MSSG: A Framework for Massive-Scale Semantic Graphs*

Timothy D. R. Hartley†, Umit Catalyurek‡†, Füsun Özgüner†
Andy Yoo+, Scott Kohn+, Keith Henderson+
†Dept. of Electrical and Computer Engineering, The Ohio State University
‡Dept. of Biomedical Informatics, The Ohio State University
+Center for Applied Scientific Computing, Lawrence Livermore National Laboratory

Abstract

This paper presents a middleware framework for storing, accessing and analyzing massive-scale semantic graphs. The framework, MSSG*, targets scale-free semantic graphs with* $O(10^{12})$ *(trillion) vertices and edges. Here, we present the overall architectural design of the framework, as well as a prototype implementation for cluster architectures. The sheer size of these massive-scale semantic graphs prohibits storing the entire graph in memory even on medium- to large-scale parallel architectures. We therefore propose a new graph database,* grDB*, for the efficient storage and retrieval of large scale-free semantic graphs on secondary storage. This new database supports the efficient and scalable execution of parallel out-of-core graph algorithms which are essential for analyzing semantic graphs of massive size. We have also developed a parallel out-of-core breadth-first search algorithm for performance study. To the best of our knowledge, it is the first of such algorithms reported in the literature. Experimental evaluations on large real-world semantic graphs show that the MSSG framework scales well, and grDB outperforms widely used open-source out-of-core databases, such as BerkeleyDB and MySQL, in the storage and retrieval of scale-free graphs.*

1 Introduction

Graphs have been used to model many interaction networks, ranging from the biological to the computational sciences. Some examples are metabolic and signaling pathways, gene regulatory networks, protein interaction networks, taxonomies of proteins and chemical compounds, and social networks [22, 23, 28, 30, 34, 35]. These types of real-world graphs are known as *semantic graphs* [18]. In a semantic graph, vertices represent certain concepts or objects and edges represent the relationships between them. Further, the vertices and edges in the semantic graph are associated with some meaningful types. These vertex and edge types form an ontology graph, which summarizes the semantic information the corresponding semantic graph carries.

While real-world semantic graphs are typically large, new graphs in some emerging fields are expected to have truly massive numbers of vertices and edges. For example, Kolda et al. [22] predict semantic graphs representing social networks of interest to the Department of Homeland Security will have 10^{15} entities. Analyzing such large graphs and answering user queries within a reasonable amount of time is therefore an important and very challenging problem.

First, a set of novel graph algorithms are needed in order to analyze these huge data sets. In order to process given user queries in a timely manner, these graph algorithms must explore large semantic graphs in parallel. More importantly, they must be out-of-core (OOC) algorithms that read and process input graphs that are stored in persistent storage, as the memory requirements of any massive semantic graph are prohibitively large. Although in the literature there are various parallel graph algorithms [8, 12, 13, 17, 20, 21, 35], and OOC (also called external memory) algorithms [2,3,14,16,24,26], to the best of our knowledge, there is no work that combines both.

Second, traditional relational database systems have been used as data managers to store the input graph data. Although they provide enough performance for business applications such as transaction processing, traditional relational databases are not ideal platforms for storing and processing massive semantic graphs. The same flexibility which makes relational databases appropriate for a wide variety of divergent applications causes them to perform poorly when faced with such strict data and speed requirements as are imposed when dealing with large semantic graphs. Therefore, a new data management system is needed. This data manager must be able to store streaming semantic graphs of massive size and provide an underly-

*This research was supported in part by UC Subcontract #B555676 and the National Science Foundation under Grant #CNS-0403342.

ing infrastructure to allow the parallel graph algorithms to access and process the stored graph in a scalable and cost-effective way.

In this paper, we present a middleware framework for storing, accessing and analyzing massive-scale semantic graphs. The framework, *Massive-Scale Semantic Graphs (MSSG)*, targets scale-free semantic graphs with $O(10^{12})$ (trillion) vertices and edges. In a scale-free graph, most of the vertices are only connected to a small number of other vertices (i.e., they have low degree), while a few vertices, known as *hubs*, are connected to a large number of other vertices. Most of the real-world semantic graphs exhibit this topological property.

The sheer size of these massive-scale semantic graphs prohibits storing the entire graph in memory even on medium- to large-scale parallel architectures; hence, these graphs will need to be persistently stored in a distributed database. The framework is architected targeting large clusters with compute nodes that have direct access to fast disk storage. One of the many possible configurations is compute nodes with large local disks. Another configuration is compute nodes that are connected to fast storage arrays via Storage Area Network switches. An example of the latter is Ohio Supercomputing Center's Mass Storage system [11].

We propose a new graph database, *grDB*, for storing and processing large scale-free semantic graphs on secondary storage. The grDB database stores the vertices and edges of a semantic graph in such a way that the number of disk I/Os required to access adjacent vertices is minimized while still efficiently utilizing the storage space.

We have developed a prototype implementation of MSSG. This prototype has been built on top of DataCutter [9, 10], which uses MPI as its low-level communication protocol. Experimental evaluations with both real-world scale-free graphs and synthetic scale-free graphs show that our prototype scales well. We have also compared grDB against other widely used open-source databases, BerkeleyDB [31] and MySQL [25], as well as in-memory graph storage implementations (for small graphs). These experiments showed the effectiveness of the proposed graph database for scale-free graphs.

The main contributions of this research are summarized as follows.

- We have architected a framework, MSSG, to store, retrieve, and process massive-scale semantic graphs.
- We have developed a novel data management system called grDB. Unlike traditional relational databases, grDB is optimized specifically for efficiently storing and accessing semantic graphs of massive size.
- We have developed a parallel OOC breadth-first search algorithm that runs on distributed parallel machines. To the best of our knowledge, this is the first of such algorithms reported in the literature.
- We have evaluated the performance of MSSG using large real-world semantic graphs. The results show that the MSSG framework scales well, and that the proposed graph database, grDB, outperforms other open-source database systems, Berkeley DB and MySQL.

The remainder of the paper is organized as follows. The architecture of the proposed MSSG framework and its implementation are described in detail in Sections 2 and 3. Experimental results are presented in Section 4, followed by concluding remarks in Section 5.

2 System Architecture

The MSSG framework is designed to provide storage, retrieval and processing of large scale-free graphs. It consists of one or more front-end nodes which provide an entry point for the user queries as well as graph data ingestion, and a set of back-end nodes that are responsible for storing and processing the graph data (Figure 1). MSSG has been built on top of DataCutter [9, 10] and its functionality is provided by a set of modular, customizable services implemented as DataCutter components and pluggable interfaces. The *Ingestion Service* provides an entry point for data storage and it is responsible for clustering and declustering of the graph data to the back-end storage nodes. The *Query Service* allows for analysis of the stored graph, while the *GraphDB Service* provides a unified mechanism for storing and accessing graph data.

Both the Ingestion and GraphDB services can draw parallels from the parallel file-system domain, particularly the Parallel Disk Model [32, 33] (PDM). The PDM provides a generic model for use in designing OOC algorithms and in calculating upper and lower bounds for OOC algorithm performance. In a sense, the Ingestion service's declustering of the input graph is equivalent to striping the graph data intelligently across multiple disks in a uniprocessor system. Within the Ingestion and GraphDB implementations, there are also PDM optimization techniques which may be applicable.

In the following subsections, we first present a brief overview of DataCutter and then discuss the details of each service.

2.1 DataCutter

DataCutter [9, 10] is a component-based middleware framework [1,4,5,15,19,27,29] designed to support coarse-grain dataflow [7] execution on heterogeneous environments. In DataCutter, the application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through *logical streams*. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). A filter is required to read data from its input streams and write data to its output streams only. The DataCutter runtime system supports both data- and task-parallelism. Processing, network,

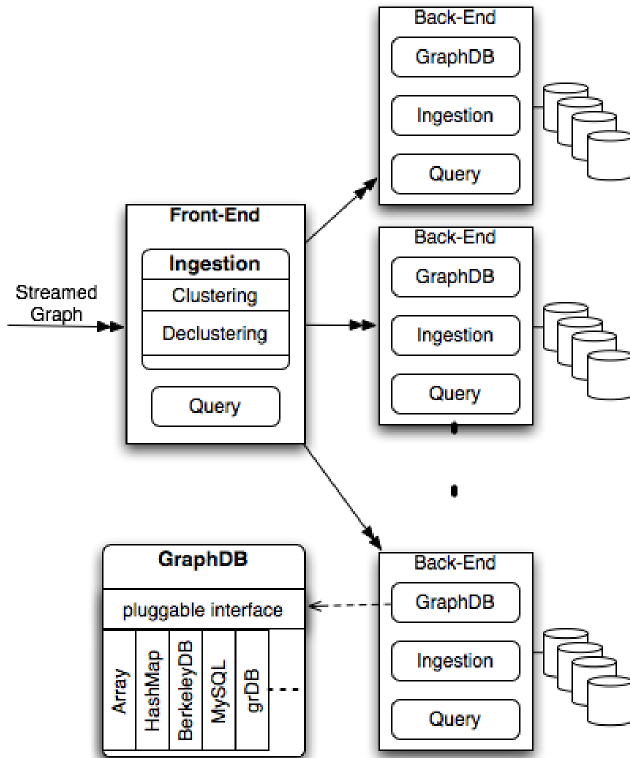


Figure 1. MSSG Overall System Architecture

and data copying overheads are minimized by the ability to place filters on different platforms. The filtering service of DataCutter performs all steps necessary to instantiate filters on the desired hosts, to connect all logical endpoints, and to call the filter's interface functions for processing work. Data exchange between two filters on the same host is carried out by memory copy operations, while a message passing communication layer (e.g. TCP sockets or MPI) is used for communication between filters on different hosts.

2.2 Ingestion Service

The Ingestion service provides the entry point for graph data to the MSSG system. Its job is to cluster and decluster (distribute) the ingested data appropriately to the GraphDB instances on the back-end nodes. Due to the sheer size of target graphs, these operations should be very efficient. The ideal approach is to perform these operations while the graph data is being ingested by the system via streaming.

The goal of this clustering and declustering is to achieve fast query processing by reducing the total number of disk I/Os incurred to access the database and increasing the parallelism during the query processing. Parallelism is relatively easier to achieve for queries which require processing a large portion of the dataset compared to queries that require processing only a localized portion of the data. For example, if a query were in the form of search between two vertices, it would be ideal if the vertices of the graph that are close to source of the search were clustered together in one GraphDB instance to reduce the I/O overhead. However, we also would like those vertices to be spread out to the nodes of the distributed storage system in order to achieve better parallelism.

A graph can be clustered and stored mainly at two granularity levels: 1) at the vertex level by storing all the edges incident to a vertex together and 2) at the edge level by storing each edge as an independent entity. MSSG supports both granularity levels. Similarly, streaming updates can arrive at those two granularity levels. Adding new vertices and new edges using vertex- and edge-level granularity respectively, however, necessitates novel clustering techniques. If vertex granularity is selected at the storage side, it would largely dictate the clustering and declustering of streaming updates. That is, if a vertex has been already clustered and assigned to storage node, all the new edges incident to that vertex have to be added to the same cluster to which the vertex belongs. Clustering would be simpler in this paradigm, but updating the data each time a new edge is stored can be very costly. Smart caching and blocking techniques help reduce the number of disk I/Os due to updates.

For clustering streaming data, MSSG processes the ingested data in *blocks* (or *windows*) of a pre-determined size, each of which fits into memory. Any streaming data can be converted into this format by accumulating incoming data to construct a block. Clustering algorithms will work on those blocks one by one. These algorithms must be very efficient in order for decisions to be made in real-time. Furthermore, these algorithms should keep some additional summary information about the data that has been already clustered and distributed to the nodes of the storage system. Using the summary information, the clustering algorithms should be able to make more intelligent decisions on where to send blocked data.

MSSG provides a customizable interface for developing clustering and declustering techniques. By default, the MSSG framework provides simple declustering techniques such as vertex- and edge-based round-robin declustering.

2.3 Query Service

The Query service provides the query interface for the client and orchestrates the execution of data analysis queries. In the MSSG framework, data analysis techniques are implemented as DataCutter filter graphs communicating via DataCutter's filter-stream interface. All implemented data analysis techniques are registered with the system and can be queried by the user. Any data analysis accesses the stored graph data via the unified graph interface provided by

GraphDB Service. Since each GraphDB instance only provides direct access to local data stored in each node, data analysis service instances need to be implemented in such a way as to take this data distribution into account. For example, if the graph is stored using vertex-level granularity, the complete adjacency list of any arbitrary vertex will be stored in only one node. Hence, any operation that requires accessing the adjacency list of a vertex needs to be either delegated to that node or the adjacency list needs to be transfered to the node that initiated the access. As an example Query Service instance, a relationship analysis method based on breadth-first search is described in more detailed in Section 3.2.

2.4 GraphDB Service

The GraphDB Service's job is to interface with a number of disparate storage mediums, such as various in-memory data structures, relational databases, and other disk-based storage methods.

One of the main innovations of the MSSG API consists of a Java interface *Graph* which exposes the smallest complete set of graph operations possible, along with one or two higher-performance methods which implement higher-level graph functions. In order to be complete, a graph-storage service only needs to store edges and retrieve lists of distance-1 neighbors (adjacent vertices).

Currently, there are several concrete classes which implement the graph interface and store the actual graph data in different formats and different storage mediums. Two of the default implementations are based on efficient in-memory storage for graphs that could fit in memory of the MSSG installation. We also provide three disk-based implementations of GraphDB services. Two of them are based on open-source databases, BerkeleyDB and MySQL. The last one, *grDB*, is a novel disk-based graph database designed for massive scale-free graphs.

2.4.1 grDB: Graph Database

We propose a novel graph database, *grDB*, which is intended to allow the efficient out-of-core storage and retrieval of scale-free graphs. A grDB instance is comprised of two components; the *storage component* and the *block cache component*. The storage component is responsible for the storage and retrieval of *blocks* which store partial adjacency lists of one or more vertices. The block cache component provides in-memory caching of the storage blocks for improved performance.

A scale-free graph in grDB is stored in multiple files that are composed of *blocks*. Blocks are smallest unit of I/O for grDB. While the optimum block size is determined by the performance characteristics of the physical storage system, we nevertheless expect the optimum block size will not be smaller than the filesystem's block size. Each block will be further divided into *sub-blocks* that are uniquely addressable. A sub-block is used to store a vertex's partial adjacency list. A grDB instance contains multiple levels of storage files. At level ℓ, each sub-block of a storage file can store up to d_ℓ adjacent vertices, where $d_\ell \geq 2 \times d_{\ell-1}$ for $\ell \geq 1$. Since our target graphs exhibit the power-law degree distribution, we suggest choosing d_ℓ values that also follow an exponential curve, such as $d_\ell = 2^{2^\ell}$.

Each vertex in grDB will have a b-byte unique Global ID (GID) in the range between 0 to n, where n is the number of vertices in the graph; hence, each sub-block in level ℓ is $b \times d_\ell$ bytes. Since each block can store one or more sub-blocks, block size B_ℓ at level ℓ is computed as $B_\ell = k_\ell \times b \times d_\ell$, for an integer $k_\ell \geq 1$. Because of file system limitations as well as performance reasons, at each level ℓ graph data is stored in multiple files with a maximum size of M-bytes, or equivalently $N_\ell = M/B_\ell$ blocks. The location of a sub-block s in the disk at a level ℓ can be found using simple modulo arithmetic as follows. Since each block stores k_ℓ sub-blocks, sub-block s is stored in s/k_ℓ-th block, which is stored in $s/k_\ell/N_\ell$-th file at offset $B_\ell \times ((s/k_\ell) \,\%\, N_\ell) + b \times d_\ell \times (s \,\%\, k_\ell)$.

The beginning of the adjacency list of a vertex v in grDB is stored in v-th sub-block at level 0. If a vertex has d_0 or less number of adjacent vertices, they are directly stored in that level. If v has more adjacent vertices than d_0, the first $d_0 - 1$ adjacent vertices are stored in the level 0 sub-block, and the last location in the level 0 sub-block is used as a pointer into the higher level files. During the ingestion of the graph data set, if the adjacent vertices are added in small groups, the adjacency list of a vertex could have entries in multiple levels of the grDB. For example, if vertex v already has d_0 adjacent vertices and one more adjacent vertex is added, a new sub-block is allocated for that vertex in level 1. A link from the v-th sub-block at level 0 to the newly allocated sub-block at level 1 is created. When the degree of vertex v achieves d_0+d_1, a new sub-block is allocated for that vertex at level 2, and either all of the contents of the sub-block at level 1 are moved to the new sub-block at level 2 and subsequent new adjacent vertices are added to that sub-block, or the sub-block at level 1 is left unchanged and simply links to the newly allocated sub-block at level 2. The former approach necessitates extra copy operations during the insertion, while the latter creates fragmentation in the adjacency list. One approach is to leave the adjacency lists fragmented during the ingestion, and later during "idle" time, the grDB service can defragment these multi-level adjacency lists in the background. Figure 2 illustrates the file format of grDB, and a small example is shown in Figure 3.

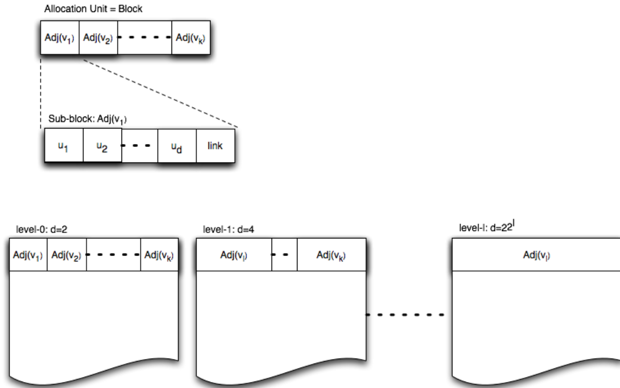


Figure 2. An illustration of grDB file format

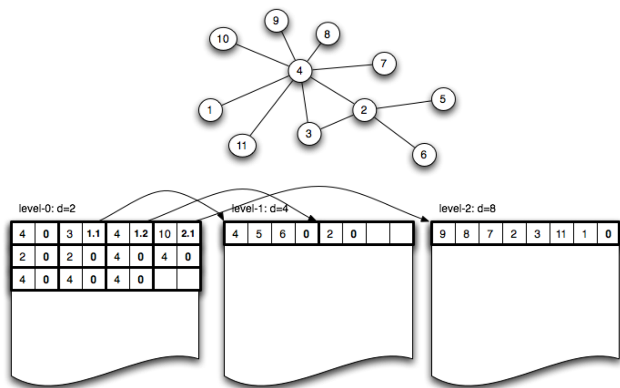


Figure 3. A sample graph and its storage representation in a 3-level grDB instance with $d_0 = 2$, $d_1 = 4$ and $d_2 = 8$

3 Prototype Implementation

We have implemented a prototype of the MSSG framework in Java. The framework allows analysis services to be implemented in Java using DataCutter's filter interface. We have implemented an analysis service which uses breadth-first search. Below we present the details of the prototype MSSG middleware and the breadth-first search analysis plug-in.

3.1 Customization of GraphDB Service

We have implemented five different instances which meet the generic GraphDB interface contract, two in-memory versions and three out-of-core versions using various persistent storage managers. Although the graphs we are targeting will not fit in memory, the two in-memory implementations provide a solid base-line comparison for our out-of-core implementations. In a sense, they represent the lower-bound we could achieve with the out-of-core implementations. Here is a brief description of these five GraphDB instances:

Array: The first in-memory implementation uses the standard *compressed adjacency list* format to store the graph in memory. When using the compressed adjacency list format, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is stored using two arrays of size $|\mathcal{E}|$ and $|\mathcal{V}| + 1$, respectively. The first array, adj, contains the adjacency list of all vertices concatenated one after the other, while the second array, $xadj$, stores a pointer to the beginning of adjacency list for each vertex. That is, the adjacency list of vertex v is stored at $adj[xadj[v]], \ldots, adj[xadj[v+1]-1]$. The advantage of this format is it provides very efficient access to the adjacency list of each vertex, by using the highest-performance in-memory data structure possible. However, this format has three major issues. First, Java only allows 32-bit integers as array indices, which restricts the input graph size. Second, this storage format is not suitable for dynamically-growing graphs. Third, it is poorly suited for storing graphs distributed to multiple machines; unless a block distribution of vertices is used, each node has to store the full $xadj$ array. Therefore, the array-based storage format's memory requirement does not scale with increasing numbers of back-end nodes. For small graphs that fit into main memory, the first and third concerns will not cause problems. For the input stage, when the graph is streaming in from the front-end nodes, we have actually used the `HashMap` implementation (see below) with integer IDs as temporary storage. After flushing the graph to disk, the `Array` GraphDB instance loads the graph into the compressed adjacency list arrays. The `Array` implementation is useful as a lower-bound on the search execution times.

HashMap: By using a *hash* data-structure one can improve on the memory requirements of the `Array` implementation when dynamic or distributed graph storage is required. There are two possible implementations, the first of which is to use a hash data-structure to map global vertex IDs to local vertex IDs. The compressed adjacency list array implementation can then be used with the local, renumbered vertices. This implementation requires *global-to-local* and *local-to-global* vertex ID translations and (like the `Array` instance) is not very suitable for dynamically growing its storage during the ingestion. The other implementation option entails storing the adjacency lists of each vertex separately and using a hash data-structure to store and retrieve the pointers to those adjacency lists. Although only global vertex IDs (64-bit longs in Java) are used and there is no need for global-to-local and local-to-global ID translation, accessing the adjacency list of a vertex still requires a hash look-up. We have implemented the latter approach using Java's `HashMap` data structure, which gives this GraphDB

instance its name. As already mentioned, this implementation's memory requirement scales well when increasing the number of back-end nodes, at the expense of additional hash look-up time in order to access the adjacency list for a vertex.

MySQL: We have implemented an out-of-core graph database instance using MySQL 4.1.12 [25]. With a standard {*src, dest*} table model, the overhead of retrieving the adjacency list of a vertex will be prohibitively high for a table with conceivably hundreds of millions of rows. Therefore, we have chosen to store the adjacency list of a vertex in one or more MySQL records indexed by the vertex ID. In order to provide a level of performance that approaches the other implementations we also have chosen to serialize the adjacency list into a *BLOB* data type in a table which is indexed by the source vertex. Since BLOBs can be of arbitrary size, however, we chose to chunk the adjacency list into standard-sized blocks (8 KB), as suggested by the MySQL documentation.

BerkeleyDB: We have also implemented an out-of-core graph database instance using BerkeleyDB version 1.7.1 [31]. The BerkeleyDB is a programming API which gives the user easy access to persistent, transactional, and storage without the overhead of using a relational database server. The chunking technique used in the MySQL implementation is also used here.

grDB: We have implemented the proposed grDB discussed in Section 2.4.1 in Java using the standard `RandomAccessFile` class. Global vertex IDs and file sub-block address pointers are 64-bit long integers where the 3 most significant bits are reserved for the grDB's internal use to mark when the value is a pointer to a higher-degree storage file. With 3 bits acting as the pointer indicator, we still allow 61 bit vertex numbers which will be sufficient for graphs with up to 2 quintillion vertices ($2 * 10^{18}$). In our experiments we have restricted the maximum file size to be $M = 256\ MB$. We have used a 6-level instance of grDB with d_ℓ values are equal to 2, 4, 16, 256, 4K and 16K, and with a block size, B_ℓ, of $4\ KB$ in the first 4 levels and 32 KB and 256 KB for the last two levels.

3.2 Parallel Out-of-core Breadth-First Search

In the prototype we provide an example instance of the Query Service which implements a parallel out-of-core Breadth-First Search (oocBFS) algorithm. BFS is one of the basic algorithms for relationship analysis [22, 35].

Algorithm 1 outlines the parallel oocBFS algorithm. The main algorithm is not much different than a sequential BFS algorithm. The main differences come from graph data distribution and storage. If an edge-level granularity is used to store the graph, it is possible that the adjacency list of a vertex is distributed to multiple nodes. In such a case, the next level's frontier vertices N need to be broadcasted to all the processors. A similar situation arises when vertex-level granularity is used but the vertex mapping is not known globally by every processor. In this case, frontier vertices have to be broadcasted to all processors again. In both of these cases, the frontier set F will be identical for all processors. However, if vertex-level granularity for distribution is used with a globally-known mapping (such as $GID \,\%\, p$, where p is the number of back-end nodes), on each processor P_i the frontier set F will only contain the vertices assigned to P_i. Algorithm 1 handles all of these cases naturally by using a simple graph interface which returns the empty set when an adjacency list of a vertex that is not assigned to that processor is requested (steps 5 and 10).

In the current implementation, we rely heavily on the underlying database's caching and clustering capabilities to hide the latency that comes from accessing the adjacency list on disk. This dramatically simplifies the oocBFS imple-

Algorithm 1 Parallel Out-of-core Breadth-First Search Algorithm

1: **function** OOCBFS($\mathcal{G} = (\mathcal{V}, \mathcal{E}), s, d$)
2: *Initial data distribution*: $\mathcal{G}$ is divided into p subgraphs $G_1 = (V_1, E_1), \ldots, G_p = (V_p, E_p)$ where $E_1, \ldots, E_p$ is a disjoint partition of the edge set $\mathcal{E}$ and $V_i \subseteq \mathcal{V}$ for $1 \leq i \leq p$.
3: **on each** processor P_i, $1 \leq i \leq p$.
4: $level[v] = \infty$ for $v \in \mathcal{V}$
5: $F \leftarrow adj_{G_i}(s)$
6: $level[s] \leftarrow levcnt \leftarrow 0$
7: **while 'found'** message has not been received **do**
8: $levcnt \leftarrow levcnt + 1$
9: **for** $v \in F$ **do**
10: **for** $u \in adj_{G_i}(v)$ **do**
11: **if** $u = d$ **then**
12: send **found** message to all processors and **return** $levcnt$
13: **else if** $level[u] = \infty$ **then**
14: $level[u] \leftarrow levcnt$
15: $N \leftarrow N \cup \{u\}$
16: **if** $\mathcal{G}$ is stored with vertex-level granularity **and** vertex mapping $map[u]$ is known by every processor **then**
17: **for** $u \in N$ **do**
18: $S_{map[u]} \leftarrow S_{map[u]} \cup \{u\}$
19: send S_q to processor P_q for $1 \leq q \leq p$.
20: **else**
21: broadcast N to all processors
22: Receive R_q from P_q for $1 \leq q \leq p$.
23: $F \leftarrow \bigcup_q R_q$
24: **return** $infty$

mentation and our experiments show that both BerkeleyDB and grDB benefit from using a cache. The performance of this algorithm can be further optimized by introducing some pre-fetching of the adjacency lists of the vertices in the frontier. It can be even further optimized by sorting the pre-fetch disk accesses by file offsets to reduce the seek overhead. As part of our future work, we will investigate both of these options.

4 Experimental Results

We carried out the experimental evaluation of the MSSG framework on a 64-node Linux cluster owned by the Department of Biomedical Informatics at The Ohio State University. Each node of the cluster is equipped with dual 2.4 GHz Opteron 250 processors, 8 GB of RAM and two 250 GB SATA drives providing 500 GB of local storage via software RAID0. The nodes are interconnected with switched gigabit ethernet and Infiniband.

The tests were performed using vertex declustering during ingestion; the vertex ownership knowledge was leveraged during the search phase. All the tests were performed with one of three graphs; two real-world semantic graphs, PubMed-S and PubMed-L, were extracted from the PubMed document database, while the third, 100M was created to exhibit the scale-free properties which MSSG targets. PubMed-S contains 3,751,921 vertices and 55,682,678 directed edges, PubMed-L has 26,676,177 vertices and 519,630,678 directed edges, and the 100M graph has 100,000,000 vertices and 1,999,999,640 directed edges. While these graphs are smaller than the trillion-vertex graphs MSSG targets, examining the performance of these preliminary experiments is a necessary first step in order to scale to larger graphs.

The first results displayed in Figure 4 represent a baseline comparison of the search performance of the MSSG framework using in-memory implementations of GraphDB. In this experiment 100 random BFS queries were executed on 16 nodes against the PubMed-S graph, and the query execution times are averaged based on the path length between the source and destination vertices. As seen in the figure, the Array graph storage performs much better than the HashMap implementation, as expected. The HashMap GraphDB storage requires a hash lookup to access the adjacency list of a vertex; with large graphs, this overhead becomes significant. This is especially true as the path length increases, since the size of the fringe at each search level increases exponentially. However, when increasing the number of processors, this overhead is spread over multiple processors and the difference between Array and HashMap is lessened.

The second set of experiments shows the importance of caching effects when dealing with out-of-core data structures. Figure 5 displays the average performance on 16 nodes of BerkeleyDB and grDB on 100 random queries against the PubMed-S graph, with their internal (block) caches enabled and disabled. As seen in the figure, caching can reduce the execution time up to 50% on both implementations, especially for longer path queries. Therefore, further results will only come from our cache-enabled out-of-core implementations.

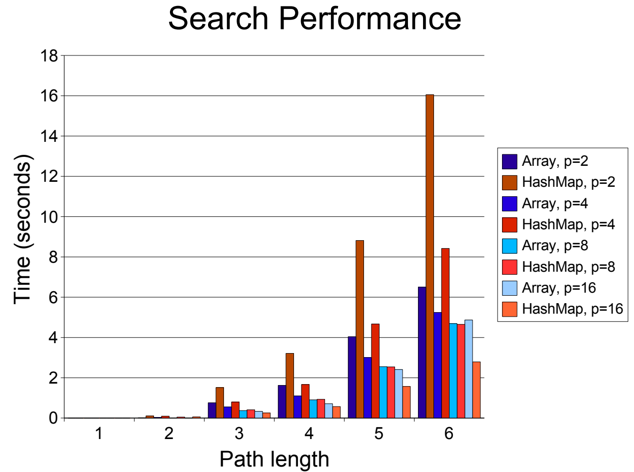


Figure 4. Performance of in-memory GraphDB implementations

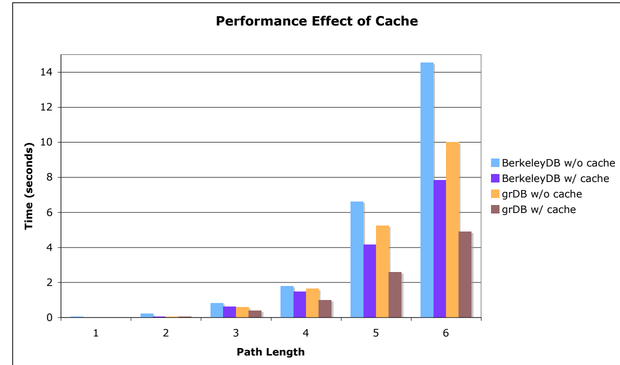
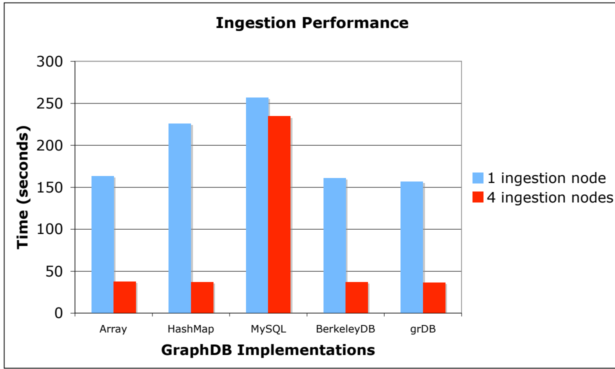
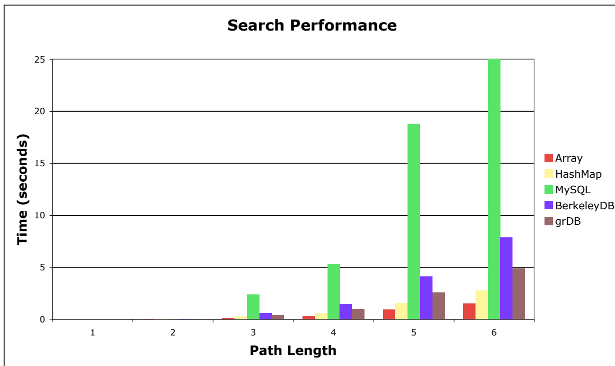


Figure 5. Performance of BerkeleyDB and grDB with and without cache

Figure 6(a) displays a comparison of the five different GraphDB implementations on 16 nodes using the PubMed-S graph. To investigate the effect of increasing the number of front-end ingestion nodes, we repeated the ingestion of PubMed-S multiple times and varied the number of ingestion nodes. The result shown in Figure 6(a) shows that Array, BerkeleyDB, and grDB achieved similar performance in both cases (1 ingestion node vs 4 ingestion nodes).



(a)

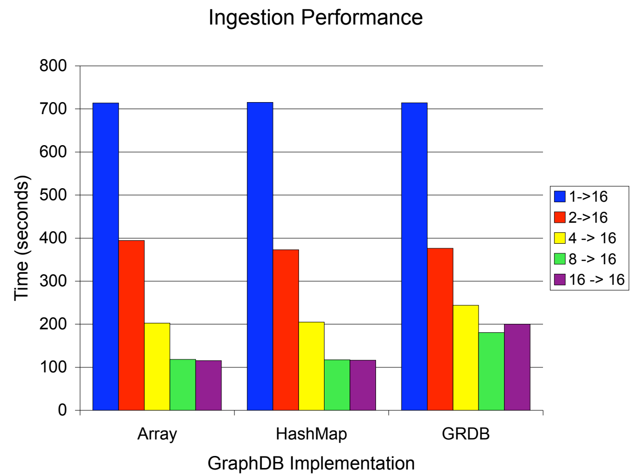


(b)

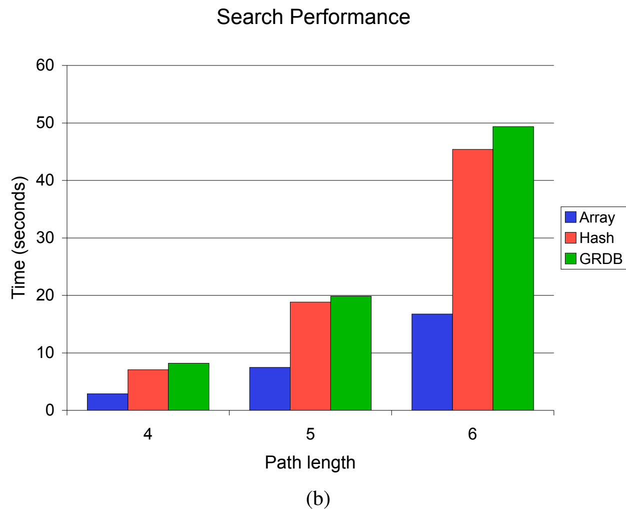
Figure 6. Performance comparison of five GraphDB implementations on PubMed-S graph

However, the HashMap and MySQL implementations were slower when only 1 ingestion node was used. When using 1 ingestion node, the ingestion speed is clearly limited by the I/O and network performance of that node. Furthermore, even though the ingestion node reads a block (window) of edges and then distributes them to back-end nodes, edge ordering in the streaming input graph could negatively affect the load balance on the back-end nodes. By increasing the number of ingestion nodes, we both remove the bottleneck from the front-end node and also achieve better load balance on the back-end nodes. The results show that the ingestion performance is more or less the same for all approaches, except for MySQL, which is slower than all other GraphDB storage implementations.

As seen in Figure 6(b), the Array implementation gives the lowest search time. Not surprisingly, the second best results are achieved with the other in-memory implementation, HashMap. MySQL performs significantly worse than all other implementations. The fastest of the three out-of-core GraphDB implementations, grDB, performs an average of 33% faster than the next fastest out-of-core implementation, BerkeleyDB. When comparing grDB with the in-memory implementations, grDB is only 1.7 times slower than HashMap and about 2.9 times slower than Array, on average. Finally, the search times for short paths (and hence small fringe sizes) is negligible for all GraphDB implementations. As such, future results will only show longer path lengths.



(a) Front-end ingestion nodes: varies, Back-end storage nodes: 16



(b)

Figure 7. Performance comparison of three GraphDB implementations on PubMed-L graph

Since the grDB instance is the fastest of the out-of-core GraphDB implementations, the experiments on PubMed-L were only carried out with grDB and the two in-memory GraphDB implementations. The results of the ingestion experiments in Figure 7(a) still show that the system is limited by the I/O bandwidth of the ingestion front-end nodes

when a small number of ingestion nodes are used. As such, only the results with four or more ingestion nodes show any significant difference between the three GraphDB implementations. The Array instance (with its integer-based HashMap storage used during ingestion) performs the best, followed by HashMap and grDB. The slight increase in ingestion time for grDB with 16 ingestion nodes is due to the thrashing of the working set in grDB's in-memory block cache.

Figure 7(b) shows the results of the search experiments, which were run on 16 nodes. As before, the Array GraphDB implementation is the fastest of the three instances. However, the HashMap implementation loses some of its speed advantage as the graph size grows, compared to the grDB storage engine. The grDB result, therefore, is only 2.95 times slower than the Array instance, and a mere 1.09 times slower than the HashMap implementation!

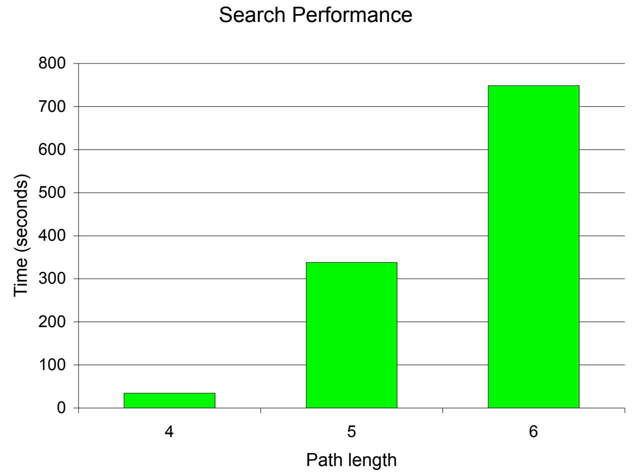
Figure 8 shows the preliminary results of MSSG's performance when working with the 100M graph. Figure 8(b), in particular, shows the search time against the total sum of unseen vertices placed into the fringe when searching for the goal vertex. The outcome of these two graphs is positive. There is a clear linear relationship between the number of unique vertices seen and the search time. However, more work needs to be done before the system could be called interactive when working with such large graphs.

5 Conclusions and Future Work

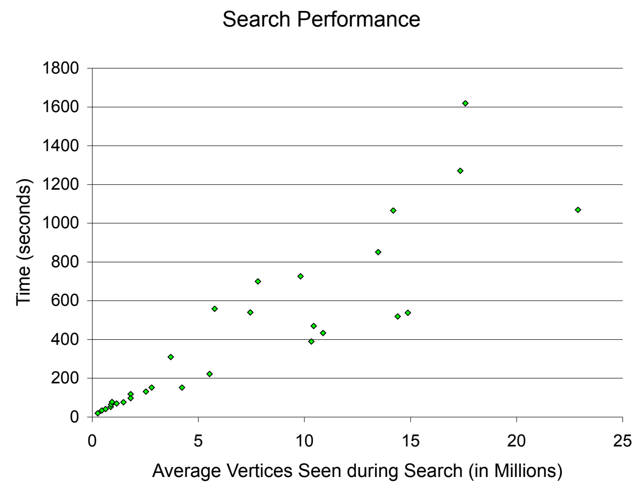
In this paper we presented a middleware framework, MSSG, for storing, accessing, and analyzing massive-scale semantic graphs. We proposed and developed a novel disk-based graph database, *grDB*, for massive scale-free graphs. We have also developed a parallel out-of-core breadth-first search algorithm. To the best of our knowledge, this is the first of such algorithms presented in the literature. Experimental evaluations on large real-world semantic graphs show that the MSSG framework scales well. Also, grDB outperforms widely used open-source databases, such as BerkeleyDB and MySQL, in storage and retrieval of scale-free graphs.

As part of our future work in this area, we will investigate the applicability of other external memory libraries, such as the Transparent Parallel I/O Environment (TPIE) [6] in our grDB implementation. TPIE is designed for providing efficient access to multiple disks attached to a single system (hence the word 'parallel' in the name). Its Random-acess Block Transfer Engine seems like a viable replacement for the low-level random access file API.

Additionally, we will continue to investigate different optimization techniques for the Ingestion, GraphDB, and Query services. We will explore various intelligent data-striping and caching techniques.



(a)



(b) Number of unique vertices considered during search for 100M on grDB

Figure 8. Search Performance for 100M graph using grDB

References

[1] The ABACUS project. *http://www.cs.cmu.edu/∼amiri/abacus.html*.

[2] J. Abello, A. L. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[3] J. Abello, M. G. C. Resende, and S. Sudarsky. Massive quasi-clique detection. In *LATIN*, pages 598–612, 2002.

[4] M. Aeschlimann, P. Dinda, J. Lopez, B. Lowekamp, L. Kallivokas, and D. O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, Las Vegas, NV, June 1999.

[5] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster com-

puting. In *the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[6] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *ESA*, pages 88–100, 2002.

[7] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster i/o with river: making the fast case common. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 10–22, New York, NY, USA, 1999. ACM Press.

[8] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *J. Parallel Distrib. Comput.*, 65(9):994–1006, 2005.

[9] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133. National Aeronautics and Space Administration, March 2000. NASA/CP 2000-209888.

[10] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[11] S. Bokhari, B. Rutt, P. Wyckoff, and P. Buerger. An evaluation of the osc fastt600 turbo storage pool. Technical Report OSUBMI_TR_2004_n02, The Ohio State University, Department of Biomedical Informatics, Sep 2004.

[12] E. G. Boman, D. Bozdağ, U. Catalyurek, A. H. Gebremedhin, and F. Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *EuroPar 2005*, 2005.

[13] D. Bozdağ, U. Catalyurek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner. A parallel distance-2 graph coloring algorithm for distributed memory computers. In *The 2005 International Conference on High Performance Computing and Communications (HPCC-05)*, 2005.

[14] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *SODA*, pages 859–860, 2000.

[15] Common Component Architecture Forum. *http://www.cca-forum.org*.

[16] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[17] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.

[18] T. Eliassi-Rad and E. Chow. Using ontological information to accelerate search in large semantic graphs: A probabilistic approach. Technical Report UCRL-CONF-20200, Lawrence Livermore National Laboratory.

[19] C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, Cancun, Mexico, May 2000.

[20] M. T. Jones and P. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.

[21] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library, version 3.1. Technical report, Dept. Computer Science, University of Minnesota, 2003. http://www-users.cs.umn.edu/ karypis/metis/parmetis/download.html.

[22] T. Kolda and et. al. Data sciences technology for homeland security information management and knowledge discovery. Technical Report UCRL-TR-208926, Lawrence Livermore National Laboratories, 2004. Report of the DHS Workshop on Data Sciences, September 22-23, 2004.

[23] E. Konstantinova. Chemical hypergraph theory. Lecture Notes from Combinatorial & Computational Mathametics Center, http://com2mac.postech.ac.kr/, 2000.

[24] U. Meyer and N. Zeh. I/o-efficient undirected shortest paths. In *ESA*, pages 434–445, 2003.

[25] MySql Database. *http://www.mysql.com/*.

[26] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[27] R. Oldfield and D. Kotz. Armada: A parallel file system for computational grids. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[28] F. Olken. Graph data management for molecular biology. *OMICS*, 7(1):75–78, 2003.

[29] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

[30] E. Ramadan, A. Tarafdar, and A. Pothen. A hypergraph model for the protein complex network in the yeast. In *Proceedings of 186th International Parallel and Distributed Processing Symposium (IPDPS), Third Workshop on High Performance Computational Biology*, Santa Fe, NM, April 2004.

[31] Sleepy Cat Software. Berkeley DB. http://www.sleepycat.com/.

[32] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society Press, Providence, RI, 1999.

[33] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. Technical Report Technical report DUKE–TR–1993–01, 1993.

[34] W. Xu, L. Krishnamurthy, M. Tasan, G. Özsoyoglu, J. H. Nadeau, Z. M. Özsoyoglu, and G. Schaeffer. Pathways database system: An integrated set of tools for biological pathways. In *SAC*, pages 96–102, 2003.

[35] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of SC2005 High Performance Computing, Networking, and Storage Conference*, 2005. Gordon Bell Finalist.