Graph Optimization for GAS based Parallel PageRank

Abhishek Verma
New York University
New York, NY, USA
av2783@nyu.edu

Shivanshi
New York University
New York, NY, USA
ss14396@nyu.edu

Ishita Jaisia
New York University
New York, NY, USA
ij2056@nyu.edu

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

***Abstract*— PageRank is a graph-based algorithm to rank the vertices in a given graph in the order of their importance. Due to the poor locality of data in non-symmetric huge real-world graphs, many implementations (even though parallel) suffer from the random DRAM access problem. We present an implementation of an efficient parallel PageRank based on an edge-centric scatter-gather model which optimizes the memory performance. We partition the vertices into bins of non-overlapping vertices such that the data of each bin is small enough to fit in the cache. Then, based on the destination vertices, we sort the outgoing edges from each bin - this significantly reduces random memory read and writes. It allows for an efficient parallel implementation of PageRank like algorithms on multicore platforms. We test our implementation on a machine with 64 AMD Opteron(TM) 6272 processors (each with 8 cores) and use two large-scale real-life datasets for evaluation. We test our implementation using various chunk sizes with static, dynamic and guided scheduling policies in OpenMP. Compared to an implementation that does not use sorted edges, our implementation achieves 2x to 3x speedups for these datasets.**

As a side project, we also implement a parallel version of PageRank using Pthreads to compare the efforts required to modify a sequential program using OpenMP and Pthreads. The results show that OpenMP performs better than Pthreads.

I. INTRODUCTION

PageRank was developed in 1999 in the paper[1]. There was a need for an algorithm to rank the ever-increasing number of web pages indexed by Google and PageRank was one of the best efforts in that direction. The internet is completely open and democratic, which while great for people like us can be a headache for search engines like Google. Since anyone can start a website on the internet, it is filled with low-quality content, fake news and even websites that are dangerous to visit. While Google's current search algorithm is bound to be much more complex and nuanced than PageRank, the PageRank algorithm does provide a wonderful insight into how the internet can be thought of as a graph.

However, the problem is more complicated than it sounds. Due to the large number of web pages on the internet, the resulting graph is huge. A single execution of sequential PageRank on this graph can take a whole lifetime to terminate and thus is impractical. This gives rise to the need for an efficient parallel version of PageRank. It is easier said than done though. It is challenging to achieve high-performance large-scale graph computation. This is mainly because the data of emerging graph applications are massive and most graph problems exhibit poor spatial and temporal locality of memory accesses [2][3][4][5]. As a result, the execution time is dominated by external memory accesses. To improve memory performance, prior work [6][7] examined improving the graph layout or reordering the computation to increase locality. These optimizations are often beneficial, but also introduce significant preprocessing overhead.

In this paper, we present an implementation of an efficient parallel PageRank based on an edge-centric scatter-gather model which optimizes the memory performance and has improved speedup compared to other implementations based on the same idea.

II. BACKGROUND AND RELATED WORK

A. PageRank

The PageRank algorithm [1] is a widely used algorithm for ranking the importance of vertices in a graph. It computes the PageRank value of each vertex which indicates the likelihood that the vertex will be reached. A higher PageRank value corresponds to more importance. The PageRank score of a vertex is the sum of the PageRank score of the vertices that it receives an edge from divided by their out-degree. Initially, the PageRank of each vertex is initialised to (1/ number of vertices in the graph). The standard serial algorithm proceeds in iterations with PageRank being computed for each vertex in an iteration until the change in the value of PageRank of a vertex in an iteration is less than a certain threshold called tolerance. The formula for PageRank that is used more frequently in practice is shown in Fig. 1.

$$PageRank(v) = \frac{1-d}{|V|} + d \times \sum \frac{PageRank(v_i)}{L_i} \quad (1)$$

Fig. 1. PageRank equation

V is the set of vertices of the graph, N+(v) is the set of vertices that receive an edge from v and N-(v) is the set of vertices that v receives an edge from. d is called the teleportation factor and is usually set to 0.85. (1-d) is the probability that a surfer may randomly navigate away from the current webpage to one that it does not contain a link to. When the input graph is altered as in analyzing streaming data, the PageRank algorithm does not traverse the entire graph[11]. In this paper, we focus on accelerating the PageRank algorithm for static graphs.

B. Gather-Apply-Scatter Model

There are three paradigms when it comes to executing PageRank in parallel on large graphs: in-memory, out-of-core and distributed. In-memory computation loads the whole graph into the main memory of the server. Out-of-core computation keeps the graph on disk and loads parts of it into main memory as required. Distributed computation splits the graph across multiple servers and coordinates the execution of PageRank between the servers. In this paper, we focus on in-memory parallel PageRank since it is the fastest.

The general algorithm for an in-memory topology driven PageRank is given in Fig. 2.

Input: graph $G = (\mathcal{V}, \mathcal{E})$, α, ϵ
Output: PageRank $\mathbf{x}$

1: Initialize $\mathbf{x} = (1-\alpha)\mathbf{e}$
2: **while** true **do**
3: **for** $v \in \mathcal{V}$ **do**
4: $x_v^{(k+1)} = \alpha \sum_{w \in \mathcal{S}_v} \frac{x_w^{(k)}}{|\mathcal{T}_w|} + (1-\alpha)$
5: $\delta_v = |x_v^{(k+1)} - x_v^{(k)}|$
6: **end for**
7: **if** $\|\boldsymbol{\delta}\|_\infty < \epsilon$ **then**
8: break;
9: **end if**
10: **end while**
11: $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$

Fig. 2. PageRank algorithm

x is the PageRank vector, e is a unit vector, alpha is 'd' that we discussed above and epsilon is the tolerance. On line 4, S_v is the set of vertices that vertex v receives an edge from. T_w is the set of vertices that receive an in-edge from vertex w. This is a vectorized version of the same algorithm that we discussed above.

The bottleneck of this algorithm is that accessing the neighbours of a vertex leads to random DRAM accesses, since the neighbours may be located anywhere in DRAM. This leads to the poor spatial locality of data access, which causes the cache to be ineffective and most of the time is spent trying to get data from the DRAM. Note that the temporal locality of data access is very poor too since a vertex's adjacency data once accessed will only be accessed again in the next iteration.

A potential solution to this problem is to apply the Gather-Apply-Scatter model(GAS) [5][8]. This divides the computation into three phases:

1) Gather: In this phase, each vertex reads the updates to it's in-neighbors' PageRank values from a dedicated location in memory where they are written by the neighbors.
2) Apply: In this phase, the vertex updates its own PageRank value since it has received the updated values from its neighbours.
3) Scatter: In this phase, the vertex writes its new PageRank value in the dedicated memory locations of its out-neighbors for them to process in the next iteration.

The advantage of doing this is that in the Gather phase, the updates are stored serially which improves spatial locality of data access.

C. Scheduling Policies In OpenMP

When dividing iteration among threads, there are various policies which can be used. The *schedule* clause in OpenMP provides three such policies out of the box, namely:

1) Static Policy: For N iterations and T threads, all threads equally get one chunk of N/T of loop iterations. The static scheduling type is appropriate when all iterations have the same computational cost.
2) Dynamic Policy: For N iterations and T threads, each thread gets a fixed-size chunk of k loop iterations and when a particular thread finishes its chunk of iterations, it simply gets assigned a new chunk. So, the relationship between iterations and threads is non deterministic and would only be decide at run-time. However, despite its good flexibility, this schedule presents a high overhead due the lots of decision regarding which thread gets each chunk. Thus, the dynamic scheduling type is appropriate when the iterations require different computational costs, that is, the iterations are poorly balanced between each other.
3) Guided Policy: For N iterations and T threads, initially the first thread gets a fixed-size chunk of k=N/T loop iterations. Afterward, the second thread gets a chunk of size proportional to the number of iterations that have not been assigned divided by the number of the threads. Therefore, the size of the chunks decreases until a minimum threshold decided by the user. However, the chunk which contains the last iterations may also have smaller size. Its advantage over the static policy is that it can better handle imbalanced load and it also takes fewer decisions than the pure dynamic version, thus causing less overhead. The guided scheduling type is appropriate when the iterations are poorly balanced between each other. The initial chunks are larger, because they reduce overhead. The smaller chunks fill the schedule towards the end of the computation and improve load balancing. This scheduling type is especially appropriate when poor load balancing occurs toward the end of the computation.

D. Related Work

The performance of PageRank is bounded by the external memory (i.e., DRAM) accesses [7]. Thus, many

prior works focus on optimizing the memory performance [6][7][9]. In [6], a graph reordering approach is proposed to improve locality and reduce cache misses. The reordering approach identifies the optimal permutation among all the vertices in a given graph by keeping the vertices that will be frequently accessed together. However, the pre-processing overhead of [6] is non-trivial. In [9], an FPGA design to accelerate the PageRank algorithm is developed. In [7] , Beamer et. al. propose the propagation blocking approach, which first stores messages in cached bins and accumulates them before writing into DRAM. This approach reduces the number of memory accesses but results in additional memory requirement. Moreover, the design in [7] does not optimize the data layout; thus, random memory accesses still occur when writing messages into the memory

III. IMPLEMENTATION AND PROPOSED OPTIMIZATION

A. Graph Partitioning

Since cache and memory performance have significant impact on the performance of PageRank implementation [7][9], we use the graph partioning approach presented in [14] to improve cache performance and eliminate random memory accesses to the DRAM. Assuming that the input graph is stored in COO format [5][10][11], all the vertices are divided into bins of equal size. If each bin has m vertices, the graph is partitioned into $|V|/m$ bins. Each bin has an edge list and a message list. The edge list stores all the edges whose source vertices are in the vertex set of the bin; the message list stores all the messages whose destination vertices are in the vertex set of the bin. The edge list of each bin remains fixed during the entire computation; the data of message list is recomputed in every scatter phase; the data of vertex set is updated in every gather phase. Fig. 3 shows an example data layout after the graph data are partitioned into three partitions.

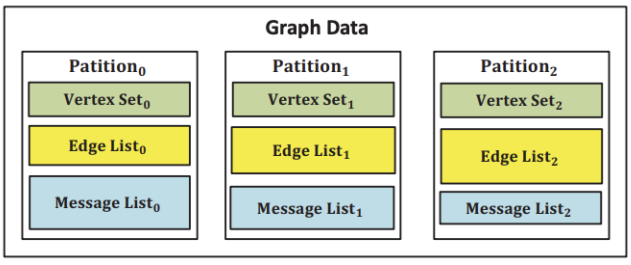

Fig. 3. Graph partitioning

Note that the data of each bin is uniform in size and hence, the memory requirement of each vertex set is identical. Edge lists and message lists can be different in size; the memory requirement of each edge list depends on the number of edges whose source vertices are in the corresponding bin; the memory requirement of each message list depends on the number of edges whose destination vertices are in the corresponding bin. Fig. 4 illustrates the PageRank algorithm based on graph partitioning.

Algorithm 1 PageRank based on graph partitioning

Let m denote the number of vertices in each vertex set

```
1: while not done do
2:   Scatter:
3:   for each partition do
4:     for each edge e in edge list do
5:       Read the data of Vertex e.src
6:       Let v = Vertex e.src
7:       Produce a message msg
8:       msg.value = v.PageRank / v.#_of_outgoing_edges
9:       msg.dest = e.dest
10:      Write msg into message list of Partition ⌊e.dest/m⌋
11:    end for
12:  end for
13:  Gather:
14:  for each partition do
15:    for each message msg in message list do
16:      Update PageRank of Vertex msg.dest
17:    end for
18:  end for
19: end while
```

Fig. 4.

Partitioning the graph leads to two benefits:

1) The scatter and gather phases of distinct partitions can be performed in parallel on multi-core platforms
2) We can choose the size of the vertex set of each partition based on the on-chip memory resources (i.e., cache size), such that vertex data can be cached for efficient data reuse.

B. Graph Layout Optimization

Every memory access sequence can be partitioned into a set of sequential memory access sequences, where a sequential memory access sequence is a sequence of accesses to contiguous memory locations. The size of such a set gives the number of random accesses in the sequence. In the algorithm above, the memory accesses to read edges are sequential and the vertex data are read from the cache. However, it is writing messages into DRAM which incurs the cost of random memory accesses. Fig. 5 shows an example of random memory access at the time of writing messages into DRAM.

In order to minimize the number of random memory accesses due to writing messages, we use an optimized data layout that sorts the edge list of each partition based on the destination vertices. Fig. 6 shows the optimized data layout for the example in Fig. 5

C. OpenMP Parallel Implementation

We use the optimized graph layout and perform parallel execution based on the following algorithm:

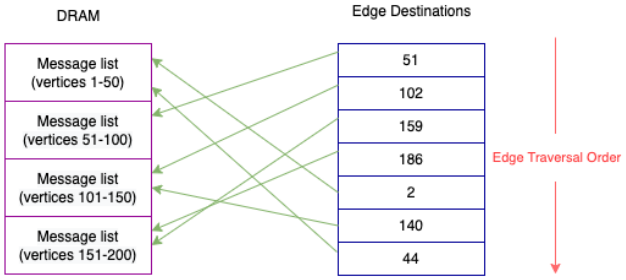

Fig. 5. Random memory accesses due to writing messages into DRAM

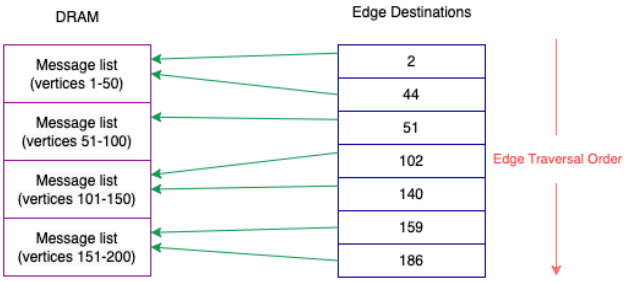

Fig. 6. Optimized data layout for the example in Fig. 5

As shown in the algorithm, assuming the multi-core platform has p cores, we divide all the computations of b bins into various chunk sizes and use static, dynamic and guided scheduling policies in OpenMP [12] to execute each chunk on one of the p cores. All the cores execute the computations of distinct partitions in parallel.

D. Pthreads Parallel Implementation

In order to quantify the efforts required to modify the sequential program using OpenMP and Pthreads, Parallel PageRank was also implemented using Pthreads. This implementation resulted in extremely fine-grained control over thread management. In each iteration of the PageRank algorithm, threads are created and terminated twice - one for initializing the vertices with the old scores and the other for computing the new scores. This implementation is not based on the graph layout optimization suggested above, Instead, this was a side project to figure out various differences between implementing the same algorithm in OpenMP v/s Pthreads.

```
Let b denote the number of bins
while not converged do
    for i from 1 to b parallel do
        Execute scatter phase of Partition i
    end for
    barrier
    for i from 1 to k parallel do
        Execute gather phase of Partition i
    end for
    barrier
end while
```

IV. EXPERIMENT AND DATASET DESCRIPTION

A. Experimental Setup

We conducted experiments on a Linux server equipped with 64 *AMD Opteron(TM) 6272* processors - each containing 8 cores - running *CentOS Linux 7 (Core)* operating system. The per-core L1 and L2 cache sizes for this CPU are 16 KB and 2048 KB respectively, and the shared L3 cache size is 6144 KB. All code is written in C++ and compiled using GCC 4.8.5 with the highest optimization -O3 flag. Cache and memory statistics are collected using *perf* tool.

B. Performance Metric and Baselines

We analyze the performance using total execution time as the metric. The total execution time is measured by running the PageRank algorithm for 20 iterations. We compare the performance of our optimized algorithm against two baselines. The PageRank Pipeline Benchmark (Base_Sequential) [13] serves as an initial baseline consisting of a sequential single-threaded PageRank implementation. The multithreaded version of the Graph Challenge benchmark serves as the second baseline (Base_Multithreaded). We denote our optimized algorithm as Opt_Multithreaded.

The multithreaded implementation in OpenMP is run using 1, 2, 4, 8 and 16 threads. The scheduling policies used are static, dynamic and guided - each with 3 different chunk sizes - 1, 16 and 32.

The Pthreads implementation is run with 1, 2, 4, 8 and 16 thread configurations.

C. Graph Representation

The baseline algorithm uses CSR format[11] of graph representation. There are two arrays in the graph: vertex array of size equal to a number of vertices and edge array of size equal to a number of edges. Element in vertex array is an offset into edge array, representing the location of the first edge to the vertex. Edge array contains source indices of all edges, sorted by the destination. In the optimized algorithm, the graph is stored as a list of vertices and edges (i.e., COO format [13]). Each vertex consists of vertex ID and PageRank value. Each edge is specified by the source, destination and weight of the edge. The messages are denoted by the destination vertex and the update to the attribute of the vertex.

D. Datasets

We use two real-world datasets to evaluate the performance of the Baseline and optimized algorithms. The two datasets are we-BerkStan and web-Google was taken from [15] The key properties of the graph datasets are summarized in the table below:

Dataset	# Vertices	# Edges	Average Degree
web-BerkStan	0.69 M	7.6 M	11.1
web-Google	0.88 M	5.1 M	5.8

In addition, the vertex with most outgoing connections had a total of 541820 outgoing edges in web-Google and 249 edges in web-BerkStan, while the node receiving most input connections had not more than 542009 incoming edges in web-Google and 84208 web-BerkStan. Thus, the graphs are not symmetric and these imbalances could result in different workloads while computing the rank of different nodes. Hence, it makes sense to try out different scheduling policies with different chunk sizes.

V. RESULTS AND ANALYSIS

A. OpenMP

We present the results of our experiments in terms of execution time and other metrics such as cache misses and memory accesses.

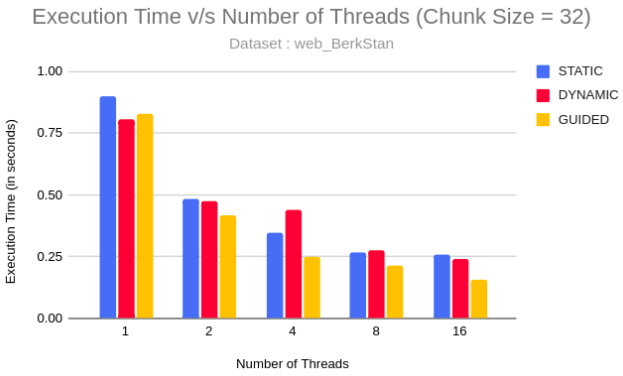

Fig. 7.

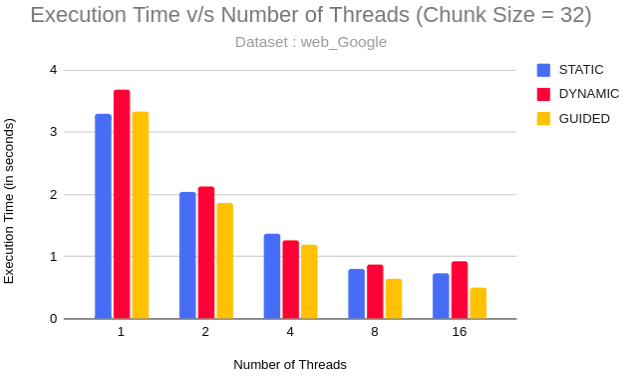

Fig. 8.

1) Scheduling Policies and Number of Threads: As expected and suggested by the Fig. 7 and Fig. 8, the execution time is least in the case of a guided policy. This is because guided scheduling can better handle the imbalanced load and takes fewer decisions than the pure dynamic version, thus causing less overhead. The speedup obtained by the dynamic scheduling suggests that the analyzed graph was very unbalanced, thus resulting in some iterations taking much more time than others. We can also note how the program's running time for different numbers of threads followed Amdahl's law. In fact, it benefits of bigger relative speedups when the number of multiple processes remains low and its increase in speedup cools down as more threads are used at the same time. The program has also been tested running with 32 threads but the results have not been reported since the running time was similar to the 16 threads version.

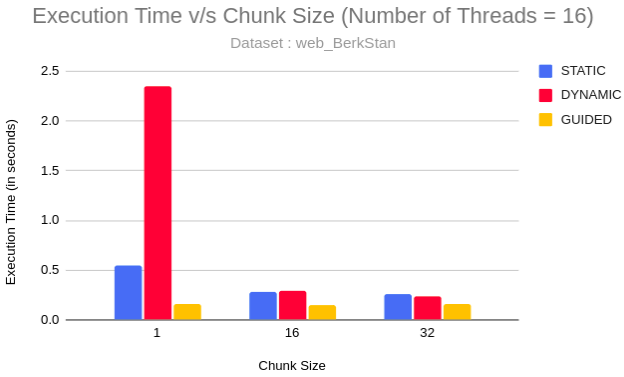

Fig. 9.

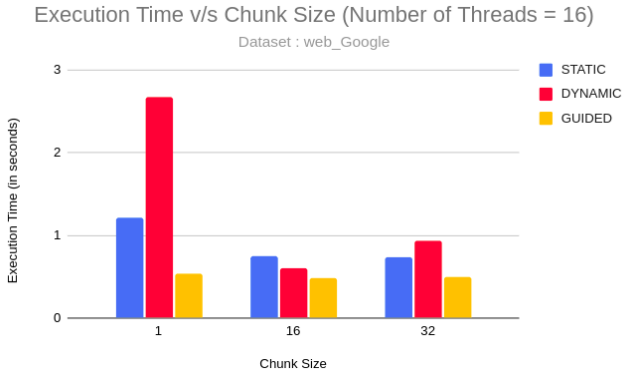

Fig. 10.

2) Chunk Size in Scheduling Policies: Fig. 9 and Fig. 10 present the effect of the increase of chunk size on the different scheduling policies. Overall, the program runs faster when using a bigger chunk size. This is due to the overhead generated by the scheduling policies while assigning chunks to threads. As expected, the improvement is much more remarked in the dynamic

and guided policy where the chunk assignment process is the main bottleneck.

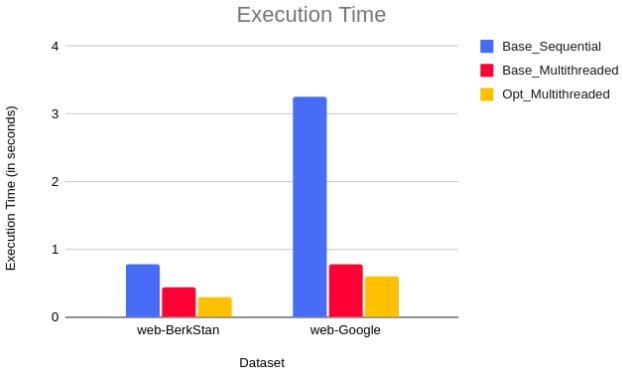

Fig. 11.

3) Execution Time: Fig. 11 illustrates the comparison of baseline and optimized algorithms. The optimized algorithms outperform both the baseline algorithms. The optimized algorithm improves the execution time by 2.6× to 5.3× compared with the sequential Graph Challenge benchmark. With respect to the multithreaded Graph Challenge benchmark, we observe an improvement of 1.3× to 1.5× in the execution time. The reasons behind these improvements are:

1) Bins are accessed in a regular manner with accesses to the vertices and edge in each bin sequential in nature (because of the sorted order of vertices in each bin).
2) Data layout optimization reduces the number of random accesses while writing the messages into memory during the scatter phase. Therefore, in comparison with the baseline, our optimized algorithm significantly reduces random accesses leading to higher sustained memory bandwidth and lower execution time.

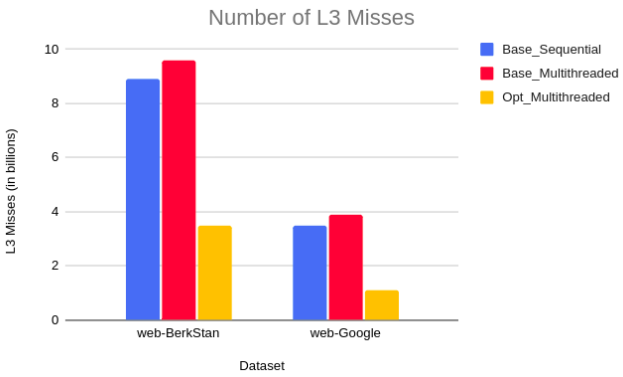

Fig. 12.

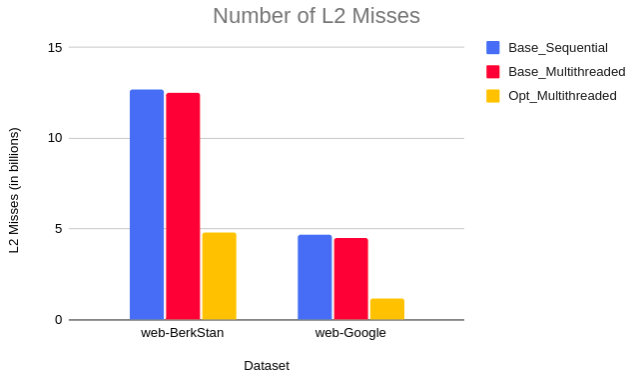

Fig. 13.

4) Cache Miss: The high number of cache misses translates to a high number of memory accesses leading to high execution time. As depicted in Fig. 12, the optimized algorithm reduces the total number of L3 misses by 2.5× to 3.6× in comparison with both the sequential and multithreaded PageRank Pipeline Benchmarks. A multithreaded implementation may cause more cache misses compared to sequential implementation due to multiple threads sharing the available cache. The high number of cache misses can be attributed to the random access nature of the baseline PageRank implementation. Furthermore, due to the nature of pre-fetching in the memory controller, the random accesses can lead to a higher number of cache lines being fetched with only a small portion of the cache line being useful. On the other hand, in our optimized algorithm, the edges are streamed from the memory and the updates are written to the memory in a streaming fashion. While processing a partition, the random accesses are limited to the vertex data which are stored in the cache of the processor. These optimizations reduce the number of cache misses. A similar analysis applies to the total number of misses in the L2 cache. As illustrated in Fig. 13, the optimized algorithm achieves a significant reduction in the number of L2 misses compared with the baselines. The multithreaded PageRank Pipeline Benchmark implementation has a similar number of cache misses as that of the sequential PageRank Pipeline Benchmark as each core has a private L2 cache.

5) Memory Access: The total number of memory accesses has a significant impact on the total execution time. Fig. 14 compares the total number of memory accesses of baseline and optimized algorithms. It can be observed that the optimized algorithm reduces the total number of memory accesses by 1.4× in comparison with both the baseline algorithms. The optimized algorithm in addition to reducing the number of memory accesses reduces the random accesses to memory thereby achieving higher sustained memory bandwidth and lower execution time.

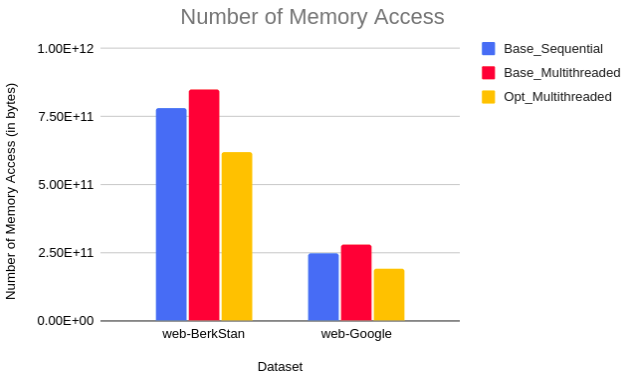

Fig. 14.

B. Pthreads

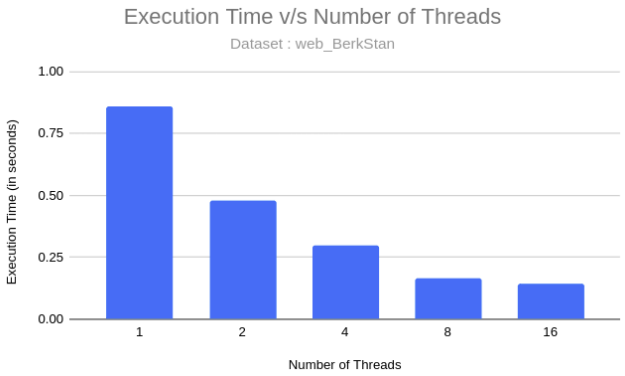

Fig. 15.

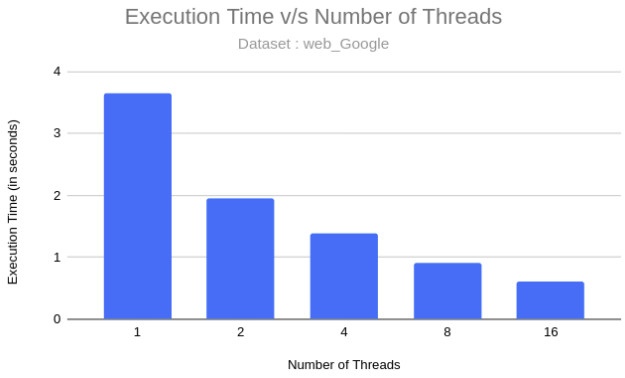

Fig. 16.

Fig. 15 and Fig. 16 suggest that the execution time decreases with an increase in a number of threads. This is expected behaviour because the data size is huge, thus resulting in significant speedup as the number of threads increase.

C. OpenMP v/s Pthreads

Pthreads and OpenMP represent two different multi-processing paradigms. Pthreads is a very low-level API for working with threads. On the other hand, OpenMP is a much higher level. With OpenMP, it was easier to rewrite the algorithm. In Pthreads, the work required to get the same performance as OpenMP is a lot more. Having said that, Pthreads provides a lot more fine-grained control over the thread management process.

VI. CONCLUSION

We presented an efficient parallel PageRank implementation on multicore platforms. Input graph was partitioned into bins and distinct threads were used to execute the computation for different bins in parallel. Results showed that our implementation achieved 2.6× to 5.3× improvement in execution time compared with the sequential Graph Challenge benchmark for 2 huge real-world datasets. With respect to the multithreaded Graph Challenge benchmark, we observed an improvement of 1.3× to 1.5× in the execution time. Our implementation reduced the total number of L3 and L2 cache misses by a significant factor. We also observed that the guided dynamic policy with a chunk size of 32 had the best performance. We also implemented non-optimized PageRank on Pthreads and found that Pthreads provides a lot more fine-grained control over the thread management process but the amount of work is increased by a significant amount compared to OpenMP.

In future, this implementation can be extended to any graph based parallel computational problem. This can also be shipped out as a C++ library for graph computations.

REFERENCES

[1] L. Page, S. Brin, M. Rajeev, and W. Terry, “The PageRank Citation Ranking: Bringing Order to the Web,” Technical Report, 1998
[2] “Graph 500,” http://graph500.org/
[3] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, “Static Graph Challenge: Subgraph Isomorphism,” IEEE HPEC, 2017
[4] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, D. Staheli, and S. Smith, “Streaming Graph Challenge: Stochastic Block Partition,” IEEE HPEC, 2017.
[5] A. Roy, I. Mihailovie, and W. Zwaenepoel, “X-Stream: Edge-centric Graph Processing using Streaming Partitions,” in Proc. of SOSP, pp. 472-488, 2013.
[6] 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.
[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] S. Zhou, C. Chelmis, and V. K. Prasanna, “High-Throughput and Energy-Efficient Graph Processing on FPGA,” in Proc. of FCCM, pp. 103-110, 2016.
[9] S. Zhou, C. Chelmis, and V. K. Prasanna, “Optimizing Memory Performance for FPGA Implementation of PageRank,” in Proc. of IEEE International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2015.
[10] S. Zhou, C. Chelmis, and V. K. Prasanna, “Accelerating Large-scale Single-source Shortest Path on FPGA,” in Proc. of IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015.

[11] A. McLaughlin, “Accelerating Graph Betweenness Centrality with CUDA,” https://devblogs.nvidia.com/parallelforall/ accelerating-graph-betweenness-centrality-cuda/
[12] “OpenMP,” https://computing.llnl.gov/tutorials/openMP/
[13] “Pre-Challenge: PageRank Pipeline Benchmark,” https://github.com/vijaygadepally/PageRankBenchmark
[14] S. Zhou, K. Lakhotia, S. G. Singapura, H. Zeng, R. Kannan, V. K. Prasanna, J. Fox, E. Kim, O. Green, D. A. Bader, "Design and Implementation of Parallel PageRank on Multicore Platforms,"
[15] Stanford Large Network Dataset Collection, "SNAP," https://snap.stanford.edu/data/