Performance of Parallel Graph Profiling Algorithms

1 Introduction

This article will be examining how parallelization and multithreading techniques affect the runtime of graph profiling and reduction algorithms. Using Intel's Thread Building Blocks (TBB) Library, we can use high level parallel algorithms and data structures to take full advantage of multicore performance, and optimize computationally heavy, long running code.

2 Algorithms

This article focuses on parallelizing several graph profiling algorithms. The algorithms considered are:

- abc: approximate betweenness centrality of a vertex - approximation of the number of shortest paths from all vertices to all others that pass through the node
- adiam: approximate diameter - approximation of the longest shortest path in the graph
- aprank: approximate page rank of the vertices of a graph
- bc: betweenness centrality of all vertices in a graph
- cc: connected components - the number of connected components and their sizes
- etri: exact triangle count - the number of triangles in a graph
- lcc: average local clustering coefficient - degree of which nodes in a graph tend to cluster together
- prank: page rank of the vertices of a graph

In addition, several graph reduction algorithms are also analyzed:

- reduce: create a reduced graph by keeping the top x neighbors of each vertex, ordered by their degree
- reducetri: similar to reduce, but avoiding triangles when possible
- reducetreetop: create a reduced graph from the spanning trees of the top x vertices, ordered by their degree
- reducehighdegreetop: similar to reducetreetop, but using a different BFS algorithm

- reducepercent: create a reduced graph by keeping the top x% of vertices past the median degree of all vertices

3 Parallelization Techniques

TBB provides several high level parallel functions and data structures that can be used. This section highlights how some of these tools were used.

Parallel Functions:

- parallel_for

  Most commonly used loop - performs parallel iteration over a range of values, which can be specified with a start and end value, or with a blocked_range object.

  Useful for:

  - repeatedly performing a calculation for an approximation (e.g. repeatedly selecting a vertex and computing a dependency score to calculate an approximate betweeness centrality)
  - performing an operation on every element of an array (e.g. calculating the page rank for each vertex, calculating heuristics for graph reduction algorithms)
  - optimizing doubly or triply nested for loops when used in conjunction with blocked_range2d or blocked_range3d (e.g. when counting triangles, or calculating betweenness centrality of every vertex)

- parallel_for_each

  Performs parallel iteration over sequences that do not have random access (for example, lists), at the cost of additional overhead. The sequence is specified with iterators pointing to the start and end of the range of the range to iterate over.

  It is useful as many functions in the boost graph library used by this project returns lists. For example, `boost::adjacent_vertices` returns a pair of iterators to a list of vertices adjacent to a specified vertex, and is used in calculating betweenness centrality, clustering coefficients, connected components, spanning trees, triangle counts, and graph reductions.

  It is important to note that parallel_for_each comes with significant overhead. Intel's documentation suggests that "For good performance with input streams that do not have random access, execution of B::operator() should take at least 100,000 clock cycles. If it is less, overhead of parallel_for_each may outweigh performance benefits."

- parallel_reduce

Computes a reduction over a range - for example finding the sum of an array. It works by first recursively splitting the range into subranges, performing the body operation (specified by the user) on each of these ranges, and then joining these ranges together (how these values are joined together is also specified by the user).

It is used over, say, a parallel_for loop, as a parallel_for loop may end up having multiple threads writing to the same variable (in this case, the running sum, or value being aggregated from the range) at the same time, or overwriting each other, leading to innaccurate results or segmentation faults. If a mutex is used in a parallel_for loop to protect the variable, then runtime is slowed down, and in worst case, the loop becomes serial.

In this project, it is used to calculate the number of triangles and clustering coefficient.

- parallel_do

  Function that processes work items in parallel, in which new items can be added as current items are processed by using the parallel_do_feeder classes. Similar to parallel_for_each, it comes with additional overhead.

  Used in parallel bfs algorithm - unvisited neighbors of visited vertices are added to a queue, as the parallel_do loop searches through the queue in parallel.

- parallel_sort

  Sorts a sequence or a container in parallel, in a non stable, non deterministic fashion. Used in graph reduction algorithms, where vertices are sorted by their degree.

- blocked_range, blocked_range2d, blocked_range3d

  blocked_range is not a function, but rather an object that can be used to represent a range of values to iterate over. When used in conjunction with parallel_for, or parallel_reduce, it splits itself into multiple, smaller blocked ranges, which can be iterated over serially in seperate threads.

  Using blocked ranges can help optimize code, as it allows the user to specify an optimal grain size for the smaller blocked ranges to improve runtime, and also lets the user write optimized nested for loops without having to nest parallel_for or parallel_reduce functions.

Concurrent Data Structures:

- concurrent_vector

  Template class for a vector that can be concurrently grown and accessed (i.e. multiple threads can append elements and access elements at the same time).

  Useful for calculating and creating the edges for a reduced graph in parallel, as many edges can be calculated in parallel, and added to the vector.

- concurrent_unordered_map, concurrent_unordered_multimap

  Template classes that support concurrent insertion and traversal, but not concurrent erasure.

  concurrent_unordered_map is used to store heuristics about vertices while an algorithm is runnning - for example, the "priority" of a vertex during graph reduction, and whether or not a vertex has been visited in BFS.

  concurrent_unordered_multimap is to create and store spanning minimum spanning trees to perform graph reductions on.

- concurrent_priority_queue

  Template class for a priority queue with concurrent operations.

  Used to add and remove vertices in order of their degree for BFS and graph reduction algorithms.

4 Data Collection

Runtimes of each algorithm was measured using the `<time.h>` library provided by C, and is calculated as the difference between before the function was called and after, rounded to the nearest second. In the future, the `<chrono>` library can be used instead, to get a time difference in milliseconds, for more accurate results. See `https://stackoverflow.com/questions/31487876/getting-a-time-difference-in-milliseconds` for details.

The parallelized and unparallelized versions of each algorithm were run 5 times each, on 6 different graphs. The program was run on the waterloo student linux environment, on which 56-64 threads were consistently available. The runtimes were recorded, and averages were used for analysis.

For reduce and reducetri, edges to the top 3 neighbors of each vertex were chosen to be kept.

For reducetreetop and reducehighdegreetop, the reduction was created from the spanning trees of the top 1% of all vertices.

For reducepercent, 50% of vertices past the median were chosen to be kept.

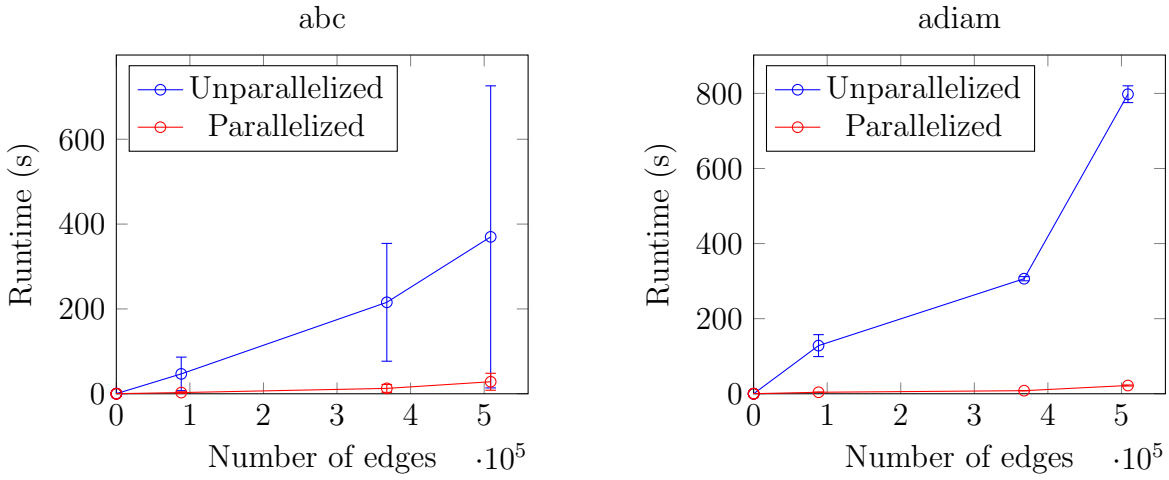
For the approximate betweeness centrality metric (abc), runtime was also measured based on the degree of the vertex chosen. 3 measurements were taken from a vertex of degree 1, 10, 100, 500 and 1000 from the largest 3 graphs.

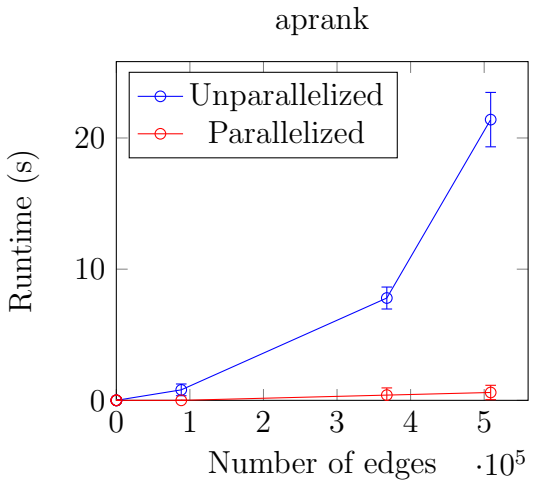
Graph Name	Num. Edges	Num. Vertices	Average Degree
5.txt	5	6	1.666667
50.txt	50	51	1.960784
small_test.txt	55	1904	0.057773
facebook_combined.txt	88234	4039	43.69101
enron.txt	367662	36692	20.04044
epinions.txt	508837	75888	13.41021

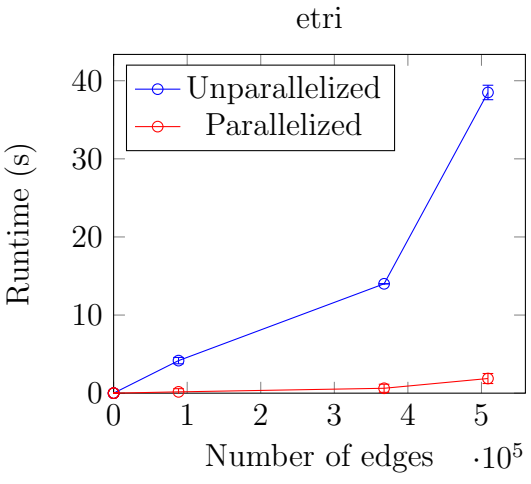
5 Results

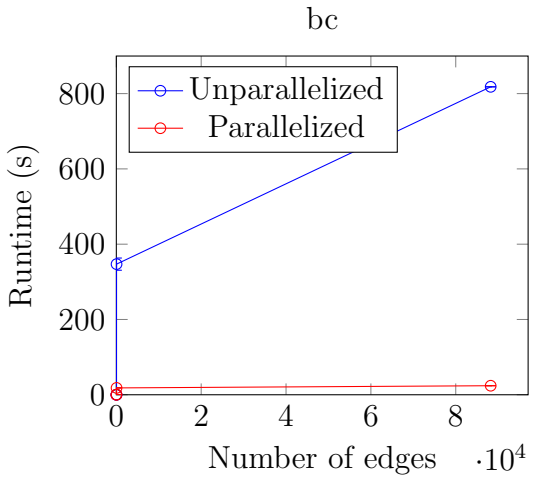
The average runtimes and uncertainties for each algorithm are shown below, plotted against the number of edges in each graph. For abc, the average runtimes were also plotted on the degree of the vertex chosen.

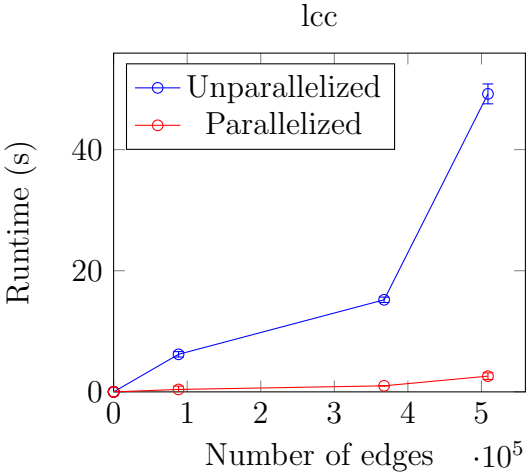
5.1 Runtime v.s. Graph Size

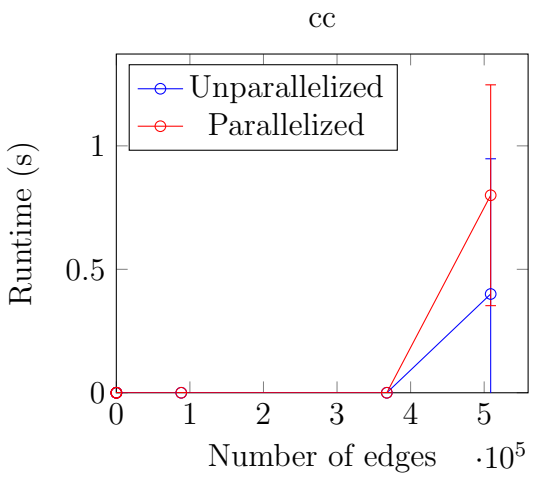


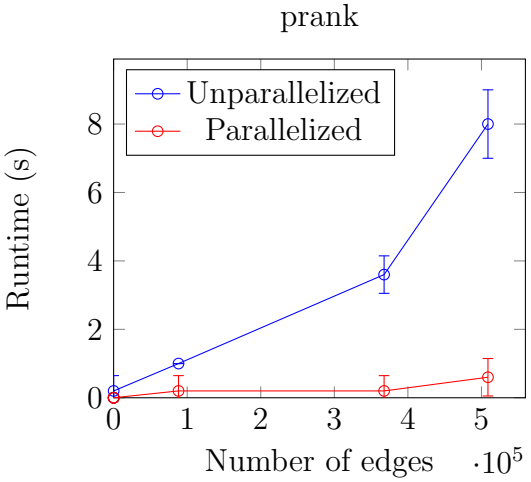

aprank
Unparallelized
Parallelized
Runtime (s)
0
10
20
0 1 2 3 4 5
Number of edges
$\cdot 10^5$

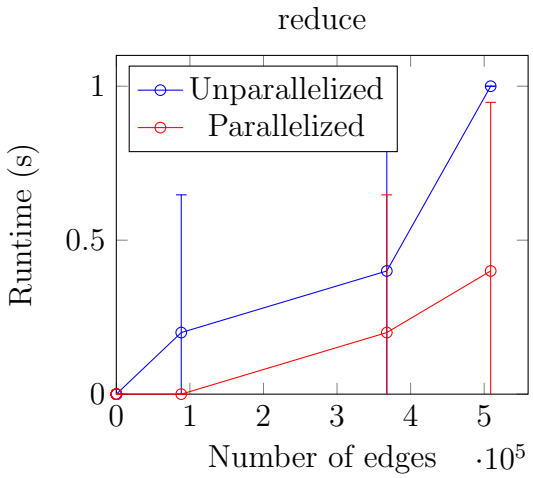

etri
Unparallelized
Parallelized
Runtime (s)
0
10
20
30
40
0 1 2 3 4 5
Number of edges
$\cdot 10^5$

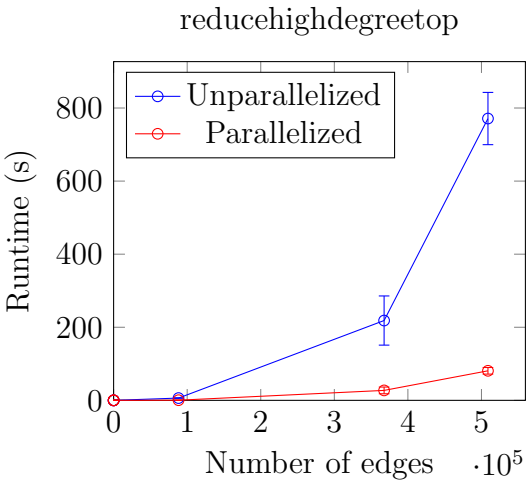

bc
Unparallelized
Parallelized
Runtime (s)
0
200
400
600
800
0 2 4 6 8
Number of edges
$\cdot 10^4$

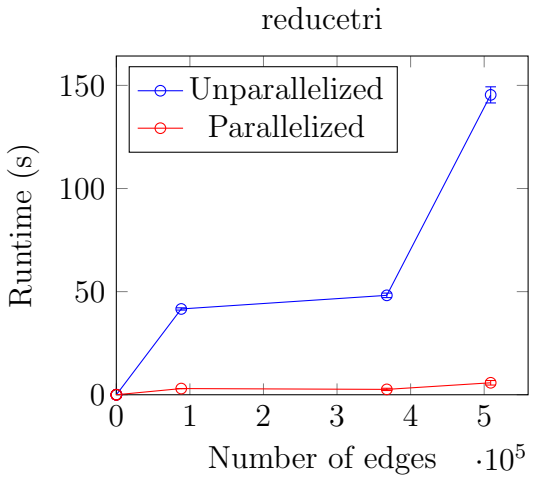

lcc
Unparallelized
Parallelized
Runtime (s)
0
20
40
0 1 2 3 4 5
Number of edges
$\cdot 10^5$

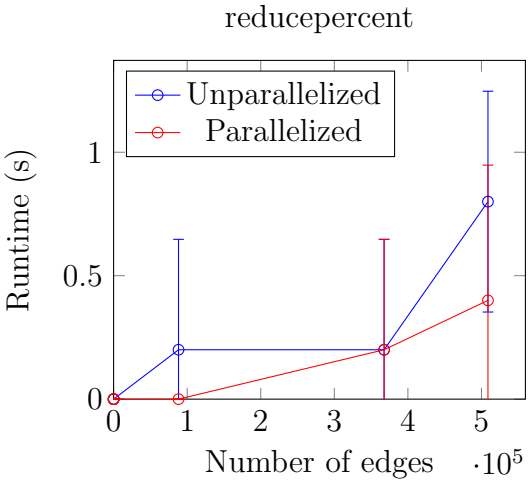

cc
Unparallelized
Parallelized
Runtime (s)
0
0.5
1
0 1 2 3 4 5
Number of edges
$\cdot 10^5$

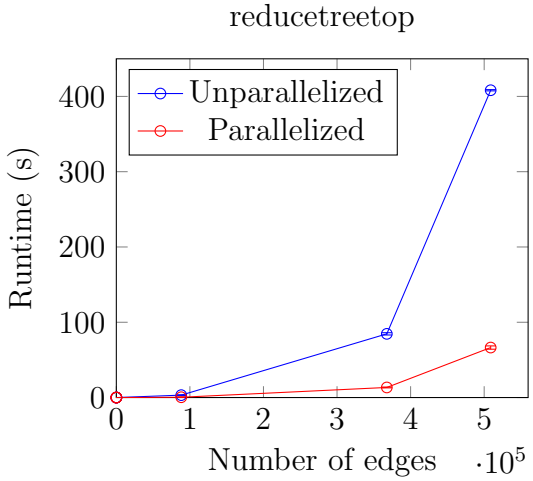

prank
Unparallelized
Parallelized
Runtime (s)
0
2
4
6
8
0 1 2 3 4 5
Number of edges
$\cdot 10^5$











Observations:

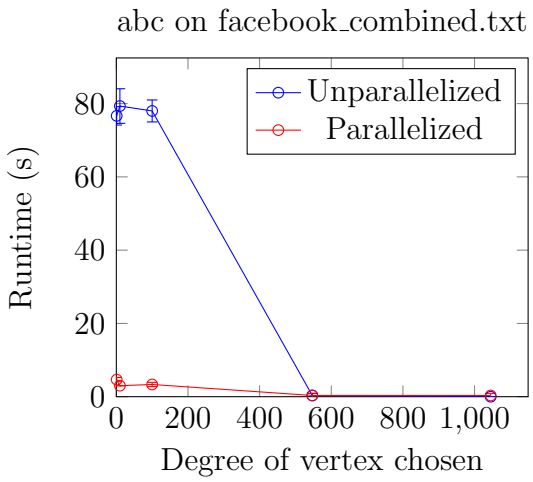
abc runtimes varied wildly based on the vertex chosen. This is explained in section 5.2.

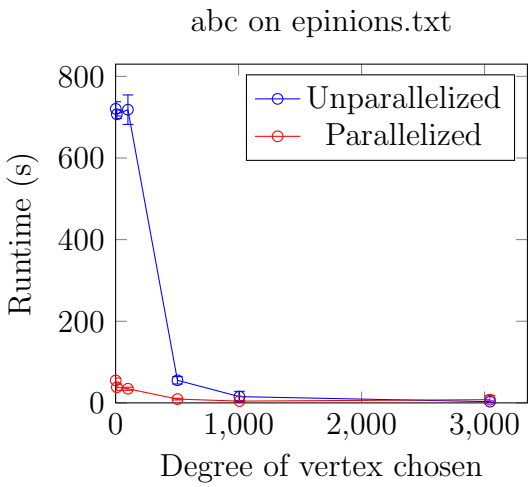
bc is highly RAM intensive, and either crashes, or is killed by the kernel on larger graphs.

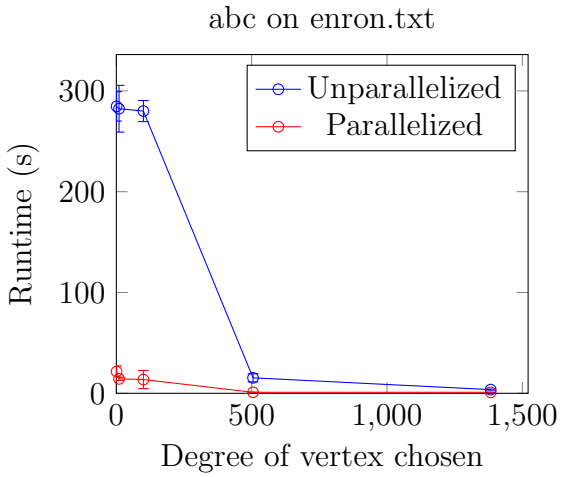
Most graphs follow a polynomial curve, which matches with the run time analysis of the algorithms.

While cc, reduce, and reducepercent all look like they have inaccurate measurements, it is important to note that both the parallelized and unparallelized versions of these algorithms ran under 1 second on all graphs, meaning no noticable performance improvement could be observed.

5.2 abc - Runtime v.s. Vertex Degree







Observations:

The runtimes of abc on all three graphs follow a very similar pattern - vertices with a low degree take much longer to run, whereas vertices with high degree run almost instantaneously. This can be explained by the nature of the algorithm.

The algorithm works by repeatedly choosing a random vertex vi, calculating the shortest path to all other vertices from vi, and computing a dependency score based on how many times our original vertex chosen appears in a path. This is repeated until the sum of the dependency scores pass a threshold, or until 1000 iterations are reached.

Vertices with many neighbors will appear in more paths and will reach the threshold much faster, so the algorithm will have a lower runtime on these vertices. On the other hand, vertices with little or no neighbors have a lower chance of appearing in a path, and go through more iterations - worst case running through all 1000 iterations, taking longer to run. This explains why vertices with higher degrees have lower runtimes than vertices with lower degrees.

6 Analysis

The improvement of the parallelized algorithms over the unparallelized versions is measured as number of times the parallelized version is faster than the unparallelized version - i.e.

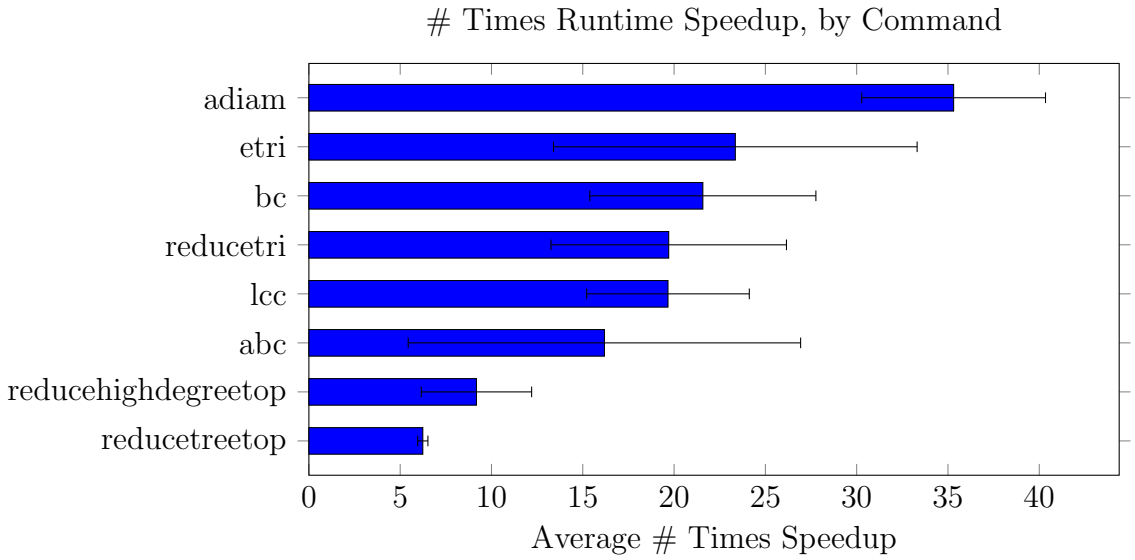
$$\text{improvement} = \frac{\text{runtime of unparallelized algorithm}}{\text{runtime of parallelized algorithm}}$$

For example, if an algorithm had a runtime of 200s unparallelized, and 10s parallelized, the improvement in time would be 20, i.e. a 20x improvement, or the parallelized version runs 20 times faster.

In our dataset, measurements where either versions of the algorithm ran in 1 second or under have been ommitted. This is because there is no way to accurately measure any runtimes under 1 second - even if measurements could be made in the milliseconds, they cannot be trusted.

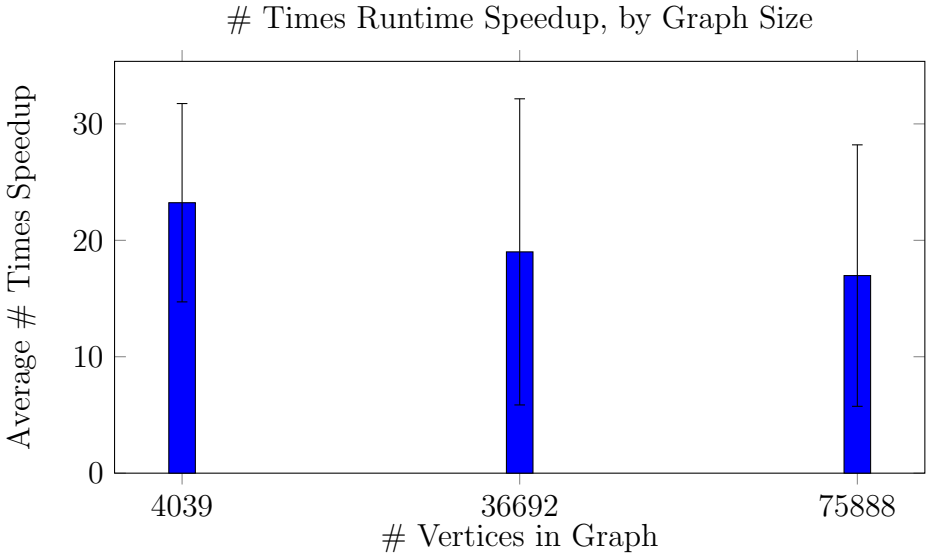
On average, runtime improved by 18.8 ± 11.2 times over all measurements.

6.1 Runtime Speedup By Command



Improvement varied widely both between commands, and within each command, based on the graph it was run on. In general, simpler algorithms, such as adiam saw much greater improvement than more complex algorithms, with many nested loops and complex control flow, such as abc, reducehighdegreetop, and reducetreetop.

6.2 Runtime Speedup By Graph Size



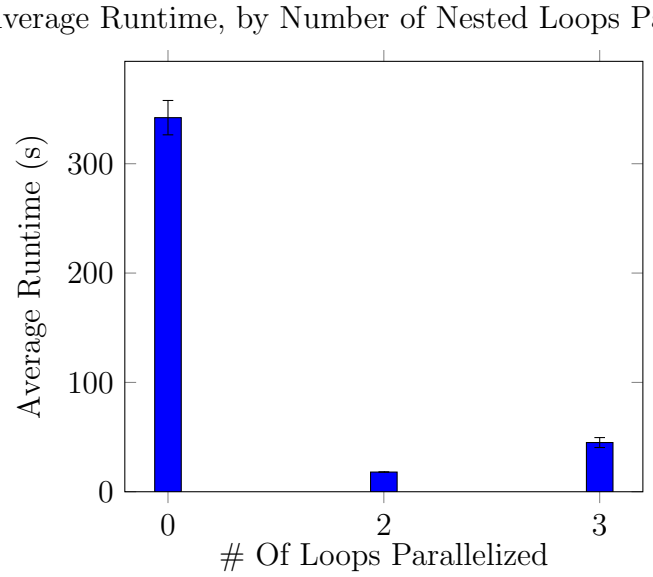
Improvement also varied widely by the size of graph, and within each graph. The smaller graphs (5.txt, 50.txt, small test.txt) have been ommitted, as virtually all algorithms saw no

improvement between parallelized and unparallelized versions when run on those graphs.

From these data points, improvement seems to decrease as graph size increases. This may be due to the extra memory used and needed by these larger graphs, which may slow down the performance of the parallel functions.

6.3 Runtime Speedup By Number of Nested Loops Parallelized

One interesting anomaly showed up when parallelizing the betweeness centrality algorithm - increasing the number of nested loops parallelized slowed down the runtime of the algorithm:



This may be due to multiple reasons - inefficient memory management, or additional overhead required for parallelization outweighing the time saved.