed

We also observed that the graphs were not symmetric and had a load of imbalance when talking in terms of the nodes and edges per partition. This can be seen in both the graphs that have been used. For instance, in web-Google we can see that a node has maximum total 541k outgoing edges and on the other hand a node has maximum total 542k incoming edges. Similarly, in the other graph, it can be seen that a node has maximum total of around 250 outgoing edges and on the other hand a node has a maximum total 84k incoming edges. This imbalance analysis led us to experiment with different scheduling with different chunk sizes.

D. Input Graph format

In the baseline models used we see that the input graph is of format where each line represents an edge from node1 to node2. This was used to get performance metrics for BaselineSeq [13] and BaselineParallel [14] to compare it with our OptParallel. In our implementation we have used the Compressed sparse row (CSR) segmenting, a design in which accesses to DRAM are sequential, and accesses to the cache are limited. With CSR Segmenting, unlike disk-based solutions, we were able to achieve higher scalability and reduced overhead through a compressed 1D-segmented graph. We have two input lists in the input file. We have a list of the size of the vertices and the values in them are the offsets into another list which is called the edge array. The offset in the vertex list transports us into the edge array to fetch the neighbours of a particular node until the offset of the next adjacent node of the source node. The edge list has the source nodes stored in a sorted manner along with the destination vertices being sorted as well. This format helps us in pre-processing the graph to generate the sorted incoming neighbour list for each node, reducing random DRAM accesses.

V. Experimental Results and Analysis

A. OpenMP Analysis

Based on our experiments, we present the results in terms of execution time and other metrics like cache misses and

memory accesses.

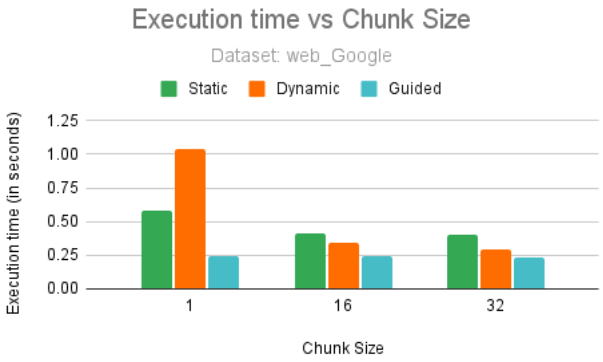

Fig. 5. Example of a figure caption.

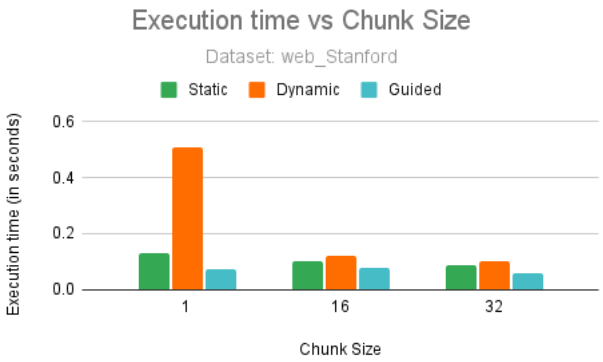

Fig. 6. Example of a figure caption.

1) Scheduling Policies and Chunk sizes: Overall, a bigger chunk size results in the faster running of the program as shown in Fig. 5 and 6. According to expectations, the improvement is more pronounced in dynamic and guided policies, where chunk assignment is the primary bottleneck. This is because the scheduling policies create overhead while assigning chunks to threads.

2) Number of threads and scheduling policies: In keeping with what is expected and suggested in Fig.7 and 8, guided policies take the shortest time to execute. As guided scheduling can handle imbalanced load more effectively and take fewer decisions than pure dynamic scheduling, there is less overhead. The speedup obtained by dynamic scheduling, indicates the graph was not well balanced, causing some iterations to take longer than others. As can also be seen from the chart above, the running time for the program is precisely determined by Amdahl's law. Rather than gaining higher performance with fewer concurrent processes, it loses this advantage when more threads are used. However, 128 threads were also tested, but no results were reported given that the running time was similar to the 64 thread version.

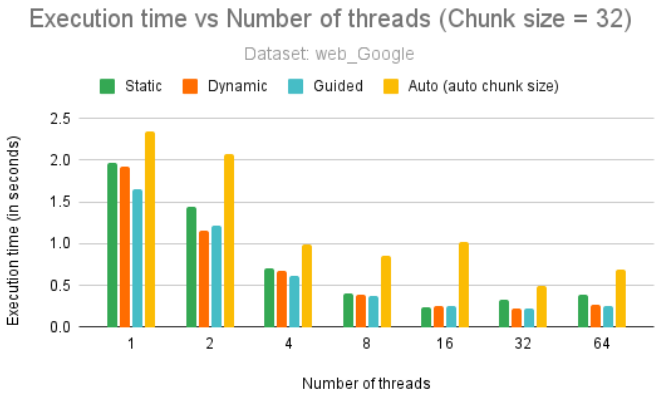

Fig. 7.

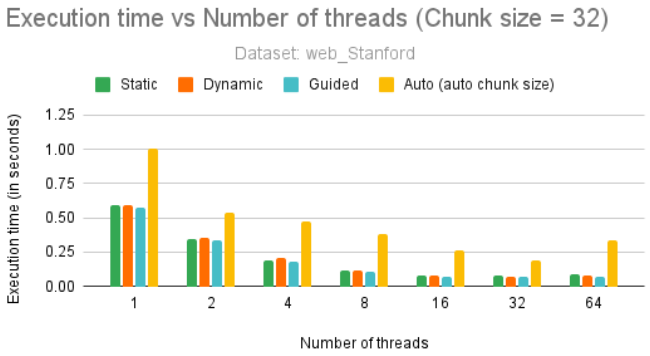

Fig. 8.

As shown in Fig.7 and 8, auto-scheduling policies have also experimented with the increasing number of threads but the results were not as good as other policies. Hence, we can infer the that auto policy is not enough for large datasets, and one needs to devise the most suitable policy based on the experiments.

Fig. 9.

3) Execution time comparison in algorithms: It's no surprise that optimized algorithms perform better than both baseline algorithms as shown in Fig 9. Comparing the sequential Graph Challenge benchmark with the optimized algorithm, the execution time improves by 1.9x to 2.6x. When looking at the multi-threaded Graph Challenge benchmark, the execution time improves by a little 1.2x to 1.6x. The optimized data layouts reduce the number of random accesses. Thus, our optimized algorithm reduces random access significantly compared with the baseline algorithm, resulting in higher sustained memory bandwidth and better execution times.

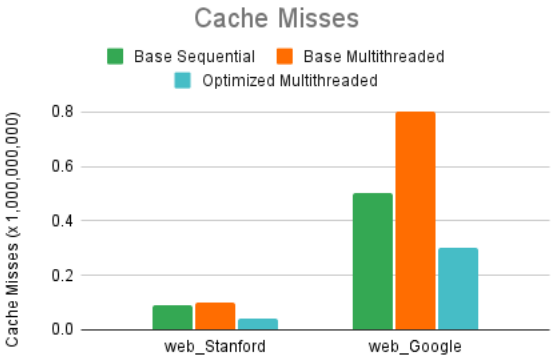

Fig. 10.

4) Analyzing Cache Misses: Many memory accesses lead to high execution time as a result of the high number of cache misses. In comparison to sequential and multithreaded PageRank Pipeline Benchmarks, the optimized algorithm cuts down the total number of cache misses by 1.7x to 2.25x. Because multiple threads are sharing the cache, multi-threaded implementations may cause more cache misses. Page Rank's random access nature accounts for the high number of cache misses. As a result of the pre-fetching nature of the memory controller, a high number of cache lines may be fetched with only a small portion of them being useful due to the nature of pre-fetching. Alternatively, we stream the edges from memory and write the updates to memory in a streaming manner in our optimized algorithm. As a result, cache misses are reduced (Fig.10). We also observed that Stanford data being smaller than Google has holistically less number of cache misses.

B. Pthreads

Fig.11 and 12 show that with an increase in the number of threads, the execution time decreases. Because of the large data size, this behavior is expected since the number of threads increases with the data size.

C. Pthreads and OpenMP usability

Multi-processing paradigms associated with Pthreads and OpenMP differ. Using Pthreads, you can work with threads on a very low level. In contrast, OpenMP provides a much higher

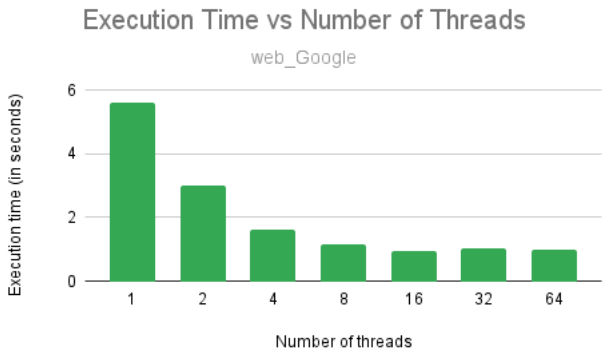

Fig. 11.

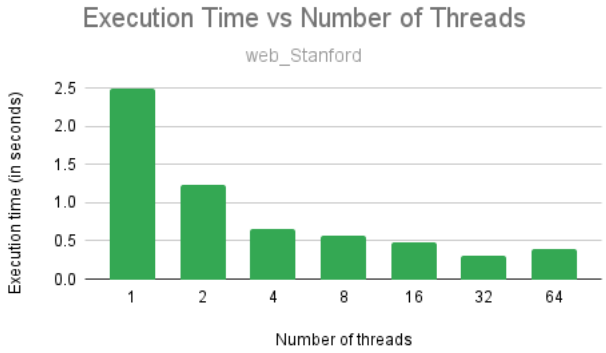

Fig. 12.

level of abstraction. The algorithm could be rewritten easier using OpenMP. For Pthreads to perform as well as OpenMP, you have to do a lot more work. Despite that, Pthreads offers a much finer level of thread management control.

VI. Conclusion

PageRank is implemented on multicore platforms in a parallel fashion. Parallel computation was carried out for each partition in the input graph by dividing the graph into partitions and executing different threads for each partition. Compared to the sequential Graph Challenge benchmark, our implementation achieved 1.9x to 2.6x faster execution times for 2 big datasets in real-world settings. We saw an improvement of 1.3x to 1.5x in the time required to run the multithreaded Graph Challenge benchmark. A significant reduction in cache misses was achieved through our implementation. A policy with a chunk size of 32, which was guided, had the best performance. Pthreads was also used and we found that it is a lot more fine-grained in its control over thread management, but that it is consuming a lot more work as compared to OpenMP.

It is possible to extend this implementation to any graph-based parallel computation problem in the future, such as Breadth-first search, Dijkstra algorithm. Moreover, by using OpenMP SIMD vector processing and using inputs as vectors for PageRank one can get significant improvements using OpenMP.

REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In Proc. of VLDB'99, 1999.

[2] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In Proc. of SIGMOD'01, 2001.

[3] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In Proceedings of PLDI, Atlanta, Georgia, USA, 1999.

[4] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In Proc. of VLDB'05, 2005.

[5] P. Lindstrom and D. Rajan. Optimal hierarchical layouts for cache-oblivious search trees. In Proc. of ICDE'14, 2014.

[6] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In Proc. of VLDB'99, 1999.

[7] S. Beamer, K. Asanovic, and David Patterson, "Reducing Pagerank Communication via Propagation Blocking," in Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017.

[8] J. Banerjee, W. Kim, S. Kim, and J. F. Garza. Clustering a DAG for CAD databases. IEEE Trans. Software Eng., 1988.

[9] H. Wei, J. X. Yu, and C. Lu, "Speedup Graph Processing by Graph Ordering," in Proc. of International Conference on Management of Data (SIGMOD), pp. 1813-1828, 2016.

[10] OpenMP, https://computing.llnl.gov/tutorials/openMP/

[11] pthreads, https://man7.org/linux/man-pages/man7/pthreads.7.html

[12] Stanford Large Network Dataset Collection https://snap.stanford.edu/data/

[13] PageRank Pipeline Benchmark "https://github.com/vijaygadepally/PageRankBenchmark

[14] GAP Benchmark Suite - https://github.com/sbeamer/gapbs