Design and Implementation of Parallel PageRank on Multicore Platforms

Shijie Zhou[1], Kartik Lakhotia[1], Shreyas G. Singapura[1], Hanqing Zeng[1], Rajgopal Kannan[2], Viktor K. Prasanna[1], James Fox[3], Euna Kim[3], Oded Green[3], David A. Bader[3]

[1]Ming Hsieh Department of Electrical Engineering, University of Southern California
[2]US Army Research Lab
[3]School of Computational Science and Engineering, Georgia Institute of Technology

***Abstract*—PageRank is a fundamental graph algorithm to evaluate the importance of vertices in a graph. In this paper, we present an efficient parallel PageRank design based on an edge-centric scatter-gather model. To overcome the poor locality of PageRank and optimize the memory performance, we develop a fast and efficient partitioning technique. We first partition all the vertices into non-overlapping vertex sets such that the data of each vertex set can fit in the cache; then we sort the outgoing edges of each vertex set based on the destination vertices to minimize random memory writes. The partitioning technique significantly reduces random accesses to main memory and improves the sustained memory bandwidth by $3\times$. It also enables efficient parallel execution on multicore platforms; we use distinct cores to execute the computations of distinct vertex sets in parallel to achieve speedup. We implement our design on a 16-core Intel Xeon processor and use various large-scale real-life and synthetic datasets for evaluation. Compared with the PageRank Pipeline Benchmark, our design achieves $12\times$ to $19\times$ speedup for all the datasets.**

I. Introduction

Graph analytics plays a critical role in many applications such as genome analysis, cybersecurity, and social networks [1]. However, it is challenging to achieve high-performance large-scale graph analytics. This is mainly because the data of emerging graph applications are massive [1], [2], [3] and most graph problems exhibit poor spatial and temporal locality of memory accesses [7]. As a result, the execution time is dominated by external memory accesses. To improve memory performance, prior work [4], [10] examined improving the graph layout or reordering the computation to increase locality. These optimizations are often beneficial, but also introduce significant pre-processing overhead. In addition, there are some graphs (e.g., social network graphs) that are less amenable to layout or reordering transformations.

The PageRank algorithm [5] was developed to determine the popularity of webpages in order to assist web search algorithms. It is also an application of the widely used linear algebra Sparse Matrix Multiplying Dense Vector (SpMV) kernel [10]. In this paper, we present a parallel implementation of the PageRank algorithm on multi-core platforms. We optimize the data layout to improve cache and memory performance. Our main contributions are:

- We develop a graph partitioning technique that improves cache performance and enables efficient parallel execution on multi-core platforms (Section III-A).
- We propose an optimized data layout that consumes little pre-processing overhead, but significantly reduces random memory accesses and improves memory performance (Section III-B).
- Compared with the PageRank Pipeline Benchmark [6], our design consistently achieves $12\times$ to $19\times$ speedup for various large graph datasets (Section IV).

The rest of the paper is organized as follows. Section II covers background and related work. Section III introduces our optimization techniques. Section IV reports experimental results. Section V concludes the paper.

II. Background and Related Work

A. PageRank

The PageRank algorithm [5] 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 by. A higher PageRank value corresponds to more importance. When the input graph is static (i.e., does not change over time), the PageRank algorithm traverses the entire graph iteratively. In each iteration, each vertex v updates its PageRank value based on Equation (1), in which d is a damping factor (usually set to 0.85); $|V|$ is the total number of vertices in the graph; v_i represents the neighbour of v such that v has an incoming edge from v_i; L_i is the number of outgoing edges of v_i.

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

When the input graph is altered as in analyzing streaming data, the PageRank algorithm does not traverse the entire graph

Funding for the researchers at Georgia Institute of Technology and University of Southern California was provided by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

[11]. In this paper, we focus on accelerating the PageRank algorithm for static graphs.

B. Edge-Centric Scatter-Gather Model

Many graph problems can be processed based on the edge-centric scatter-gather model [7], [8]. In this model, the computation is iterative; each iteration consists of a scatter phase followed by a gather phase. In the scatter phase, each edge produces a message, which carries the data of the source vertex of the edge and is used to update the destination vertex of the edge. In the gather phase, all the messages produced in the previous scatter phase are traversed to update the corresponding destination vertices.

C. Related Work

The performance of PageRank is bounded by the external memory (i.e., DRAM) accesses [10]. Thus, many prior works focus on optimizing the memory performance [4], [9], [10]. In [4], 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 [4] is non-trivial. In [9], an FPGA design to accelerate the PageRank algorithm is developed. Similar as our partitioning approach, the design in [9] also partitions all the vertices into vertex sets such that the data of each vertex set can fit in the on-chip BRAMs of FPGA. However, the FPGA accelerator can only process one partition at a time, while our design can process distinct partitions in parallel. In [10], Beamer et. al. propose the propagation blocking approach, which first stores propagations (i.e., 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 [10] does not optimize the data layout; thus, random memory accesses still occur when writing messages into the memory.

III. Optimizations

A. Graph Partitioning

Since cache and memory performance have significant impact on the performance of PageRank implementation [9], [10], we develop a fast graph partitioning approach to improve cache performance and eliminate random memory accesses to the DRAM. We assume that the input graph is stored in COO format, which is widely used for graph representation [7], [12], [13]. We divide all the vertices into vertex sets of equal size. Assuming each vertex set has m vertices, the graph is partitioned into $\lceil \frac{|V|}{m} \rceil$ partitions; the i-th partition maintains the vertex set that includes the vertices whose indices are from $i \times m$ to $(i+1) \times m - 1$ $(0 \le i < \lceil \frac{|V|}{m} \rceil)$. Each partition also 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 partition; the message list stores all the messages whose **destination** vertices are in the vertex set of the partition. The edge list of each partition 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. Figure 1 shows an example data layout after the graph data are partitioned into three partitions. Note that the data of each vertex 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 vertex set; the memory requirement of each message list depends on the number of edges whose destination vertices are in the corresponding vertex set. Algorithm 1 illustrates the PageRank algorithm based on graph partitioning.

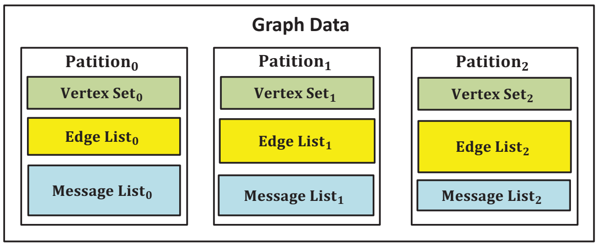


Figure 1: Graph partitioning

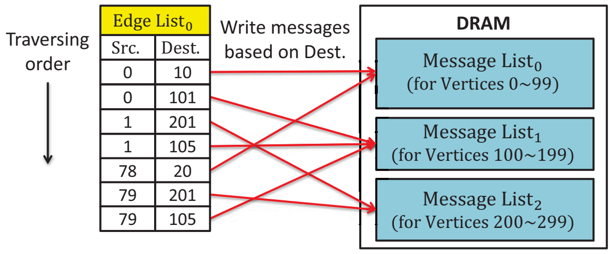
Algorithm 1 PageRank based on graph partitioning

```
Let $m$ denote the number of vertices in each vertex set
while not done do
  Scatter:
  for each partition do
    for each edge $e$ in edge list do
      Read the data of Vertex $e.src$
      Let $v$ = Vertex $e.src$
      Produce a message $msg$
      $msg.value = \frac{v.PageRank}{v.\#\_of\_outgoing\_edges}$
      $msg.dest = e.dest$
      Write $msg$ into message list of Partition $\lfloor \frac{e.dest}{m} \rfloor$
    end for
  end for
  Gather:
  for each partition do
    for each message $msg$ in message list do
      Update PageRank of Vertex $msg.dest$
    end for
  end for
end while
```

Partitioning the graph leads to two benefits. First, the scatter and gather phases of distinct partitions can be performed in parallel on multi-core platforms (Section III-C). Second, 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. Data Layout Optimization

We define sequential memory access sequence as a sequence of accesses to contiguous memory locations. Every memory access sequence can be partitioned into a set of sequential memory access sequences; we define the number of such sequential memory access sequences in a given memory access sequence as the number of random accesses in the sequence. In Algorithm 1, the memory accesses to read edges (Line 4) are sequential and the vertex data are read from cache (Line 5); however, writing messages into DRAM (Line 10) incurs random memory accesses. This is because the produced messages are written into DRAM based on their destination vertices, which can belong to any message list. In the worst case, writing messages into DRAM results in $O(|E|)$ random memory accesses in the scatter phase. Figure 2 shows an example in which writing each message into DRAM results in a random memory access.


Traversing order
Write messages based on Dest.
DRAM
Message List$_0$ (for Vertices 0~99)
Message List$_1$ (for Vertices 100~199)
Message List$_2$ (for Vertices 200~299)

Edge List$_0$	
Src.	Dest.
0	10
0	101
1	201
1	105
78	20
79	201
79	105

Figure 2: Random memory accesses due to writing messages into DRAM

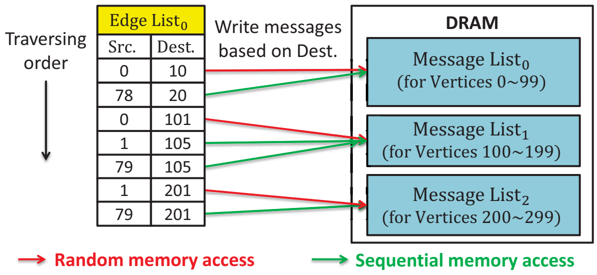
In order to minimize the number of random memory accesses due to writing messages, we propose an optimized data layout which sorts the edge list of each partition based on the destination vertices.

Theorem III.1. *In the scatter phase, based on our optimized data layout, the number of random memory accesses due to writing messages into DRAM is $O(k^2)$, where k is the number of partitions.*

Proof: The destination vertices of messages are the same as the destination vertices of the traversed edges. Since each edge list has been sorted based on the destination vertices, the messages based on each edge list are also produced in a sorted order. Thus, the messages whose destination vertices belong to the same partition are produced consecutively and written into the same message list in DRAM. Random memory accesses only occur when a message belonging to a different partition (i.e. other than the partition that the previous message belongs to) is produced. Therefore, writing the messages produced by traversing one edge list results in $O(k)$ random memory accesses. Since scatter phase traverses k edge lists, the total number of random memory accesses is $O(k^2)$, which is far less than $O(|E|)$ when k is a small number. In Figure 3, we show the optimized data layout for the example in Figure 2.

C. Parallel Implementation

Further, we parallelize the execution of Algorithm 1 on multi-core platforms. As shown in Algorithm 2, assuming the multi-core platform has p cores, we divide all the computations of k partitions into p chunks and use the dynamic scheduling in OpenMP [14] to execute each chunk on one of the p cores. All the cores execute the computations of distinct partitions in parallel.


Traversing order
Write messages based on Dest.
DRAM
Message List$_0$ (for Vertices 0~99)
Message List$_1$ (for Vertices 100~199)
Message List$_2$ (for Vertices 200~299)
Random memory access
Sequential memory access

Edge List$_0$	
Src.	Dest.
0	10
78	20
0	101
1	105
79	105
1	201
79	201

Figure 3: Optimized data layout for the example in Figure 2

Algorithm 2 Parallel PageRank on multi-core platform

```
Let $k$ denote the number of partitions
Let $p$ denote the number of cores
while not done do
    for $i$ from 0 to $k-1$ pardo [schedule(dynamic, $\frac{k}{p}$)]
        Execute scatter phase of Partition $i$
    end for
    barrier
    for $i$ from 0 to $k-1$ pardo [schedule(dynamic, $\frac{k}{p}$)]
        Execute gather phase of Partition $i$
    end for
    barrier
end while
```

During the scatter and gather phases of processing a partition, the data of the vertex set of the partition are repeatedly accessed. Thus, it is desirable to store the data of vertex set in the cache. We propose to choose the number of vertices in each vertex set (i.e., m) based on the size of private cache of each core (i.e., L2 cache), such that the data of each vertex set can fit in the private cache of each core. As a result, when a core is executing the scatter phase or gather phase of a partition, the core can access the vertex data from its private cache, rather than from the memory or the private caches of the other cores.

IV. Experimental Results

A. Experimental Setup

We conducted experiments on a linux server equipped with Intel Xeon E5-2650 CPU@2.6GHz running Ubuntu 14.04 operating system. The per core L1 and L2 cache sizes for this CPU are 64 KB and 256 KB respectively, and the shared L3 cache size is 20 MB. All code is written in C++ and compiled using G++ 4.7.1 with the highest optimization -O3 flag. The cache and memory statistics are collected using the *PCM* tool [18].

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.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

We compare the performance of our optimized algorithm against two baselines. The PageRank Pipeline Benchmark ($Base\_Sequential$) [6] serves as 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 the optimized algorithm as $Opt\_Multithreaded$. The multithreaded implementations are run using 16 threads since our target platform has 16 cores.

C. Graph Representation

The baseline algorithm uses CSR format of graph representation [13]. There are two arrays in the graph: vertex array of size equal to number of vertices and edge array of size equal to number of edges. Element in vertex array is an offset into edge array, representing location of first edge to the vertex. Edge array contains source indices of all edges, sorted by the destination. Additional arrays such as attribute and weight can be used to store attributes of graph vertices and the edge weights, respectively.

In the optimized algorithm, the graph is stored as list of vertices and edges (i.e., COO format [13]). Each vertex consists of vertex ID and application specific attribute (i.e., 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 a mixture of real world and synthetic graph datasets to evaluate the performance of Baseline and Optimized algorithms. The graph sizes vary from 16 million nodes to 94 million nodes and 536 million edges to 1963 million edges with average degree ranging from 15 to 37. The key properties of the graph datasets are summarized in Table I.

Table I: Graph datasets

Dataset	# Vertices	# Edges	Average degree
Kron24 [19]	16.7 M	536 M	16.1
Google+ [15]	28.9 M	463 M	15.2
Kron25 [19]	33.5 M	1073 M	16.0
Pld [16]	43.0 M	623 M	14.5
Twitter [17]	52.5 M	1963 M	37.4
Sd1 [16]	94.9 M	1937 M	20.4

Kron24 and Kron25 are synthetic graphs generated by the Graph Challenge benchmarks [19] with scales of 24 and 25 respectively. Google+ and Twitter are real world social networks. Pld and Sd1 are hyperlink graphs.

E. Results

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

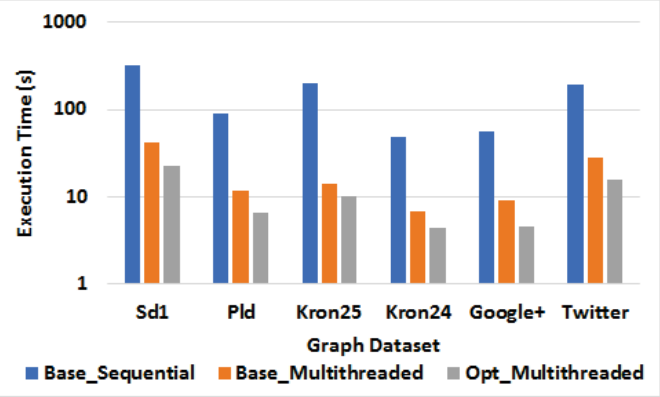


Figure 4: Execution time comparison

1) Execution Time: Figure 4 illustrates the performance comparison of the baseline and the optimized algorithms. The optimized algorithm consistently outperforms both the baseline algorithms. Our Optimized algorithm improves the execution time by 12× to 19× compared with the sequential Graph Challenge benchmark. With respect to the multithreaded Graph Challenge benchmark, we observe an improvement of 1.4× to 2× in the execution time. The improvement in execution time is due to the following reasons: (1) partitions are accessed in a regular manner with accesses to the vertices and edge in each partition sequential in nature and (2) our 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.

2) Cache Miss Ratio: In this section, we compare the cache access statistics of baseline and optimized algorithms. High number of cache misses translate to high number of memory accesses leading to high execution time. As depicted in Figure 5, the optimized algorithm reduces the total number of L3 misses by 9× to 19× in comparison with both the sequential and multithreaded PageRank Pipeline Benchmarks. The performance of multithreaded version of the graph challenge benchmark is similar in terms of number of cache misses although multithreaded implementation may cause more cache misses 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 prefetching in the memory controller, the random accesses can lead to 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 cache of the processor. These optimizations reduce the number of cache misses and

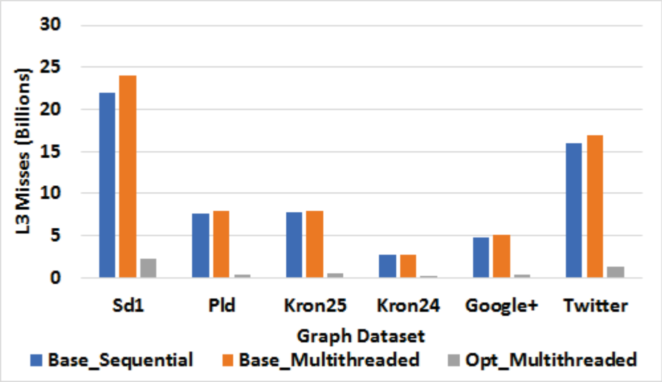


Figure 5: Number of L3 misses

cache pollution in comparison with the baselines.

Similar analysis applies to the total number of misses in the L2 cache. As illustrated in Figure 6, the optimized algorithm achieves significant reduction in the number of L2 misses compared with the baselines. The multithreaded PageRank Pipeline Benchmark implementation has similar number of cache misses as that of the sequential PageRank Pipeline Benchmark as each core has a private L2 cache.

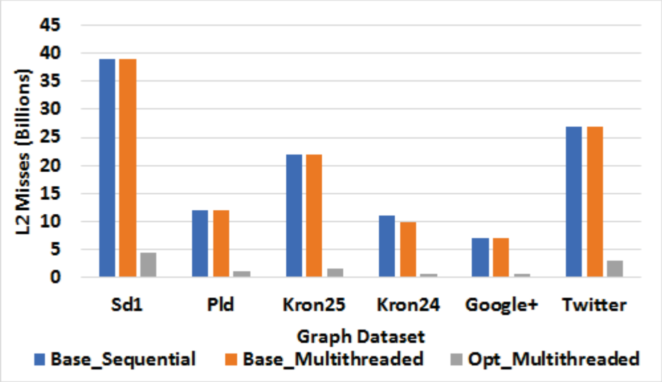


Figure 6: Number of L2 misses

3) Memory Accesses: The total number of memory accesses has a significant impact on the total execution time. Figure 7 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 $1.5\times$ in comparison with both the baseline algorithms. We believe that this is the most important factor in determining execution time for graph applications since computation time is much smaller in comparison with the memory access time. Further, the baseline algorithms have larger number of random accesses, resulting in much higher execution time in comparison with the optimized algorithm. Our 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.

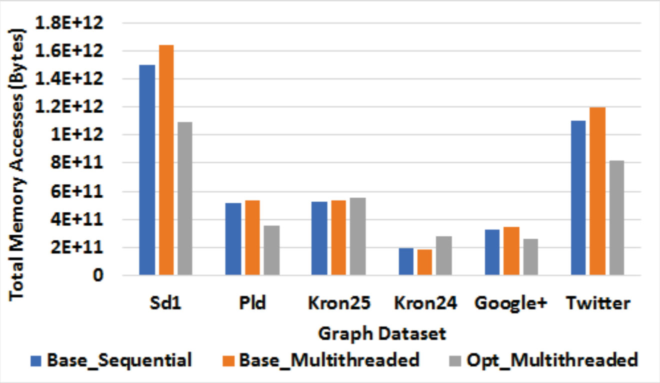


Figure 7: Total number of memory accesses

Figure 8 compares the total number of memory reads. It can be observed that the optimized algorithm reduces the number of memory reads by up to $2\times$. This is due to the fact that the there are a large number of random reads in the baseline algorithms, which result in a low cache line utilization. However, in the optimized algorithm, the cache line is completely used as we perform streaming accesses to the memory. The write statistics is illustrated in Figure 9. The total number of writes to the memory in the optimized algorithm is $5\times$ higher than that of baseline algorithm. This can be attributed to the nature of the algorithms. In the baseline algorithm, the total number of writes is $O(|V|)$ as each vertex is updated once in each iteration. In the optimized algorithm, the updates are written for each edge and gathered later into a vertex, i.e., there are $O(|E|)$ writes to the memory. However, as the total number of reads are much higher than the number of writes, by reducing the number of reads to the memory, the optimized algorithm reduces the total number of memory accesses.

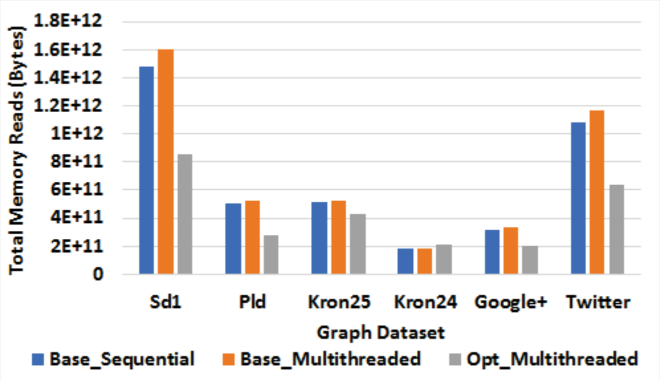


Figure 8: Number of reads from the memory

4) Effect of Partition Size: In our optimized algorithm, we divide the graph into multiple partitions. Each partition has a vertex set whose size is determined by the amount of on-chip memory or cache of processors. Smaller the size of the vertex set, higher the number of partitions in the graph. In this

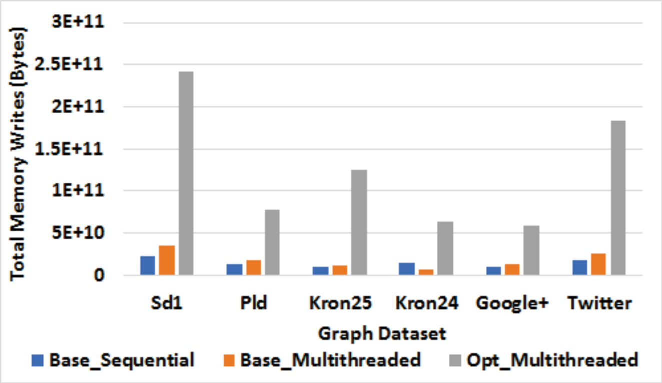


Figure 9: Number of writes to the memory

section, we evaluate the effect of varying the size of vertex set.

Figure 10 illustrates the effect of varying the size of the vertex set on total execution time of the optimized algorithm. We vary the number of vertices in the vertex set from 64K to 256K. We choose the size of the vertex set to be multiples of 2 so that the affiliation of a vertex to a vertex set can be obtained by a simple right rotate operation. We observe that the total execution time reduces with the size of vertex set until 128K and then increases with the size of vertex set. As we increase the size of vertex set, the number of partitions reduces and thereby the random accesses during the scatter phase reduce. However, if the size of vertex set is made very large, the amount of work done by a core during the gather phase is very high and it is challenging to balance the amount of work among various cores of the processor. Another problem might be the fact that the vertices in a vertex set simply do not fit in the L1/L2 cache and hence, gather phase will have more random accesses to the main memory. In this paper, we have empirically chosen vertex set to consist of 128K vertices based on our experimental results. We leave the optimal selection of size of vertex set for future work.

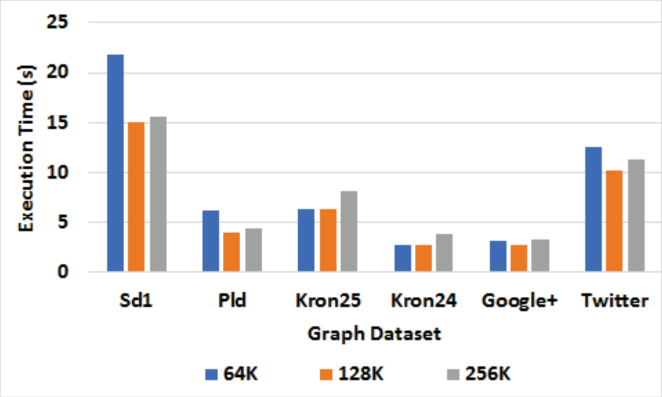


Figure 10: Impact of vertex set size on optimized algorithm performance

V. Conclusion

In this paper, we presented an efficient parallel PageRank implementation on multi-core platforms. We partitioned the input graph and used distinct cores to execute the computations for distinct partitions in parallel. Experimental results showed that our implementation achieved 12× to 19× speedup compared with the PageRank Pipeline Benchmark for 6 large graph datasets. The proposed optimization techniques in this paper can be easily extended to any SpMV-based applications.

In the future, we will develop more sophisticated graph partitioning technique to partition the graph such that the computations of distinct partitions are balanced. We also plan to extend our optimization techniques to emerging parallel architectures such as the proposed HIVE architectures [20].

References

[1] "Graph 500," http://graph500.org/

[2] 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.

[3] 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.

[4] 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.

[5] L. Page, S. Brin, M. Rajeev, and W. Terry, "The PageRank Citation Ranking: Bringing Order to the Web," Technical Report, 1998.

[6] "Pre-Challenge: PageRank Pipeline Benchmark," https://github.com/vijaygadepally/PageRankBenchmark

[7] A. Roy, I. Mihailovie, and W. Zwaenepoel, "X-Stream: Edge-centric Graph Processing using Streaming Partitions," in Proc. of SOSP, pp. 472-488, 2013.

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

[11] J. Riedy, "Updating PageRank for Streaming Graphs," in Proc. of Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016.

[12] 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.

[13] A. McLaughlin, "Accelerating Graph Betweenness Centrality with CUDA," https://devblogs.nvidia.com/parallelforall/accelerating-graph-betweenness-centrality-cuda/

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

[15] N. Z. Gong, W. Xu, L. Huang, P. Mittal, E. Stefanov, V. Sekar, and D. Song, "Evolution of Social-attribute Networks: Measurements, Modeling, and Implications using Google+," in Proc. of Internet Measurement Conference, pp. 131-144, 2012.

[16] O. Lehmberg, R. Meusel, and C. Bizer, "Graph Structure in the Web: Aggregated by Pay-level Domain," in Proc. of ACM Conference on Web Science, ser. WebSci, 2014, pp. 119-128, 2014.

[17] J. K. Konect, "The Koblenz Network Collection," in Proc. of International Conference on World Wide Web, 2013.

[18] "Intel Performance Counter Monitor," www.intel.com/software/pcm

[19] "Graph500 Kronecker generator," https://graph500.org/?page$_$id=12$#$sec-3$_$3

[20] T. Tran, "Hierarchical Identify Verify Exploit (HIVE) Program," http://www.darpa.mil/attachments/HIVE_Proposers_Day_PM_Briefing.pdf