Study Of Parallel Graph Algorithms On Shared Memory Systems

Andres Davila
`ad2291@nyu.edu`

April 23, 2018

Abstract

The focus of this paper is to benchmark and study parallel implementations of graph algorithms on multicore shared memory systems using OpenMP. Graph algorithms have properties that can make parallelizing the algorithms challenging and impact scaling opportunities. It is important identify more opportunities for optimizations in parallelizing graph algorithms because real world graph datasets continue to grow and may require further optimizations to handle larger data sizes. The two algorithms that will be of focus in this paper are Breadth First Search and PageRank. Since both algorithms have distinct properties in terms of the portion of the graph that is active during each step of the computation involved and the complexity of the computation, focusing on them will help show how these different properties can pose challenges to parallelization. In order to benchmark and study their parallel versions, several implementations were setup and executed with different sized graphs to observe timings and get hardware profiling information using perf, a Linux profiling tool. By employing different optimization techniques and / or changing implementation details to address locality, the study shows how parallelization can be found in graph algorithms regardless of the inherent challenges.

1 Introduction

Graphs are a fundamental data structure that have been used extensively in various domains. With the rise of big data analytics and the need to find relationships in data, graphs have become even more important due to its flexibility to describe these relationships. Social networks, computational sciences, recommendation systems and web searches are examples of problems that can be presented as graphs to be used in analysis. But as the datasets for these graphs rapidly continue to grow along with their complexity, the processing time required for these graph computations has also grown where current techniques and / or hardware may no longer be sufficient to process larger graph data sizes.

Hardware like multicore CPUs have helped gain significant performance enhancements of computation-intensive analytics by enabling developer defined parallelization but this is not a guaranteed improvement nor are all algorithms or analytics well suited for parallelization. Graph analytics, unfortunately, do have characteristics that make it difficult to achieve efficient parallelism on multicore systems. Traversing a graph typically has little computation but still involve long-latency memory accesses which are difficult to hide and limit parallel speedup due to random access patterns, which are a fundamental property of graph algorithms. And because of these random access patterns, graph algorithms usually have poor cache locality. Poor cache locality means that last-level cache misses will be high and could happen in any of the three key steps of graph computation: fetching an edge, fetching the metadata associated with the source of the vertex and fetching the metadata associated with the destination vertex of the edge.[12] Hardware can also contribute to issues with parallelization. On shared-memory systems, attention is needed in dealing with complex memory hierarchy and cache-coherence protocols that have an impact on scalability of the algorithms. Also, in order to ensure proper execution of an algorithm, protocols for thread synchronization and scheduling need to be considered when threads are trying to access the same region of memory.[11]

In order to study how these algorithms are impacted by these challenges, the present study considers different implementations using different techniques for each algorithm to observe the impact they have when working with reasonably sized datasets. Timings tools like clock and "omp get wtime" are used to get a sense of general performance but to really see some of the issues that can hinder performance in graph algorithms profiling tools like perf need to be used as well.

2 Proposed Idea

The aim of this paper is to study two graph algorithms and observe how the characteristics of graphs, the particular algorithm and shared-memory systems impact parallelized OpenMP versions of these algorithms. Depending on any performance issues / bottlenecks that were encountered during profiling and performance testing, literature review is used to helped understand why this these issues may be occurring and potentially use their optimizations where feasible / possible.

Breadth First Search (BFS) and PageRank graph algorithms will be the focus of this paper as they are distinctive both in terms of the portion of the graph that is active during each step of the computation involved in the algorithm and the complexity of the computation.

1. **Breadth First Search (BFS)**: used as a building block for other graph algorithms and because of this its the first algorithm researchers tend to use when trying to find new innovative ways of enhancing other graph algorithms.
2. **PageRank**: a link analysis algorithm that was created by the founders of Google to assess how popular a website.

In order to benchmark and profile the algorithms, different implementations are setup for each algorithm that follow similar handling...

1. Sequential
2. Naive Parallel Implementation
3. Two optimizations based on observations or from literature review

By setting up different implementations, the study will show how the changes in implementation can affect performance of the algorithm. To further achieve this, synthetic datasets are used that are relatively large in size and can fit within the limits of what is possible on CIMS. These synthetic graphs mimic the properties of real world social network graphs that have low density and very few vertices with high degree. Two different sizes of the datesets are used to see if any scaling is observed when increasing the data size or if if there are any issues related to increasing the number of threads for the actual computation. In order to mitigate any issues using CIMS shared resources, each algorithm is executed 5 times for each type of implementation or optimization and for the number of threads in order to get average times. Pre-processing, as mentioned in section 4, can overwhelm the actual timings for the computational work for the algorithms so within the code "clock" and "omp get wtime" are used to differentiate the timings between pre-processing and computation respectively.

To profile the algorithms during computation, the Linux tool "perf" was used for each implementation. perf is a profiling tool that uses hardware counters to measure stats like cache misses which will be extremely helpful in seeing the inherent locality issues that are observed with graph algorithms.

Finally, the GAP Benchmark Suite [5], is used in the analysis in order to draw comparisons to its efficiency and very low execution times.

3 Literature Review

3.1 Understanding Parallel Graph algorithms

In order to profile parallel graph algorithms on a shared memory / multicore system, it is important to understand a graph's characteristics and any challenges that these properties may present. The following graph characteristics that are commonly referred to are described below.

1. **Type of algorithm and type of graph**: Both the type of algorithm used and the structure of the graphs can have an impact on how well a graph can be processed. For example, social networks are typically considered as *small-world* and *scale-free*. These properties make it difficult to parallelize graph algorithms for social network analysis. A graph with the small-world property are usually hard to partition because they are typically low diameter. And a graph with the scale-free property will make it challenging to load balance because the work per vertex will often be proportional to the degree which can vary significantly. [6]
2. **Unstructured and Irregular problems**: The data in graph analytics tend to be highly irregular and unstructured making it difficult to partition the data for parallelism. Uneven partitioning can lead to imbalance in computations which can limit the scalability of the algorithm.
3. **Poor locality**: Locality is typically used to improve performance by trying to reuse data as much as possible which take advantage of caching and pre-fetching. But given that the data is typically highly irregular and unstructured, the computations and random data access patterns tend not to have much locality. Poor locality means that last-level cache misses will likely be higher leading to algorithms that are both memory bound and memory intensive.
4. **High data access to computation ratio**: Graph analytics are usually based on exploring the structure of a graph as opposed to performing intensive computations on the graph data which leads to a high data access to computation ratio. This leads to ineffective utilization of computation and bandwidth resources. Having little computation during traversal also makes it difficult to hide memory latencies.

3.2 Breadth First Search (BFS)

BFS is a popular algorithm for graph applications and is used as a building block for other graph algorithm implementations. BFS has a worst case performance of $O(|V| + |E|)$ [13] since it has to pass through all edges and vertices in the graph sequentially. The pseudocode in Figure 1 shows a naive implementation of a parallel BFS where parallelization is found primarily in processing the neighbors of a given vertex. It follows a conventional top-down traversal approach where current level vertices are each checked in parallel for their neighboring vertices while tagging them as their child vertices which then become the next level of the graph. The algorithm can be easily augmented for other information like the levels that the vertices are found in. This naive implementation actually hurts performance when compared to a sequential version of the algorithm since as mentioned previously there is little computation being done during traversal beyond updating an array that keeps track of the vertices parents. Because of this, BFS is limited by the throughput of sending vertex and edge data so the cost of having threads start up and dealing with cache coherence policies will slow down the algorithm. Also, processing the queue(s) that is typically used in BFS to keep track of what is in the current level and will be in the next level must be handled atomically which adds to the cost of using threads. Finally, traversing graphs will have random and irregular data access patterns when accessing vertex and edge data which will lead to low locality.

In order to address low locality and random access patterns, [4] used a bitmap data structure to mark vertices that have already been visited (i.e. already visited / current level). The process of checking if vertices have already been visited can be costly, so any way to minimize this process would be beneficial. As seen in the pseudocode in Figure 2, the parent array that was used for the check previously has been

```
Input: $G(V,E)$, source vertex $r$
Output: Array $P[1..n]$ with $P[v]$ holding the parent of $v$
Data: $CQ$ : queue of vertices to be explored in the current level
      $NQ$ : queue of vertices to be explored in the next level

for all $v \in V$ in parallel do
    $P[v] \longleftarrow \infty$;
$P[r] \longleftarrow 0$;
$CQ \longleftarrow$ Enqueue $r$;
while $CQ \neq \phi$ do
    $NQ \longleftarrow \phi$;
    for all $u \in CQ$ in parallel do
        $u \longleftarrow$ Dequeue $CQ$;
        for each $v$ adjacent to $u$ in parallel do
            if $P[v] = \infty$ then then
                $P[v] \longleftarrow u$;
                $NQ \longleftarrow$ Enqueue $v$;
    Swap($CQ, NQ$);
```

Figure 1: BFS Naive Parallel[4]

replaced with a bitmap data structure. One bit in the array represents a vertex and the bit is set to 1 if the vertex is in the current level. By using bits the entire data structure could potentially fit into memory. [4] could store visited information for 32 million vertices in just 4MB. This greatly reduced the working set size and avoided the performance degradation that occurs when there is overflow into each level of cache memory thereby improving locality and cache utilization. Other papers like [6] also use the same bitmap data structure in their algorithm but this does come at a cost. Atomic operations are now needed to avoid any issues from multiple threads writing to the different bits of the same 64-bit word but using atomic operations will lead to extra traffic being generated due to cache coherence protocols. To address this, [8] uses two bitmaps that store the vertices in the current level and in the next. By partitioning and leveraging OpenMP's dynamic scheduling, threads are able write to the bitmaps without contention avoiding the cost of using atomic operations.

The queue(s) used in keeping track of what vertices to check in the current level and also for the next, also have an impact on parallelization. While [4] was able to achieve noticeable gains by utilizing bitmaps, it did not address the need for locks when enqueing and dequeing the queues. In doing a BFS, it doesn't matter which vertex is visited first and which parent a vertex is matched with so the use of locked queues lead to benign races which affect the performance and not the correctness of the algorithm. [10] addresses this by using a implementation of a multiset data structure called a "bag". The bag is an unordered collection that allows redundant insertions of the same vertex if required and random selection of a vertex. The redundancy is not an issue since it doesn't matter which vertex is visited first in the current level as stated previously. Since BFS needs to check every vertex in the current level, being able to go through the data structure quickly without waiting on locks should be extremely helpful. [10]'s implementation also allowed the splitting and merging of the data structure enabling even further parallelization as the processing of the bag can then be handled recursively using divide and conquer methods.

The way that the algorithm processes a graph can also have a significant impact on performance as well. The majority of the computational work in BFS is traversing the edge to then check a vertex so any optimizations that would reduce the number of checks would help significantly. [6] introduced a new method of breadth first search on low-diameter graphs that would be later used by other researchers including [8]. The paper focuses on a hybrid implementation of BFS that improves performance by combining the conventional top-down traversal algorithm with bottom-up traversal to reduce the number of edges that need to be examined. Instead of having vertices in the current level search for an unvisited child, the bottom-up search will have vertices search for a parent. Pseudocode for the bottom-up method is in Figure 3.

The reason this works is because instead of having each vertex in the current level attempting to become the parent of all of its neighbors, each unvisited vertex instead searches for any parent among its neighbors.

```
for $all\ v \in V\ in\ parallel$ do
    $P[v] \longleftarrow \infty$;
for $i \longleftarrow 1..n\ in\ parallel$ do
    $Bitmap[i] \longleftarrow 0$;
$P[r] \longleftarrow 0$;
$CQ \longleftarrow Enqueue\ r$;
fork;
while $CQ \neq \phi$ do
    $NQ \longleftarrow \phi$;
    while $CQ \neq \phi\ in\ parallel$ do
        $u \longleftarrow LockedDequeue\ (CQ)$;
        for $each\ v\ adjacent\ to\ u$ do
            $a \longleftarrow Bitmap[v]$;
            if $a = 0$ then
                $prev \longleftarrow LockedReadSet\ (Bitmap[v],1)$;
                if $prev = 0$ then
                    $P[v] \longleftarrow u$;
                    $LockedEnqueue\ (NQ, v)$;
    Synchronize;
    Swap$(CQ, NQ)$;
join;
```

Figure 2: Parallel BFS with Bitmaps [4]

```
function bottom-up-step(vertices, frontier, next, parents)
  for v ∈ vertices do
    if parents[v] = -1 then
      for n ∈ neighbors[v] do
        if n ∈ frontier then
          parents[v] ← n
          next ← next ∪ {v}
          break
        end if
      end for
    end if
  end for
```

Figure 3: Bottom Up Traversal [6]

"Any" is the key word because once a vertex finds a parent no other edge needs to be traversed to check other vertices to identify a parent. The bottom-up traversal also doesn't require atomic operations since the child vertex only writes to itself. The work for any one vertex does need to be executed serially but [6] reports that this cost is minimal compared to the parallelism achieved in the algorithm. Also, a hybrid approach is necessary because the bottom-up traversal is only effective when the level is large as it will result in more work when it is small while top top-down traversal is most effective when the level is a small fraction of the total vertices.

A heuristic is used to switch between the two traversals where different data structures (queue for top-down and bit vector for bottom-up) are used to represent the current level since the size of the level will be different. This hybrid approach lead to speedups of 3.3-7.8 on synthetic graphs and 2.4-4.6 real world graphs compared to their baseline with noticeable scalability with each new core on a 40-core system. Other papers tried to enhance heuristic to further improve performance. [8] switches between the two traversals and their data structure representations (array to bit vector) when the size of the level is $\frac{n}{32}$ which is when the data structures are using about the same amount of space.

3.3 PageRank

PageRank is a graph algorithm that was originally developed by the founders of Google to assess how popular a website is to rank search results by computing a weighted value for each website based on inbound links to it. Since then, it has become a popular graph algorithm to rank the importance of vertices in a graph. The actual computation is straightforward…

$$\text{PageRank(u)} = \frac{1-d}{|V|} + d * \sum \frac{PageRank(v_i)}{L_i}$$

where the vertex's PageRank score $PR(u)$ relies on the PageRank of the other vertices $(PR(v_i))$ that have incoming edges to this vertex. These PageRank values are computed from the previous iteration which is then divided by the number of outgoing edges for each of the vertices to get the required aggregate needed to compute their "contribution" value for $PR(u)$. The rest of the formula is a damping factor to deal with vertices that have no outgoing edges. This process is repeated iteratively either by a set number of iterations or by computing the convergence.

```
Input: $G(V, E)$, number of iterations $I$, number of bins $B$
Output: PageRank scores for all vertices $PR[:]$
  $PR[:] \leftarrow 1/|V|$
  for each iteration $i \in 1 \ldots I$ do
      $bins[:] \leftarrow \{\}$
      for each vertex $u \in V$ do                    ▷ Binning Phase
          $contribution \leftarrow PR[u]/\textsc{OutDegree}(u)$
          for each outgoing neighbor $v$ of vertex $u$ do
              insert $(contribution, v)$ into $bins[v/B]$
      $sums[:] \leftarrow 0$
      for each index $b \in 1 \ldots B$ do             ▷ Accumulate Phase
          for each $(contribution, v) \in bins[b]$ do
              $sums[v] \leftarrow sums[v] + contribution$
      for each vertex $u \in V$ do
          $PR[u] \leftarrow (1 - d)/|V| + d \times sums[u]$
```

Figure 4: PageRank by Propagation Blocking [4]

```
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
```

Figure 5: PageRank via Vertex Sets [4]

The reliance on the contribution value for each vertex's PageRank means that for each vertex the computation must check the graph's adjacency lists to identify the vertices that have incoming links to it, access the contribution values of each the incoming vertices and the sum of the source vertex to recompute its PageRank value. Since it is possible for a vertex to be connected to any other vertex in the graph, accessing the required values will likely not be consecutive memory accesses leading to poor locality. And since computation itself is minimal, the runtime can be dominated by these memory accesses.[7]

Because of this, most of the research work done has been to improve locality. [7] uses a technique they call *propagation blocking* where propagation refers to the propagation of the vertex scores to other vertices. To address poor locality typically seen when using an array to keep track of the contributions, a bin data structure is used to store pairs of *(contribution, destination vertex*) which is referred to as the *binning phase*. This results in good locality since the data will be stored in consecutive memory addresses. And since the number of bins required should be small enough, the bins should all fit in cache simultaneously. After the process of gathering the contribution values, referred to as the *accumulate phase*, each bin is then processed to compute the sum that is used to compute the PageRank of the destination vertex. The pseudocode for this is in Figure 4.

Each phase can be handled in parallel but the implementations are different to address any need for atomicity or assigning the vertex ranges to the threads. The paper showed significant performance improvements especially for low diameter graphs, which are typical for social network graphs, but it does have implementation requirements that could be potentially problematic. To store the *(contribution, destination vertex*) pairs, additional memory may be required which can be substantial depending on the size of the graph since this pair is required for each directed edge in the graph.

Instead of partitioning the data like [7], [14] partitions the graph by vertices into equally sized vertex sets. Each partition of vertices has two additional data structures: an edge list which store the edges of the source vertices in the current partition and a message list which store vertex contribution values. The processing of the graph is similar to [7] in that there are two phases where the first phase stores the contributions after recomputing the contribution and then the second phase uses the values to compute the aggregated amount to be used for the PageRank computation for the destination vertex. Figure 5 has the pseudocode for this implementation.

As in [7], parallelization is done on the two phases (*scatter* and *gather* but it's done on distinct vertex set partitions that can be computed in parallel. And since the size of the vertex set can be adjusted, the partition sizes can be adjusted so that data structures can be fit into cache memory for efficient data reuse. A further optimization is also done to reduce random memory access by sorting the edge list of each partition by the destination vertices so that the writes to the message list will be sequential in memory. This implementations lead to significant speedup when compared to an available PageRank Benchmark[3] without the potential increase in memory requirements that [7] introduced.

4 Experimental Setup

4.1 Tools

4.1.1 System

The performance testing and profiling for the present project was all done on Courant's CIMS compute servers. 3 of the 4 compute servers have four AMD Opteron 6272 CPUs with 64 cores and 256GB of memory. Most of the work was done on compute servers 1 and 6 depending on the current load.

4.1.2 Benchmarks

The GAP Benchmark Suite (GAP) [5] was used to compare performance of BFS and PageRank implementations. It both supports and generates edge lists for input. While it uses OpenMP for parallelization, it does not provide options to adjust the number of threads for processing. Regardless, the difference in performance is dramatic in favor of GAP as it uses sophisticated techniques to take advantage of lower level / faster cache and optimizations to the algorithms.

The Benchmark Suite source code and compiled files are included with the submission. Details on how to use it is included in the README.txt.

4.1.3 Apps and Frameworks

1. Stanford Network Analysis Project (SNAP) - graph generator used to generate directed graphs based on the R-MAT algorithm [9]. The python libraries and script used to generate are included in the zip and a Dropbox link to the files used in the present paper is included in the README.txt. Can be compiled and used on CIMS.
2. GAP Graph generator used to generate undirected graphs based on the Kronecker algorithm. The scripts and how instructions on how to generate graphs are included in the README.txt. A Dropbox link to the files used in the present paper is also included.
3. OpenMP for parallelizing on shared memory systems using threads. Can be compiled and used on CIMS.
4. Intel's TBB - Used for Concurrent Queues for BFS. Source code included with submission and can be compiled on CIMS
5. Bitmap implementation from GAP [5] which is included in submission and can be compiled on CIMS.
6. perf - used for profiling the algorithms. The same parameters were used for all tests including the same number of threads for comparison. All outputs used are included in the submission.
7. clock - used to get timings within source code for pre-processing which has no parallelization
8. omp_get_wtime - used to get timings within source code for traversing the graph where there is parallelization

4.2 Data

In order to benchmark and profile parallel implementations of various graph algorithms, large graph datasets are required. A lot of the focus in papers tend to be around social network graphs which are typically characterized as *small-world* and *scale-free* which means that the graphs tend to have low diameter and few nodes with very high degree nodes. Graphs are growing to massive sizes but real world data sources typically don't have equivalent graph sizes that can be used for benchmarking. To address this, algorithms have been created to build synthetic graphs that mimic the properties of real world social network graphs. One in particular is the Kronecker algorithm, a generalization of R-MAT, which can generate edge lists for undirected graphs based on an edge factor and a power of 2 for number of vertices. The algorithm is used for the Graph500 benchmark [1] and was used in this present paper for testing BFS implementations. The graph generator available in [5] was used to generate two Kronecker graphs using 2^{20} (Kron-20) and 2^{21} (Kron-21) vertices.

Table 1 has edge stats for Kron-21.[1]

	Edges
Count	63.5m
Average per Vertex	41
Max	102,440
$> 5{,}000$	1,562
$> 10{,}000$	232
$> 50{,}000$	22

Table 1: Stats for Kron-21

R-MAT was used to generate directed graphs to test PageRank as directed graphs are typically used in PageRank computations. Two graphs were generate using the same number of vertices as was done for the undirected graphs and are referred to as as RMAT-20 and RMAT-21. The graph generation does requires some values to initialize a matrix that is used for a computing graph. The parameters used were based off of Graph500 specifications - A - .57, B - .19 and C - .19.

The size of the graphs were chosen based on limitations on CIMS for data storage and because the compute servers are shared resources. Larger graphs can lead to consuming resources for a substantial amount of time limiting the use of resources for other users but, at the same time, its own performance can be impacted by other processes running on the system.

4.3 Pre-processing

In the Graph500 specification, graphs of 2^{26} vertices are actually considered as "toy" size and requires 17GB of space but as mentioned in the *Data* section the largest size graph used in this study is Kron-21 / RMAT-21 which has 4m vertices and a file size that is close to 1GB. Pre-processing the edge list into a data structure that represents the graph's adjacencies is more of an implementation detail but it does play a major role in processing time and memory usage limiting the graph sizes used in the present study to 2^{21} vertices. As [12] pointed out, a lot of the published work seem to ignore the cost involved in converting the edge list into an appropriate data structure. For example, Table 2 shows the count in page faults that was observed in early implementation testing of a Kron-20 graph when going from a linked list representation of adjacency lists to arrays.

[1]Average adjusted to exclude vertices with edges greater than 2,500.

(in millions)	Page faults
Linked-Lists	4.2
Arrays	1.6

Table 2: Page faults When Switching from Linked-Lists to Arrays

Changing the data structure to be a 2d array where each row represents a vertex's adjacency list brought down page-faults significantly while reducing execution times of both BFS and PageRank. In order to set up the 2d array, there is a hard limit for the row arrays of 103,000 to accommodate the Kron-21 graph that has a small fraction of vertices that very large degrees. This is a key property of a sparse graph so the data structure used must accommodate this.

A better way of handling the data structure to reduce cache misses and page faults would be to represent the graph as a Compressed Sparse Row matrix (CSR) which avoids the sparsity inherent to social network graphs since these graphs are low density with very few vertices that have high edge count. Unfortunately due to timing, this was not implemented. As of now, it takes about 30 seconds to parse and generate the data structure for Kronecker 21 which has over 63 million edges but takes less than 2 seconds to BFS the same graph using the best implementation. In order to get proper timings for the algorithm, clock and omp_get_wtime were used within the scripts for pre-preprocessing and graph processing respectively.

5 Experiments & Analysis

As mentioned in the "Proposed Idea" section, the goal of Experiments and Analysis is to see how performance timings and profiling information can be impacted by changing any of the following...

1. Implementation details like changing data structures or changing how data structures are updated to address data locality
2. Using different sized datasets
3. Increasing the number of threads to see scalability
4. Utilizing features available in the OpenMP framework

And as the charts and tables will show, these changes can have a significant impact on the algorithms.

5.1 BFS

As baselines, both the sequential and naive parallel implementations of BFS were used for comparisons. Then two of the optimizations from the literature review were used for implementation ideas.

1. Bitmaps as implemented in [4] and uses the available code from [5]. As discussed previously, the data structure was used to improve locality and reduce random accesses which can be high due to the process of checking if neighbors have been visited
2. Intel's TBB Concurrent Queues instead of the "bag" data structure implemented in [10]. As discussed previously, having threads wait due to locks can have a significant impact on processing time so using a data structure that doesn't require locks should be beneficial.

All parallel implementations use OpenMP's "pragma omp parallel for" when iterating over the vertices that will be checked for its neighbors. While "pragma omp critical" were used for non concurrent queue parallel implementations. All implementations processed Kron-20 and Kron-21 undirected edge lists on CIMS compute servers but, as mentioned previously, the servers are fairly active and can have an impact

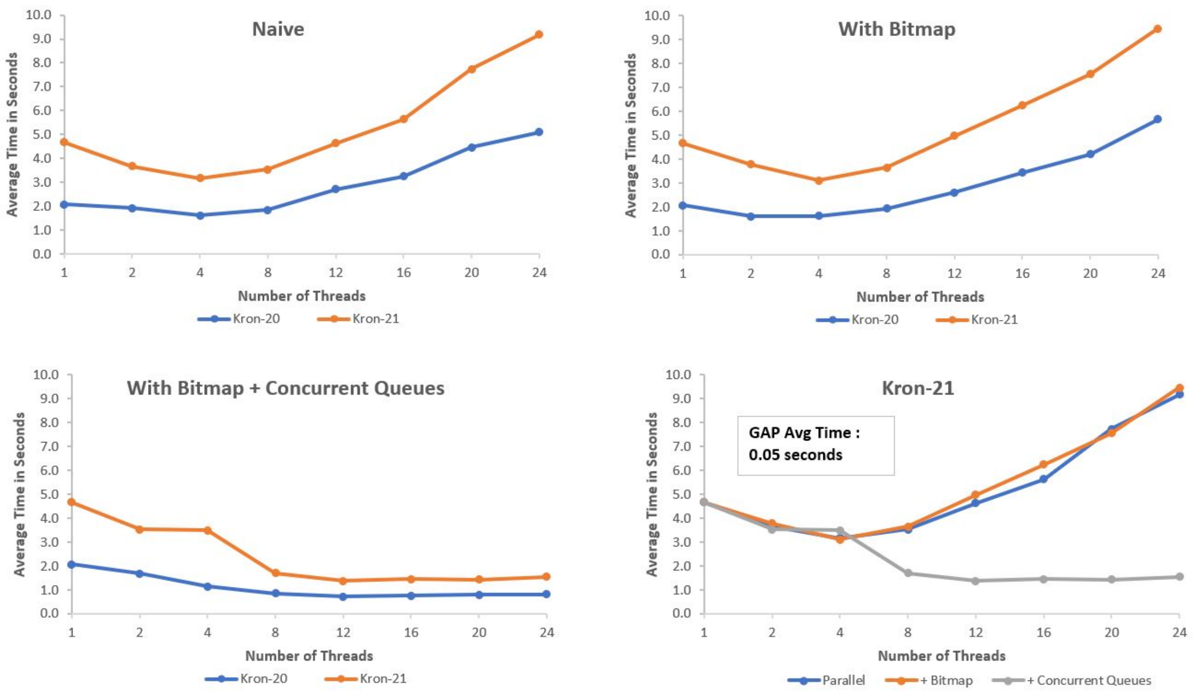
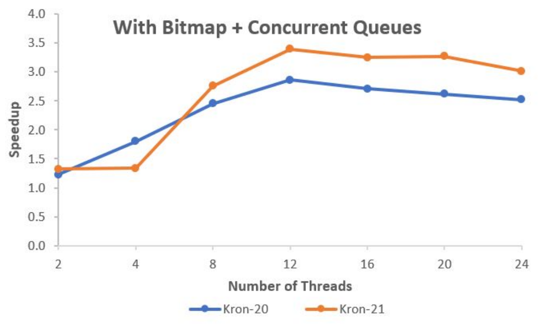


Figure 6: BFS Timings - Average Time in Seconds

on timings. To help mitigate this, average time in seconds of 5 runs for each implementation were done for the following thread counts: 1, 2, 4, 8, 12, 16, 20 and 24 threads where 1 is actually the sequential version of BFS.

The first thing to notice is that while there are some minor improvements to timings from the sequential algorithm in Figure 6, the use of a bitmap does not contribute to any more parallelization than what is achieved from the naive implementation. And at the same time, suffers the same scalability issues that the naive implementation has where it loses its advantage over sequential after 4 threads. The use of a bitmap to check if a vertex has been visited instead of the parent array in the naive implementation should have helped since the idea is that the bitmap should be compact enough to fit into cache. The sizes of the graphs could be a reason why the performance isn't any better which is contradictory to what other published papers have shown. For example, [4] referred to their bitmap implementation being able to store visited information for 32 million vertices, which is a dramatic difference in vertex count. One reason could be that the bitmap's need for atomic operations added some overhead.

While the bitmap optimization seemed to help little in the present study, the use of concurrent queues has had a dramatic impact on timings. Intel's TBB [2] Concurrent Queue (CQ) implementation allows for multiple threads to concurrently access and update the queue removing the need for locks on the queue. While it does not have the same divide and conquer capabilities of the "bag" data structure from [10], not relying on locks provides a clear advantage over all the other implementations. Queues are generally efficient due to $O(1)$ operations to enqueue and dequeue elements from the data structure. But without changing or replacing the queue, sparse graphs will have minimal parallelization [10]. This is because the queue operations while checking the vertices will need to be serialized but BFS' correctness does not rely on sequential processing of the vertices so the locks are unnecessary and only impede performance gains. The Speedup chart in 7 shows that by not requiring locks on the queue, the change does help considerably with scaling and actually has a greater speed up on the larger Kron-21 dataset than the Kron-20 dataset as the number of threads increase. And when comparing to the other implementations, we see in the right Speedup chart 7 that the naive and bitmap implementations actually degrade in performance, which is likely due to being overwhelmed by resource contention and cache coherence protocols.



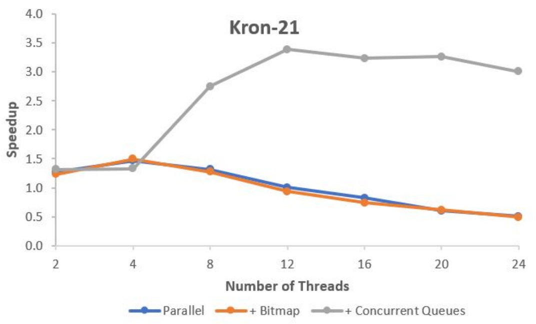


Figure 7: BFS Speedup Respective to Sequential

Using Perf, we can get profile data on the algorithm to see what might be occurring in the system while the graph is being pre-processed and traversed. Kron-21 was used on all implementations with 8 threads (where applicable) to keep things consistent. The GAP Benchmark Suite [5] was used with Kron-21 to get an idea as to what makes its performance significantly better. **NOTE: GAP actually executes the algorithm 15 times so it isn't directly comparable but it can still be helpful to compare to.**

Stat (in billions[2])	Naive	Bitmap	CQ	GAP
page faults	2.22m	2.23m	2.21m	0.22m
cycles	160.88	162.64	114.37	309.35
stalled-cycles-backend	87.52	88.65	45.99	63.51
instructions	110.15	112.67	103.67	164.29
L1-dcache-loads	53.23	54.22	50.27	76.16
L1-dcache-load-misses	0.51	0.77	0.49	0.46
LLC-loads	1.24	1.50	1.17	0.70
LLC-load-misses	0.28	0.25	0.22	0.28

Table 3: perf Stats After Processing Kron-21

GAP is significantly faster than any of the implementations done in this study taking half the time to pre-process the graph and 0.054 seconds to traverse the graph versus the 1.37 seconds it takes for the best implementation using 12 threads. As mentioned previously, page faults were 2.6 times more when using the linked list representation for adjacency lists vs. arrays. Given that, the higher page faults when comparing Concurrent Queues (CQ) to GAP could be one of the reasons for the significant difference. GAP uses a Compressed Sparse Row matrix format for its adjacency representation which avoids issues with sparsity in the data structure.

The performance difference in traversing the graph is likely explained by how efficient GAP is in utilizing memory. Given the multiple runs GAP performs on BFS, the stats show how the right optimizations can help with locality to achieve much better performance. When considering GAP runs BFS multiple times, the parallel implementations in this study show substantial locality issues. We can see from the implementations in this study that last level cache loads and misses are much higher when compared to GAP even though it's only one full traversal of the graph versus the 15 done for GAP. Given the higher misses at the last level cache, these implementations are accessing memory more frequently leading to performance degradation. This also likely explains why stalled-cycles-backend is quite high where stalls are likely due to threads waiting for memory access. With that said, we can see that CQ actually helps mitigate some of the issues with memory access and backend stalls introduced in the Bitmap implementation as there are noticeable differences between these implementations.

As mentioned, the Bitmap implementation barely added parallelism and in some cases made it worse when

```
# pragma omp parallel for
for (int u = 0; u < vertex_count; u++) {
    int* incoming = G.getAdjList(u);
    for (int j = 0; j < in_edge_counts[u]; j++) {
        int v = incoming[j];
        pr_temp[u] += pr_values[v] / out_edge_counts[v];
    }
    pr_temp[u] = pr_temp[u] * damping_factor + adj;
}
```

Figure 8: PageRank Naive Computation

```
Input: $G(V, E)$, number of iterations $I$
Output: PageRank scores for all vertices $PR[:]$
  $PR[:] \leftarrow 1/|V|$
  for each iteration $i \in 1 \dots I$ do
    for each vertex $u \in V$ do
      $contributions[u] \leftarrow PR[u]/\text{OUTDEGREE}(u)$
    for each vertex $u \in V$ do
      $sum \leftarrow 0$
      for each incoming neighbor $v$ of vertex $u$ do
        $sum \leftarrow sum + contributions[v]$
      $PR[u] \leftarrow (1 - d)/|V| + d \times sum$
```

Figure 9: PageRank using Pull Technique [7]

compared to the naive implementation. There is likely some overhead incurred when having to use atomic operations when manipulating the bitmap but the perf profile data also shows worse use of cache memory. The Bitmap implementation has significantly more L1 cache misses leading to overflow. These misses and backend stalls are likely the reason that there is no noticeable gains over the naive implementation.

5.2 PageRank

As done with BFS, both the sequential and naive parallel implementations of PageRank were used for comparisons and two optimizations were then made.

1. PageRank algorithm was changed to a "pull" implementation as was detailed in [7] and shown in Figure 9
2. OpenMP scheduling was used

All parallel implementations use OpenMP's "pragma omp for" when iterating over the vertices that will be checked for its neighbors though scheduling is used in the best performing implementation. All implementations processed RMAT-20 and RMAT-21 directed edge lists on CIMS compute servers. Averages of 5 runs for each implementation were done for the following thread counts: 1, 2, 4, 8, 12, 16, 20 and 24 threads where 1 is actually the sequential version of PageRank. The charts in Figures 10 and 11 show the average time in seconds and also the speedup respective to sequential. Due to the fact that PageRank is a bit more computation intensive, the timings are significantly slower in BFS where the only processing required was updating an array to identify parent vertices. The average time it took for the sequential version to process RMAT-21 was over a minute. Because of this difference, it has been excluded from timings in Figure 10 but still used for Speedup in 11. The speedup is dramatic for the best implementation due to the sequential version's poor performance in computing PageRanks for a set of vertices.

The naive implementation as shown in Figure 8 was essentially a direct implementation of the formula for PageRank. We can see from the performance timings between naive and pull implementations that there is a dramatic difference leading to timings that lead to timings that are almost twice as long to compute the PageRank values. perf is used to get some quick stats on the two implementations using RMAT-21 and 12 threads. Table 4 show a subset of information provided by perf. **NOTE: GAP actually executes the algorithm 15 times so it isn't directly comparable but it can still be helpful to compare to.**

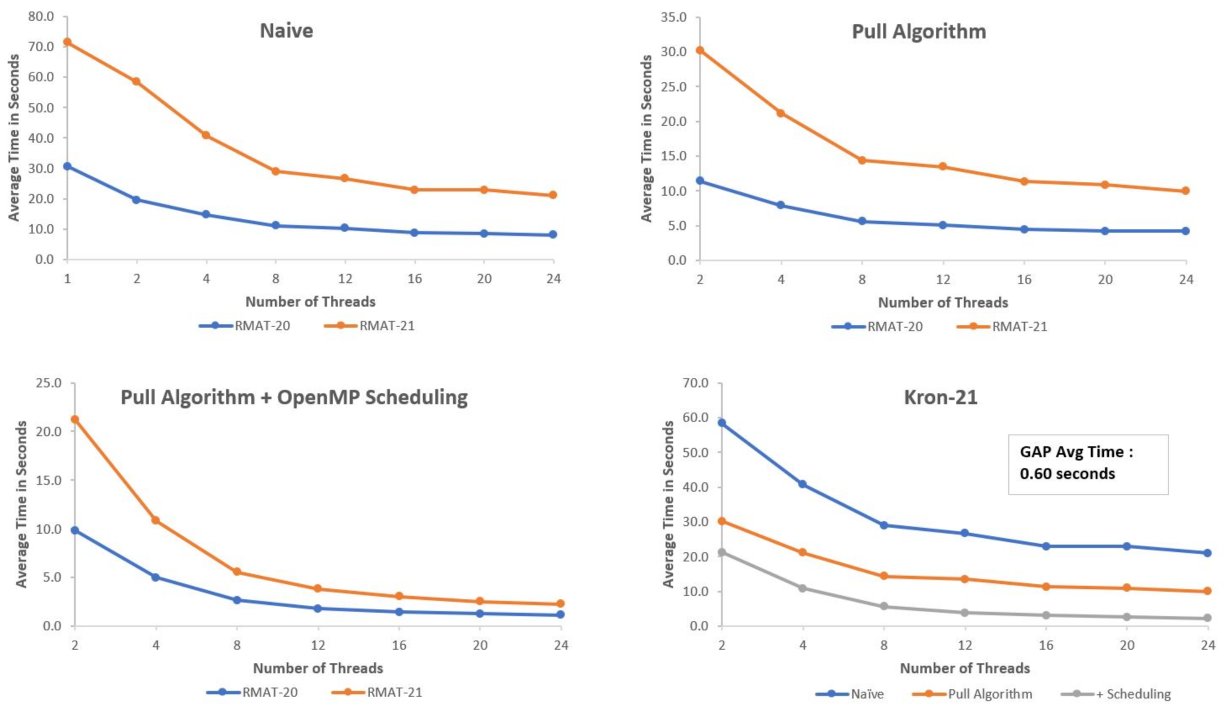


Figure 10: PageRank Timings - Average Time in Seconds

Stat (in billions[3])	Naive	Pull	Scheduling	GAP
page faults	2.28m	2.26m	2.28m	0.09m
cycles	344.91	254.24	255.92	1,608.88
stalled-cycles-backend	214.77	163.85	153.944	550.87
instructions	186.20	149.32	148.41	392.76
L1-dcache-loads	105.89	78.50	78.6	182.88
L1-dcache-load-misses	2.09	2.09	2.08	17.54
LLC-loads	5.02	3.17	3.10	19.17
LLC-load-misses	1.74	0.93	0.96	4.96

Table 4: perf Stats After Processing RMAT-21

In the naive implementation, it is obvious that the algorithm is using the cache in an unfriendly manner leading to substantial loads and misses at the last level cache. Last level misses are almost twice as much as it is after switching to the pull method leading to more memory accesses and stalled backend cycles as there are stalls for memory access. As mentioned in [7], accessing the contribution information from an incoming vertex (*pr_values[v] / out edge_counts[v]* in the inner loop vs. a separate loop in pull) and the sum of the destination vertex (*pr[u]*) may experience much lower locality since the data accesses are likely to not be consecutive due to the potential of a vertex to be connected to any other vertex. By not separating these computations, we can see that cache misses in the naive confirms this point.

Like with BFS, GAP computes PageRank both efficiently and fast where it takes on average 0.65 seconds to compute the values on RMAT-21 vs. an average of 10.0 seconds with 24 threads for the pull algorithm implementation. Part of this can be attributed to load balancing. Both the naive and pull implementations achieve parallelism by processing across vertices. Given that sparse graphs will have vertices with varying degrees where some may have greater degree counts than other vertices the variance in work per thread can be substantial. And since a vertex can be connected to any other vertex, the random data access patterns required to get the necessary information from the data structures for the computation can also have an

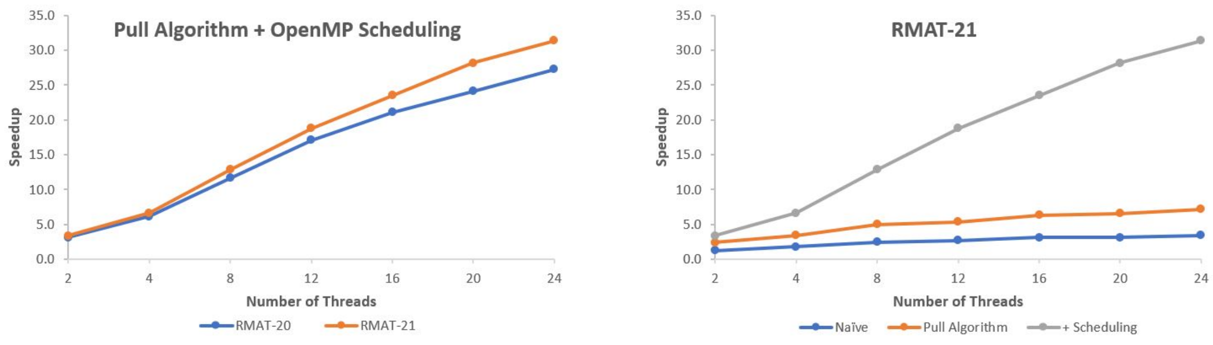


Figure 11: Pagerank Speedup Respective to Sequential

impact on locality as we saw in the naive implementation.

To address this, dynamic scheduling was used at that same line where the ”pragma omp parallel for” line is in Figure 8. While perf isn't an appropriate tool to measure the differences between the two implementations, which can be observed from the similarity in stats between pull and scheduling, the timings in 10 do show how much of an impact using dynamic scheduling can have on an algorithm. While GAP still outperforms, processing of RMAT-21 went from on average 10.0 seconds using 24 threads to 2.3 seconds.

6 Conclusion

Even though BFS and PageRank are different in computational intensity, their parallel performance can hindered by the same challenges that graph algorithms usually face when parallelizing due to their random data access patterns. But parallelism opportunities can be found in both by considering how data access flow and data structures used in the implementations work.

Key observations from the study…

BFS

1. Standard implementations of algorithms can pose significant challenges when parallelizing algorithms. Queues is the default choice for BFS when keeping track of the current and next set of vertices but this causes serialization of threads when manipulating the queue. Using non locking implementations or other data types can be of substantial help

2. Data structures that can fit into low levels of cache memory can increase locality and hence improve performance. Bitmaps in place of an array to check if a vertex has been visited is one way to do this as the data structure should be compact enough to fit. Unfortunately, it did not help with parallelizing in this study which may be due to the much smaller sizes of graphs that were used in testing versus what is typically used in parallel graph algorithm studies.

PageRank

1. Due to more work being done than BFS for the actual computational work, load balancing was problematic where OpenMP's scheduling had a a dramatic impact on timings

2. Direct implementations of a formula can lead to very poor performing code if opportunities for data locality are not considered when handling data structures in the computations

Work continues to be done to further find more opportunities to parallelize graph algorithms since there are no signs of slowdown with how fast real world graph datasets continue to grow.

Though, the purpose of the study was not to directly compare the two algorithms we can see from the timings the difference in computations but PageRank sees the most benefit from sequential to "best" implementation. And unlike BFS, PageRank shows better scalability as the number of threads and data size grows.

References

[1] Graph500. *https://graph500.org/*.

[2] Intel TBB. *https://www.threadingbuildingblocks.org/tutorial-intel-tbb-concurrent-containers*.

[3] Pre-Challenge: PageRank Pipeline Benchmark. *https://github.com/vijaygadepally/PageRankBenchmark*.

[4] Agarwal, V., Petrini, F., Pasetto, D., and Bader, D. Scalable Graph Exploration on Multicore Processors. *SC10: Proceedings of the 2010 ACM/IEEE International Conference* (2010).

[5] Beamer, S., Asanović, K., and Patterson, D. The GAP Benchmark Suite. *https://github.com/sbeamer/gapbs*.

[6] Beamer, S., Asanović, K., and Patterson, D. Direction-Optimizing Breadth-First Search. *Scientific Programming 21* (2013).

[7] Beamer, S., Asanović, K., and Patterson, D. Reducing Pagerank Communication via Propagation Blocking. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2017), 820–831.

[8] Belova, M., and Ouyang, M. Breadth-First Search with A Multi-Core Computer. *IEEE International Parallel and Distributed Processing Symposium Workshops* (2017).

[9] Chakrabarti, Zhan, Y., and Faloutsos, C. R-MAT: A Recursive Model for Graph Mining.

[10] Leiserson, C., and Schardl, T. B. A Work-efficient Parallel Breadth-first Search Algorithm (or How to Cope with the Nondeterminism of Reducers). 303–314.

[11] Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. Challenges in Parallel Graph Processing. *Parallel Processing Letters 17* (2007), 5–20.

[12] Malicevic, J., Lepers, B., and Zwaenepoel, W. Everything you always wanted to know about multicore graph processing but were afraid to ask. *USENIX Annual Technical Conference* (2017).

[13] T. H. Cormen, C. E. Leiserson, R. L. R., and Stein, C. Introduction to Algorithms, 3rd ed. *The MIT Press* (2009).

[14] Zhou, S., Lakhotia, K., Singapura, S. G., Zeng, H., Kannan, R., Prasanna, V. K., Fox, J., Kim, E., Green, O., and Bader, D. A. Design and implementation of parallel pagerank on multicore platforms. *2017 IEEE High Performance Extreme Computing Conference (HPEC)* (2017), 1–6.