STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation

David A. Bader
Georgia Institute of Technolgy

Jonathan Berry
Sandia National Laboratories

Adam Amos-Binks
Carleton University, Canada

Daniel Chavarría-Miranda
Pacific Northwest National Laboratory

Charles Hastings
Hayden Software Consulting, Inc.

Kamesh Madduri
Lawrence Berkeley National Laboratory

Steven C. Poulos
U.S. Department of Defense

May 8, 2009

Abstract

In this document, we propose a dynamic graph data structure that can serve as a common data structure for multiple real-world applications. The extensible representation for dynamic complex networks is space-efficient, allows parallelism over vertices and edges independently, and can be used for efficient checkpoint/restart of the data.

1 Motivation

A graph is a set of vertices and edges and has proved to be a useful abstraction for solving real-world problems across many domains such as transportation, biological sciences, electrical engineering, and social networks. The abstract data structures to date for representing graphs primarily focus on static graph instances, such as the adjacency matrix for dense graphs and adjacency lists for sparse graphs.

Many problems today can be formulated as dynamic spatio-temporal graph problems. For example, one may wish to track communities within social networks on Facebook as edges (friendship pairs) are added or removed. Or more interestingly, one may look for people who bridge between different social communities, or switch allegiances over time.

As the computer science community increases its development of algorithms and codes for large-scale graph problems, no canonical graph representation has yet to emerge. Without a standard graph representation, algorithms that are implemented for one framework may require substantial programming efforts to port to a different framework. Even worse, algorithms within a single framework may use different data structures for each graph kernel, requiring costly data transformations between each graph kernel subroutine. These inefficiencies in time, space, and productivity, could be reduced or eliminated through a canonical graph representation.

Several efforts are focused on developing parallel graph libraries/frameworks for the Cray XMT, a special-purpose parallel computer system that contains architectural features that make it one an ideal system for large-scale graph processing. For instance, Pacific Northwest National Laboratory (PNNL) operates the Center for Adaptive Supercomputing Software for Multithreaded Architectures (CASS-MT), Sandia National Laboratories' Jon Berry is developing the MultiThreaded Graph Library (MTGL), and Andrew Lumsdaine at Indiana University is developing a parallel version of the Boost Graph Library. In addition, my Spatio-Temporal Interaction Networks and Graphs (STING) research group at Georgia Tech along with Kamesh Madduri at Lawrence Berkeley National Laboratory continue our development of native XMT graph algorithms, and the open source Social Network Analysis in Parallel (SNAP) graph framework.

While some open research has experimented with data structure than can handle insertions and deletions efficiently (see Madduri/Bader 2009), the community would be helped by a more general dynamic attribute graph data structure (STINGER) with the following objectives:

1. Portability

   (a) Algorithms written for STINGER can easily be translated/ported between multiple languages and frameworks (e.g. XMT C, MTGL, PBGL).

   (b) Optimizations made on algorithms using STINGER in one framework should give reasonable confidence that these same optimizations will have the same trends when applied to algorithms using STINGER on other frameworks.

2. Productivity

   (a) STINGER should provide a common abstract data structure such that the large graph community can quickly leverage each others' research developments. This is similar in philosophy to the numerical algorithms community implicit use of sparse and dense matrices.

3. Performance

   (a) It is recognized that no single data structure is optimal for every graph algorithm. The objective of STINGER is to configure a sensible data structure that can run

most algorithms well. There should be no significant performance reduction for using STINGER when compared with another general data structure across a broad set of typical graph algorithms.

(b) STINGER should assume a shared memory address space, and allow both sequential or parallel algorithms. The data structure should allow parallel algorithms to exploit concurrency where possible.

2 Description of the data structure and its basic operations

2.1 STINGER Data Structure

Figure 1 contains a diagram of the STINGER extensible data structure.

The STINGER data structure permits the operations of query, insert, and delete, on both vertices and edges. Algorithms use the data structure in a read-only fashion so that multiple algorithms may run concurrently. An algorithm uses its own separate data structures for storing auxiliary information required by the algorithm. In addition, asynchronous processes may concurrently insert and delete vertices and edges from the STINGER data structure. It is assumed that timestamps are used to avoid any insertion and deletion events from conflicting with the running algorithms. Algorithms may be active or passive. A passive algorithm may be posed in terms of static queries such as finding the shortest path between two vertices during a timestamp interval. An active algorithm is a continually running process, such as alerting the user when the shortest path between two vertices significantly decreases, or computing the current distribution of sizes of connected components in the graph.

A structured Logical Vertex Array resides in a contiguous block of memory and holds vertex records in a semi-dense fashion.

Each physical vertex id (a 64-bit integer) maps one-to-one to an entry in the Logical Vertex Array. The logical vertex id is a value between 0 and $N-1$, where N is approximately the size (number of vertices) in the in-memory graph. Each record in the Logical Vertex Array contains the physical vertex id, a vertex type ($VType$), a vertex weight, the in-degree of the vertex, the out-degree of the vertex, and a pointer to the first edge block. A nonnegative physical vertex id in an logcal vertex record indicates that the logical vertex record is valid; otherwise, a negative entry indicates an invalid record. The implementation of the physical to logical vertex id mapper is not specified, as long as the correct synchronizations are put in place to allow for multiple readers and writers.

The logical vertex id may not be persistent. For instance, a garbage collection routine is allowed to compact the Logical Vertex Array as long as a change to a logical vertex id is atomically updated in its instance in the physical to logical vertex id mapping data structure as well as all instances in edge records. Also, it is permissible for logical vertex ids to change when an in-memory graph is checkpointed to disk and/or restarted in memory from the disk

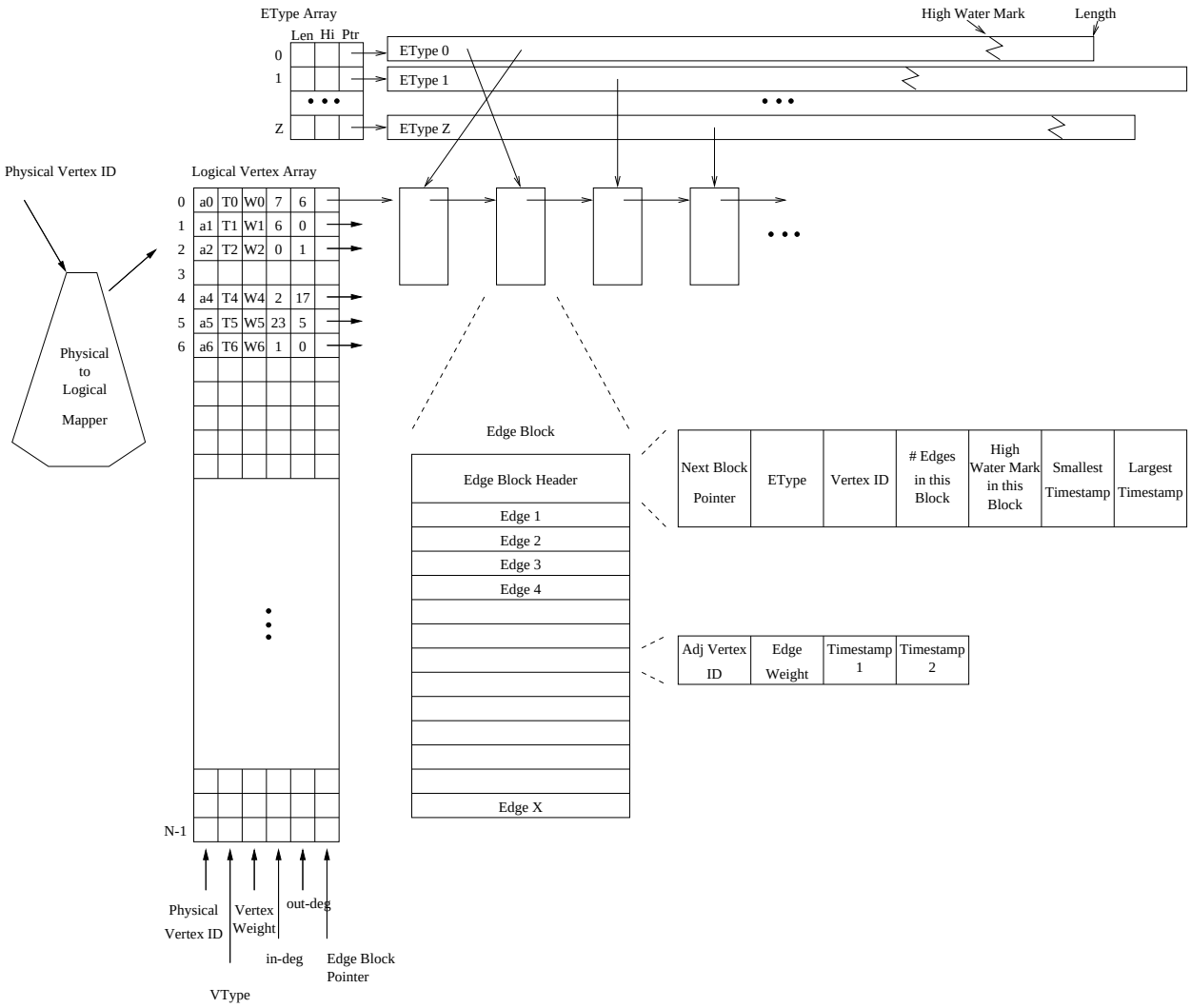

Figure 1: A diagram of the STINGER data structure.

checkpoint file.

Edges are held in semi-dense structured *edge blocks*. An edge block contains a header with a pointer to the next edge block, an edge type (*EType*), the source logical vertex id, the number of valid edge records, the high water mark (highest index of any valid edge record in this block), the smallest timestamp, and the largest timestamp, in this edge block. The *EType* in the edge block is a 64-bit integer, although *ETypes* are expected to fall within in a small range. Only edges that share the same *EType* can be held within the same edge block. When traversing the list of edge blocks, the STINGER representation requires that the list of edge blocks be arranged in monotonically increasing order by *EType*. For simplicity of implementation, each edge block may hold a fixed number X of edge records; for instance $X = 1024$. We recognize that it would be possible to use multiple block sizes for each type in order to conserve space, increase concurrency, and/or improve efficiency, and suggest that more sophisticated memory management for these blocks is left to implementation. Edge records within an edge block are in arbitrary order. A nonnegative adjacent vertex id in an edge record indicates that the edge record is valid; otherwise, a negative entry indicates an invalid record. An edge record contains a 64-bit logical vertex id, a 64-bit integer edge weight, and two 64-bit integer timestamps. The two edge timestamps hold the first time and the most recent time, respectively, that an edge is added to the graph.

STINGER uses an indexing structure to provide a more efficient mechanism for locating all edges of a certain *EType*. An EType Array is indexed by each *EType*, and each entry contains a header with *Length* and *High Water Mark* and a pointer to an EType Block. Each EType Block has *Length* entries, and *High Water Mark* is the highest index of any valid entry in the EType Block. The EType Array enables the ability to concurrently search all edges (or all edges of particular *ETypes*) without the need for traversing each linked list. When a new Edge Block is allocated, a pointer to it is added to the appropriate EType Block. A null pointer represents an invalid entry in an EType Block.

2.2 The Physical-to-Logical Vertex Id Mapper

The choice of implementation for the physical to locical vertex id mapping is left to the implementor, as long as the correct synchronizations are put in place to allow for multiple readers and writers. The mapping must be a one-to-one function. The mapper should permit efficient implementations of insert, search, and delete, of (key, value) pairs, where the physical vertex id is the key and the logical vertex id is the value.

2.3 STINGER Basic Operations

The STINGER dynamic graph representation allows for read-only queries of vertices and edges, insertion and deletion both vertices and edges, aging off entities by timestamp, and checkpoint/restart. Each dynamic operation is described next.

2.3.1 Inserting a new vertex

We must insert a new logical vertex into the STINGER data structure when a physical vertex id is searched and the mapper fails to return a corresponding logical vertex id. The operation proceeds as follows. First, a logical vertex id must be determined, and it may reuse an available logical vertex id (an entry in the Logical Vertex Array with a negative value in the physical vertex id field) or use a new entry at the end of the array. The Logical Vertex Array entry enter the physical vertex id (implicitly making this record in the Logical Vertex Array valid), and the fields such as *VType*, vertex weight, in-degree, and out-degree, set to the appropriate values for this vertex. The Edge Block Pointer for this vertex is initialized as a null pointer. At any time, the Logical Vertex Array may be grown by reallocating it to a larger array. The new (physical id, logical id) key-value pair is inserted into the mapper.

2.3.2 Deleting an existing vertex

Once the logical vertex id is found through a search of the mapper, the appropriate vertex record of the Logical Vertex Array is removed as follows. First, each edge from this logical vertex is examined and the degrees of the adjacent vertices are updated accordingly. Second, the linked list of edge blocks can be freed from memory, and the pointers to them from the EType Blocks set to null pointers. Third, all edges that point to this logical vertex must be located and deleted. The EType Array can be used to efficiently search through the edge blocks for the edges that must be deleted. Fourth, the Logical Vertex Array entry is cleansed by setting the physical vertex id to a negative value to invalidate the record, and zeroing the in- and out-degrees. Finally, the physical-to-logical key is deleted from the mapper.

2.3.3 Inserting a new edge

Each physical vertex is searched in the mapper. If either or both searches fail, the procedure above is run to insert each new vertex into the STINGER graph representation. The appropriate degree counts are incremented on the *from* and *to* vertices. The edge is added to the appropriate edge records for the two vertices and edge block headers are updated. Edge records can be added to the block whenever space remains. (If possible, when multiple blocks have slack, the block that has the maximum *smallest* timestamp should be used.) Should all of the existing edge blocks for that vertex and *EType* be filled, then a new edge block will be added to the linked list, and a pointer to it will be added to the appropriate EType Block.

2.3.4 Deleting an existing edge

Each physical vertex is searched in the mapper. If both logical vertex ids are not returned, then the edge is ignored. Otherwise, the edge is deleted from the appropriate edge block by setting the appropriate adjacent vertex id to a negative value. Each edge block can have slack (that is, available space for additional edge records). However, an implementation can determine an appropriate threshold for compacting edge blocks of a given *EType*. If the

threshold is triggered and additional edge blocks of the same *EType* for this logical vertex exist, then a garbage collection procedure should try to compact the edge blocks into a fewer number of edge blocks, if possible. An empty edge block should be released from memory and the pointer to it from the EType Block should be set to a null pointer.

2.3.5 Aging-off

The STINGER data structure should allow for an independent garbage collection process that ages-off all edges with timestamps less than a given value. Edges are deleted using the procedure described above. During the aging-off process, a vertex with both its in-degree and out-degree of zero should be deleted from the STINGER graph using the procedure above to delete an existing vertex. In this case, the procedure can be shortcut, because the logical vertex's entry in the Local Vertex Array will already have in- and out-degrees set to zero, and a null pointer for its edge block pointer. Hence, no searching is necessary through all of the edge blocks (such as was described by using the EType Array) is necessary.

2.3.6 Checkpoint/Restart

Checkpointing refers to halting all graph operations and saving the STINGER graph data structure from memory to disk. Restarting refers to loading the graph data structure from disk into memory, and restarting operations. It is allowable to compact the graph on a checkpoint or restart. On a checkpoint, the Logical Vertex Array and all Edge Blocks are stored to disk. The physical-to-logical mapper and the EType Array and EType Blocks are discarded during the checkpoint, and recreated on a restart. Each edge block contains its owner logical vertex id and EType so that the linked list of edge blocks for each logical vertex id can be re-strung together, and the EType Array and EType Blocks recreated.