FemtoGraph: Lightweight Shared-Memory Graph Processing Framework

[Extended Abstract]

Alex Ballmer
Illinois Institute of Technology
alexandersballmer@gmail.com

Benjamin Walters
Illinois Institute of Technology
bwalter4@hawk.iit.edu

Ioan Raicu
Illinois Institute of Technology
iraicu@cs.iit.edu

ABSTRACT

The emerging applications for large graphs in big data science and social networks has led to the development of numerous parallel or distributed graph processing applications. The need for faster manipulation of graphs has driven the need to scale across large core counts and many parallel machines. While distributed memory parallel systems continue to be used for high performance computing, some smaller systems make use of shared memory (SMP) and larger core counts. We have implemented a graph processing framework for shared memory systems capable of scaling past 48 parallel cores. This system leverages and scale to large core counts and provide a framework for later incorporating distributed processing across multiple nodes.

CCS Concepts

•**Theory of computation** → **Shared memory algorithms;**

Keywords

Graph processing; Parallel Algorithms; Shared Memory

1. INTRODUCTION

FemtoGraph is a graph processing application which will use a vertex-centric approach. This approach involves calling a function in the context of a vertex for each and every vertex. This function can modify or read from edges and other vertices. Computation occurs in intervals or steps, with some form of communication between steps. Vertex-centric algorithms can be either synchronous, meaning every vertex function must finish before the next step begins, or asynchronous, which means that the next step can begin in the context of one vertex immediately. [1]

FemtoGraph is based off of the pregel model. Pregel is a vertex-centric graph processing model. [3] Pregel is synchronous, with computation occurring in steps called supersteps. There is a messaging system for sending data and state to vertices in the next superstep, but not to any vertex in the current superstep. In each step, vertices can do computations, modify neighbor vertices and edges, send messages to vertices in the next superstep, and vote to halt, meaning it cannot run its compute function until it receives a message or all vertices have voted to halt. When all vertices have voted to halt, the simulation ends. [3]

ACM ISBN 978-1-4503-2138-9...$-24.32
DOI: 10.1145/1235

2. PREGEL MODEL

The Pregel model is a type of vertex-centric graph processing paradigm. It is designed to make it easy for vary large amounts of worker threads or processes to modify the graph at one time with minimal collision of resources.

Pregel is vertex-centric, which means that all code runs in the context of a vertex in the graph. Vertices can preform modifications on edges and other neighboring vertices. They can modify tags, data, and other aspects, and add or remove vertices and edges.

Computation in Pregel occurs in steps called supersteps. A single superstep consists of calling a function in the context of every vertex of the graph theoretically in parallel. This can be fully in parallel in the case of very small graphs, but in most case vertices are called in sequence with a number of parallel threads.

Communication between vertices is done by sending messages. Messages can be sent at any time, but only received in the next superstep, as to preserve the parallel nature of Pregel supersteps. Messages can be sent to any number of vertices at once. The messaging system serves as the main method of communicating across supersteps.

At the end of each vertex's compute step, it can choose to vote to halt. This is usually done when the particular piece of computation that it was working on has ended. A vertex that has voted to halt does not perform a compute step and enters a halted state. The vertex can be 'woken up' from the halted state upon receiving a message. When all vertices enter a halted state, the entire simulation halts and returns the data and tags from the vertices.

3. RELATED WORK

There are many existing applications in the world of graph processing. These applications can run on a single node using shared memory, or on multiple nodes using a distributed memory paradigm like MPI or MapReduce.

One of the main similar applications in this area is GraphLab. GraphLab is another vertex-centric graph processing framework that can either run on a single node using shared memory, or as a distributed application. GraphLab is the simplest to compare to FemtoGraph as it can run with shared memory without the harsh overhead of frameworks like Hadoop or Spark. GraphLab is asynchronous, which gives it the edge of not having to wait for the fist superstep to complete before starting the next superstep. [4] Graphlab was the main point of comparison of FemtoGraph

Apache Giraph is another application for processing graphs. Giraph runs on top of the Hadoop framework, and uses a modified vertex-centric approach on top of Hadoop's MapReduce architecture Giraph is used by Facebook to extract meaningful data from the stored social network.

Most work on graph processing is done in parallel, on multiple nodes. These parallel algorithms make little use of shared memory.

4. IMPLEMENTATION

FemtoGraph is implemented in C++ using parts of the Boost C++ library. There are 3 main parts vertex storage, the message queue, and the compute function.

Vertices are stored in an adjacency list using C++ vectors. Edges are represented with symbolic references to another vertex's unique id. Each vertex has a unique id that serves as a simple hashing function to generate its index in the adjacency list. Each vertex has a data or tag object that can be extended or replaced depending on the type of graph being stored. Graph vertices can be added or removed quickly by updating the references. All graph storage is done in memory as of now.

The message queue is implemented using Boost lockfree queues, [5] one for each vertex. Queues are presorted by vertex in a single vector in order to minimize the cost of sorting messages by vertex during the computation. There is one short queue per vertex in the current graph. This could result in very large queue structures for large graphs. This structure improves computation speed at the expense of potential memory fragmentation.

The compute function takes the role of the Pregel update function, and is called once for every vertex in the context of every vertex during a pregel superstep. Compute functions are called theoretically in parallel with a user defined number of threads operating at a single time. This means that computation may as well be occurring in parallel across all vertices even if there is only a single thread.

Threading is done with std::thread threads from the C++ standard library. The only data shared between threads during runtime is the vertex storage and message queue. The vertexes are not accessed frequently enough to need anything more that a simple mutex to lock between threads. The message queue is accessed much more frequently and needs a lockfree approach.

Graphs are input from a file. The format defaults to the Stanford SNAP graph data format, as this is the primary data sets we used for testing. Outputs are written as a csv file with a list of vertices and their associated data, but this can be changed easily depending on the algorithm run. File input and output, memory initialization, and graph creation are not parallelized. Only the compute step is parallelized.

5. EVALUATION

5.1 Results

We collected runtime information at varying core counts and function call graphs for FemtoGraph and GraphLab. FemtoGraph and GraphLab were compared for scaling performance on a large shared memory machine.

We used profiling tools such as valgrind and callgrind, along with basic runtime measurement, to measure the efficiency and speed of FemtoGraph in relation to Graphlab.

5.2 Testing Conditions

All tests were done:

- On a single node system with 48 cores, 2 sockets, and NUMA
- using a modified pagerank algorithm defined as $\forall t \in P : r^{(t)}(t) = (1-\alpha) \cdot r(t) + \alpha \sum_{(s,t)\in L} \frac{r^{(t-1)}(s)}{|L(s)|}$
- Counting only the compute steps (not initialization or reading in data) for both FemtoGraph and GraphLab.

6. GRAPHLAB

GraphLab is our main point of comparison for FemtoGraph. GraphLab is vertex-centric, but has a few main differences from the vertex-centric Pregel model.

On of the most glaring differences in GraphLab is that its equivalent of the Pregel superstep is asynchronous. This means that a single vertex or a new group of vertices can begin a new superstep before the others finish. This means that algorithms for Graphlab must take into account that any vertex that they are communicating with may be in a totally different step. It also means that there is a slight theoretical speed increase in relation to synchronous algorithms.

In practice, graphlab is very efficient at low core counts (Figure **??**). It does not scale very well, peaking at at about 20 threads (Figure **??**) and scaling in reverse afterwards.

7. DIFFICULTIES ENCOUNTERED

In the pregel model for graph processing, the main bottleneck encountered when scaling to many cores is the message queue. The message queue and the vertex storage are the only two data structures accessed from multiple threads. Vertices are not modified enough to warrant anything more than simple mutex based locking.

The message queue, made clear by profiling with the callgrind function call analysis tool, is accessed at a very high rate from vertex compute functions from all threads. This can lead to race conditions, slowdowns, and deadlocks. A mutex based locking system resulted in an extreme slowdown to the point where the system scaled in reverse (Figure 1).

The message queue also had a performance issue with allocating memory for dynamic structures like queues and vectors. This performance issue was evident in the callgrind profiling, but was not noticed until later.

8. PROFILING

Profiling to find bottlenecks was done with Callgrind, a part of the Valgrind suite of profiling tools. Callgrind maps out

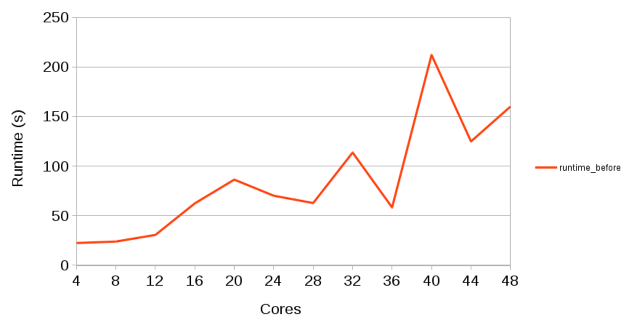
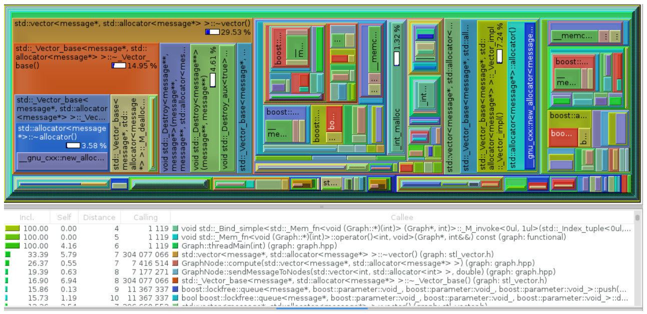


Figure 1: FemtoGraph scaling using mutex based message queue



std::vector<message*, std::allocator<message*> >::~vector()
29.53 %
std::_Vector_base<message*, std::allocator<message*> >::_Vector_base()
14.95 %
std::allocator<message*>::~allocator()
3.58 %
__gnu_cxx::new_alloc...
int_malloc
boost::...

Incl.	Self	Distance	Calling	Callee
100.00	0.00	4	1 119	void std::_Bind_simple<std::_Mem_fn<void (Graph::*)(int)> (Graph*, int)>::_M_invoke<0ul, 1ul>(std::_Index_tuple<0ul,...
100.00	0.00	5	1 119	void std::_Mem_fn<void (Graph::*)(int)>::operator()<int, void>(Graph*, int&&) const (graph: functional)
100.00	4.16	6	1 119	Graph::threadMain(int) (graph: graph.hpp)
33.39	5.79	7	304 077 066	std::vector<message*, std::allocator<message*> >::~vector() (graph: stl_vector.h)
26.37	0.55	7	7 416 514	GraphNode::compute(std::vector<message*, std::allocator<message*> >) (graph: graph.hpp)
19.39	0.63	8	7 177 271	GraphNode::sendMessageToNodes(std::vector<int, std::allocator<int> >, double) (graph: graph.hpp)
16.90	6.94	8	304 077 066	std::_Vector_base<message*, std::allocator<message*> >::~_Vector_base() (graph: stl_vector.h)
15.86	0.13	9	11 367 337	boost::lockfree::queue<message*, boost::parameter::void_, boost::parameter::void_, boost::parameter::void_>::push(...
15.73	1.19	10	11 367 337	bool boost::lockfree::queue<message*, boost::parameter::void_, boost::parameter::void_, boost::parameter::void_>::d...

Figure 2: Callgrind profiling after optimization

the graph of function calls and records how long the application spends in each function. Using callgrind, we can locate areas of the program that are hanging, due to an lack of optimization or a threading error.

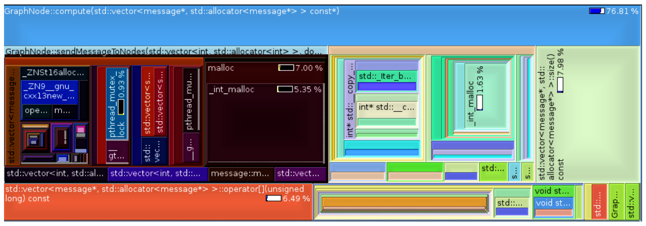
Callgrind can be visualized by a tool called kcachegrind, which arranges functions into a cluster. The function's size is based on the time spent in the function. The largest functions are the ones that have spent the most time running, and are therefore a potential cause of bottleneck. Some long running functions are not cause for alarm. The main() function runs for the entire duration of the program, for instance.

The profiling showed problems with allocating memory for message queue subqueues. This showed up as a large time spent in the malloc function. This slowdown was not noticed until after work on optimizing the queue locking was done The highlighted area in figure 2) shows the large malloc functions taking up a quarter of the plot. This bottleneck was eventually fixed. The fix can be seen in a later callgrind plot in figure 3.

The bottleneck created by locking in the message queue showed up as a large amount of time spent in the chain of functions related to pushing a message to the queue, and popping a message off the queue. This can be seen in the large number of large-sized vector related functions in the callgrind plot in figure 3.

Profiling led us to find the bottleneck in the message queue was caused by two main problems: locking between threads and allocating memory. Once these bottlenecks were fixed, FemtoGraph was able to perform normally.

9. SOLUTION



GraphNode::compute(std::vector<message*, std::allocator<message*> > const*)
76.81 %
GraphNode::sendMessageToNodes(std::vector<int, std::allocator<int> >, do...
malloc
7.00 %
_int_malloc
5.35 %
std::vector<message*, std::allocator<message*> >::operator[](unsigned long) const
6.49 %
17.98 %

Figure 3: Callgrind profiling before optimization

My main solution for the message queue was to use a vector of Boost::lockfree queues for the message queue. [5] The vector was used to presort messages by vertex in order to minimize compute time sorting message when they were received. Adding new vertices still requires a mutex, as the vector is not lockfree. This mutex is not a problem for the algorithms that I tested, as they spend most of their time updating current vertices. The lockfree queues minimize the total bottleneck in the message queue.

A we tried a few intermediate solutions before implementing the final solution using lockfree queues. We tried various locking patterns including mutexes across the entire queue, across sections of the queue, and across individual sections of the array of queues. The full mutex across the entire queue had too much of a performance impact, essentially scaling in reverse. The partial mutex caused some race conditions on inserting messages. The. The local mutex on sections of the array proved too difficult to implement in the time allowed.

The problem with memory allocation was solved by allocating all needed memory for the boost lockfree queues at startup, instead of at runtime. Fixed sized queues do not need to block while they allocate more memory. As there is only one queue per vertex, there is little risk of running out of memory, even at a fixed size.

10. RESULTS

We compared FemtoGraph and GraphLab on running Pagerank on a large Stanford SNAP graph (Figure 4). The graph used was the Wikipedia Talk Network, a directed graph based off of discussion about Wikipedia article edits with 2394385 Vertices and 5021410 edges. [2] The final version of FemtoGraph is capable of scaling to 48 cores on a single large system. At 28 cores, it begins to overtake graphlab in terms of runtime. GraphLab scales very weakly, only increasing in performace below 20 cores. After 20 cores, GraphLab scales in reverse (Figure 5). In the case of both applications, only compute time was counted. reading in data and initializing graphs were outside of the measured data.

11. CONCLUSION

FemtoGraph is a lightweight, single node graph processing system capable of scaling to large core counts and outperforming some of the current graph processing standards while using the pagerank algorithm. FemtoGraph shows that the pregel model performs well under shared memory situations at scale.

12. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under awards NSF-1461260 (REU).

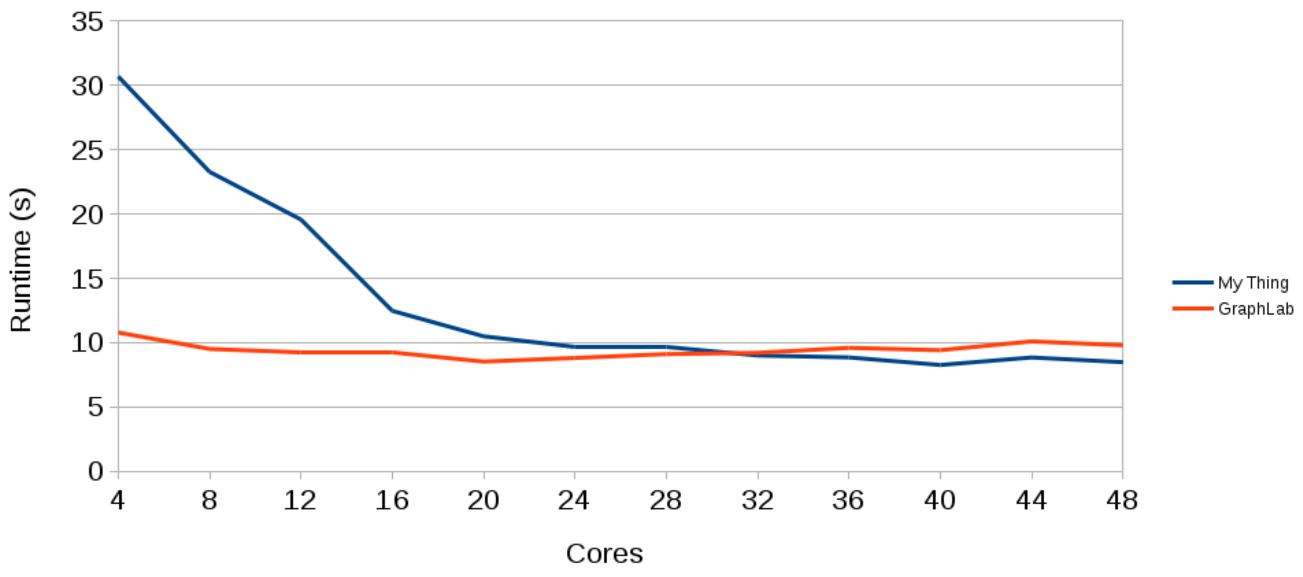


Figure 4: FemtoGraph in comparison to GraphLab on a single node

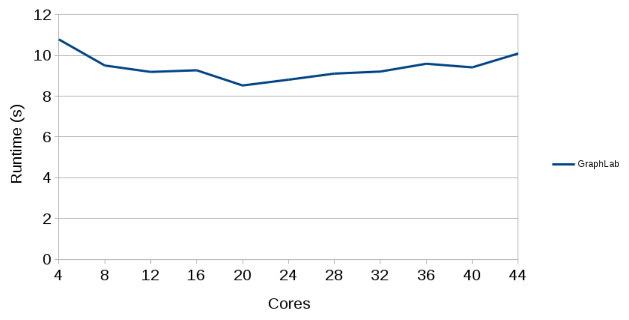


Figure 5: Graphlab scaling performance

13. REFERENCES

[1] O. Batarfi, R. El Shawi, A. G. Fayoumi, R. Nouri, A. Barnawi, S. Sakr, et al. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

[2] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[4] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, Oct. 2015.

[5] B. Schling. *The Boost C++ Libraries*. XML Press, 2011.