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
- parallel_sort
- 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:

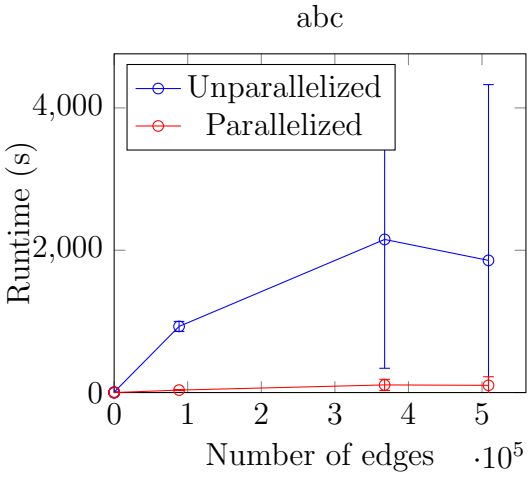
- concurrent_vector
- concurrent_unordered_map
- concurrent_unordered_multimap
- concurrent_priority_queue
- mutex

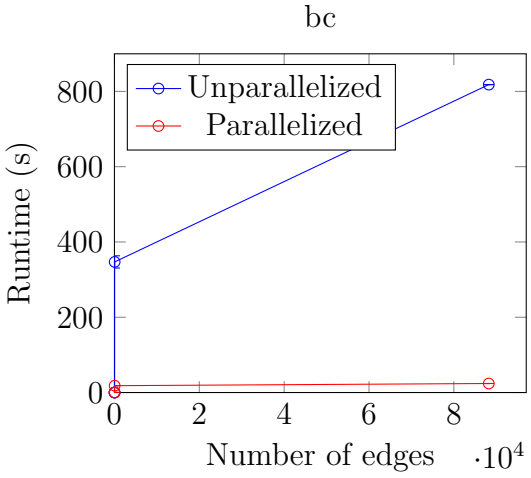
4 Testing

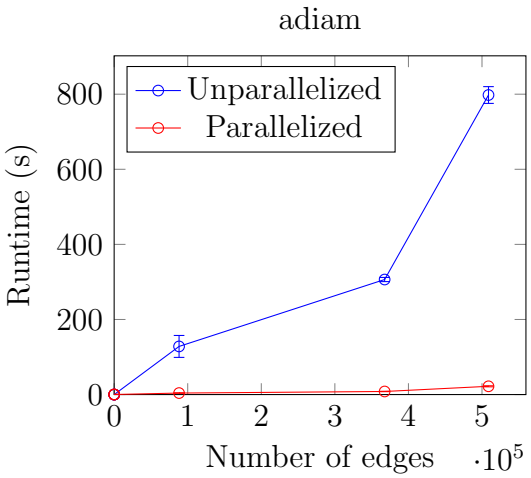
ran the parallelized and unparallelized version of each algorithm 5 times on 6 different graphs
waterloo student linux environment
56-64 threads available
benchmarked time, rounded to seconds

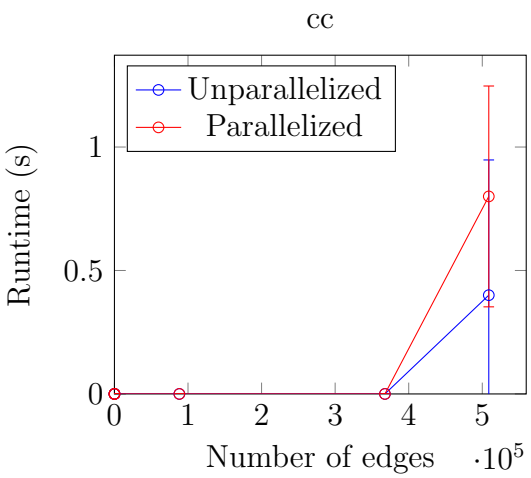
Graph Name	Num. Edges	Num. Vertices	Average Degree
5.txt	5	6	1.666667
50.txt	50	51	1.960784
small_test.txt	55	1904	0.057773
facebook_combined.txt	88234	4039	43.69101
enron.txt	367662	36692	20.04044
epinions.txt	508837	75888	13.41021

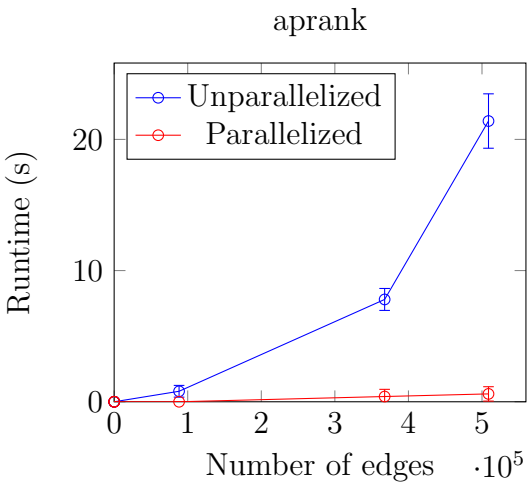
5 Results

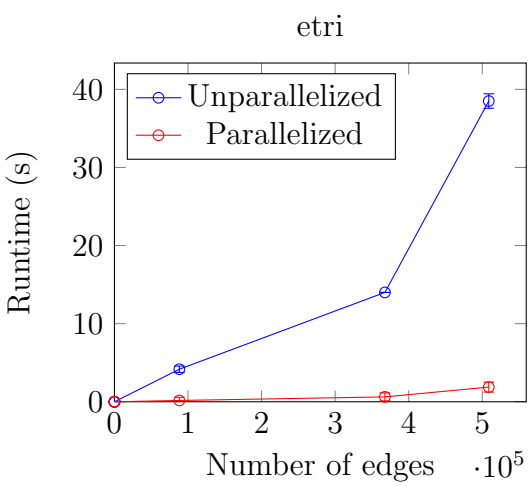

abc
Unparallelized
Parallelized
Runtime (s)
4,000
2,000
0
0 1 2 3 4 5
Number of edges
$\cdot 10^5$

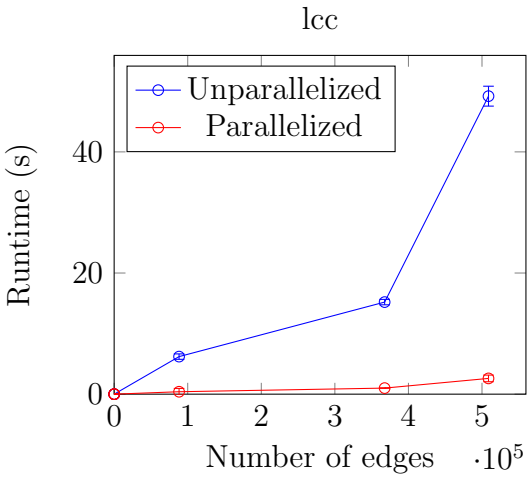

bc
Unparallelized
Parallelized
Runtime (s)
800
600
400
200
0
0 2 4 6 8
Number of edges
$\cdot 10^4$

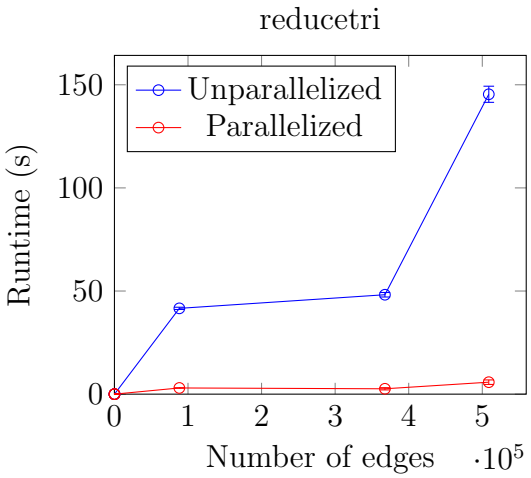

adiam
Unparallelized
Parallelized
Runtime (s)
800
600
400
200
0
0 1 2 3 4 5
Number of edges
$\cdot 10^5$

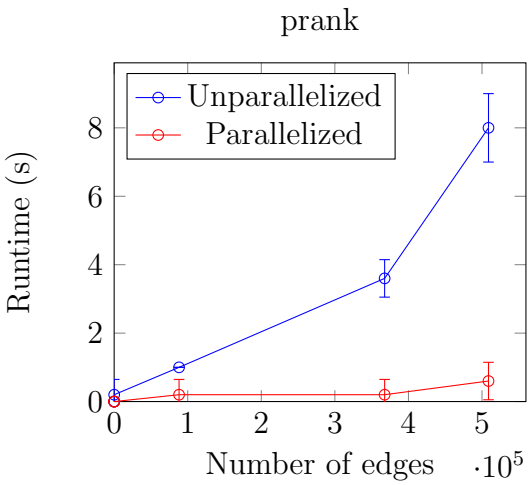

cc
Unparallelized
Parallelized
Runtime (s)
1
0.5
0
0 1 2 3 4 5
Number of edges
$\cdot 10^5$

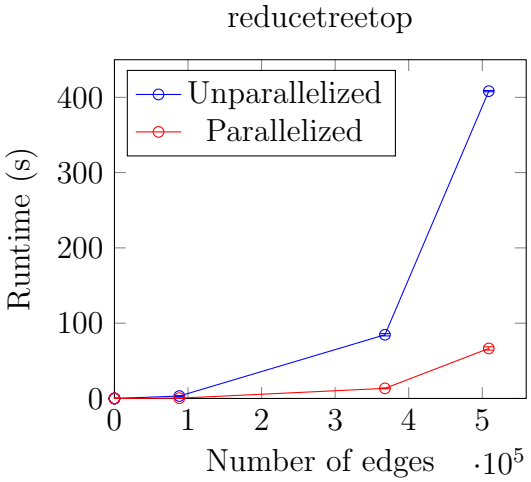

aprank
Unparallelized
Parallelized
Runtime (s)
20
10
0
0 1 2 3 4 5
Number of edges
$\cdot 10^5$

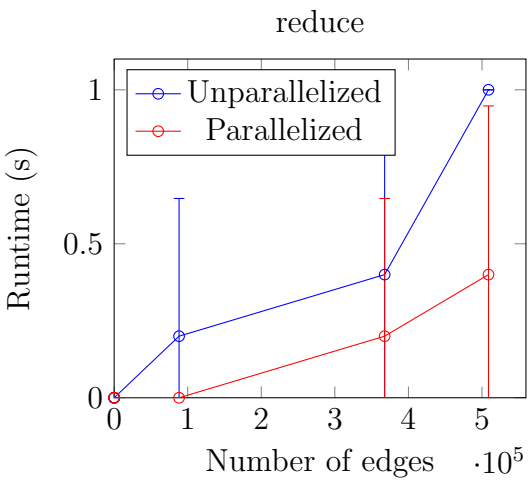

etri
Unparallelized
Parallelized
Runtime (s)
40
30
20
10
0
0 1 2 3 4 5
Number of edges
$\cdot 10^5$

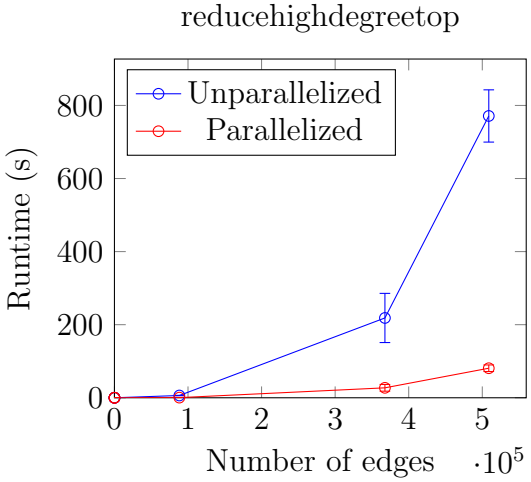

lcc
Unparallelized
Parallelized
Runtime (s)
0
20
40
0
1
2
3
4
5
Number of edges
$\cdot 10^5$

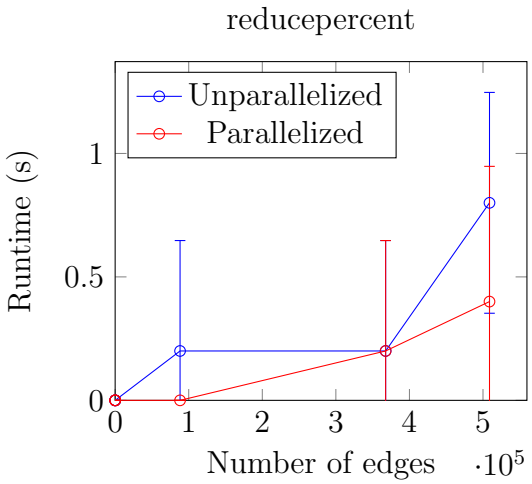

reducetri
Unparallelized
Parallelized
Runtime (s)
0
50
100
150
0
1
2
3
4
5
Number of edges
$\cdot 10^5$


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

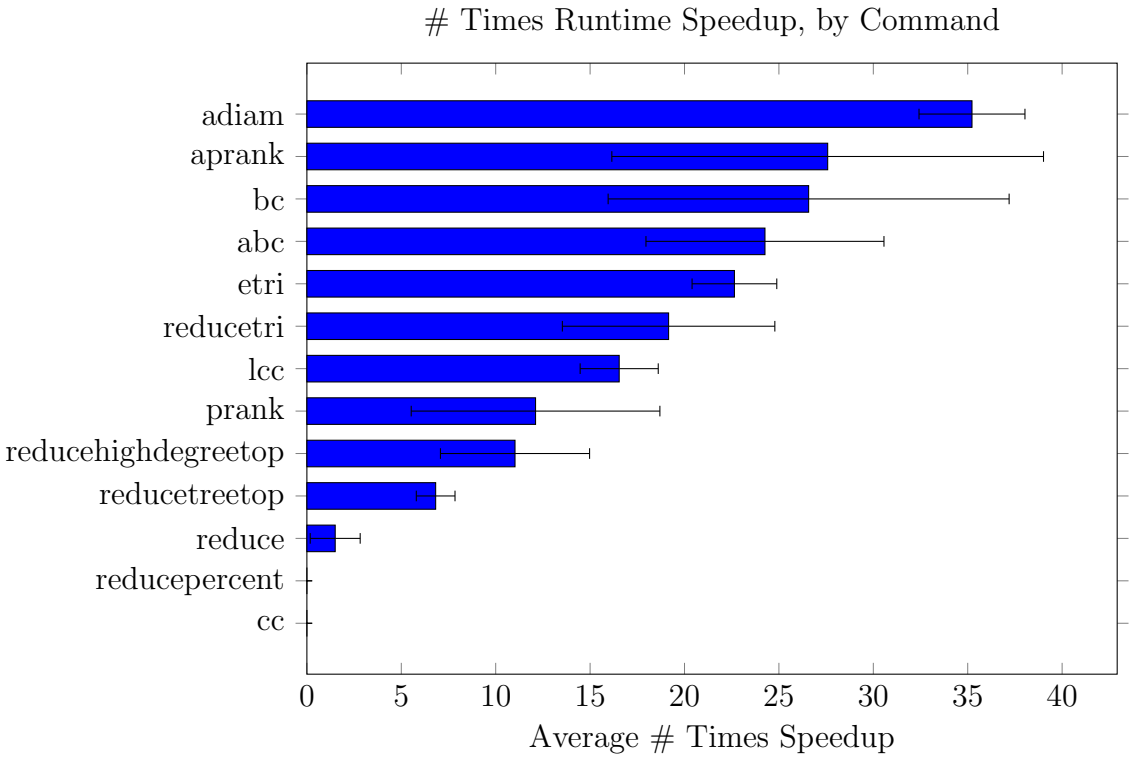

reducetreetop
Unparallelized
Parallelized
Runtime (s)
0
100
200
300
400
0
1
2
3
4
5
Number of edges
$\cdot 10^5$

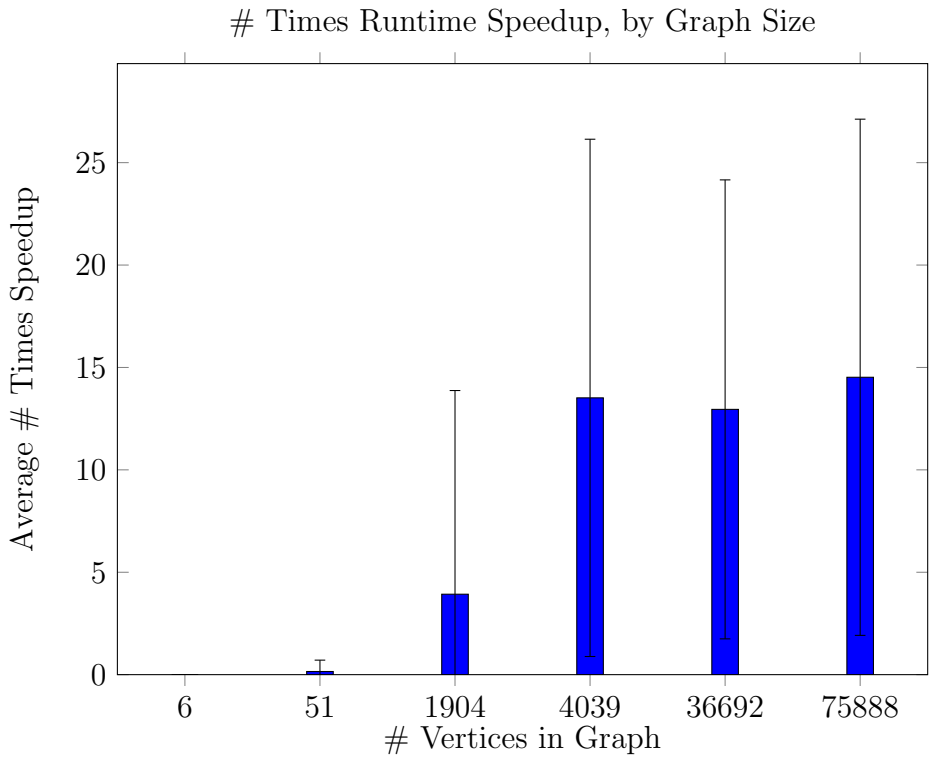

reduce
Unparallelized
Parallelized
Runtime (s)
0
0.5
1
0
1
2
3
4
5
Number of edges
$\cdot 10^5$


reducehighdegreetop
Unparallelized
Parallelized
Runtime (s)
0
200
400
600
800
0
1
2
3
4
5
Number of edges
$\cdot 10^5$


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

6 Analysis


Times Runtime Speedup, by Command
adiam
aprank
bc
abc
etri
reducetri
lcc
prank
reducehighdegreetop
reducetreetop
reduce
reducepercent
cc
0
5
10
15
20
25
30
35
40
Average # Times Speedup


Times Runtime Speedup, by Graph Size
Average # Times Speedup
0
5
10
15
20
25
6
51
1904
4039
36692
75888
Vertices in Graph