Graph Analytics: Complexity, Scalability, and Architectures

Peter M. Kogge
CSE Dept.
Univ. of Notre Dame
Notre Dame, IN USA
kogge@cse.nd.edu

***Abstract*—Big Data as expressed as "Big Graphs" are growing in importance. Looking forward, there is also increasing interest in streaming versions of the associated analytics. This paper develops an initial template for the relationship between "traditional" batch graph problems, and streaming forms. Variations of streaming problems are discussed, along with their relationship to existing benchmarks. Also included is a discussion of classes of parallel architectures (including newly emerging ones) and how such kernels are liable to scale on them. Preliminary projections for some of these systems is presented.**

***Keywords*-graph analytics; streaming analytics; scalability; emerging architectures**

I. Introduction

A graph consists of a set of objects (vertices) and links (edges) between pairs of those objects that represent some sort of relationships. Computing over such graphs is of increasing importance to a wide spectrum of application areas ranging from "conventional" communication and power networks, transport, and scheduling, to rapidly growing applications such as recommendation systems, social networks, medical informatics, genomics, and cyber-security.

Outputs of such analysis may range from computing properties of individual vertices (such as vertex *out-degree*: the number of incoming or outgoing edges) to properties of the graph as a whole (such as the *diameter*: maximum distance between any two vertices, or a *covering*: minimum set of edges that connects all vertices). It also includes properties of pairs of vertices (such as the shortest path between them) and the properties of subgraphs (such as components that are *connected* or form a *spanning tree*). More recently, similarities between vertices or sub-graphs have become important in applications such as community detection and link prediction.

Today most graph analytics are "batch," that is execution of a function that is applied to an entire graph or a major subset of the graph as it exists at a particular point in time. Further, in many cases these are analogs of analytics applied to traditional big data that treat data as multiple large sets of related tabular data.

A particularly important emerging class of graph analytics are ones that are *streaming*, that where an incoming stream of individually small-scale *updates*, such as additions or deletions to vertices or edges, or modification of their properties, is performed. Analytics in such cases may be more localized, but include conditions that may trigger larger analytics where the results are needed "in real time."

Benchmarks that attempt to represent the computations for such analytics are just emerging, especially for the streaming cases. Initial indications are that such problems are very irregular with little locality. This means that the characteristics needed by high end graph computing systems are liable to be significantly different from that of high end dense scientific flop-intensive systems. Handling minimal locality and high sparsity in particular are characteristics that need to become first-order requirements for truly efficient future systems.

In outline, this paper[1] proceeds as follows. Section II discusses graphs and existing graph benchmarks. Section III overviews an initial description of a processing flow that encompasses both batch and streaming graph analytics. Section IV gives some data on execution on current architectures. Section V does the same for emerging architectures. Section VI concludes.

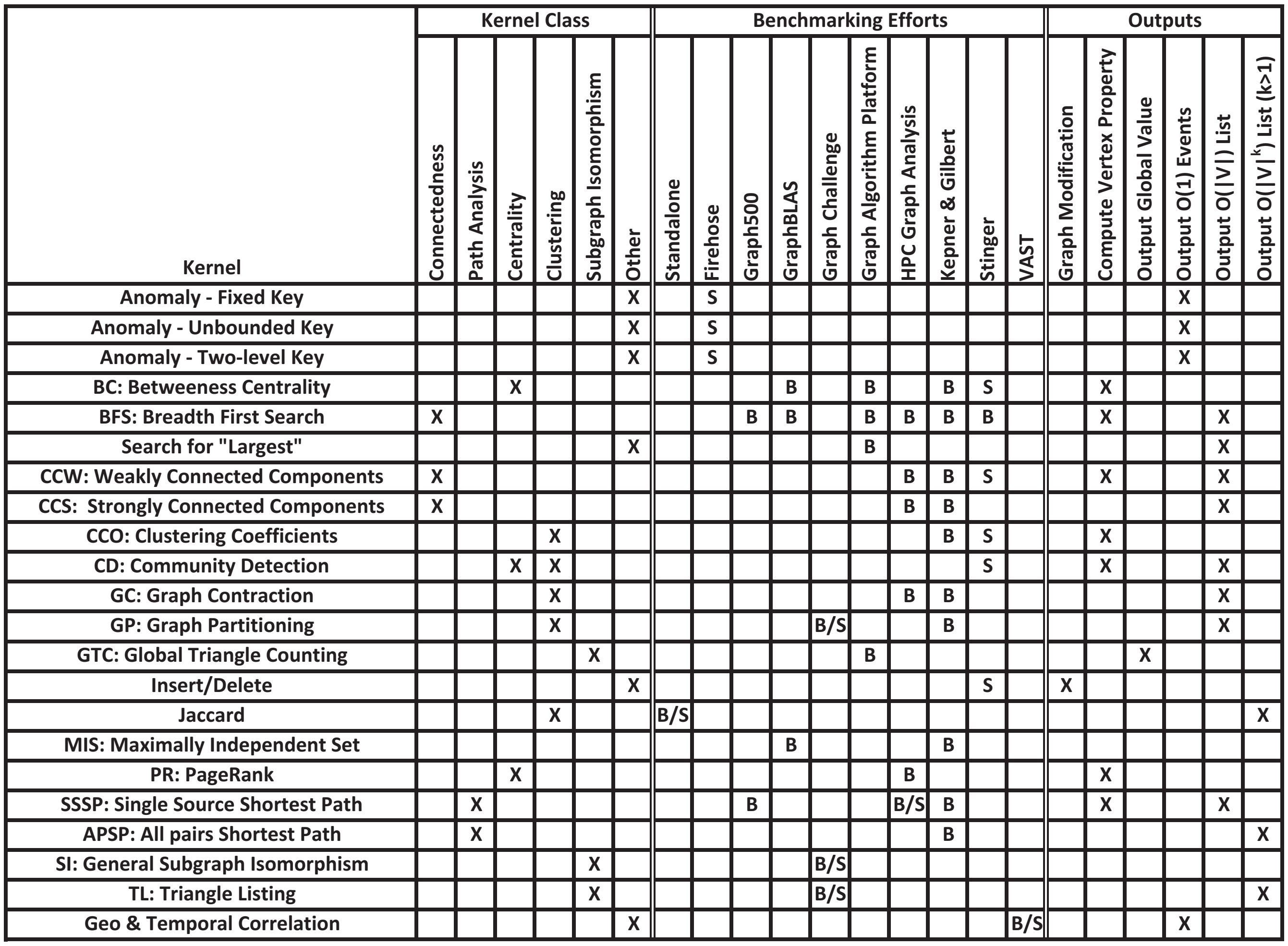
We note that the discussion of kernels and benchmarks here does not include either general graph database systems such as Neo4j[29] or RDF/SPARQL[7], [13], or the growing number of graph programming systems (PREGEL[25], GraQL[14], or KEL[24], etc.).

II. Graph Kernels and Benchmarks

For this paper we define a **kernel** as a function that is typical of the kinds of functions that are found in real applications, but simplified in some way. Likewise a **benchmark** is a collection of one or more kernels applied in a relatively structured way to data sets with well-controlled characteristics, with the goal of using the controlled environment to gain insight into performance characteristics of the underlying system.

In a typical kernel or benchmark today, graph objects are largely defined by generic vertices and edges of single classes, with at best a few vertex **properties** (values attached to the vertex) such as in and out-degree, and a few overall

[1]This paper is based on a previous invited presentation [22].



	Kernel Class						Benchmarking Efforts										Outputs					
Kernel	Connectedness	Path Analysis	Centrality	Clustering	Subgraph Isomorphism	Other	Standalone	Firehose	Graph500	GraphBLAS	Graph Challenge	Graph Algorithm Platform	HPC Graph Analysis	Kepner & Gilbert	Stinger	VAST	Graph Modification	Compute Vertex Property	Output Global Value	Output O(1) Events	Output O(|V|) List	Output O(|V|^k) List (k>1)
Anomaly - Fixed Key						X		S												X		
Anomaly - Unbounded Key						X		S												X		
Anomaly - Two-level Key						X		S												X		
BC: Betweeness Centrality			X							B		B		B	S			X				
BFS: Breadth First Search	X								B	B		B	B	B	B			X			X	
Search for "Largest"						X						B									X	
CCW: Weakly Connected Components	X												B	B	S			X			X	
CCS: Strongly Connected Components	X												B	B							X	
CCO: Clustering Coefficients				X										B	S			X				
CD: Community Detection			X	X											S			X			X	
GC: Graph Contraction				X									B	B							X	
GP: Graph Partitioning				X							B/S			B							X	
GTC: Global Triangle Counting					X							B							X			
Insert/Delete						X									S		X					
Jaccard				X			B/S															X
MIS: Maximally Independent Set										B				B								
PR: PageRank			X										B					X				
SSSP: Single Source Shortest Path		X							B				B/S	B				X			X	
APSP: All pairs Shortest Path		X												B								X
SI: General Subgraph Isomorphism					X						B/S											
TL: Triangle Listing					X						B/S											X
Geo & Temporal Correlation						X										B/S				X		

Figure 1. The Spectrum of Existing kernels.

graph metrics such as diameter considered. In real applications, there are often many different classes of vertices and/or edges, vertices may have 1000s of properties, and edges may have time-stamps in addition to properties.

Likewise, academic operations as found in graph kernels are relatively straightforward, with common ones as follows:

1) Compute vertex properties
2) Search vertex properties
3) Follow an edge to a neighbor
4) Determine a neighborhood
5) Find a path
6) Look at all paths
7) Compute global properties of graph
8) Identify subgraphs in a larger graph

Algorithms for benchmarking graph analytics are more complex:

1) Search for a/all vertices with a particular property or neighborhood
2) Explore the region around some number of vertices
3) Compute a new property for each vertex
4) Compute/output a list of vertices and/or edges
5) Compute/output a list of all subgraphs with certain properties

Streaming graph algorithms come in two forms: perform incremental targeted graph updates or answer a stream of independent local queries. The former typically refer to an incoming stream of edges and/or vertices that are incrementally added to or deleted from a large graph. In the latter, many streaming applications have for each stream input a specification of some vertex to search for, and an operation to perform to some property(ies) of that vertex, once found.

Either form may actually have several stages. First is the basic operation; next is a test of some sort that, if passed, may trigger larger computations.

Fig. 1 attempts to provide some comparison between existing kernels (the rows) and current benchmark suites that contain them. The rows in the table specify a variety of kernel operations.

The table has three sets of columns. First is a set of columns that indicate in what general category of graph operations the kernel is part of. Connectedness kernels (CCW, CCS, BFS) look for subsets of the graph whose edges satisfy some property. Path kernels (SSSP, APSP) are a variant of this, and look at aggregations of the properties of the edges that connect certain vertices. Centrality metrics (BC, CD, PR) look for the "most important" vertices in a graph, based on some combination of the vertex degrees, length of paths to other vertices, how often vertices are on paths between other vertices, or a derived "influence" of each vertex. Clustering kernels (CCO, Jaccard) look for the degree to which groups of vertices "cluster" together. Jaccard coefficients [10], [12], [21] are a growing subset of this where for pairs of vertices what is computed is the fraction of all neighboring vertices that the two vertices have in common. Graph contraction and partitioning kernels (CD, GC, GP) attempt to find higher level views of graphs where vertices are in fact subgraphs of the original graph. Subgraph isomorphism looks for subgraphs of certain shapes, of which triangle counting (GTC, TL) are the most well-known.

It should be clear that in terms of the above kernel classification, many of these kernels "overlap" in functionality.

Next is a series of columns that identify benchmark suites, taken from [12], [1], [2], [6], [5], [3], [4], [19], [8]. An "S" in an intersection indicates that the specified kernel is used in the associated benchmark in a "Streaming" mode; a "B" indicates a "Batch" mode.

Streaming forms of centrality metrics address questions such as "if edge e is added, how does it change its associated vertex centrality metrics, and does that cause a change in the "top n" vertices in terms of the metric." Streaming forms of triangle counting look to identify the change in either/both the associated vertices triangle count or the overall number of triangles in the graph.

Streaming Jaccard coefficient kernels may take both forms of streaming. On addition of an edge, a Jaccard kernel may ask what the graph modification does to the maximum Jaccard coefficient the two vertices may have with any other, or (less frequently) update all the coefficients with all other vertices (infrequent because of the near quadratic storage requirements to remember all coefficients). The second form of streaming for Jaccard may be a sequence of vertices, where for each provided vertex the kernel should return what other vertices have a non-zero Jaccard coefficient (perhaps greater than some threshold).

The third set of columns specify the type of graph modification and/or output that a kernel may form. An "X" indicates that a particular kernel may have the specified effect. The "Output O(*) Events" columns are particularly relevant to streaming where the local streaming process passes some threshold which causes reporting of an event. The $O(1)$ column specify that some fixed amount of data is generated from each event. The $O(|V|)$ column specifies that an output may grow in size proportional to the number of vertices in the graph. The $O(|V|^*)$ may generate data that grows far faster, but where typically only some "top k" values are chosen.

There are other benchmarking efforts that define problems requiring the composition of many of these kernels. VAST[9] is one such example that changes the problem each year, with recent years including both batch and streaming benchmark descriptions.

The key take-away from this table is that no one kernel is universal, and that there is a significant difference between streaming and batch kernels. This lays the rationale for the next section.

III. Canonical Graph Processing

While useful in an academic setting, the kernels and benchmarks discussed in the prior section are missing some major attributes if they are to be used to help understand the characteristics of systems that run real, more complex, graph applications. This section discusses a canonical processing flow that is perhaps closer to real practice, and demonstrates the potential interaction of multiple kernels with more realistic data structures. The goal for the discussion is that there is room for significant work in defining benchmarks that are more complex than any of the existing benchmark suites.

Fig. 2 diagrams a processing flow that includes both batch (middle and right side) and streaming graph processing (left side) in a more realistic combination.

Unlike many of the kernels discussed in the prior section, real applications start with large graphs built from not one but many classes of vertices and edges, often of multi-terabyte size, with thousands of properties per vertex, and time-stamps on edges. These graphs are persistent; their existence is independent of any single analytic or set of analytics. Today, these graphs are initially created via some large batch processing **dedup** processes [15], [17] that "clean up" multiple data sets by checking spelling, removing duplicates (*post-process deduping*), identifying faulty or missing values, and combining to create properties to be associated with vertices and edges. In a streaming form called *in-line deduping*, once established, updates will be from streams of incoming data. In Fig. 2 bulk undeduped data enters from the bottom, with the deduping process handled by a batch analytic before being used to build the initial graph.

Once a graph is established, real application analytics typically are not run over the entire graph. Instead, an analytic may start with identification of some **selection criteria** (on right) that are used to identify some initial **seed** entries. This may be as simple as specifying some particular vertex, or more involved such as scanning for the "top k" vertices with the highest values of some properties.

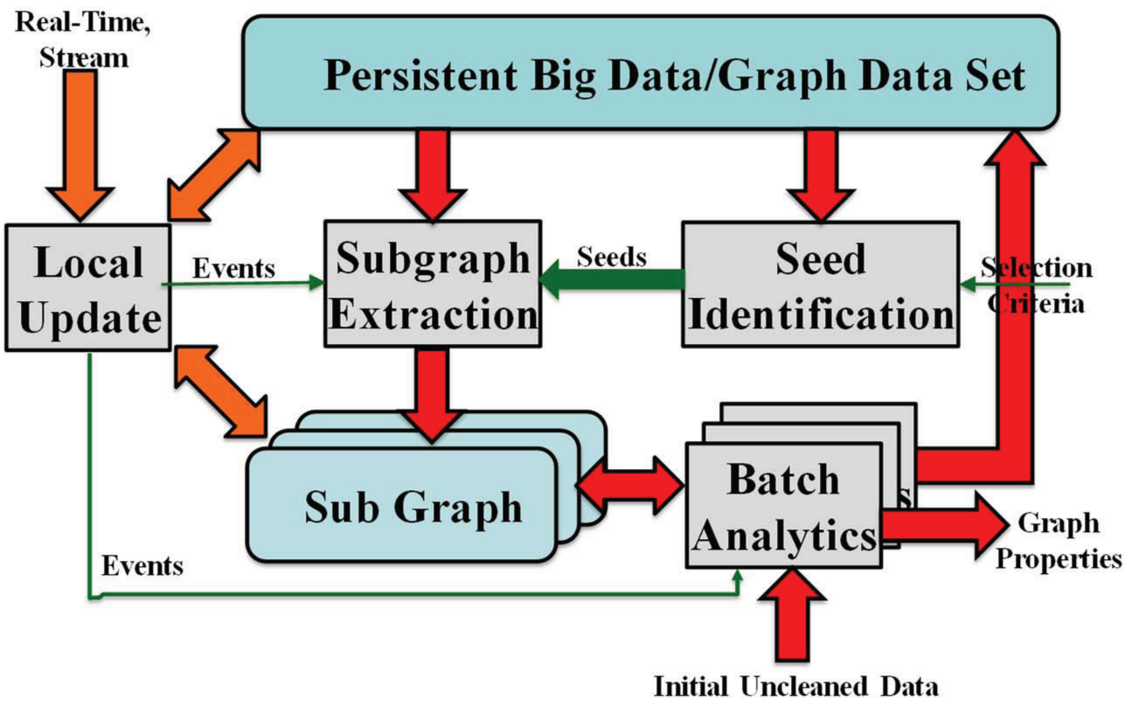


Figure 2. A Canonical Graph Processing Flow.

Once identified, these seeds may then be used to perform some sort of **subgraph extraction**, whereby a subset of the large graph is identified. A simple example of such a process may be a breadth-first search from individual seed vertices out to some depth, or perhaps out some distance from some path between two or more seeds. Once identified, it may be appropriate to physically copy such a subgraph out of the memory holding the large persistent graph into a smaller, but faster access rate, memory from which more complex analytics can be run. This copy process may also include some sort of projection to copy only a small subset of the properties.

Potentially long-running batch analytics may then be run on these subgraphs. Typical outputs may be metrics of the overall graph and/or even smaller subgraphs or vertex/edge sets ("neighborhoods"). Of growing importance, however, is the use of the analytic to compute/update properties of vertices or edges that are to be sent back to update the original persistent graph. This is in fact how many real-world applications end up with thousands of vertex properties, as analysts often find that some new property of vertices is useful, and it is easier to define a "one-time" analytic that computes this property for all vertices at once, and then use the property values in later repeated calls to application-specific analytics.

Stream processing (left-hand side) takes a different path. A real-time stream of data arrives incrementally. Processing may take a variety of forms. First, the inputs may specify specific vertices and some update to one or more of the vertex's properties. This is the case for the Firehose benchmark [1]. Alternatively, inputs may specify new edges (less frequently new vertices). Processing either one may involve checking if it is already in the graph and then either adding the edge or updating some properties associated with an existing edge. Less often are deletions. A good example of this is the in-line deduping process discussed earlier.

In both cases, the initial operations against the graph are relatively simple and rather local. However, it is not uncommon for the stream processing to look for changes in local or global graph parameters, and only if those parameters exceed some threshold, to use the modified vertices/edges as seeds into a subgraph extraction process similar to that described for the batch process. The extracted subgraph may then be the target of a specific batch analytic to more fully analyze the effects of the change. As before, the execution of this analytic may result in either alerts back to some external system and/or updates to properties in the larger graph.

An example of a matching real-world application can be found in [23]. A 2012 batch-only implementation of an insurance application problem periodically combined and cleaned 40+ TB of public data into a persistent database of 4-7+ TB (bigger now). Once a week this data set is "boiled" (over the weekend) to pre-compute answers to a set of queries, in two forms and for each of all multi-million people in the set. The first form is a simple indexed database where simple SQL-like selects can quickly retrieve data relevant to a particular insurance applicant, such as credit score. The more valuable answers are, however, the result of searches for "relationships" between people, such as "who has shared an address with what other individuals 2 or more times, especially if they have shared a common last name." These latter computations represent the bulk of the weekly computations, and are close to the Jaccard coefficient kernel mentioned earlier, and have been called **Non-Obvious Relationship Analysis** (**NORA**).

Such a resulting data set is then used, for example, to provide real-time data back to an insurance company when a potential client has entered data for a quote. This data includes both the specific client data and the results of NORA relationships involving the client, and is used as input to the company's quoting process.

A streaming real-time version of such applications would have both types of streaming as discussed above. One stream would be updates to the persistent graph, and determine for each updated vertex if the update is likely to change any of the key relationships. Simply adding more validity to a pre-identified relationship needs no more processing, but when there is the potential for crossing some threshold, a more complete computation of the particular metric may be warranted. Such a process makes the data set less stale.

The second type of streaming would take a sequence of applicants and compute in real-time whatever relationships are relevant for the type of application they are applying for. This has the significant advantage of removing much of the need for the pre-computation that takes so much time. It also increases the fidelity of the responses, as the results include updates since the last pre-computation.

IV. Using Today's Architectures

There are published results for several of the kernels discussed earlier, with perhaps the most exhaustive the twice-yearly reports on several hundred implementations on many different systems of the Breadth First Kernel used in the GRAPH500 benchmark[2]. Other than for BFS [20], however, very few of these results have been analyzed for how different architectural and system characteristics affect performance, especially for large systems with multi-kernel applications.

The NORA application mentioned earlier is a partial exception. In [23], a model of the multi-step algorithm was built to estimate four different system parameters as a function of problem size: required compute cycles, disk bandwidth, network bandwidth, and memory access rate. Several different system configurations were studied:

- The 2012 baseline consisted of 10 racks totalling 400 2012 dual-socket 6-core 2.4GHz Intel Xeon® server blades, each with the equivalent of local disks with about 0.16GB/s bandwidth, 96 GB of DRAM, and a network port with an injection bandwidth of about 0.1 GB/s.
- An upgrade to the servers to reflect a more modern design, with options of more cores (24) at a higher clock rate (3GHz), 3X more memory bandwidth, SSDs or RAMdisks, and Infiniband links with up to 24 GB/s injection bandwidth.
- A system based on "lightweight" processors such as ARMs rather than high end server processors, similar to systems such as HPE's Moonshot line [18].
- A system based on an architecture initially postulated by Sandia National Labs [26] called "X-caliber" that resembled the current generation of Intel Knight's Landing chips where a two-level memory systems has close-in memory provided by 3D memory stacks with very significant increases in memory bandwidth. Such two-level memories may very naturally support the two levels of graphs pictured in Fig. 1.
- A system like X-Caliber where all the processing is moved to the bottom of the memory stacks, and both DRAM and non-volatile memory are included in each stack, again supporting the two levels of graphs pictured in Fig. 1. Both separate central processing chips and network interface chips are deleted. Such resulting systems would be "seas" of just memory stacks.

The last architecture is definitely not one of "today's" architectures, but is included for reference as to what might be possible.

Fig. 3 provides an update to the 2013 model results using variations of the above configurations. In each graph, each bar represents usage of one of the four key resources during one of the steps in the application: network bandwidth, disk bandwidth, memory bandwidth, and instruction processing rate. At each step the highest bar represents the bounding execution time for that step. The total time is computed from these peaks.

As can be seen, although disk and network bandwidth represent the tall poles for the baseline system. However, no one type of resource is uniformly the bounding peak for all steps. In fact, updating from the 2012 baseline one resource at a time provides relatively small benefit. For example upgrading the microprocessor alone provided only a 45% increase in performance, with the other options individually providing less. As shown, however, upgrading all but the microprocessor provides over a 3X growth in performance (far more than the product of the individual factors). Then, in addition to all of the above, upgrading the microprocessor did provide an 8X growth in performance for the same 10 racks.

The Lightweight projection, with its entirely different mix of resources, has a strikingly different profile, as its lower processing capability causes computational rate to dominate for 4 of the 9 steps. Even so, it provides near equal performance in 1/5'th of the hardware (2 racks rather

[2]www.graph500.org

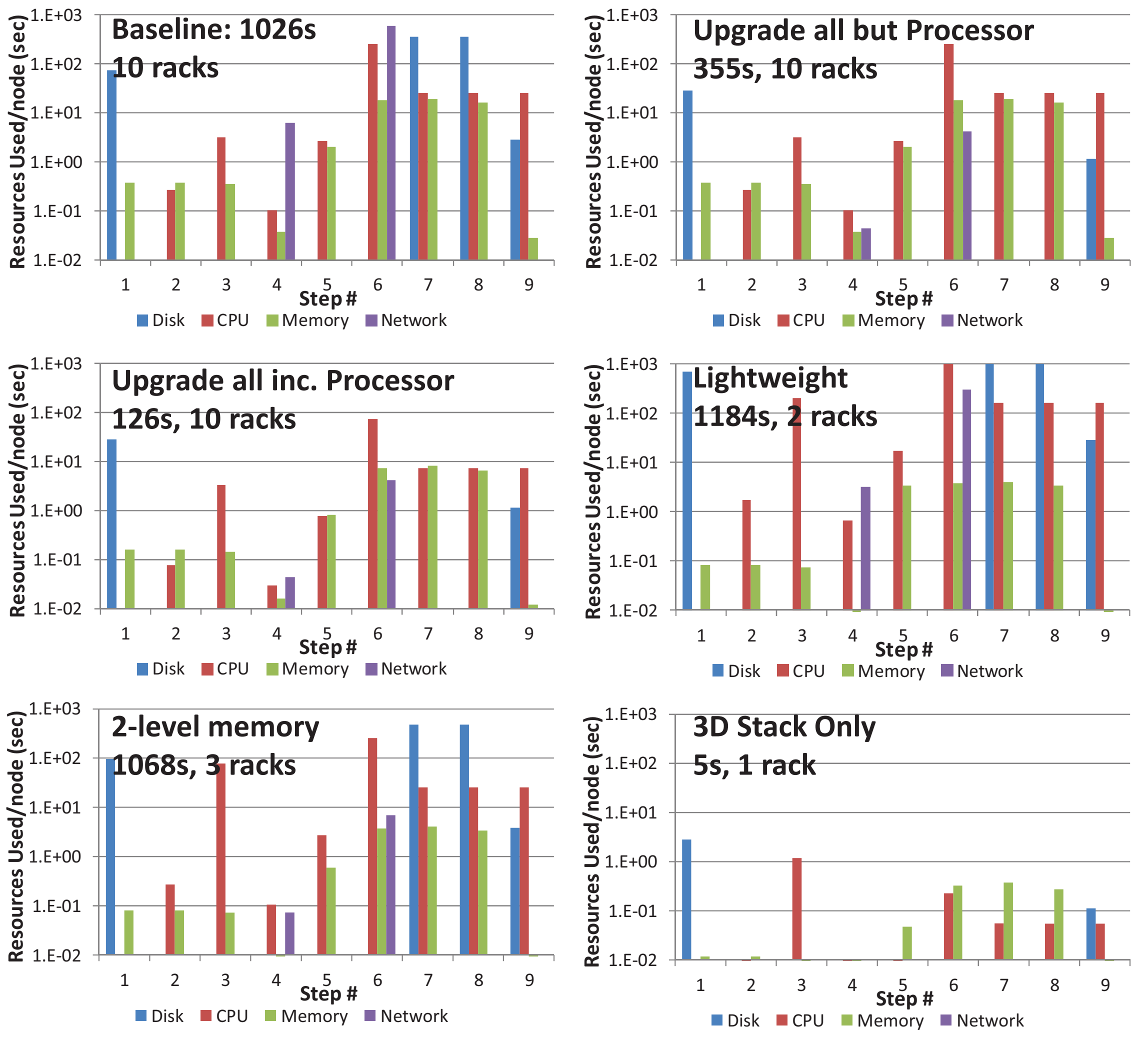


Figure 3. Performance Modeling of NORA Problem.

than 10).

The two-level memory system also has yet a different profile, achieving equal performance in only 3 racks.

The 3D stack-only system doesn't resemble any existing architecture but provides possibly up to 200X performance in 1/10th the hardware.

V. Emerging Architectures

There are at least two interesting emerging architectures that are radically different from today, but both have unique attributes suitable for performing the graph analytics such as proposed in Fig. 2.

A. *A Sparse Linear-Algebra-based Architecture*

Fig. 4 depicts the microarchitecture of a single accelerator node of a machine [27], [28] designed explicitly for performing the kinds of linear algebra-based kernels compatible with many basic graph algorithms[19], where graphs are expressed as boolean adjacency matrices[3]. The architecture particularly focuses on sparse to very sparse situations where most of the matrix entries are 0. Multiple of these nodes are combined in up to a 3D topology under the control of a conventional host processor.

The dotted and dashed lines in Fig. 4 represent two streams of matrix component references that start with address generation of multiple sparse vectors, proceed through a memory designed to support irregular accesses, then through a sorter to align the individual components from pairs of sparse vectors that are both non-zero, go through an ALU to perform multiply-accumulates, and then go back into memory (solid line - again in a sparse format). Condensed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats are "hardwired" into the architecture.

[3]In a typical adjacency matrix the (i,j)th element is 1 if there is an edge from vertex j to vertex i.

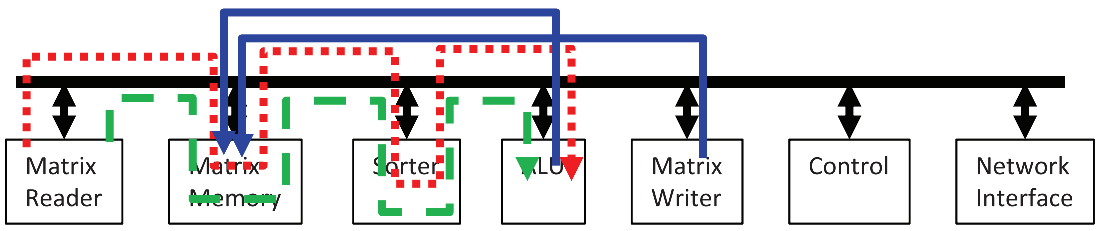


Figure 4. A novel Sparse Graph Processor.

Measured data on a prototype 8-node FPGA-based system for sparse matrix-matrix multiplies indicate that there is perhaps more than an order of magnitude performance advantage over a node for a Cray XT4, and 4 racks of such nodes would exceed 10X a rack of a Cray XK7 (the system used in the Titan supercomputer[4]). Performance per watt, even for the FPGA implementation is even more striking. Projections to ASIC-based designs imply a possibility of another order of magnitude advantage in both metrics.

This architecture seems excellent for accelerating batch analytics where the kernel operations can be expressed in linear algebra as discussed in Section II. No data is available on combinations of kernels, or how streaming might be supported.

B. A Migrating Thread-based Architecture

Fig. 5 diagrams a second architecture[16] that takes perhaps the opposite direction by focusing on fast edge-following. This scalable system implements a single very large shared memory domain out of an interconnection of multiple nodes, each of which contains a large number of **nodelets**. Each nodelet is built around a separate memory channel, and has a collection of heavily multi-threaded **Gossamer Cores** (GC) that drive it. A **mobile thread** executes within some GC until it makes a memory reference to a location not in the current nodelet. In such cases, the GC hardware suspends the thread, packages up its internal state, and sends it over the system's internal network to the correct nodelet within the correct node that owns the targeted memory location. Here the state is unpackaged and given to any of the target nodelet's GCs that have room. When the thread is restarted, it has no knowledge that it had moved, other than the memory access it was attempting is now local. The thread then executes locally until it attempts yet another non-local memory access, at which point it is again suspended and shipped to the correct target.

The net result is that all memory references are local, regardless of how big the system is. This locality also means that very simple hardware within the memory controller of each nodelet can execute a rich suite of atomic memory operations (AMO) that occur very quickly, permitting simple and efficient synchronization operations.

The current hardware has a conventional microprocessor embedded in each 8-nodelet node. This processor executes a conventional Linux and initiates the original parent thread into the memory system. The programming tool chain is a variation of Cilk[11].

With this type of functionality, operations such as "pointer-chasing with atomic updates to list elements" become trivial to write, and more importantly, consume half or less the bandwidth and latency of a conventional thread trying to do the same thing via remote memory operations or, even worse, message passing.

In addition, where a programmer knows that the number of operations to be applied to some address is limited, instructions may be invoked that launch tiny single-function threads to perform single operations at a target location. This is useful for performing such things as random updates into a very large table.

In addition to mobility, a thread may also spawn a child thread with as little as a single instruction. The child thread is then free to travel as its program sees fit, independent of the parent.

The current production system supports 8 nodes, each with 8 nodelets in a deskside box. Each nodelet has 4 GCs, each capable of holding 64 concurrent threads. Expansions to rack-scale systems are in design. Expected future implementations will implement each node as first an ASIC and then a 3D memory stack having dozens of nodelets in the same package. This latter configuration is similar to the 3D stack configuration mentioned in the prior section.

As with the sparse matrix machine, orders of magnitude improvement over conventional systems appear reachable. Measured results from real hardware should be available shortly. However, an extension of the modeling exercise discussed in the prior section on the NORA problem lead to projections pictured in Fig. 6, where 3 generations of this architecture[5] are compared to the conventional upgrades from the prior section. In 1/10th the hardware, projected

[4]https://www.olcf.ornl.gov/titan/

[5]Emu1 is the current design extended to rack-size; Emu2 uses an ASIC in place of the FGPA; Emu3 is a 3D stack utilizing the Emu architecture for the base logic chip

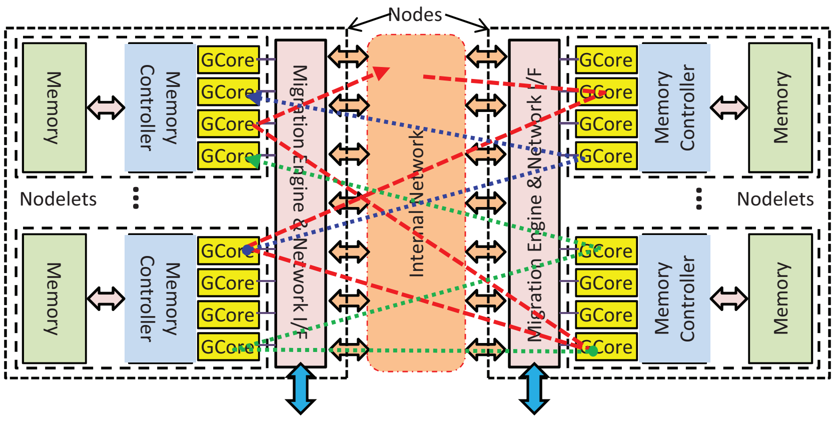


Figure 5. Emu's Migrating Thread Architecture.

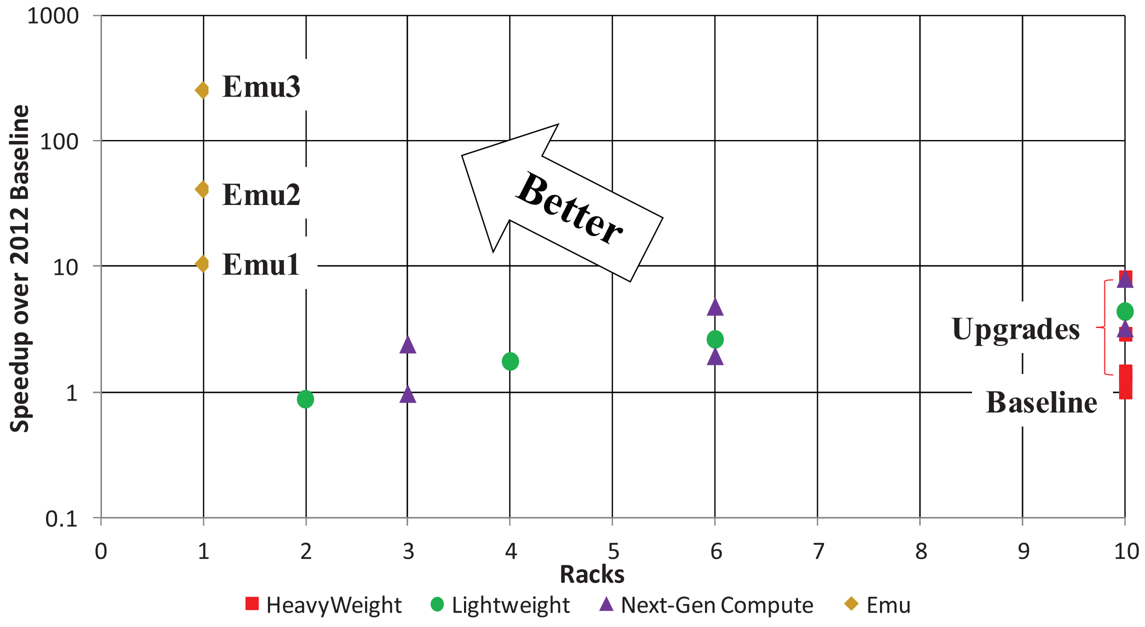


Figure 6. Size-Performance Comparison for the NORA problem.

performance for the Emu system are up to 60X that of the best of the upgraded clusters.

Also very preliminary studies of streaming queries for Jaccard-like problems indicate that individual response times in the 10s' of microseconds are possible, with throughputs that are large multiples of what can be achieved with conventional systems.

This architecture can support both batch and, in particular, streaming applications with complex classes of vertices and edges, and substantial sets of properties for each.

VI. Conclusions

This paper has overviewed a range of existing graph kernels and benchmarks for batch mode applications, and discussed their extension to streaming modes where data arrives incrementally. A general process flow was introduced to begin a discussion of a more realistic integration of multiple analytics of both batch and streaming form. Several possible forms of streaming were identified. A variety of system architectures were compared, and several identified that offer the potential of significant gain over today's systems, especially as 3D memory stacks that provide huge bandwidths. Two emerging architectures in particular have the potential for very large gains, with the interesting observation that they employ almost opposite execution models. One performs graph operations after translation into sparse matrix operations; another is based around "pointer chasing" where the computational state moves with a pointer dereference.

An important next step would be to develop a multi-kernel benchmark that mirrors Fig. 2, especially in the combined batch and streaming mode. An early parameterized model, similar to that used for the NORA problem, could identify the most potentially valuable configurations. A potentially interesting option might be to combine both the emerging models into a single system.

In addition to this, a reference implementation, with explicit instrumentation, of a combined benchmark would allow calibration of the model. Kernels extracted from this implementation can then be provided as extensions to existing kernels.

ACKNOWLEDGMENT

This work was supported in part by the Univ. of Notre Dame, and in part by NSF under grant CCF-1642280.

REFERENCES

[1] Firehose benchmarks. http://firehose.sandia.gov/.

[2] Graph 500. http://www.graph500.org/.

[3] Graph Algorithm Platform. http://gap.cs.berkeley.edu/.

[4] Graph analysis benchmark suite. http://graphanalysis.org/benchmark/index.html.

[5] Graph challenge. http://graphchallenge.org/.

[6] GraphBLAS. http://www.graphblas.org/home/.

[7] Sparql query language for rdf. https://www.w3.org/TR/rdf-sparql-query/.

[8] Stinger. https://trac.research.cc.gatech.edu/graphs/wiki/STINGER.

[9] VAST. http://vacommunity.org/HomePage.

[10] J. Bank and B. Cole. Calculating the Jaccard Similarity Coefficient with Map Reduce for Entity Pairs in Wikipedia. Wikipedia Similarity Team, Dec. 16 2008.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[12] P. Burkhardt. Asking Hard Graph Questions: Beyond Watson: Predictive Analytics and Big Data. Beyond Watson Workshop, Feb. 2014.

[13] K. S. Candan, H. Liu, and R. Suvarna. Resource description framework: Metadata and its applications. *SIGKDD Explor. Newsl.*, 3(1):6–19, July 2001.

[14] D. Chavarria-Miranda, V. G. Castellana, A. Morari, D. Haglin, and J. Feo. Graql: A query language for high-performance attributed graph databases. *Workshop on Parallel and Distributed Computing for Large Scale Machine Learning and Big Data Analytics (ParLearning'2016)*, May 27, 2016.

[15] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 24(9):1537–1555, Sept 2012.

[16] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein. Highly scalable near memory processing with migrating threads on the emu system architecture. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '16, pages 2–9, Piscataway, NJ, USA, 2016. IEEE Press.

[17] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, Jan 2007.

[18] Hewlett Packard Enterprises. HPE Moonshot System. https://www.hpe.com/h20195/v2/getpdf.aspx/4AA4-6076ENW.pdf.

[19] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[20] P. Kogge. Performance analysis of a large memory application on multiple architectures. In *7th Int. Conference on PGAS Programming Models*, 2013.

[21] P. Kogge. Jaccard coefficients as a potential graph benchmark. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 00:921–928, 2016.

[22] P. Kogge. Streaming graph analytics: Complexity, scalability, and architectures. Keynote presentation: Chesapeake Large Scale Analytics Conf., Oct. 2016.

[23] P. Kogge and D. Bayliss. Comparative performance analysis of a big data nora problem on a variety of architectures. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pages 22–34, 2013.

[24] LexisNexis Risk Solutions. Kel language reference version 5.4.2, 2015.

[25] 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.

[26] Sandia National Labs. Darpa selects sandia national laboratories to design new supercomputer prototype. https://share-ng.sandia.gov/news/resources/news_releases/supercomputer-prototype/#.WMg9mmfaupo, August 2010.

[27] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner. Novel graph processor architecture, prototype systems and results. *HPEC COnference*, 2016.

[28] W. S. Song, J. Kepner, V. Gleyzer, H. T. Nguyen, and J. I. Kramer. Novel graph processor architecture. *LINCOLN LABORATORY JOURNAL*, 20(1):92–104, 2013.

[29] J. Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 217–218, New York, NY, USA, 2012. ACM.