OMRGx: Programmable and Transparent Out-of-Core Graph Partitioning and Processing

Gurneet Kaur
University of California Riverside
Riverside, USA
gkaur007@ucr.edu

Rajiv Gupta
University of California Riverside
Riverside, USA
rajivg@ucr.edu

Abstract

Partitioning and processing of large graphs on a single machine with limited memory is a challenge. While many custom solutions for out-of-core *processing* have been developed, limited work has been done on out-of-core *partitioning* that can be far more memory intensive than processing. In this paper we present the OMRGx system whose programming interface allows the programmer to rapidly prototype existing as well as new partitioning and processing strategies with minimal programming effort and oblivious of the graph size. The OMRGx engine transparently implements these strategies in an out-of-core manner while hiding the complexities of managing limited memory, parallel computation, and parallel IO from the programmer. The execution model allows multiple partitions to be simultaneously constructed and simultaneously processed by dividing the machine memory among the partitions. In contrast, existing systems process partitions one at a time. Using OMRGx we developed the first out-of-core implementation of the popular MtMetis partitioner. OMRGx implementations of existing GridGraph and GraphChi out-of-core processing frameworks deliver performance better than their standalone optimized implementations. The runtimes of implementations produced by OMRGx decrease with the number of partitions requested and increase *linearly* with the graph size. Finally OMRGx default implementation performs the best of all.

CCS Concepts: • **Computing methodologies** → **Parallel computing methodologies**; • **Information systems** → **Computing platforms**.

Keywords: irregular graphs, out-of-core graph partitioning, out-of-core graph processing, map-reduce

ACM Reference Format:
Gurneet Kaur and Rajiv Gupta. 2023. OMRGx: Programmable and Transparent Out-of-Core Graph Partitioning and Processing. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3591195.3595268

ISMM '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0179-5/23/06.
https://doi.org/10.1145/3591195.3595268

1 Introduction

With the growing popularity of single machine analytics, out-of-core systems (e.g., GraphChi [15], GridGraph [29], etc.) have been developed enabling processing of graphs that are too large to fit in available memory. The graphs are therefore partitioned and then partition-based processing is performed where partitions, that individually fit in machine memory, are loaded one at a time and processed. Graph partitioning is even more memory intensive. The popular multilevel graph partitioner MtMetis [9, 17] requires 4.8×–13.8× [14] the memory that it takes to hold the original graph as it creates many versions of the graph via graph coarsening and uncoarsening. However, work on out-of-core graph partitioning is quite limited – to the best of our knowledge, GO [14] is the only out-of-core graph partitioner. All of the above systems are custom systems whose development required substantial effort.

In this paper we present the OMRGx system that enables rapid prototyping of existing and new graph partitioning and partition-based processing algorithms with minimal effort. The system consists of an *execution engine* and a *programming interface*. The *execution engine* transparently deals with the complexities of out-of-core processing required to successfully perform graph partitioning and processing of a given graph no matter its size. The *programming interface* allows partitioning and processing tasks to be expressed with minimal programming effort oblivious of the graph size in relation to machine memory size. The ease of expressing wide variety of algorithms using our map-reduce based programming interface and the highly specialized map-reduce out-of-core engine for graphs respectively deliver programmability and performance for partitioning and processing of large graphs. The key aspects of this system are are as follows.

Transparent Out-of-Core Execution Engine. The out-of-core map-reduce engine implements a novel runtime that transparently supports management of limited memory, parallel mappers and reducers, and parallel IO to achieve highly optimized implementation of partitioning and processing

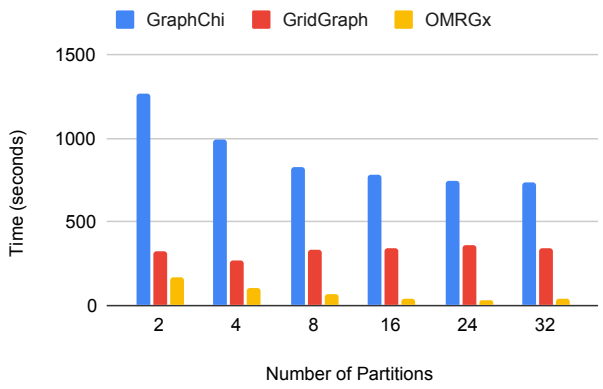

Figure 1. Comparison of PageRank Processing Times (in Seconds) for GraphChi, GridGraph, and OMRGx on an input UKDomain-2007 graph with over 3 billion edges.

logic provided by the user. The novel features of this engine include the following:

- *Simultaneously Creating or Processing Partitions.* The single machine resources, limited memory and parallel threads, are distributed among the partitions allowing all partitions to be simultaneously created or processed. In doing a partition does not need to fully reside in memory but rather it is streamed through the memory buffer assigned to the partition. As a consequence the number of partitions used is not dependent upon the memory size of the machine, the user can configure OMRGx to consider any number of partitions while dividing available memory among those partitions. Parallel loading of partitions reduces overall I/O overhead. In contrast, existing out-of-core graph processing systems (e.g., [15]) process one partition at a time and require that each partition fit in memory.
- *In-Memory and On-Disk Graph Representations.* To enable simultaneous processing of multiple partitions, each partition is held partially in memory and on disk. Thus, we employ both in-memory and on-disk graph representations as well as runtime support that transforms the graph between these representations as it is transferred between memory and disk. Runtime efficient serialization and deserialization is achieved via protocol buffers [31] and storage on disk via Infinimem object store [12]. Moreover, protocol buffers allows use of custom formats for the graph structure.

The benefits of above features of OMRGx translate into significant speedups over the GraphChi and GridGraph systems as shown in Figure 1. Moreover, the runtime of OMRGx decreases with the number of partitions via greater parallelism.

Graph Partitioning and Processing Programming Interface. Our interface allows graph partitioning and processing logic to be easily expressed via map and reduce operations. The programming is independent of the graph size and machine memory size. To express graphs in this programming model, a *key* corresponds to a vertex (node) id and the edges, represented by destination vertex ids, correspond to the *value* for the key. The partitioning strategies can be expressed easily in form of a *map* function. Complex partitioning strategies like MtMetis [17] that make repeated passes to refine partitions can implement these passes via an additional *reduce* function. Given a partitioned graph, the processing can also be easily specified by the user using a *reduce* function. *Multiple mappers and reducers in the engine carry out partitioning and processing in parallel.*

Demonstration of System Capabilities. We show that OMRGx enables rapid prototyping of graph partitioning and partition-based graph processing strategies while delivering scalable performance. We developed multiple graph partitioners ranging from simple hash partitioner to sophisticated GO [14] and MtMetis [17, 18] partitioners. We also present default OMRGx processing algorithm while also programming strategies proposed in GraphChi [15] and GridGraph [29] out-of-core graph processing systems. The effectiveness of our approach is evident from the following:

- *Graph Partitioning with* OMRGx: OMRGx generated MtMetis implementation is the first out-of-core implementation of popular MtMetis [17] partitioner. This implementation successfully partitions large graphs while existing implementation of MtMetis fails as it runs out of memory. We also performed rapid prototyping of the GO [14] partitioner using OMRGx.
- *Graph Processing with* OMRGx: The runtimes of implementations produced by OMRGx decrease with the number of partitions requested and increase linearly with the graph size. OMRGx implementations of existing GridGraph [29] and GraphChi [15] out-of-core frameworks deliver performance better than their standalone optimized implementations. OMRGx default implementation performs the best of all.
- OMRGx *Programmability*: Using 573 lines of C++ code we were able to prototype two sophisticated out-of-core graph partitioners (GO and MtMetis) and two well known out-of-core graph processing frameworks (GraphChi and GridGraph).

Thus, OMRGx simultaneously delivers high-performance and programmability for rapid prototyping.

The remainder of the paper is organized as follows. In Section 2 we present the novel features of our out-of-core engine, describe the programming interface and illustrate its use in programming first simple and then complex partitioning and processing algorithms. In Section 3 we present detailed evaluation. Section 4 presents our conclusions.

2 OMRGx: Programming Interface and Out-Of-Core Execution Engine

OMRGx provides a rich programming interface that allows programmers to remain *oblivious* of the need for Out-of-Core graph handling and consequently simply express the logic of different graph partitioning and processing schemes using simple *map* and *reduce* functions. Based on this partitioning and processing logic, the runtime takes care of the entire process of handling large amounts of data in parallel using limited memory and optimizing parallel IO operations for transfer of data between disk and memory whenever needed. The system, shown in Figure 2, provides programmers with the flexibility to program different applications and strategies. Next, we first describe OMRGx's application programming interface that enables graph partitioning and processing, and then show how the system transparently handles processing of large graphs beyond the size of main memory. Finally, we illustrate implementation of advanced partitioning and processing algorithms using OMRGx.

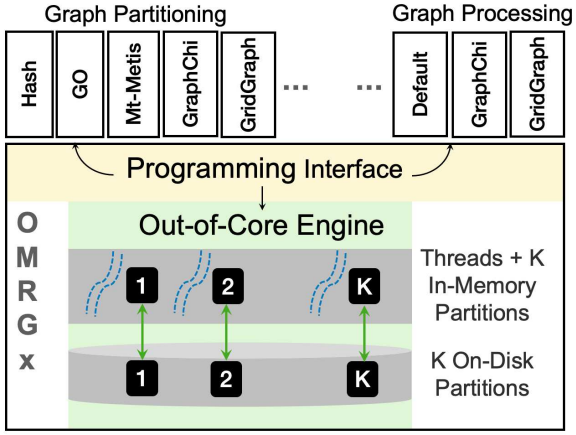

Figure 2. Overview of OMRGx Graph Partitioning and Processing System.

2.1 The Programming Interface

The high-level interface provided by OMRGx allows programmers to express different graph partitioning strategies and also perform partition-based graph processing. OMRGx does not limit the user to any specific partitioner or processing approach for large graphs. Instead it provides the programmer with the ability to rapidly prototype new and existing graph partitioning and processing approaches of choice. The programmer transparently benefits from the use of out-of-core capabilities of the system.

The set of APIs shown in Table 1 represent bulk of OMRGx's programming interface. The core functions represent the minimal set of functions a programmer must use to implement graph partitioning and/or processing. The additional functions listed have default implementations that the user can rely upon. However, the user has the option of providing custom versions of these functions when programming more sophisticated frameworks. We will discuss and illustrate the use of core functions in this section and in a later section describe the usage of others.

The programmer specified *map* function is used to split the input into $< key - [values] >$ pairs. Here *keys* are the *vertices*/*nodes* and *values* represent an edge `adjacencylist` for the vertex. In order to partition graphs, we need to define a partitioning strategy which is expressed using the `setPartitionID` API. This API is used to set the *PID* in memory for the input $key - value$ pair which is then passed to the *map* function to store the $< key - value >$ to the specified partition. If during the *map* phase, the in-memory buffer partitions grow in size, their entire contents are written off to disk to make room for further $< key - value >$ pairs. The *reduce* function processes all the $< key - [values] >$ pairs in the partitions on disk in parallel, refines them based on the logic provided by the application programmer and emits the final partitions. This process repeats until the entire graph is read and processed. Note, if the refinement logic is not provided by the programmer, the *default* behavior is to merge all the *values* for each *key* as $< key - [values] >$ pairs into in-memory buffer partitions and write these off to disk.

The graph processing algorithms tend to be iterative in nature (e.g., PageRank). Therefore, OMRGx provides APIs to support iterative graph processing. The APIs $beforeMap$,

Table 1. *OMRGx* APIs to implement graph *partitioning* and perform partition-based graph *processing*.

Core Functions Provided by the Programmer
setPartitionId (unsigned PID);
map (const unsigned tid, std::string& input, unsigned PID);
reduce (unsigned PID, const InMemoryContainer<KeyType, ValueType>& partition);
updateReduceIteration (const unsigned PID);
Additional Functions With Default Implementations
diskReadPartition (unsigned PID, unsigned count);
diskWritePartition (unsigned PID, unsigned count, InMemoryContainer<KeyType, ValueType>& partition);
beforeMap (unsigned tid);
afterMap (unsigned tid);
beforeReduce (unsigned PID);
afterReduce (unsigned PID);
setIterations ();

```
template <typename KeyType, typename ValueType>
class Hash : public MapReduce<KeyType,ValueType> {
 unsigned setPartitionId(const unsigned tid,
                         const KeyType& key){
   return hashKey(getkey(key));
 }

 void* map(const unsigned tid,
           std::string& input, unsigned PID){
   std::stringstream inputStream(input);
   std::string token;
   inputStream >> to >> from;

   foreach(to, from) /* Key to and Value from */
     EdgeType e;
     e.src = from;  e.dst = to;
     e.rank = e.vRank = 1.0/nvertices;
     e.numNeighbors = from.size();
     writeToBuffer(to, e, PID);
 }

 void* reduce(unsigned PID, InMemoryContainer<KeyType,
              ValueType>& partition) {
   double sum = 0.0;
   done = true; // TermCondition = true
   foreach(vertex v: partition){
        if (v->numNeighbors > 0)
            sum += (v->rank / v->numNeighbors);
       double old = v->rank;
       v->rank = (DAMPING_FACTOR * sum) +
                          (1 - DAMPING_FACTOR);
       foreach(neighbor n) { n->vRank = v->rank; }
       if (fabs(old - v->rank) > TOLERANCE)
           done = false;
   }
 }

 void* updateReduceIter(const unsigned tid) {
    if (done) return;
    ++iteration;  }
};
```

Figure 3. PageRank algorithm using Hash partitioner programmed in *OMRG*x.

$afterMap$, $beforeReduce$, and $afterReduce$ are used to set and clear graph related structures. During each iteration *reduce* operation is performed. The *updateReduceIteration* API is used to update the computation data structures to prepare for the next iteration. It also checks the termination condition if one is provided by the programmer. Alternatively, the user can decide to run the algorithm for a fixed number of iterations specified using the *setIterations* API.

For the iterative algorithms, the entire graph is read from or written off to disk multiple times causing processing that propagates and updates values from one iteration to next. To support the iterative processing, OMRGx internally uses *diskReadPartition* and $diskWritePartition$ functions to read/write from/to disk. The programmers also have the choice to read/write the entire partition or a part of it based on their implementation. Therefore, the programmers also have the access to *diskReadPartition* and $diskWritePartition$ functions as extended APIs which they can directly call in their programs.

To illustrate the use of graph partitioning and processing APIs, Figure 3 shows the implementation of PageRank in OMRGx using a Hash partitioner. The Hash partitioner, as shown, is a simple partitioner that hashes vertices into different partitions (using `setPartitionId` API) based on their keys during the *map* phase and calculates the PageRank of each vertex within the partition during the *reduce* phase. During *map* phase, vertex *key* and all its corresponding edges *value* are written off to buffers in memory in the form of $< key - value >$ pairs. Here the incoming vertex edge $from$ is stored along with the other information like its src, $rank$, $num - of - neighbors$ etc. This incoming vertex along with its key *to* is written to in memory buffer based on the specified *PID*. All edges corresponding to each key *to* are aggregated as $< to - [EdgeType] >$ pairs in their PIDs.

During the *reduce* phase, all the edges corresponding to each vertex within a *PID* are merged together by the reducer and made available in the *partition*. Therefore, for each key *to*, its *rank* is calculated based on its previous rank which is stored as its attribute along with its neighbor edges $from$. It then updates this new *rank* for all of the neighbors of key *to*. Once the entire partition is processed, the $updateReduceIter$ API increments the number of iterations for the PageRank algorithm until it finally converges, thereby, writing the final partition using the $afterReduce$ API. Note that, one iteration of a PageRank is said to be completed when the entire graph is read from disk and processed in memory to update the rank for each vertex. As mentioned earlier OMRGx internally uses its $diskWritePartition$ API to propagate the updated values for the next iteration. The entire process of reading and writing graphs from disk in parallel and optimized IO operations is *oblivious* to the application programmer. The users have the flexibility to program any kind of partitioning and processing framework.

While OMRGx can be used to express wide variety of graph partitioning and processing algorithms, its powerful out-of-core capabilities combined with its simple architecture makes it a self-contained system in its *default* state. In *default* state OMRGx uses hash partitioning and default graph representations. Application programmers can simply express the partitioning logic using the *map* API and express the PageRank logic using the *reduce* API to output the final $key - [values]$ pairs.

2.2 Transparent Out-Of-Core Engine

OMRGx provides a runtime that manages all the complexities associated with out-of-core processing transparently to the programmer. The runtime manages the memory available to carry out partitioning and processing of large graphs. The two key features of this runtime are as follows.

- First, OMRGx uses the available memory to form multiple memory buffers that are used by multiple threads to simultaneously create or process partitions.
- Second, it uses a novel strategy for representing multiple partitions in memory and on disk while seamlessly

moving them between memory and disk giving the illusion of entire graph being available in memory.

Together, the above two features enable exploitation of parallelism in computation and IO while allowing partitioning and processing tasks to proceed for large graphs on a limited memory machine.

As shown earlier, the logic is specified by the user via *setPartitionId*, *map*, and *reduce* functions. While *setPartitionId* assigns partition ids to graph entities, *map* and *reduce* functions are used to perform tasks in parallel via creation of mapper and reducer threads as shown in Figure 4. The *run()* method initializes the engine and initiates the threads.

Each *mapper* thread owns a row of in-memory buffers, one buffer per partition, as shown in Figure 5 (left half). A *mapper* thread maps the vertex *key* along with its *EdgeType* value based on its *partitionID* to an appropriate buffer in its row. Each *key* (node) can hold a list of values (edges); thus, multiple $< key - EdgeType >$ pairs with the same key are aggregated as $< key - [EdgeType] >$ pairs as shown in Figure 6. Once the in-memory buffer reaches its capacity, its contents are emptied to disk as a batch of contiguous records. As mentioned earlier, the contents within each batch are serialized as *RecordType*. At the end of *map* phase, any given *key* will be present in some or all of the batches with the *same* color and the entire graph is partitioned and stored on disk as shown in Figure 5 (right view).

During the *reduce* phase, the engine initiates the *reducer* threads based on the number of partitions (k) of the graph to be produced. Each *reducer* thread independently operates on batches with the same color. This means, *threads* during the *reduce* phase own batches with the same color which correspond to an entire partition of a graph. Each thread reads as much from each batch within its assigned partition, deserializes the *RecordType* into its corresponding $< key - EdgeType >$ pair and also ***merges* the** *EdgeType* **split across batches within the same partition for each** *key*. This in-memory partition which contains a portion of its on-disk partition is then used to run the user specified processing logic using the *reduce* API as shown in Figure 6. A single *iteration* of graph is complete when each on-disk

```
int main(int argc, char** argv)
{
 Hash<IdType, EdgeType> hs;
 std::cout << "Usage: " << argv[0] << " <folderpath>
<nmappers> <nreducers> <graphSize> <topK> <optional -
nvertices> <optional - iterations> <optional - partition
output prefix>" << std::endl;

 hs.setInput(folderPath);
 hs.setMappers(nmappers);
 hs.setReducers(nreducers);
 …
 hs.run();
 return 0;
}
```

Figure 4. *OMRGx* Engine's main method.

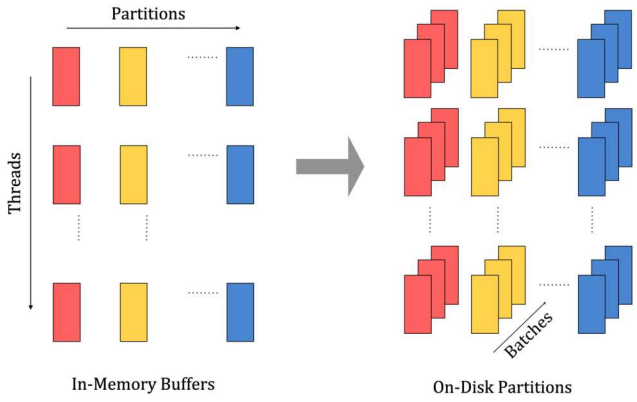

Figure 5. Organizing Memory into Buffers and Disk Usage.

partition is read in its entirety and processed in memory. Due to iterative nature of PageRank, the *reduce* phase will go through more such iterations until convergence.

Note if the entire graph fits in memory, then all the $< key - EdgeType >$ pairs represent all values for their corresponding keys and hence, can be directly sent to reducer where processing logic is applied.

When the input graph is read from disk, it is stored in a partitioned form in memory buffers. The initial partitioning logic is expressed in the *map* API along with specifying the partitioning scheme using the `setPartitionId` API. When memory buffers are filled to the capacity, their contents are serialized into batch of contiguous records using the custom format defined by users in Protocol Buffers and written to disk (using the Infinimem object store) to empty memory space for further processing. *OMRGx* internally uses *diskWritePartition* API to store the partitions from memory buffers to disk and provides seamless memory to its users. When the entire input graph is read, all its contents are organized on disk as partitions of input graph. It is important to

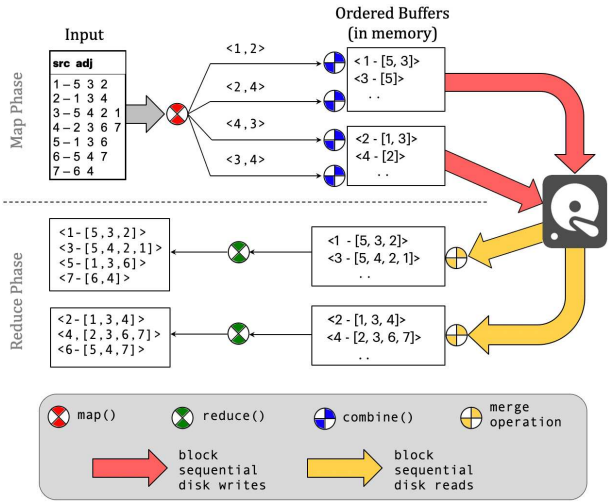

Figure 6. Organization of a Graph as Partitions In Memory.

```
message EdgeType{
  required uint64 src = 1;
  required uint64 dst = 3;

  optional double vRank = 4;
  optional double rank = 5; // of src
  optional uint64 nNbrs = 6; // of src
}

message RecordType {
  required uint64 key = 1;
  repeated EdgeType values = 2;
}
```

Figure 7. Customized attributes for PageRank Algorithm expressed using Protocol Buffers.

note that if memory buffers are large enough to accommodate all the partitions in memory then their contents are not written to disk and the graph processing is also fully carried out in memory. Only the final partitions are written to disk.

The $key - value$ record is defined using protocol buffers. For example, Figure 7 shows the custom *EdgeType* value which contains the information like source, destination, number of neighbors, rank etc for each vertex. This *EgdeType* along with the *key* is passed to the engine when creating the object of the class *Hash* and calling the methods to pass input to the engine within the main method in Figure 4. The $key - EdgeType$ pair is further serialized as *RecordType* by the engine when storing these records on disk.

To parallelize the IO for graph partitioning and processing, we use t threads that read from disk in parallel and create k partitions in parallel. To ensure that the t threads do not have to synchronize with each other when updating the buffers, each buffer is subdivided into t sub-buffers, one for each thread. This leads to the organization of the graph as shown in Figure 5 where the number of in-memory buffers is $t \times k$; t represents all the rows and k represents all the columns. Corresponding to each of $t \times k$ buffers, the disk contains a series of batches that are written to disk when the buffers are emptied.

For a simple Hash partitioner, each partition is read in parallel by its k assigned threads. All threads, in parallel, load as much of the partitioned subgraph of their assigned partition to their respective in memory buffer. The graph processing logic is then applied to the loaded subgraphs in memory. This process continues until the graph processing (e.g. PageRank) logic is applied to the entire graph. OMRGx internally uses *diskReadPartition* and *diskWritePartition* APIs to provide seamless memory managing experience to its users. For sophisticated graph partitioners like GO, MtMetis which also perform the graph refinement step after the initial partitioning, the refinement logic is applied to the subgraphs fetched in memory before carrying out graph processing.

Some of the partitioning algorithms like MtMetis further perform coarsening and uncoarsening steps to refine the graph. *disReadPartition* and *diskWritePartition* APIs make it easier to fetch and store the coarsened subgraphs from/to disk to avoid running out of memory. Our experiments show the implementation of MtMetis using OMRGx API which can even process larger graphs which are not processed by the original standalone implementation of MtMetis. Once the graph processing algorithm runs through all the partitions and updates are applied to all the vertices, the output is written to disk.

The records expressed using Protocol Buffers provide efficient serialization and deserialization. In addition to that, we define *custom records* based on their processing logic making it extremely easier to express wide variety of algorithms which require different number and types of attributes. These attributes are then stored as serialized records on disk. Figure 7 illustrates how different attributes needed by PageRank algorithm are expressed using Protocol Buffers. When reading the records from disk, they are deserialized into these different attributes and processed to get useful information.

2.3 Complex Partitioning & Processing Algorithms

Next we use OMRGx to program complex partitioning and processing algorithms including GO in Figure 8, MtMetis in Figure 9, GridGraph in Figure 10, and GraphChi in Figure 11a. The GO and MtMetis partitioners being sophisticated graph partitioners, refine the partitions during the *reduce* phase and hence perform additional processing. Once all the partitions are read from disk and refined during the *reduce* phase, the final partitions are available at the end which can be used for further processing by the graph algorithms. Since MtMetis employs *multilevel* graph partitioning strategy where a graph is transformed into a sequence of smaller graphs during the *coarsening* phase. The *partitioning* phase generates the initial partitioning of the coarsest graph. The *uncoarsening* phase refines the partitions produced and projects them to their finer level graph and all the way to the original graph. Figure 9 shows the ease of programming *multilevel* algorithm using the simple APIs provided by OMRGx. If the input graph is large, MtMetis stores all its *coarsened* graphs on disk using OMRGx's *diskWritePartition* API by specifying the *count* for the $< key{-}value >$ pairs to be stored on disk. During the *uncoarsening* phase, it simply reads the *finer* level graphs using the *diskReadPartition* API provided by OMRGx, thus, utilizing the disk space to avoid running out-of-memory. Similarly, GO performs the refinement step by computing the gains of the vertices across different partitions using *Kernighan-Lin* algorithm [11]. GridGraph uses a 2-level hierarchical strategy to produce refined partitions.

For implementing the graph processing systems such as GraphChi [15] that uses the shard representation to process large datasets, users have the flexibility to use different data structures to store the shards which are then processed in

```
template <typename KeyType, typename ValueType>
class Go : public MapReduce<KeyType, ValueType> {
 void* reduce(unsigned PID, InMemoryContainer<KeyType,
                ValueType>& partition){
      /* Refine this Partition wrt others */
     foreach(PID, PIDi){ /* Compute Edgecuts */
       ComputeBEdgeCut(PID, PIDi, partition);
       /* Compute Gain using KL Algorithm */
       ComputeGain(PID, PIDi, partition)
     }
 }
};
```

Figure 8. GO programmed in OMRGx.

```
template <typename KeyType, typename ValueType>
class MtMetis : public MapReduce<KeyType, ValueType>
{
 void* reduce(unsigned PID, InMemoryContainer<KeyType,
              ValueType>& partition){
   coarsened_graph = coarsen(PID, partition);
   diskWritePartition(PID, count, InMemoryContainer<KeyType,
                      ValueType>& coarsened_graph);
   init_partition(PID, coarsest_graph);
   foreach(coarsened_graph){
     refine_partition(PID, coarsened_graph);
     /* Reads records specified by count */
     finer_graph = diskReadPartition(PID, count);
   }
 }
};
```

Figure 9. MtMetis programmed in OMRGx.

```
template <typename KeyType, typename ValueType>
class GridGraph : public MapReduce<KeyType, ValueType>
{
 unsigned setPartitionId(unsigned PID const KeyType& key)
    { return -1;}
 void* map(const unsigned tid, std::string& input, unsigned PID){
   std::stringstream inputStream(input);
   std::string token;
   inputStream >> to >> from;

   foreach(to, from){
  /* create edgeblock partitioning based on destination vertex */
     bufferId = hashKey(from[i] % this->getParts();
     writeToBuffer(to, from, PID);
   }
 }

 void* reduce(unsigned PID,
          InMemoryContainer<KeyType, ValueType>& partition){
   /* create access sequence within each active partition */
   foreach(to : partition){
    foreach(from : partition)
     /* Apply Updates on to vertices using PageRank Algorithm */
   }
 }
};
```

Figure 10. GridGraph programmed in OMRGx.

parallel by the OMRGx system. Figure 11 illustrates how we can define simple C++ *structs* to capture the intervals of each shard and to store GraphChi meta data. In addition to the simplicity of programming GraphChi, OMRGx also provide users with the option to program their custom partitioners to partition the input data first and then process them using GraphChi based data structures. Similar to our simple Hash partitioner, GraphChi implements the *map* phase by specifying the $< key - EdgeType >$ pairs. Along with that it also stores the interval information for each shard. It is important to note that in this case each partition (PID) is considered as a separate shard. During the *reduce* phase, parallel

```
template <typename KeyType, typename ValueType>
class GraphChi : public MapReduce<KeyType, ValueType>
{
 void* map(const unsigned tid,
             std::string& input, unsigned PID){
    ..... // same as Figure 3
   /* Add shard's interval info */
   IntervalInfo[tid%this->getParts()].ubEdgeCount =
   edgeCounter;
 }
 void* reduce(unsigned PID,
         InMemoryContainer<KeyType, ValueType>& partition){
      /* Process shard: vs = vertex shard PID = Memory Shard*/
      /* Initialize subgraph for memory shard */
        buildGCD(PID); // GCD =  GraphChi Data
     /* Fetch sliding shards */
     for(i=0; i<nShards; i++)
       vs[i].diskReadPartition(.,.,.);
     /* update vertex in shard */
     double sum = 0.0;
     done = true; // termination condition
     foreach(v in subgraph)
          /* update rank same as Figure 3 */
 }
};
```

(a) map and reduce for GraphChi.

```
typedef struct __intervalInfo
{
  unsigned lbEdgeCount; /* lower bound edge count */
  unsigned ubEdgeCount; /* upper bound edge count */

  unsigned lbIndex;     /* lower bound vertex */
  unsigned ubIndex;     /* upper bound vertex */

} IntervalInfo;
```

(b) Structures used to store the shard information in GraphChi.

```
typedef struct __intervalLengths
{
    unsigned startEdgeIndex;
    unsigned endEdgeIndex;
    unsigned length;
} IntervalLengths;

IntervalLengths **gcd; /* GraphChiData */
```

(c) Structures defined to store meta data.

Figure 11. GraphChi in OMRGx.

reducer threads fetch records from their respective on-disk partitions using *diskReadPartition* API into in-memory partitions which act as a memory shard using which we build a subgraph and create GraphChi meta data for that shard. Once we have all the information about all the shards loaded in memory, we fetch the sliding shards and update each vertex within that shard. The updated shard is written to its on-disk partition using *diskWritePartition* API.

3 Evaluation

OMRGx is implemented in C++ (compiled with gcc version 8) and can be used to *partition* large graphs and perform *partition-based* graph processing. Therefore to evaluate OMRGx we consider a number of partitioning and processing strategies. The detailed evaluation demonstrates OMRGx's

programmability, performance scalability, and its favorable performance in comparison with handcoded custom systems. Following graph partitioning algorithms and partition-based graph processing frameworks are considered.

- *Graph Partitioning in* OMRGx: OMRGx-Hash, OMRGx-GO, and OMRGx-Mt correspond to the Hash, GO, and MtMetis partitioners implemented using OMRGx. We also implemented OMRGx-GC and OMRGx-GG that correspond to partitioning algorithms that produce partitioned representations used by GraphChi [15] and GridGraph [29] out-of-core frameworks.
- *Graph Processing in* OMRGx: OMRGx-D, OMRGx-GC, and OMRGx-GG correspond to the default, GraphChi, and GridGraph processing implementations carried out using OMRGx. We compare these with GraphChi-S and GridGraph-S that refer to standalone custom implementations of GraphChi and GridGraph.

All the experiments were performed on a machine with 32 cores (2 sockets, each with 16 cores) with Intel Xeon Processor E5-2683 v4 processors, 425GB memory, 1TB SATA Drives, and running CentOS Linux 7. The four input graph datasets used in the experiments are listed in Table 2 – they range from medium sized graphs - OK, WK with around $120M-378M$ edges to large sized graphs - TW, UK consisting of $1,202M-3,407M$ edges. UK is the largest graph needing 55GB of disk space. To ensure that out-of-core features were exercised, each run was given less memory that was needed to hold the entire graph.

3.1 Programmability and Versatility of OMRGx

Prototyping complex graph partitioning and processing algorithms using OMRGx's programming interface is quite straightforward as the user can be oblivious of the graph size, memory size, and out-of-core complexities. The programmer may simply need to provide *map* and *reduce* functions which encode the custom logic and data structures that may be needed while the processing engine takes care of the rest.

Table 3 quantifies the programming effort for implementing various partitioning approaches using OMRGx in terms of the lines of C++ code written by the programmer for various partitioning and processing approaches.

A simple hash partitioner can be implemented with a fewer lines of code. It is implemented with around 10 lines of code in OMRGx and is also provided as the default partitioner in OMRGx. While the hash partitioner does not refine the partitions, the OMRGx-GO partitioner refines the partitions during the *reduce* phase requiring additional coding. While the standalone version of GO [14] is implemented using around 1300 lines of code, only 176 lines of code are needed to implement OMRGx-GO. This is because OMRGx's engine does all the heavy lifting. The standalone implementation of MtMetis [17] uses around 22,489 lines of code to implement the multilevel coarsening and uncoarsening algorithms – this implementation is not an out-of-core implementation and hence runs out of memory for large graphs. **On the other hand, OMRGx-MtMetis was implemented using 200 lines of code and it is the first out-of-core implementation of MtMetis that reliably works for large graphs on a single machine.**

Implementing OMRGx-GraphChi was relatively easy and took 123 (40+83) lines of code including code for loading memory with corresponding sliding shards, building subgraph they represent in memory, and processing the subgraph to run *PageRank* algorithm. On the other hand, the standalone implementation of GraphChi [15] is done with around 1323 lines of code – this is the engine that processes the shards. The extra lines of code for preprocessing the shards, building subgraphs and meta data are not included. Similarly OMRGx-GridGraph implementation took 74 lines of code for both partitioning and processing. OMRGx-Default is the default processing provided by OMRGx which uses a hash partitioner to partition the vertex-edge pairs based on their vertex ID and runs PageRank on the partitioned graph. It is implemented using only 29 lines of code.

Table 2. Input Graphs: Orkut (OK), Wikipedia-eng (WK), Twitter-WWW (TW), and UKdomain-2007 (UK).

G	Vertices $\lvert V \rvert$	Edges $\lvert E \rvert$	Graph Size $\lvert E \rvert + \lvert V \rvert$	$\frac{\lvert E \rvert}{\lvert V \rvert}$
OK	3,072,441	117,185,083	120.3 million	38.1
WK	12,150,976	378,142,420	390.3 million	31.1
TW	41,652,230	1,202,513,195	1,244.2 million	28.9
UK	105,153,952	3,301,876,564	3,407.0 million	31.4

Table 3. Programmer written Lines of Code (LoC) for various *partitioning* and *processing* algorithms in *OMRG*x.

Partitioning Algorithm	OMRGx User LoC
OMRGx-Hash	10
OMRGx-GO	176
OMRGx-MtMetis	200
OMRGx-GraphChi	40
OMRGx-GridGraph	32
Processing Algorithm	**OMRGx User LoC**
OMRGx-Default	29
OMRGx-GraphChi	83
OMRGx-GridGraph	42

Table 4. Scalability of PARTITIONING Algorithms Implemented via OMRGx w.r.t Input Graph Size for 8 partitions.

Input Graphs	Total Edges	OMRGx-Hash	OMRGx-GO	OMRGx-MtMetis	OMRGx-GraphChi	OMRGx-GridGraph
OK	117,185,083	16s	34s	55s	15.9s	36.6s
WK	3x	3.5x	3.6x	2.7x	3.5x	2.4x
TW	10x	14x	15x	18x	14.7x	9.8x
UK	28x	32x	19.5x	15x	37.5x	26.5x

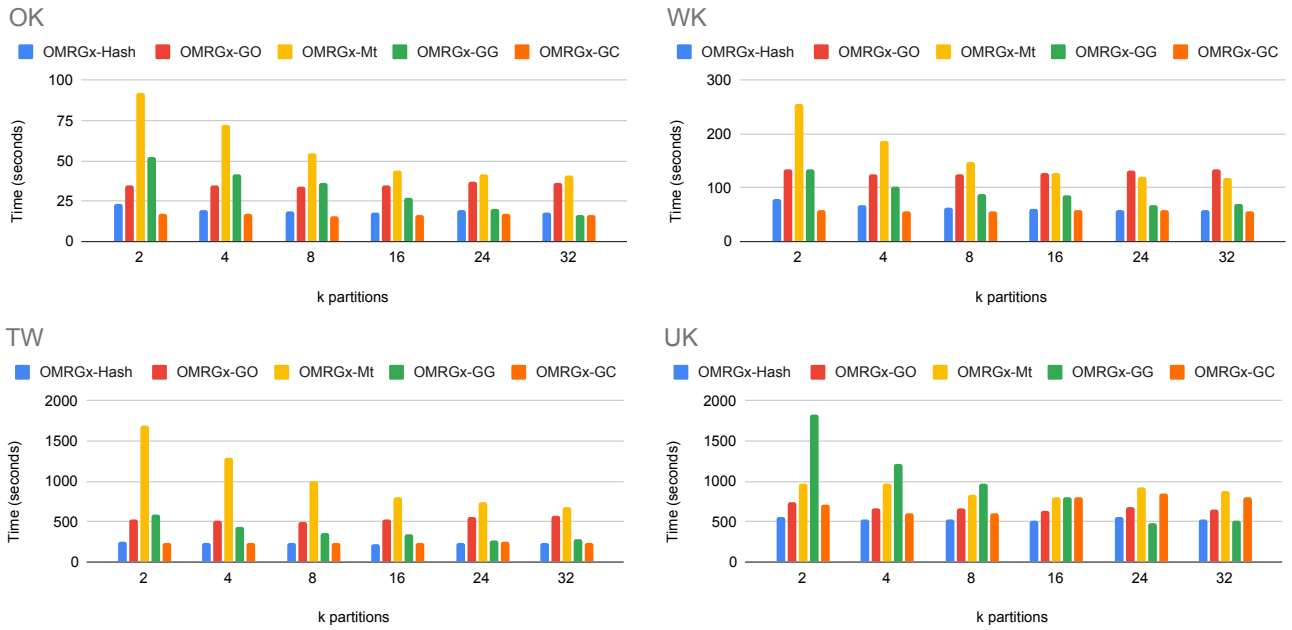

Figure 12. Graph **PARTITIONING** times (in seconds) using different partitioning schemes implemented in OMRGx.

Overall we observe that in 573 lines of code we were able to prototype two sophisticated out-of-core graph partitioners (OMRGx-GO and OMRGx-MtMetis) and two well known out-of-core processing frameworks (OMRGx-GraphChi and OMRGx-GridGraph). Moreover, OMRGx-MtMetis is the first out-of-core implementation of MtMetis.

3.2 Performance of OMRGx-generated Graph Partitioners

Next, we study the scalability of five different graph partitioning algorithms developed using OMRGx. First, we fix the number of partitions to 8 (i.e., $k = 8$) and see how performance scales with increasing graph size. Second, we vary the number of partitions from 2 to 32 to study the sensitivity of performance of partitioning algorithms to k.

Table 4 shows that the cost of partitioning increases in proportion to the increase in input graph size. We use number of edges as a proxy for graph size. As we can see, the number of edges in WK, TW, and UK are 3×, and 10×, 28× relative to the number of edges in OK. When we examine the performance, we see similar trend. For example, UK is 28× bigger than OK while its partitioning time ranges from 15× to 37.5× that of OK across five different partitioners developed using OMRGx. **It is also worth noting that the performance of OMRGx-MtMetis partitioner scales very well according to graph size and this is the first known out-of-core implementation of MtMetis graph partitioner**. Finally, all the graph partitioners were able to process all the input graphs successfully.

Now let us consider the sensitivity of runtime performance to the number of partitions produced. Figure 12 shows the performance of all five partitioners generated using OMRGx when number of partitions is varied from 2 to 32. **We observe a trend that applies to most of the partitioners, as k increases from 2 to 32, the partitioning time for any given partitioner reduces**. This is entirely due to OMRGx execution model that allows partitions to be construced in parallel, in particular, to create k partitions OMRGx uses k threads. Therefore, as we increase k from 2 to 32, greater amount of parallelism is exploited and the time for constructing partitions reduces.

The execution times of different partitioners relative to each other are as expected. The cost of partitioning using OMRGx-Hash, OMRGx-GG (GG-GridGraph), and OMRGx-GC (GC-GraphChi) are reasonably small. The OMRGx-GC takes a bit more time because of the time spent on building the shard representation that is greater in size than a simple adjacency list representation. The most expensive partitioners are OMRGx-GO and OMRGx-Mt (MtMetis) because both perform partition refinement using the KL-algorithm [11]. However, OMRGx-Mt is significantly more expensive than OMRGx-GO due to graph coarsening and uncoarsening that is used which requires multiple versions of the graph to be created, stored, and refined. In other words, OMRGx-Mt performs multilevel refinement while OMRGx-GO performs singlelevel refinement.

Finally, note that very limited work has been carried out on out-of-core graph partitioning. While MtMetis is a popular single machine partitioner, it runs out of memory for large graphs. On the other hand, with modest effort, using OMRGx we have generated performant implementations of five different graph partitioners that successfully handle large graphs while delivering good runtime performance.

Table 5. Scalability of Graph PROCESSING Frameworks implemented via *OMRGx* vs. Standalone GraphChi [15] and GridGraph [29] systems. Runtimes are for PageRank and 8-partitioning of input graphs normalized w.r.t OK.

Input Graph	Total Edges	OMRGx-Default	OMRGx-GraphChi	GraphChi-Standalone	OMRGx-GridGraph	GridGraph-Standalone
OK	117,185,083	18.5s	22.9s	41.1s	4.8s	13s
WK	3x	3.4x	3.3x	2.6x	2.5x	3.5x
TW	10x	13.8x	14x	14.4x	9.6x	5.6x
UK	28x	32x	26x	34x	23.5x	25.9x

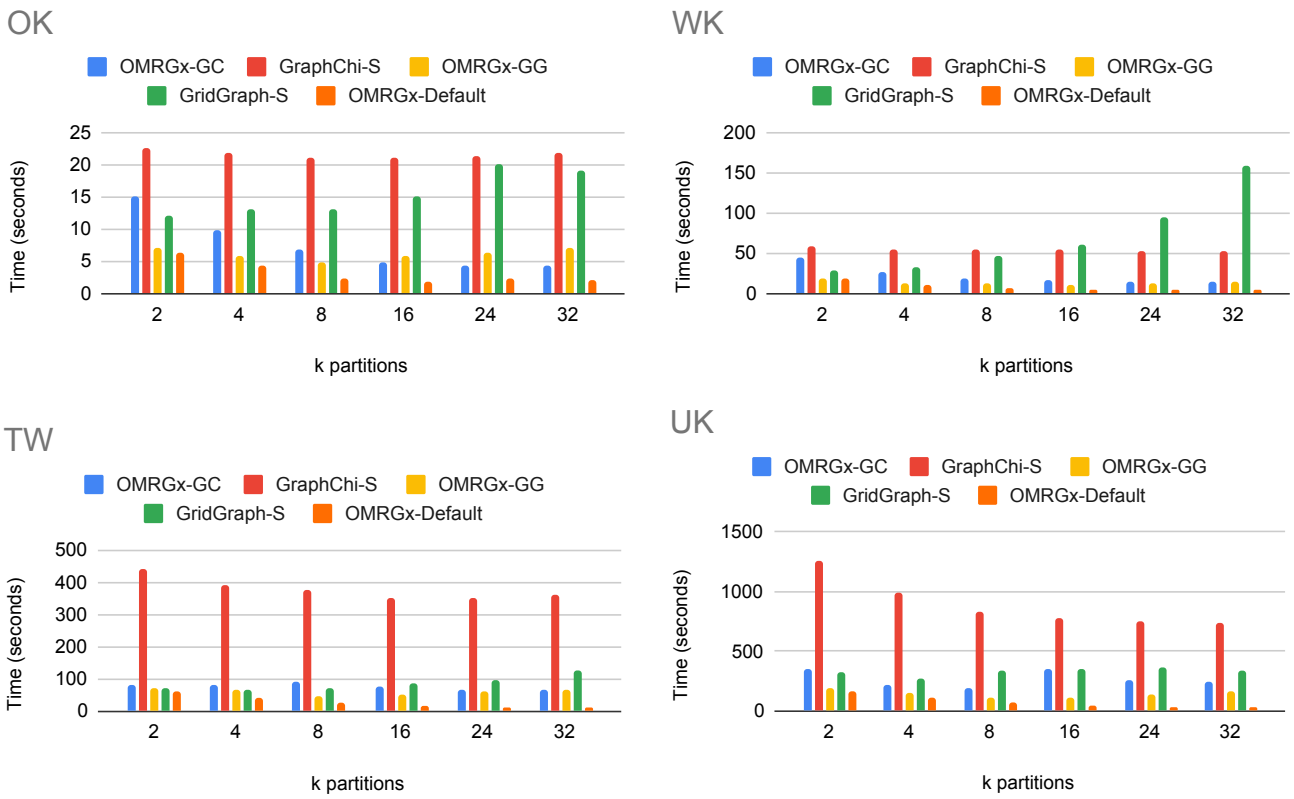

Figure 13. Comparison of PageRank **PROCESSING** Times (in Seconds) for standalone implementations of GraphChi-S and GridGraph-S with their corresponding and default implementations in OMRGx.

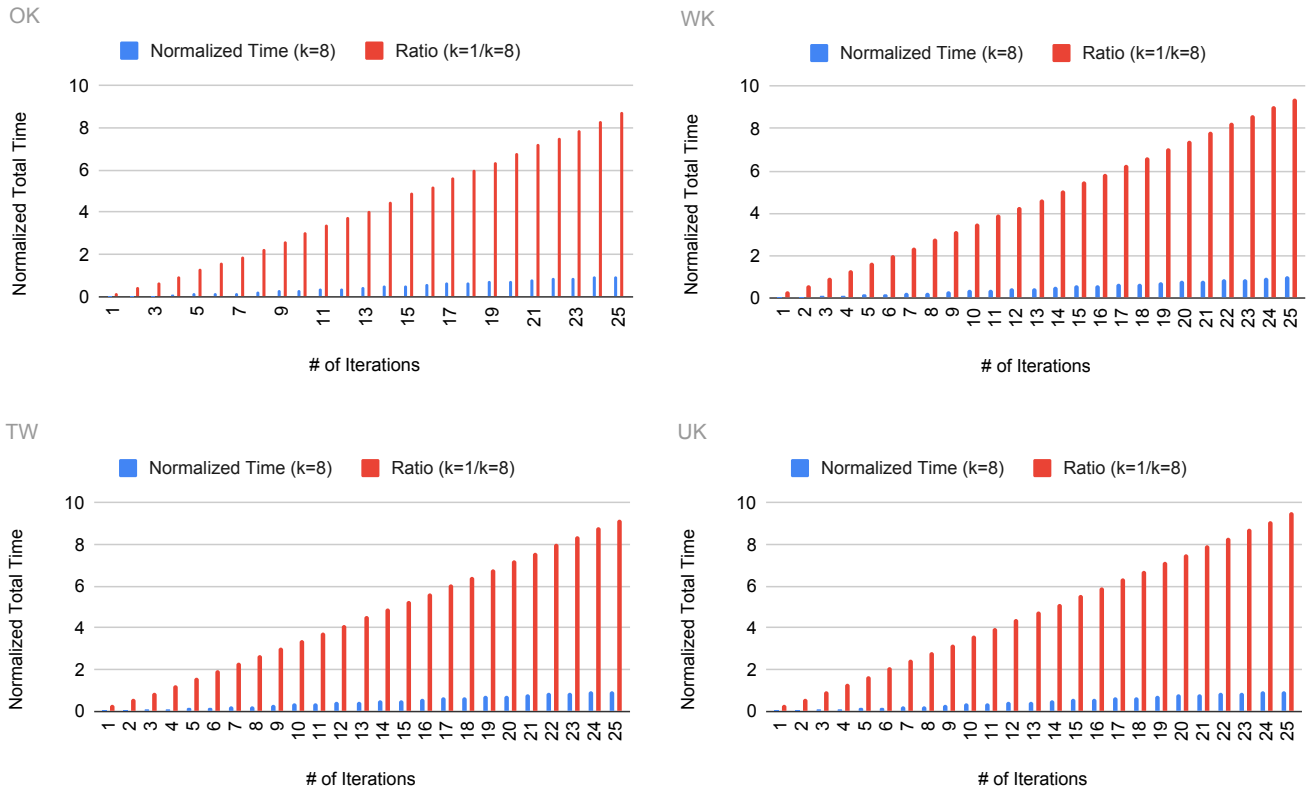

Figure 14. Benefit of Processing Partitions in Parallel in OMRGx Memory Available for Buffers is Half the Graph Size. All times are normalized with respect to total time taken by OMRGx with k=8 and 25 iterations.

3.3 Performance of OMRGx-generated Graph Processing Frameworks

Next, we compare the runtime performance of standalone GraphChi [15] and GridGraph [29] frameworks with their corresponding implementations that were created using OMRGx. We also compare the performance of these implementations with the OMRGx-Default version. For comparing the graph processing runtime costs, we ran the PageRank algorithm on all four input graphs using all of the above five implementations. All of the above implementations are synchronous; thus, they perform the same work in each iteration. We ran all versions for first few iterations as for PageRank the initial iterations are the most expensive.

Table 5 studies the scalability of all implementations with respect to the input graph size using 8 partitions. Here the runtimes of the all the systems are normalized w.r.t OK. While the UK graph size is 28× the size of OK, the runtimes of different versions for UK are 25.9× to 34× that of runtimes for OK. In other words the processing times increase less proportionally to the graph size indicating good scalability.

We also studied the sensitivity of processing times to number of partitions. The runtimes for all the systems, OMRGx-generated and standalone systems, are shown in Figure 13 as number of partitions is varied from 2 to 32. We observe that the processing runtimes of any given approach using OMRGx is less insensitive to number of partitions, though runtimes for $k > 2$ are typically lower than for $k = 2$. We also observe that OMRGx generated implementations are superior to standalone implementations. In other words, OMRGx-GC and OMRGx-GG perform better than well-tuned and hand-optimized GraphChi-S and GridGraph-S systems. This is due to the parallel processing of all partitions by OMRGx based systems as opposed to standalone systems that process one partition at a time. It should be noted that GridGraph-S performance is significantly superior to GraphChi-S and the same is true for their corresponding OMRGx-GG and OMRGx-GC. **We also observe that OMRGx-Default implementation that is essentially based upon the simple default hash partitioning performs the best overall.** It is substantially faster than both OMRGx-GC and OMRGx-GG as well as stand alone implementations of GraphChi and GridGraph.

As mentioned earlier, OMRGx benefits from processing multiple partitions in parallel. In order to show the benefit achieved by processing partitions in parallel, we ran 25 iterations of PageRank and compared the processing time for running a single partition with running 8 partitions in parallel. Figure 14 shows how the normalized cumulative

execution time grows with iteration count – in each of these runs we provided memory equal to half the size of the input graph. As expected, the runtime shows a significant improvement in overall execution time (around 9×) when processing multiple partitions in parallel. This experiment also explains the reason for the faster execution times for OMRGx-GC and OMRGx-GG when compared to standalone implementations of GraphChi and GridGraph that process one partition at a time. Both OMRGx-GC and OMRGx-GG benefit from the loading and processing of partitions in parallel by the OMRGx execution engine and hence achieve speedups over the standalone implementations of GraphChi and GridGraph.

4 Related Work

OMRGx is a general system that can be used for prototyping new and old strategies for partitioning and processing. It is the *only* out-of-core graph partitioning and processing system that simultaneously constructs and simultaneously processes multiple partitions. This is why the implementations of existing strategies in OMRGx deliver higher performance than their standalone implementations [14, 15, 29] as it hides the I/O times by processing partitions in parallel. It is well known that out-of-core systems spend vast majority of their time on disk I/O. Therefore, much of the speedups we have demonstrated are due to parallel I/O, i.e. simultaneous loading of multiple partitions.

APIs extensions to MapReduce- Mapreduce by itself does not provide any support for graph partitioning or processing. Our APIs extend the MapReduce system [13] to allow prototyping of graph partitioning and processing strategies.

Graph Partitioning- OMRGx enables prototyping of out-of-core graph partitioning techniques that have high memory demands (e.g., METIS [9] requires memory several times the size of the graph). In our prior work we developed the GO [14] out-of-core partitioner. The approach taken there has been incorporated in OMRGx so that not only GO, but any other out-of-core graph partitioner can be transparently programmed using OMRGx.

Graph Processing- Existing out-of-core systems [15, 29] make fundamental assumption that only one partition at a time will be processed, and all threads will process that partition in parallel. The partitions are created so each one will fit in available memory. OMRGx does not have that limitation. It is designed to handle multiple buffers of fixed size that cannot hold the entire partition, but rather stream each partition through its buffer. This approach allows OMRGx to simultaneously process different partitions. It is important to note that since OMRGx derives speedups via streaming of edges from multiple partitions in parallel, a standalone system such as X-Stream [21] that also streams edges may have performance comparable to its prototype developed via OMRGx.

5 Conclusions

In this paper we presented a Map-Reduce based out-of-core system with a programming interface using which the user can rapidly prototype variety of graph partitioning and graph processing frameworks. The complexities of limited memory management, parallel computation, and parallel I/O are hidden from the user allowing programming to be carried out oblivious to the relative sizes of the graph and machine memory. Our evaluation showed that OMRGx derived implementations of existing frameworks, namely GraphChi and GridGraph, outperform their corresponding hand coded and optimized implementations. This is because OMRGx ability to simultaneously create and simultaneously processes all partitions. Using OMRGx we created the first out-of-core implementation of the popular MtMetis [17] graph partitioner. Finally, using 573 lines of C++ code we were able to prototype two sophisticated out-of-core graph partitioners (GO and MtMetis) and two well known out-of-core processing frameworks (GraphChi and GridGraph). Moreover, OMRGx generated MtMetis implementation is the first out-of-core implementation of popular MtMetis [17].

ACKNOWLEDGEMENTS

This work is supported in part by National Science Foundation grants CCF-1813173, CCF-2028714, CCF-2002554, and CCF-2226448 to the University of California Riverside.

References

[1] T. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, pages 153-159, 1992.

[2] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing*, 5(3), 13, 2019.

[3] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, (16):498-513, 1987.

[4] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[5] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[6] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Technical Report SAND93-1301*, Sandia National Labs, 1993.

[7] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *ACM/IEEE Conference on Supercomputing*, 1995.

[8] G. Karypis, and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. In *Journal of Parallel Distributed Computing*, 48(1):96-129, 1998.

[9] G. Karypis, and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359-392, Dec. 1998.

[10] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *ACM/IEEE Conference on Supercomputing*, 1996.

[11] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49: 291–307, 1970.

[12] S-C. Koduru, R. Gupta, and I. Neamtiu. Size oblivious programming with InfiniMem. In *Workshop on Languages and Compilers for Parallel Computing*, pages 3–19, 2016.

[13] G. Kaur, K. Vora, S-C. Koduru and R. Gupta. Out-of-Core MapReduce For Large Data Sets. In *Proceedings of 2018 ACM SIGPLAN International Symposium on Memory Management*, ACM, NY, USA, 2018.

[14] G. Kaur and R. Gupta. GO: Out-of-Core Partitioning of Large Irregular Graphs. In *15th IEEE International Conference on Networking, Architecture and Storage*, IEEE NAS 2021.

[15] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 31–46, 2012.

[16] J. Leskovec. "Stanford large network dataset collection," *http://snap.stanford.edu/data/index.html*, 2011.

[17] D. LaSalle, and G. Karypis. Multithreaded Graph Partitioning. In *IEEE International Parallel and Distributed Processing Symposium*, 2013.

[18] D. LaSalle, Md M. A. Patwary, N. Satish, N. Sundaram, G. Karypis and P. Dubey. Improving Graph Partitioning for Modern Graphs and Architectures. In *Workshop on Irregular Applications: Architectures and Algorithms*, 2015.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *Proceedings of the VLDB Endowment 5*, 8 (2012), 716-727.

[20] G. Malewicz, M.H. Austern, A.J.C Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD International Conf. on Management of Data*, pages 135-146, 2010.

[21] Roy, Amitabha and Mihailovic, Ivo and Zwaenepoel, Willy. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[22] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *ACM Symposium on Operating Systems Principles*, pages 456-471, 2013.

[23] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *International Conference and Exhibition on High-Performance Computing and Networking*, pages 493-498, 1996.

[24] P. Sanders and C. Schulz. Distributed evolutionary graph partitioning. *CoRR*, vol. abs/1110.0477, 2011.

[25] J. Shun and G. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 135-146, 2013.

[26] K. Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *USENIX Annual Technical Conference*, pages 429-442, 2019.

[27] K. Vora, G. Xu, and R. Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *USENIX Annual Technical Conference*, pages 507–522, 2016.

[28] C. Walshaw and M. Cross. Parallel Optimization Algorithms for Multilevel Mesh Partitioning, In *Parallel Computing*, 26(12):1635-60, 2000.

[29] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX Annual Technical Conference*, pages 375–386, 2015.

[30] Konect: http://konect.cc/networks/

[31] Protocol Buffers. Google's data interchange format, https://developers.google.com/protocol-buffers

Received 2023-03-03; accepted 2023-04-24