2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Designing Hybrid Architectures for Massive-Scale Graph Analysis

David Ediger
PhD Candidate
Georgia Institute of Technology

David A. Bader
PhD Advisor
Georgia Institute of Technology

***Abstract*—Turning large volumes of data into actionable knowledge is a top challenge in high performance computing. Our previous work in this area demonstrated algorithmic techniques for massively parallel graph analysis on multithreaded systems. This work led to the development of GraphCT, the first end-to-end graph analytics platform for the Cray XMT and x86-class systems with OpenMP, and STINGER, a high performance, multithreaded, dynamic graph data structure and algorithms. Both of these packages are freely available as open source software. This dissertation research culminates in experimental and analytical techniques to study the marriage of disk-based systems, such as Hadoop, with shared memory-based systems, such as the Cray XMT, for data-intensive applications. David Ediger is a fifth year PhD candidate in Electrical and Computer Engineering.**

I. Introduction

The quantity of rich, semi-structured data generated by sensor networks, scientific simulation, business activity, and the Internet, grows daily. In the past, collection and analysis of data using relational queries was sufficient. Today, complex analytics and near real-time responses to new data are required. To meet the demand and keep up with ever-increasing data rates, novel solutions in the form of hardware, software, and algorithms are required.

The objective of this research is to investigate architectural requirements for data-intensive applications in massive graph analysis, such as community finding and anomaly detection. Using emerging hybrid systems, we will map applications to architectures and close the loop between software and hardware design in this application space. Algorithm engineering, experimental techniques, and modeling and simulation are used to evaluate the design space of large-scale parallel systems for graph analytics that use both shared memory and storage devices.

Shared memory systems for graph analysis leverage the benefits of DRAM (fast single-word access) while not requiring data to be partitioned. Storage device-based systems enable scale-out to larger data sets than is possible with DRAM at a cost of additional time and physical size. MapReduce has attracted the attention of the large graph analytics community for its ability to perform operations on petabyte-scale datasets [1]. The core strength of MapReduce is its ability to store truly massive datasets using commodity hardware. However, the programming model gives the programmer no control over data movement or locality. Pregel is a distributed graph processing system with a C++ API developed by Google [2]. To avoid issues of deadlock and data races, Pregel uses a bulk synchronous parallel (BSP) programming model, as opposed to the BSP algorithmic model [3]. In each superstep, a vertex can 1) receive messages from the previous iteration, 2) do local computation or modify the graph, and 3) send messages to vertices that will be received in the next iteration. Similar to MapReduce in many ways, chains of iterations are used to solve a graph query in a fault-tolerant manner across thousands of distributed systems. Unlike MapReduce, however, vertices in Pregel can maintain state between iterations, reducing the communication cost.

The dissertation begins with our previously published work [4], [5], [6], [7], [8], [9], [10] as a foundation of new algorithms and techniques for large-scale graph analysis. This paper previews new analysis on performance aspects of programming models and hardware characteristics to give insights into the design space of future systems on these problems.

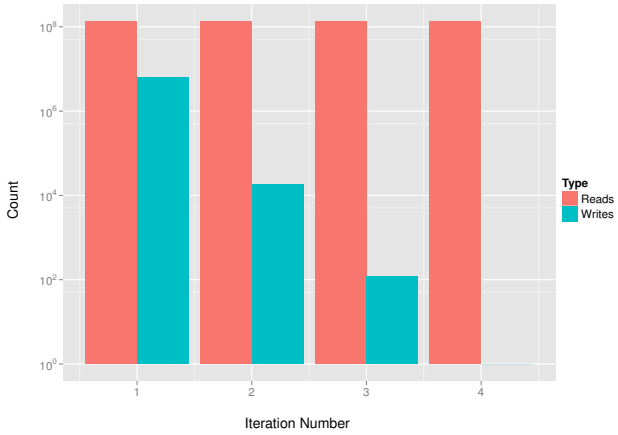
II. Algorithmic Effects of BSP

To determine the algorithmic effects of the bulk synchronous parallel programming model on graph algorithms, we devised a shared memory BSP environment written on top of GraphCT [10]. The BSP environment supports lightweight messaging between vertices. Incoming and outgoing message queues are transferred between supersteps with a single pointer swap. An array of vertex state is maintained between iterations. To measure performance, we track the time to execute each superstep as well as the number of messages sent and received.

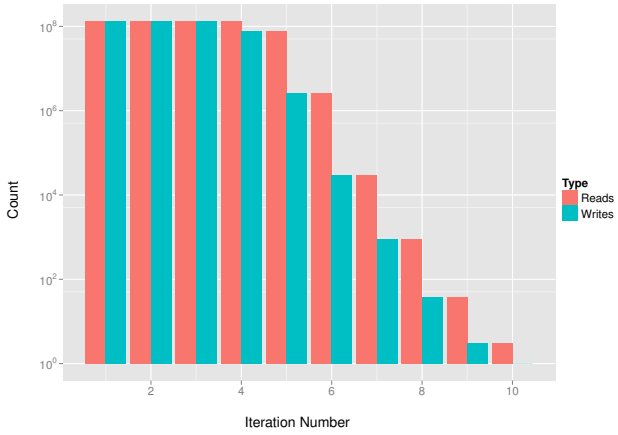
The comparison code is a hand-written C-language implementation of connected components in GraphCT. The implementation is instrumented to measure the time per iteration, time to solution, as well as the number of reads and writes. Both the shared memory and BSP implementations are run on the same input graph on the same machine and we examine intermediate data for correctness.

We run Shiloach-Vishkin connected components on an undirected, scale-free RMAT [11] graph with 2 million vertices and 131 million edges. The shared memory implementation from GraphCT completes in four iterations. In Figure 1(a), each iteration performs approximately 132 million reads. A write is performed when a component label change is detected. The number of writes recorded drops quickly from over 6

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.172



(a) Shared Memory Connected Components



(b) Bulk Synchronous Parallel Connected Components

Fig. 1. The number of reads and writes per iteration performed by connected components algorithms on a scale-free graph with 2 million vertices and 131 million edges.

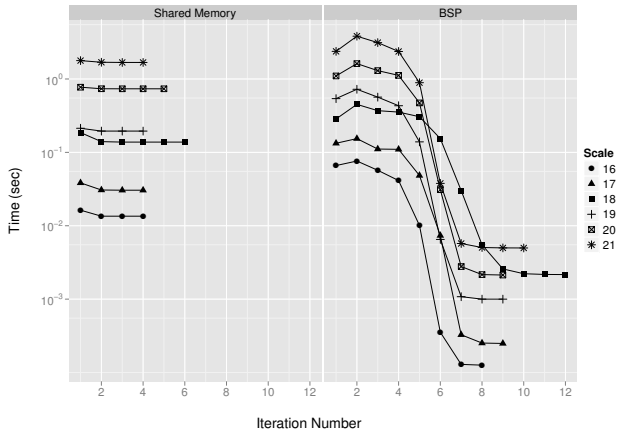


Fig. 2. Execution time per iteration performed by connected components algorithms. Scale is the log base 2 of the number of vertices and the edge factor is 8. For Scale 21, shared memory completes in 8.3 seconds, while BSP takes 12.7 seconds.

million in the first iteration to only 121 in the third iteration. The ratio of total reads to writes is approximately 85:1.

In Figure 1(b), the BSP components algorithm takes 10 iterations to complete. In the BSP algorithm, the only active vertices are those that received messages from the previous superstep. In the first several supersteps, nearly all vertices are active and modifying their component labels. Beginning with the fifth superstep, the number of component label changes drops off dramatically. However, the ratio of total reads to total writes for BSP is 1.3:1.

On a dual-socket 2.4 GHz Intel Xeon E5530 with 12 GiB of main memory, the shared memory connected components takes 6.83 seconds compared to the BSP implementation that completes in 12.7 seconds. The shared memory implementation processes at a rate of 19.3 million edges per second compared to 10.4 million edges per second for the BSP.

Despite the fact that the BSP connected components algorithm computes only on the active vertex set and performs only 15 percent more reads than the shared memory algorithm, it requires double the number of iterations and takes twice as long to complete.

Note that the time per iteration for the shared memory implementation is constant in Figure 2. In the BSP model, the time is proportional to the number of messages being sent and received. Early iterations take two orders of magnitude longer than later iterations. However the early iterations are too long to make this approach competitive with the shared memory implementation.

Since messages in the BSP model cannot arrive until the next superstep, vertices processing in the current superstep process on stale data. Because data move forward in the computation, the number of iterations required until convergence is at least a factor of two larger than in the shared memory model. In the shared memory algorithm, once a vertex discovers its label has changed, that new information is available to all of its neighbors immediately and can be further consumed. While the shared memory algorithm requires edges and vertices to be read and processed that will not change, the significantly fewer iterations results in a shorter execution time.

We can apply these execution measurement techniques to other graph algorithms, such as breadth-first search and triangle counting, and use their results to model larger-scale problems on future systems. Please reference the completed dissertation for these details.

III. Modeling Data Access Time

Using algorithmic analysis, we seek to model data access patterns for large-scale graphs on future systems that use shared memory and disk-based storage.

Let us consider the performance of a hypothetical future system on a large graph algorithm. We will first model shared memory connected components on a graph with 17.2 billion vertices and 137.4 billion edges. The memory footprint is

given by $(|V| + 2|E|)$ 8-byte elements, or approximately 2.3 TiB. We assume that in-memory Shiloach-Vishkin will converge in seven iterations, although this will depend on graph diameter (BSP would require approximately double). We do not consider network bandwidth or latency.

Considering only the shared memory algorithm for connected components on a compressed sparse row graph, in each iteration we read $|E|$ source vertices and $|E|$ destination vertices, which are both stored and accessed contiguously. We also read $|V|$ source component labels and $|V|$ destination component labels, which are random accesses. For a graph with an average degree of 8, we calculate that 11 percent of reads will be random and 89 percent of reads with be linear.

The total execution time is given in Equation 1. I is the number of iterations required for completion and T_i is the time for a single iteration i.

$$T_{total} = \sum_{i=1}^{I} T_i \tag{1}$$

For shared memory connected components, the work per iteration is constant, so the total time is the product of the number of iterations and the time for a single iteration. Under this model, each iteration will perform 309 billion memory references per iteration of connected components, for a total of 2.16 trillion memory reads. Let us consider a hypothetical in-memory system with 4096-way concurrency in the memory. This level of concurrency is equivalent to a Cray XMT2 with 4-channel memory. When measuring the performance of connected components running on a current system in memory, the processing time per edge was determined to be 8.33 nanoseconds, or 120 million references per second. At this rate, we estimate the time per iteration to be 629 milliseconds for a total computation time of 4.4 seconds in memory.

The computation time estimates for the disk-based models are more complicated. A random access will be charged according to the latency of access. Linear accesses must be charged according to the linear access bandwidth. The total time will be a combination of the two costs based on the frequency of random accesses with respect to linear accesses. We assume these to be uniformly distributed regardless of system layout.

The time per iteration is given in Equation 2. The latency of a unit random access is L seconds. B is the peak bandwidth of the device in bytes per second. C is the number of concurrent storage devices. The total number of memory references (M_{ref}) is divided between random (M_{random}) and linear (M_{linear}), as shown in Equation 3.

$$T_i = \frac{1}{C}\left(L(1 + M_{random}) + \frac{8 \cdot M_{linear}}{B}\right) \tag{2}$$

$$M_{ref} = M_{random} + M_{linear} \tag{3}$$

The algorithmic analysis estimated that 11 percent of reads in connected components are random with the remaining 89 percent linear. Considering spinning hard disk drives, 11 percent of the memory references will be charged 2 milliseconds, while the remainder will be charged based on the linear access rate of 12.5 million references per second. Assuming 65,536 concurrent disk devices, the computation time is estimated to be 66,100 seconds.

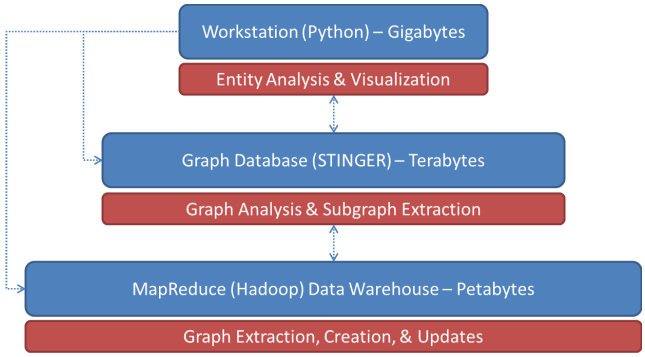


Fig. 3. A proposed hierarchy of data analytics the includes raw, unstructured data in a commodity cluster, a special purpose high-performance graph representation, and many client analytics that operate in parallel.

The solid state disk has both a high linear access bandwidth (550 MiB per second) and a comparatively low random access latency (12.5 microseconds). Assuming 11 percent of accesses are random, the per-iteration time is reduced to 6.56 seconds for a total time of 45.9 seconds.

In order for hard disk drives to match main memory performance, approximately 107 million drives would be needed under this model. Alternatively, the percentage of random accesses would need to be reduced to 0.04 percent. For SSDs, the concurrency required is 675,000 or a random access percentage of 1.1 percent. It is unlikely that this extreme level of spatial reuse can be found in graph algorithms that have been shown to contain little locality.

IV. A Hybrid System Architecture

Future systems will combine shared memory systems with disk-based clusters and user workstations. Figure 3 depicts the proposed hierarchy of data analytics. The commodity cluster contains the largest volume of data storage and sits at the bottom. It holds the raw, unstructured data. Above it, a graph database is built on a shared memory platform that has an order of magnitude less memory, but is faster and more flexible. Above the graph database, the analyst workstation analyzes smaller subgraphs extracted from the graph database.

Raw, unstructured data flows into the data warehouse as it is created and collected. The data warehouse, which specializes in data parallel operations, extracts the entities and relationships from the text. These new updates are sent to the graph database as a stream of new edges. The graph database (STINGER [6] in our example) processes these new edge updates in its internal graph representation and updates continuously running analytics. Analysts using advanced pro-

gramming environments, such as Python, extract subgraphs from the graph database for further investigations that do not scale to large datasets. It is impossible to visualize graphs with more than several hundred vertices.

The hierarchy of graph systems and representations behaves much like a memory hierarchy, and we can apply many well-understood aspects of the memory hierarchy to this problem. All data that resides in a higher level of the hierarchy must also reside in all lower levels. The data that an edge represents in the graph database must remain in the data warehouse. A strategy for good performance throughout the system is to keep the most relevant data in the graph database so that a minimum number of queries must require intervention by the data warehouse.

When the data representation becomes full, the stream of edges will not cease. Rather, new edges will need to be evaluated and stored, if relevant. Inserting a new edge into the representation requires other data to removed and overwritten. One possibility is to purge the oldest data in the graph. Other approaches may utilize analytic results to determine the data least used (or containing the least information).

In this hybrid system, we must consider where to run analytics on the data. Although there could be many criteria, we will focus on minimizing time-to-solution. We assume that the data warehouse contains the ground truth, with the graph database server holding a representation of most, but possibly not all, of the edges in the graph.

If the graph under consideration is too large to run on an analyst workstation, it must be run on the graph database or in the data warehouse. Because the graph database is a shared memory system specifically designed for graph queries, it may be more efficient to run the query there provided that all of the data needed is present in the representation.

If edges are not present in the graph database, and need to be retrieved from the data warehouse, there will be an additional cost to doing so. To complete this research, we will apply the data access model to this scenario. The key research question is to determine what percentage of edges must be missing from the graph representation to move the computation to the data warehouse given the algorithm access pattern. This research will help to better understand large-scale system design trade-offs for these massively irregular data-intensive problems.

V. Remaining Objectives and Challenges

The remaining objective in this research is to apply performance measurements taken from shared memory and disk-based systems and the performance model to the proposed hybrid system. With this data, we will have a better understanding of how to build and model hybrid memory- and disk-based systems for data-intensive science. We will be able to predict, for a given algorithm or application, where in the hierarchy of data analytics the query should be run. The need for integrated system architectures pushes large-scale, streaming graph analytics closer to real-time query and response.

Acknowledgments

This work was supported in part by the Pacific Northwest National Lab (PNNL) Center for Adaptive Supercomputing Software for MultiThreaded Architectures (CASS-MT).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] 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 international conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[3] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[4] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarría-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP'09)*, Rome, Italy, May 2009.

[5] K. Jiang, D. Ediger, and D. A. Bader, "Generalizing k-Betweenness centrality using short paths and a parallel multithreaded implementation," in *The 38th International Conference on Parallel Processing (ICPP 2009)*, Vienna, Austria, Sep. 2009.

[6] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *Workshop on Multithreaded Architectures and Applications (MTAAP)*, Atlanta, Georgia, Apr. 2010.

[7] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. Farber, and W. N. Reynolds, "Massive social network analysis: Mining twitter for social good," *Parallel Processing, International Conference on*, pp. 583–593, 2010.

[8] D. Ediger, E. J. Riedy, D. A. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks," in *5th Workshop on Multithreaded Architectures and Applications (MTAAP)*, May 2011.

[9] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *9th International Conference on Parallel Processing and Applied Mathematics (PPAM)*. Springer, Sep. 2011.

[10] D. Ediger, K. Jiang, J. Riedy, and D. Bader, "Graphct: Multithreaded algorithms for massive graph analysis," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, 2012.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM)*. Orlando, FL: SIAM, Apr. 2004.