GO: Out-Of-Core Partitioning of Large Irregular Graphs

Gurneet Kaur
CSE Dept., UC Riverside

Rajiv Gupta
CSE Dept., UC Riverside

2021 IEEE International Conference on Networking, Architecture and Storage (NAS) | 978-1-7281-7744-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/NAS51552.2021.9605433

***Abstract*—Single-PC, disk-based processing of large irregular graphs has recently gained much popularity. At the core of a disk-based system is a static graph partitioning that must be created before the processing starts. By handling one partition at a time, graphs that do not fit in memory are processed on a single machine. However, the multilevel graph partitioning algorithms used by the most sophisticated partitioners cannot be run on the same machine as their memory requirements far exceed the size of the graph. The popular memory efficient Mt-Metis graph partitioner requires 4.8× to 13.8× the memory needed to hold the entire graph in memory. To overcome this problem, we present the GO out-of-core graph partitioner that can successfully partition large graphs on a single machine. GO performs just two passes over the entire input graph, *partition creation pass* that creates *balanced* partitions and *partition refinement pass* that reduces *edgecuts*. Both passes function in a memory constrained manner via disk-based processing. GO successfully partitions large graphs for which Mt-Metis runs out of memory. For graphs that can be successfully partitioned by Mt-Metis on a single machine, GO produces balanced 8-way partitions with 11.8× to 76.2× fewer edgecuts using 1.9× to 8.3× less memory in comparable runtime.**

***Index Terms*—irregular graphs, out-of-core processing, multi-level graph partitioning**

I. Introduction

Graph analytics is employed in many domains to gain insights by analyzing large graphs representing entities and interactions among them. Real world graphs often contain millions of vertices and billions of edges (see Table I), and iterative graph analytics queries require multiple passes over the graph until convergence. Therefore, there has been a great deal of interest in developing scalable graph analytics systems that exploit parallelism available on distributed systems [2], [4], [16], [17]) as well as shared memory systems [18], [21].

Before parallel or distributed analytics on large graphs can be performed, they typically must first be partitioned. In context of distributed systems the graph is partitioned across multiple machines such that each machine is primarily responsible for computations that operate on its assigned partition. In context of a single shared-memory machine, the graph is partitioned to enable *out-of-core* or disk-based processing of a large graph on a single machine [12], [22], [23], [25]. When the graph is too large to fit in the memory available on the machine, it is divided into smaller partitions and stored on disk. This enables partitions to be loaded one at a time into memory and processed.

TABLE I: Input Graphs of Varying Sizes: Flickr (FL), PokeC (PK), LiveJournal (LJ), Orkut (OK), UKdomain2002 (UK02), Wikepedia-eng (WK), Twitter-WWW (TW), Twitter-MPI (TM), and UKdomain-2007 (UK07). [13], [26]

Graph G	Vertices $\mid V \mid$	Edges $\mid E \mid$	Graph Size $\mid E \mid + \mid V \mid$	$\frac{\mid E \mid}{\mid V \mid}$
FL	1,715,255	15,551,249	17.3 million	9.1
PK	1,632,803	30,622,564	32.3 million	18.7
LJ	4,036,537	34,681,189	38.7 million	8.6
OK	3,072,441	117,185,083	120.3 million	38.1
UK02	18,520,486	261,787,258	280.3 million	14.1
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
TM	999,999,987	1,614,106,343	2,614.1 million	1.6
UK07	105,153,952	3,301,876,564	3,407.0 million	31.4

For superior performance, partitioning algorithms endeavor to create partitions that are well balanced and minimize edgecuts (i.e., number of edges that cross partition boundaries).

Although the problem of graph partitioning is known to be NP-hard [1], highly effective *multilevel graph partitioning* algorithms have been developed and are widely used [5]–[8], [24]. A large graph goes through a series of coarsening phases which, by merging vertices, produces significantly smaller graphs at each subsequent level. Once a sufficiently small graph is obtained, it is partitioned. Next this partitioning is projected to the coarsened graph at the next level to obtain its partitioning that is further refined using the Kernighan-Lin algorithm [10] to reduce edgecuts. This preceding step is repeated through the levels eventually producing a partitioning for the original full graph. Many implementation frameworks for multilevel graph partitioning are available for distributed systems (e.g., ParMetis [9], Scotch [19], KaFFPa [20]). These frameworks enable *end-to-end processing* of large graphs on distributed systems as both partitioning and subsequent analytics tasks can be performed on the same distributed platform.

While a framework that implements multilevel graph partitioning on a single shared-machine, called Mt-Metis [14], [15], is also available, it does not enable *end-to-end* processing of large graphs on a single machine because Mt-Metis requires that not only the input graph be held in memory but also the coarsened graphs. Since the out-of-core processing of large graphs is required in the first place because the entire graph does not fit in memory, executing Mt-Metis to partition such a graph fails as it runs out of memory. Therefore current out-of-core graph processing systems employ very simple partitioning

978-1-7281-7744-1/21/$31.00 ©2021 IEEE

schemes that simply distribute vertices among partitions [22]. Consequently, end-to-end processing of large graphs on a single machine using sophisticated partitioners is an open problem.

We present GO, an Out-of-core Graph Partitioner, that given a fixed amount of memory on a machine, successfully partitions large graphs that cannot be held in the given amount of memory. GO performs just two passes over the entire input graph, the *partition creation pass* that creates *balanced* partitions and the *partition refinement pass* that reduces *edgecuts*. Both passes are designed to function in a memory constrained manner. During the *partition creation* phase parallel threads read the graph from disk and assign vertices, along with their adjacency lists, to different partitions. Once the available memory is full, the subpartitions created thus far are written to disk to free up the memory. Thus, the threads can resume reading the remainder of the graph from disk and partitioning it. This process produces an initial partitioning that resides on disk. Next the *partition refinement* phase reads portions of all partitions into memory, refines these subpartitions against each other using the KL-algorithm [10], and maintains the refined partitioning. This process is repeated until entire initial partitioning has been read and processed to generate the final refined partitioning.

Since GO performs partitioning without creating coarsened graphs, it is a *single level* partitioner with greatly reduced memory requirement. In other words, given a graph that can be held in available memory, GO can partition the graph without requiring the initial partitioning to be written to and then read back from disk during refinement. That is, entire partitioning can be performed in-memory. On the other hand, since Mt-Metis requires additional memory to hold coarsened graphs, total memory that it requires to hold all graphs is several times (4.8× to 13.8×) the memory needed to simply hold the original graph. Thus, Mt-Metis cannot partition the graph successfully even if enough memory is available to hold the entire input graph in memory. The quality of partitions produced by GO were found to be superior to those produced by Mt-Metis both in the terms of *balance* and *edgecuts*. This is due to careful design of initial partitioning algorithm and our modified application of KL-algorithm.

Our experiments with GO prototype on nine input graphs of varying sizes show that GO can successfully partition large graphs for which Mt-Metis runs out of memory on the machine used. For graphs that can be successfully partitioned by **Mt-Metis** on a single machine, **GO** produces balanced 8-way partitions with 11.8× to 76.2× fewer edgecuts using 1.9× to 8.3× less memory and comparable runtime. To further compare the quality of partitions produced by GO and Mt-Metis, we used these partitions to execute PageRank and Weakly Connected Components workloads on the GridGraph [25] disk-based processing system. Running times of GridGraph using GO partitions were always found to be less than those using Mt-Metis partitions.

II. Limitation of Multilevel Partitioning

Given a graph $G = (V, E)$, the goal of *k-way* graph partitioning is to partition V into k non-empty disjoint subsets $V_1, V_2, \cdots, V_k$. In general all vertices and edges in the graph have a weight but to simplify the discussion let us assume all weights are one. The quality of resulting partition is measured by its *balance* and *edgecuts*. Balance is defined as $k\frac{MAX_i(|V_i|)}{|V|}$, which is ideally 1, and when it exceeds 1, the greater the value the greater is the degree of imbalance. Edgecuts corresponds to the number of edges that connect vertices from different partitions. A partitioning with lower edgecuts is preferred.

The multilevel graph partitioning scheme employed by Mt-Metis [14], [15] has three phases. The *Coarsening Phase* transforms the given graph G into a sequence of smaller graphs $G_1, G_2, \cdots, G_L$ such that $|V| > |V_1| > |V_2| > \cdots > |V_L|$. The *Partitioning Phase* generates k-way partitioning P_L of G_L. The *Uncoarsening Phase* projects P_L back to partitioning P of G going through intermediate partitions $P_{L-1}, P_{L-2}, \cdots, P_1$. A detailed evaluation of Mt-Metis carried out by Lasalle and Karypis [14] shows that in comparison to ParMetis [9] and Scotch [19] which are both distributed implementations, the memory requirements of Mt-Metis is significantly lower. In fact, the memory requirements of Mt-Metis are only slightly higher than the serial implementation KMetis [7] – the extra memory is needed primarily for thread local data structures and it increases with the number of threads. Moreover, Mt-Metis is optimized to work for irregular graphs [15]. Table III compares Mt-Metis and KMetis in terms of edgecuts, peak memory used, and execution times on $425GB$ machine.

However, multilevel algorithms have a high memory requirement because in addition to holding G in memory, the coarsened graphs $G_1, G_2, \cdots, G_L$ must also be held in memory. While the original graph size is $|E| + |V|$, the cumulative graph size of coarsened graphs is $\sum_{i=1}^{L} |E_i| + |V_i|$. Thus, the combined size of the original graph and the coarsened graphs is $1 + \sum_{i=1}^{L} \frac{|E_i|+|V_i|}{|E|+|V|}$ times the size of the original graph.

We collected the values of this ratio for the sample graphs in Table I by running Mt-Metis and the results are presented in Table II. All experiments in this paper were performed on a machine with 32 cores (2 sockets, each with 16 cores) with Intel Xeon Processor E5-2683 v4 processors, 425 GB memory, 1TB SATA Drives, and running CentOS Linux 7. We observe that the ratio varies from **4.8**× to **13.8**× for the first six graphs. The number of levels of coarsening L is also given for each run. We could not collect the data for the three largest graphs because, on the machine used, Mt-Metis ran out of memory.

Next we present GO, a two-phase algorithm that does not employ multilevel partitioning, and can partition the larger graphs even when provided with memory less than what is needed to hold the original graph.

III. GO: Out-of-core Graph Partitioner

Given a limited amount of memory that cannot even hold the input graph in adjacency list format, GO uses the given memory to form memory buffers that are used by multiple threads to perform parallel graph partitioning in two phases: *Initial Partition Creation*; and *Partition Refinement to Create Final Partitions*. Both phases are designed to function in *memory*

TABLE II: Given Original Graph of Size $|E| + |V|$ and ($+L$) Coarsened Graphs Generated by Mt-Metis: Ratio is the times by which Cumulative Graph Size of Mt-Metis is Greater than the Original Graph Size.

Input Graph	**Ratio**: $1 + \sum_{i=1}^{L} \frac{\|E_i\|+\|V_i\|}{\|E\|+\|V\|} \times$ **Coarsened Graph Levels**: (+L)											
	k=2		k=4		k=8		k=16		k=24		k=32	
FL	9.8×	(+6)	11.5×	(+8)	12.2×	(+9)	12.9×	(+10)	13.5×	(+11)	13.5×	(+11)
PK	6.0×	(+4)	6.9×	(+5)	8.3×	(+7)	8.2×	(+7)	8.9×	(+8)	8.9×	(+8)
LJ	10.9×	(+5)	13.0×	(+7)	12.9×	(+7)	13.7×	(+8)	13.7×	(+8)	13.8×	(+8)
OK	9.3×	(+5)	10.2×	(+6)	10.1×	(+6)	10.9×	(+7)	10.1×	(+6)	10.1×	(+6)
UK02	4.8×	(+6)	4.9×	(+8)	4.9×	(+8)	4.9×	(+9)	4.9×	(+9)	4.9×	(+9)
WK	7.0×	(+6)	8.0×	(+8)	7.9×	(+8)	8.2×	(+9)	8.2×	(+9)	8.2×	(+9)
TW, TM, UK07	✗		✗		✗		✗		✗		✗	

TABLE III: Comparison of serial implementation KMetis with the multithreaded Mt-Metis in terms of the number of Edgecuts, Memory Consumption (GB) and Execution Time (sec). 8 partitions produced for each input graph on a machine with $425GB$ main memory.

Input Graph	KMetis			Mt-Metis		
	Edgecuts	**Mem**	**Time**	**Edgecuts**	**Mem**	**Time**
FL	3,371,075	1.70	18.0	3,974,999	2.30	6.0
PK	4,351,130	2.20	25.0	4,196,212	4.00	8.0
LJ	7,377,230	4.5	58.0	7,563,933	6.4	15.0
OK	24,257,372	11.10	103.0	24,163,859	22.40	32.0
UK02	2,107,793	10.4	62.0	2,241,490	15.4	34.0
WK	45,803,435	25.5	269.0	44,993,565	55.1	121.8

constrained manner, i.e. they successfully execute using the given memory that cannot hold the large input graph.

Figure 1 provides an overview of GO. The input graph is stored on disk in adjacency list format. The available memory is organized as multiple in-memory buffers. During the first phase threads read the graph from disk in parallel, assigning vertices to create balanced initial partitions and accumulating their adjacency lists in the same buffers. When a buffer is full, it is written to disk so that processing of rest of the graph can continue. At the end of the first phase, the initial partitions are created and each partition is written to disk (Infinimem object store [11]) as an ordered sequence of batches. In the second phase once again the in-memory buffers are created, one for each partition. Portions of partitions from Infinimem are read into these buffers until the buffers are full, and then the in-memory portions of the partitions are refined against each other. Once initial partitions of the graph have gone through the in-memory buffers, the final refined partitions are available and written to the disk. Very high degree vertices are treated differently from other vertices during the above phases to obtain partitioning with good balance and low edgecuts.

Note that if the buffers are large enough to accommodate the entire graph during the first phase, then the initial partitions are not written to disk and re-read for refinement. Instead the refinement is also fully carried out in memory and final partitions are finally output to the disk. In subsequent subsections we present each of these phases in greater detail.

A. Memory Constrained Initial Partition Creation

For k-way partitioning of graph G, the memory available to the GO partitioner is divided into k buffers $B_1, B_2 \cdots B_k$, one for each partition. The graph is read using blocked serial reads from disk. Each source vertex v is assigned to some partition P_i, and the vertex v and its adjacency list $Adj(v)$ are stored in buffer B_i. By using a simple hashing function for assigning partitions to source vertices, a *balanced* partitioning is ensured. The above process is repeated as long as there is room in the buffers to accommodate more of the graph. Once some buffer B_* is full, its contents are written to the disk, that is, B_* is emptied and processing of the graph is resumed. Emptying of B_* is referred to as writing of a *batch* to disk. When the entire graph has been processed, all buffers are emptied and the partitioned graph is available on disk. On disk the graph is now organized according to *partitions*, where each partition is made up of series of *batches*. Consequently, after this phase, the next phase (refinement) can read each partition using blocked serial reads of its batches from the disk. While the partitioned graph is written to disk, an auxiliary in-memory PID array, indexed by vertex id, remembers the partition ids of all vertices.

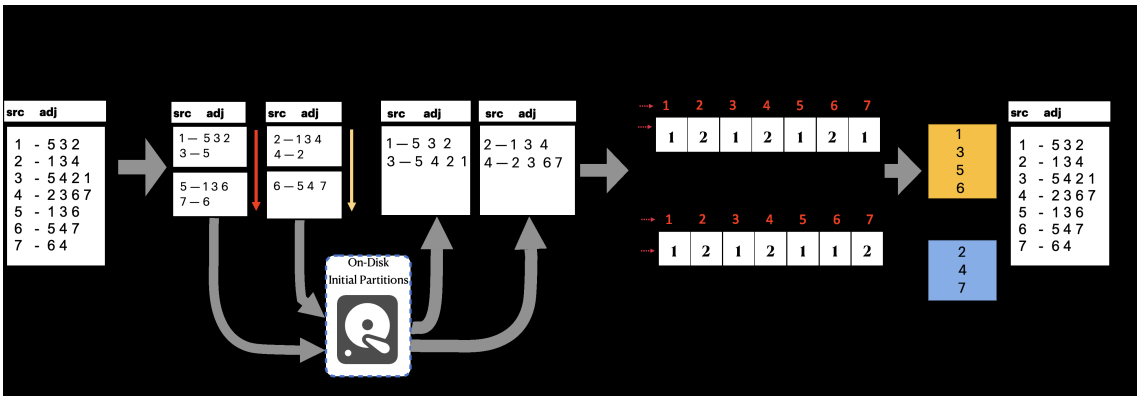


Fig. 1: Overview of Out-of-Core GO Graph Partitioning.

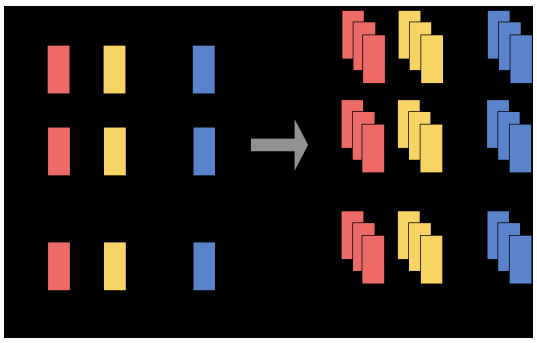
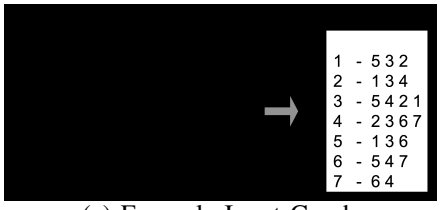


Fig. 2: Organizing Memory into Buffers and Disk Usage.

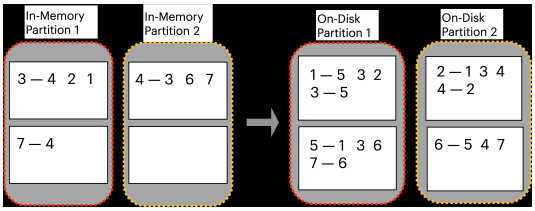
Parallelization: To parallelize the I/O and processing, we use t threads that read from disk in parallel and create the partitions in parallel. To ensure that the t threads do not have to synchronize with each other when updating the buffers, each buffer is subdivided into t sub-buffers, one for each thread. This leads to the organization of the graph as shown in Figure 2 where the number of in-memory buffers is $t \times k$. 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.

Vertex-Ordered Representation: Note that the source vertices in the original graph, and corresponding adjacency lists, are organized on disk in the order of source *vertex ids*. Thus, they are read in the order of source vertex ids by the above partitioning phase, they appear in the order of source vertex ids in B_*'s, and thus appear in order inside each batch written to disk. To illustrate this phase let us consider the example shown in Figure 3. In Figure 3(a) an example graph and its adjacency list representation is shown. Let us assume that we are carrying out 2-way partitioning using 2 threads and hence the memory is organized into 4 buffers. Figure 3(b) shows the representation of the graph where all of the graph has been read and processed, part of it has been written to the disk in its batched partition form while part resides in memory buffers. Here each buffer has been emptied once. Once the buffers are emptied, the completed initial partitioning of the graph that resides on the disk is shown in Figure 3(c). Note that the contents of in-memory buffers and on-disk batches are all ordered according to vertex ids of source vertices in them.

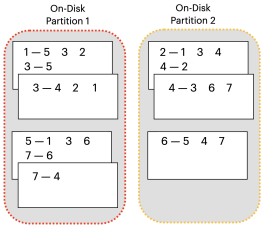
Split Adjacency Lists for Irregular Graphs: In out-of-core processing by systems such as GraphChi [12] and GridGraph [25], the graph partitions are represented as a subset of edges from the graph. The edges present in the subgraph corresponding to a partition are essentially contained in the partition's representation. Balancing edges across partitions is important because the work performed by analytics tasks (e.g., single source shortest paths) is proportional to the number of edges. Since power-law graphs, due to their irregular skewed degree distribution, contain vertices with very high degrees, we obtain a balanced partitioning by distributing the edges incident to such vertices across the partitions. This can be carried out simply by adding a threshold parameter δ such that when source vertex v is processed by a thread, after placing δ edges in its current partition, the partition is changed causing



(a) Example Input Graph.



(b) Ongoing Partitioning: In-Memory (left); & On-Disk (right).



(c) Completed Initial Partitioning.

Fig. 3: Representation of example graph in memory and on disk where $k = 2$ and $t = 2$.

the next δ edges to be placed in a different partition. This approach enables the edges for vertex v to be split across all k partitions in batches of δ.

Note that it is also possible for the adjacency list of some vertex v to get split because the buffer to which the list is being written to has become full. This can be observed in Figure 3(c) where vertex 7 is assigned to partition 1 and its adjacency list consisting of two vertices, 6 and 4, are split across two consecutive batches on disk. However, note that in this kind of splitting, the adjacency list is not split across partitions but rather it is split across batches that belong to the same partition. Moreover, as we will show later, when partitions are read from their batches in the refinement phase, these split adjacency lists will be merged together.

B. Memory Constrained Partition Refinement

For a k-way partitioning, the refinement phase is executed in parallel by k threads where each thread is assigned the task of refining a single partition. The memory available is divided into k buffers of equal size. All threads (T_is), in parallel, load as much of the subgraphs (wP_is) of their respective assigned initial partitions (P_is) into their buffers ($Buffer(T_i)$s). After refining the loaded wP_i's against each other, the buffers are loaded again and refined. This process continues until the entire graph has been refined and final partitioning has been found.

Parallel Loading: When buffer $Buffer(T_i)$ is loaded from partition P_i the vertices assigned to P_i are loaded in increasing order of vertex id. This is achieved by performing a merge sort across batches written to P_i by different threads.

Note that when vertices and their adjacency lists are loaded into $Buffer(T_i)$ in this fashion, if the adjacency list of a vertex v in P_i had been split across different batches due to emptying of filled buffers in the initial partitioning phase, then the spilt adjacency list of v will get merged during loading. Consequently, after loading, the contents of $Buffer(T_i)$ are organized as follows. A low degree vertex v_l that is assigned to a unique partition P_i, appears in $Buffer(T_i)$ along with $Adj(v_l)$, its *complete* adjacency list. A high degree vertex v_h whose edges are spread across all k partitions appears in each $Buffer(T_*)$ along with the *complete* subset of its adjacency list assigned to each partition P_*.

Refinement: The goal of refinement using the Kernighan-Lin (KL) algorithm [10] is to reduce edgecuts in the existing partitioning by swapping pairs of vertices between partitions. Doing so does not alter the *balance* of the partitions that is achieved during the initial partitioning phase, the swap operations are chosen merely to reduce edgecuts. The initial partitioning is held in memory in PID[*] array where using vertex id to index the array, we can read the partition id of the vertex. When swap operations are applied, PID[*] contents are modified to reflect the change in partitioning.

Given a pair of vertices (v, w) from two different partitions, PID$[v]$ and PID$[w]$, the decision to swap v and w between the two partitions is based upon the extent to which the swap reduces edgecuts, which is also called the GAIN. The *external cost* of v with respect to PID$[w]$ (i.e., $EC(v)\backslash PID[w]$) is the number of edges from v to vertices in PID$[w]$. The *internal cost* of v, $IC(v)$, is the number of edges from v to vertices in PID$[v]$. If the difference between external and internal costs of v and w, $D(v)$ and $D(w)$, are both positive, then swapping of the vertices will definitely reduce edgecuts. The GAIN is the sum $D(v)$ and $D(w)$ if v and w are not directly connected by an edge; otherwise it is $D(v) + D(w) - 2$.

When refining a pair of partitions against each other, all pairs of vertices are considered and the one that provides the highest Gain are chosen for swapping. This process is repeated – each application is called a *Pass* that finds one pair to swap – until no more pairs with positive Gain are available.

Taming KL-algorithm's Complexity for Large Graphs: In large graphs with millions of vertices in each partition, it is not practical to consider every pair of vertices from every pair of partitions and consider them for swapping. To reduce the complexity we first observe that only *boundary vertices* – vertices that have edges cut by the initial partition – for a pair of partitions need to be considered during refinement. To further lower the complexity of refinement we limit the scope of refinement by dividing boundary vertices belonging to each partition into equal-sized NUMINTS *intervals*. When refining two partitions with respect to each other, only all pairs of vertices from corresponding intervals in the two partitions are considered. This greatly reduces the pairs considered and hence the cost of refinement. Note that high-degree vertices are not included in such pairs as, having partitioned their edges across partitions, their is no benefit from swapping them. The low degree vertices are plentiful in a large irregular graph and hence refinement of intervals that are smaller subgraphs is still highly effective as will be observed from our experimental results for GO in comparison to Mt-Metis.

Interval-based Algorithm and Example: Algorithm 1 presents the complete algorithm of our refinement phase. Lines 4-15 show threads loading portions of partitions into buffers in parallel, identifying boundary vertices, dividing them into intervals, and calling REFINEPARTITION to refine partitions. Lines 16-33 show the function including how a partition refines

Algorithm 1 Interval-based KL Algorithm

```
1:  P_1 ··· P_k – Subgraphs Representing k Partitions
2:    PID[*] – Initial Partition Ids of Vertices
3:  NUMINTS – Number of Intervals
4:  for all threads T_i ∈ {T_1, ··· , T_k}  do
5:    do
6:      ▷ Load Partitions
7:      Read Subgraph wP_i of P_i into BUFFER(T_i)
8:        merge sorting over source vertex ids causing
9:        split adjacency lists to be merged
10:     Identify Boundary Vertices in wP_i using PID[*]
11:     Divide Boundary Vertices into NUMINTS Intervals
12:     ▷ Refine Partition wP_i against partitions wP_j, j > i
13:     REFINEPARTITION (wP_i ⊆ P_i)
14:   while (P_i is exhausted)
15: end for
16: ▷ Refine Partition Pairs
17: function REFINEPARTITION( wP_i )
18:   for  wP_j = wP_{i+1} ··· wP_k  do
19:     ▷ Refine wP_i wrt wP_j
20:     for interval id x = 1 ··· NUMINTS  do
21:       ▷ refine interval pair (I_x^i ∈ wP_i, I_x^j ∈ wP_j)
22:       loop ▷ Making a Pass
23:         (g, (v, w)) ← FINDMAXGAINPAIR(I_x^i, I_x^j)
24:         break when g = 0
25:         Add (v, w) to SWAPSET
26:       forever
27:       ▷ Update Partitions using SWAPSET
28:       for each (v, w) ∈ SWAPSET do
29:         SWAP ( PID[v], PID[w] )
30:       end for
31:     end for
32:   end for
33: end function
34: ▷ Find Pair of Vertices to Swap
35: function FINDMAXGAINPAIR(I_x^i, I_x^j)
36:   MAXGAIN ← 0; MAXPAIR ← null
37:   for  v ∈ I_x^i  do
38:     for  w ∈ I_x^j  do
39:       D(v) ← EC(v)\I_x^j − IC(v)\I_x^i
40:       D(w) ← EC(w)\I_x^i − IC(w)\I_x^j
41:       if D(v) ≥ 0 ∧ D(w) ≥ 0 then
42:         THISGAIN ← D(v) + D(w) − 2 × Edge(v, w)
43:       end if
44:       if THISGAIN > MAXGAIN then
45:         MAXGAIN ← THISGAIN;
46:         MAXPAIR ← (v, w)
47:       end if
48:     end for
49:   end for
50:   return (MAXGAIN, MAXPAIR)
51: end function
```

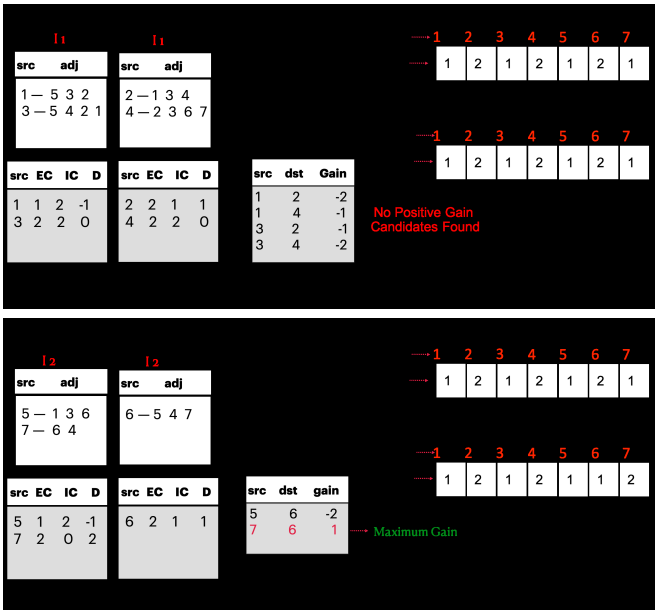


Fig. 4: Illustration of Refinement Algorithm.

itself against all other partitions (Line 18), considering pairs of intervals (Line 20), making multiple *passes* (Lines 22-26), and finally applying atomic SWAP operations that update partition ids stored in array PID. When considering a pair of intervals, function FINDMAXGAINPAIR (Lines 34-51) is used to consider all relevant vertex pairs in those intervals and finds the pair with the highest gain and returns that pair.

Figure 4 illustrates the above algorithm. We assume two partitions with two intervals each. The top part of the figure refines the first intervals from both partitions against each other and does not swap any vertices as no vertex pair with a positive gain is found after a pass. The bottom part refines second intervals from both partitions and this time the first pass identifies a vertex pair with positive gain and in the second pass (not shown) none is found. The figure shows how the partitioning as expressed in PID[*] array changes. The EC, IC, and D values of the source vertices belonging to intervals are shown. The gains computed in each pass are also shown. However, note that the tables that show these are for illustration as no tables are maintained by the algorithm, only the pair with maximum gain is remembered.

IV. GO PROTOTYPE AND ITS EVALUATION

GO is implemented in C++ and it uses the Infinimem I/O runtime [11] to leverage its support for seamless batch disk I/O for variable size records. The records are expressed using Protocol Buffers [27] which provide efficient serialization/deserialization. The graph is divided among multiple threads for parallel processing where each thread reads its assigned part of the graph and puts it into the in-memory buffers. The graph is represented in memory using the adjacency list format and written to disk by the threads to which vertices are assigned using Infinimem's batch I/O.

The goal of our evaluation is to compare the quality and cost of graph partitioning as performed by GO and Mt-Metis. We study the scalability of GO with increasing graph size and varying memory availability. For modest sized graphs that could be successfully partitioned by both GO and Mt-Metis, we compare the partitions produced by both the systems in terms of number of edgecuts, balance and peak memory used in handling graphs of varying sizes.

The graph partitioning experiments were performed on a machine with 32 cores (2 sockets, each with 16 cores) with Intel Xeon Processor E5-2683 v4 processors, 425 GB memory, 1TB SATA Drives and running CentOS Linux 7. Table I includes graphs of varying sizes ranging from 15.5 million edges to 3.3 billion edges. This allows us to compare the scalability of partitioning algorithms and demonstrate that eventually for large graphs Mt-Metis runs out of memory while GO can successfully partition them even when only given part of the memory is available on the machine used in the experiments.

Our evaluation considers following partitioning algorithms:

- GO-100 corresponds to the amount of memory so the graph fits in memory and initial partition is not written to and re-read from disk;
- GO-75, GO-50, and GO-25 correspond to running GO with 75%, 50% and 25% of graph in memory. GO will need to write and re-read the initial partition from disk.
- Mt-Metis is a *multithreaded* version of Metis [14]. We compare GO's performance with Mt-Metis 0.6.0 that incorporates enhanced coarsening scheme [15] for graphs with highly variable degree distribution (e.g., power-law).

A. Quality of Partitions: Edgecuts and Balance

We demonstrate that GO can successfully partition all graphs in Table I using varying amounts of memory that holds 100%, 75%, 50%, or 25% of the graph. Since different configurations have different amounts of memory available, the partitionings they produce are different. We compare the quality of partitions produced using Mt-Metis with the various GO configurations. Table IV and Figure 5 presents *edgecuts* data while Table V presents the *balance* data for partitionings produced. For Mt-Metis no data is presented for the three largest graphs (TW, TM and UK07) because it ran out of memory in the coarsening phase and terminated (indicated by ✗ in the tables).

Edgecuts: From the number of edgecuts given in Table IV, we observe that the number of edgecuts produced by all GO configurations are typically far fewer than Mt-Metis, i.e. irrespective of the amount of memory available to GO. This is because the interval size employed by the KL algorithm during the refinement is similar for all the GO configurations. The variation in edgecuts for different GO configurations is due to the difference in refinement intervals caused by batches merged during the *partition refinement phase*. Mt-Metis creates 8-way/16-way/24-way partitioning with 11.8/4.2/2.1× to 76.2/28.9/10.3× more edgecuts than GO-100. And this is in spite of GO-100 using much less memory as, being a single-level algorithm, it does not create coarsened graphs. As expected, the percentage of edges that are cut increases with number of partitions.

TABLE IV: Number of Edgecuts for GO-100 and Relative Number for Mt-Metis and Other GO Configurations.

k	Algo.	Input Graphs								
		FL	PK	LJ	OK	UK02	WK	TW	TM	UK07
8	GO-100	337,908	173,738	452,443	316,918	113,265	655,057	5,837,535	112,048	6,760,828
	Mt-Metis	11.8×	24.2×	16.7×	76.2×	19.8×	68.7×	✗	✗	✗
	GO-75	2.4×	0.8×	0.4×	1.1×	1.4×	0.5×	1.5×	1.6×	0.6×
	GO-50	2.4×	1.3×	0.5×	1.4×	1.2×	0.9×	1.6×	2.5×	0.7×
	GO-25	2.8×	1.7×	0.6×	2.4×	2.2×	1.1×	1.7×	4.3×	0.6×
16	GO-100	1,145,151	366,450	493,762	2,640,637	649,569	1,923,263	14,249,266	1,078,944	10,342,321
	Mt-Metis	4.2×	15.2×	20.3×	12.5×	4.0×	28.9×	✗	✗	✗
	GO-75	2.1×	1.3×	0.9×	0.4×	0.4×	0.8×	1.9×	0.9×	0.5×
	GO-50	2.1×	1.5×	1.1×	0.4×	0.3×	1.0×	1.9×	0.8×	1.1×
	GO-25	2.2×	2.1×	1.4×	0.6×	0.7×	1.1×	1.9×	1.0×	1.0×
24	GO-100	2,469,269	1,672,698	2,260,283	9,278,356	869,908	6,232,550	22,486,575	3,923,882	13,110,702
	Mt-Metis	2.1×	3.8×	5.3×	3.8×	3.3×	10.3×	✗	✗	✗
	GO-75	2.0×	0.5×	0.4×	0.2×	0.6×	0.6×	1.7×	0.2×	1.8×
	GO-50	2.0×	0.7×	0.5×	0.2×	0.4×	0.7×	1.7×	0.3×	1.8×
	GO-25	2.1×	0.7×	0.6×	0.3×	1.1×	0.8×	1.8×	0.6×	1.7×

TABLE V: *Balance* of Partitions – GO Configurations vs. Mt-Metis: Values are $MAX_i(|\ V_i\ |)$ as a percentage of $|\ V_i\ |$. The ideal balance percentage for 8, 16 and 24 partitions is 12.50%, 6.25% and 4.17% respectively.

k	Algo.	Input Graphs								
		FL	PK	LJ	OK	UK02	WK	TW	TM	UK07
8	GO-100	12.50%	12.50%	16.12%	12.50%	12.50%	12.50%	12.50%	12.51%	12.50%
	Mt-Metis	13.76%	13.70%	16.65%	13.70%	12.54%	14.61%	✗	✗	✗
	GO-75	12.50%	12.50%	16.12%	12.50%	12.50%	12.50%	12.50%	12.50%	12.50%
	GO-50	12.51%	12.51%	16.12%	12.51%	12.50%	12.50%	12.50%	12.51%	12.50%
	GO-25	12.52%	12.52%	16.12%	12.51%	12.50%	12.50%	12.50%	12.51%	12.50%
16	GO-100	6.25%	6.25%	8.06%	6.25%	6.25%	6.25%	6.25%	6.26%	6.25%
	Mt-Metis	7.62%	6.76%	8.76%	7.21%	6.28%	6.67%	✗	✗	✗
	GO-75	6.25%	6.25%	8.06%	6.25%	6.25%	6.25%	6.25%	6.26%	6.25%
	GO-50	6.27%	6.27%	8.07%	6.26%	6.25%	6.25%	6.25%	6.26%	6.25%
	GO-25	6.30%	6.31%	8.08%	6.29%	6.26%	6.26%	6.25%	6.26%	6.25%
24	GO-100	4.17%	4.17%	5.37%	4.17%	4.17%	4.17%	4.17%	4.17%	4.17%
	Mt-Metis	4.86%	4.82%	5.70%	4.48%	4.19%	5.14%	✗	✗	✗
	GO-75	4.17%	4.17%	5.37%	4.17%	4.17%	4.17%	4.17%	4.17%	4.17%
	GO-50	4.21%	4.23%	5.40%	4.19%	4.17%	4.18%	4.17%	4.17%	4.17%
	GO-25	4.22%	4.24%	5.41%	4.24%	4.18%	4.19%	4.17%	4.17%	4.17%

When we look at the ratio of edgecuts for MT-Metis and GO-100, we also make another observation. The higher the $\frac{|E|}{|V|}$ ratio for the input graph, the higher is the edgecuts ratio of Mt-Metis over GO-100. Among the six graphs that are successfully handled by Mt-Metis, the three largest $\frac{|E|}{|V|}$ ratios are 38.1 for OK, 31.1 for WK, and 18.7 for PK. Their Mt-Metis over GO-100 edgecut ratios are the worst – 76.2× for OK, 68.7× for WK, and 24.2× for PK. In other words, GO is more effective in dealing with irregular nature of graphs.

Finally, Figure 5 presents the percentage of edges that are cut by partitioning. As we can see, the percentage is generally greater for smaller graphs than for larger graphs, the edgecut percentage increases with the number of partitions.

Balance of partitions: Next we examine how well balanced are the partitions that are produced. We present the percentage of vertices that belong to the largest partition in a partitioning in Table V. Note that the ideal percentage for best balance is 12.5% for 8-way, 6.25% for 16-way, and 4.17% for 24-way partitioning. We observe that for seven out of nine graphs – FL, PK, OK, WK, UK2002, TW, and UK2007 – GO-100 produces almost perfect partitioning. On the other hand, with Mt-Metis the largest partitions vary from 12.54% to 16.65% in size. Thus, the partitions generated by GO-100 are more balanced and have fewer edgecuts than partitions produced by Mt-Metis. When GO is given less memory, balance of partitions is not adversely impacted but only very slightly. The highly balanced partitions produced by GO is due to its superior initial partitioning phase. This is because the swapping of vertices performed during refinement, by design, reduce the edgecuts without altering the balance of the partitions.

B. Memory Usage

Table VI (Left) shows the peak memory used by GO and Mt-Metis for varying the number of partitions. The peak memory is noted using the *top* utility on Linux. For the first six graphs, since Mt-Metis could successfully partition the graphs, we calculate the ratio between peak memory used by Mt-Metis versus that used by GO configurations.

From the data in Table VI and Table VII we observe the following. As expected, GO-100 is far more memory efficient than Mt-Metis as GO is a single level algorithm while Mt-

TABLE VI: (Left) Peak Memory in **GB** and (Right) Execution Time in **Seconds** for GO configurations and Mt-Metis.

k	Algo.	FL	PK	LJ	OK	UK02	WK	TW	TM	UK07	FL	PK	LJ	OK	UK02	WK	TW	TM	UK07
8	**GO-100**	0.61	0.72	1.7	2.70	8.0	7.6	29.8	72.9	79.1	7.4	6.2	18.9	18.2	70.9	71.9	397	1,722	1,026
	Mt-Metis	2.30	4.00	6.4	22.40	15.4	55.1	✗	✗	✗	6.0	8.0	15.0	32.0	34.0	121.8	✗	✗	✗
	GO-75	0.27	0.31	0.8	0.96	3.3	2.5	10.0	52.0	26.5	7.5	6.5	19.1	20.1	83.4	75.1	427	1,794	1,057
	GO-50	0.26	0.27	0.6	0.93	2.6	2.4	9.8	46.0	26.1	7.9	6.6	19.2	20.2	84.1	78.0	450	1,814	1,073
	GO-25	0.14	0.16	0.4	0.48	1.6	1.5	6.4	40.4	13.4	8.0	8.2	19.4	20.4	87.6	79.9	453	2,068	1,113
16	**GO-100**	0.67	0.78	1.9	2.80	8.6	8.0	31.2	104.2	82.4	7.2	5.8	17.4	17.5	65.1	66.2	396	1,763	965
	Mt-Metis	2.40	2.70	7.8	20.50	16.2	53.1	✗	✗	✗	7.0	7.0	19.0	34.0	34.0	123.6	✗	✗	✗
	GO-75	0.37	0.38	0.9	1.00	3.8	2.9	11.0	83.0	29.0	7.4	6.9	17.7	19.1	65.7	69.1	405	1,789	1,020
	GO-50	0.31	0.32	0.8	0.99	3.2	2.8	10.3	77.8	28.0	8.2	8.2	18.5	20.4	68.3	70.6	413	1,811	1,049
	GO-25	0.23	0.22	0.6	0.52	2.2	1.9	7.9	71.4	17.2	9.4	8.5	19.2	21.0	69.5	74.1	432	1,929	1,094
24	**GO-100**	0.45	0.50	2.1	2.90	9.2	8.4	32.6	135.6	85.7	7.4	6.1	18.6	19.9	60.4	69.0	424	1,869	1,028
	Mt-Metis	2.60	3.70	8.1	17.40	15.9	47.1	✗	✗	✗	6.0	8.0	17.0	30.0	35.0	126.6	✗	✗	✗
	GO-75	0.41	0.45	1.0	1.10	4.2	3.3	12.0	114.0	32.0	7.9	6.5	19.1	21.2	64.8	71.7	425	1,911	1,075
	GO-50	0.38	0.40	0.9	1.00	3.8	3.1	11.5	109.0	31.0	8.2	6.6	19.1	21.9	70.6	73.7	435	1,937	1,080
	GO-25	0.35	0.30	0.8	0.64	2.8	2.1	8.8	104.0	19.3	9.0	7.5	19.2	29.0	71.9	79.2	462	1,948	1,095

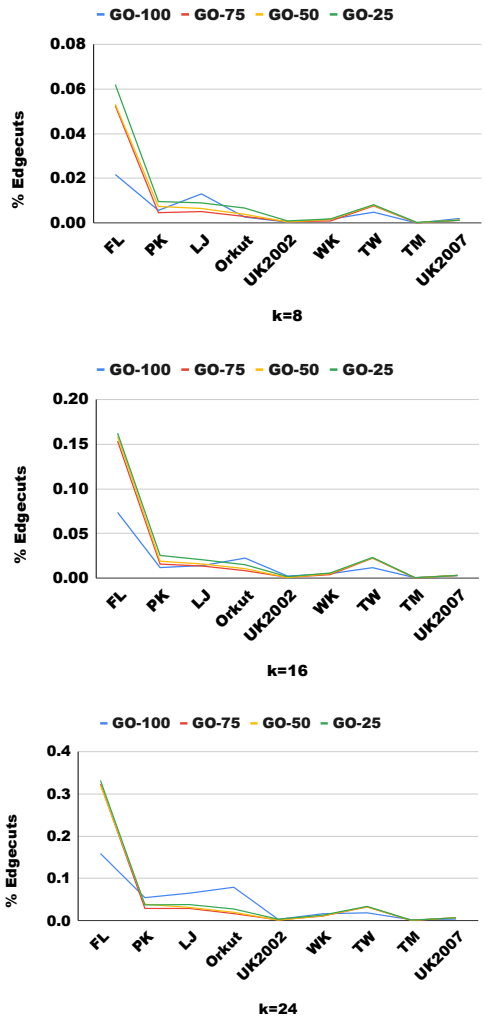


Fig. 5: Edgecuts as a percentage of total number of edges for GO configurations.

Metis is a multilevel algorithm that must additionally hold coarsened graphs in memory. For first six graphs, GO-100 uses 1.9× to 8.3× lesser amount of memory than Mt-Metis for an 8-way partitioning. For producing greater number of partitions (i.e., 16 and 24), both GO-100 and Mt-Metis require greater amount of memory although this increase is modest (less than 10%). For the largest three TW, TM, and UK2007 graphs, GO-100 uses 29.8-32.6GB, 72.9-135.6GB, and 79-85.7GB of memory respectively. Given that typically Mt-Metis uses many times the memory used by GO-100, it is not surprising that Mt-Metis runs out of memory and fails to partition the graphs.

TABLE VII: Memory Consumption of Mt-Metis Over GO.

k	Mt-Metis vs.	Input Graphs					
		FL	PK	LJ	OK	UK02	WK
8	**GO-100**	3.7×	5.5×	3.7×	8.3×	1.9×	7.3×
	GO-75	8.5×	12.9×	8.0×	23.3×	4.7×	22.0×
	GO-50	8.8×	14.8×	10.7×	24.1×	5.9×	23.0×
	GO-25	15.3×	25.0×	16.0×	46.7×	9.6×	34.4×
16	**GO-100**	3.6×	3.5×	4.1×	7.3×	1.9×	6.6×
	GO-75	6.4×	7.1×	8.6×	20.0×	4.2×	18.3×
	GO-50	7.7×	8.4×	9.7×	20.7×	5.1×	18.9×
	GO-25	10.4×	12.3×	13.0×	39.4×	7.4×	26.5×
24	**GO-100**	5.8×	7.4×	3.8×	6.0×	1.7×	5.6×
	GO-75	6.3×	8.2×	8.1×	15.8×	3.8×	14.2×
	GO-50	6.8×	9.2×	9.0×	17.4×	4.2×	15.2×
	GO-25	7.4×	12.3×	10.1×	29.0×	5.7×	21.4×

The $\frac{|E|}{|V|}$ ratio impacts the memory used by the algorithms, much more so for Mt-Metis than for GO-100 resulting in the following observations. Although the sizes of WK and UK2002 are fairly close, their $\frac{|E|}{|V|}$ ratios are quite different – 31.1 for WK versus 14.1 for UK2002. This impacts the relative memory usage of Mt-Metis and GO. As Table VII shows MT-Metis uses 7.3× for WK and 1.9× for UK2002 in comparison to GO-100. This is because the coarsened graphs for Mt-Metis contain greater numbers of edges for WK than for UK2002 causing greater need for additional memory by WK than UK2002.

The memory used by GO-25 is at least four times less than the memory used by GO-100. The exception is the TM input graph for which GO-100 uses around 1.5× to 2× the memory used by GO-25. This is because TM has the smallest $\frac{|E|}{|V|}$ ratio of 1.6 and hence edges account for smaller fraction of memory needs. In fact the auxiliary array for holding partition ids of vertices accounts for significant fraction of memory. Thus, reducing buffer memory impacts peak memory consumption far less than for all other graphs.

TABLE VIII: I/O Time in Seconds for GO Configurations.

k	GO-Mem Config.	Input Graphs								
		FL	PK	LJ	OK	UK02	WK	TW	TM	UK07
8	GO-100	0 (4.7)	0 (3.7)	0 (11.9)	0 (8.9)	0 (47.0)	0 (42.9)	0 (200.5)	0 (1187.0)	0 (478.4)
	GO-75	1.02 (5.1)	0.6 (4.2)	3.1 (12.4)	1.2 (9.6)	14.5 (50.6)	5.4 (46.1)	26.6 (493.3)	50.5 (1281.5)	68.6 (499.9)
	GO-50	1.52 (8.7)	1.2 (5.0)	5.0 (12.8)	1.5 (10.7)	15.3 (60.7)	8.2 (47.7)	34.2 (509.8)	53.7 (1473.7)	82.5 (523.3)
	GO-25	2.63 (6.3)	1.4 (5.7)	5.4 (14.7)	1.6 (11.9)	20.8 (61.4)	8.7 (53.8)	45.4 (572.6)	69.2 (1502.8)	47.9 (622.9)
16	GO-100	0 (4.6)	0 (3.5)	0 (11.0)	0 (8.5)	0 (40.4)	0 (34.8)	0 (173.5)	0 (1216.4)	0 (413.6)
	GO-75	1.23 (4.6)	0.41 (5.5)	2.1 (11.4)	1.0 (9.0)	10.7 (41.4)	5.1 (42.3)	26.6 (514.8)	31.9 (1224.6)	49.0 (441.0)
	GO-50	1.33 (5.8)	0.82 (5.7)	3.9 (33.7)	1.6 (10.9)	12.7 (42.1)	7.2 (42.9)	34.2 (541.7)	39.0 (1409.2)	62.4 (497.2)
	GO-25	1.42 (6.0)	0.83 (6.8)	4.2 (14.1)	1.7 (11.7)	17.5 (45.6)	8.4 (44.2)	45.4 (584.7)	77.0 (1471.0)	64.7 (683.8)
24	GO-100	0 (4.6)	0 (3.8)	0 (11.2)	0 (9.2)	0 (33.4)	0 (34.8)	0 (163.3)	0 (1179.7)	0 (302.5)
	GO-75	1.04 (5.2)	0.3 (3.9)	1.3 (11.9)	0.9 (13.9)	5.0 (36.6)	6.4 (122.2)	10.0 (452.4)	36.1 (1257.9)	24.8 (355.3)
	GO-50	1.44 (6.2)	0.7 (4.2)	3.4 (12.6)	1.4 (17.4)	6.4 (37.3)	4.0 (120.3)	15.7 (533.4)	49.4 (1463.1)	42.0 (365.0)
	GO-25	1.84 (7.4)	0.9 (5.5)	3.6 (14.3)	1.5 (19.3)	7.9 (45.7)	6.2 (138.4)	17.1 (534.8)	87.0 (1499.7)	46.0 (391.1)

TABLE IX: Mt-Metis Over GO Execution Times.

k	Mt-Metis Speedup	Input Graphs					
		FL	PK	LJ	OK	UK02	WK
8	GO-100	0.8×	1.3×	0.79×	1.7×	0.5×	1.6×
	GO-75	0.8×	1.23×	0.78×	1.59×	0.41×	1.6×
	GO-50	0.7×	1.21×	0.78×	1.58×	0.40×	1.5×
	GO-25	0.7×	0.9×	0.77×	1.6×	0.4×	1.5×
16	GO-100	1.0×	1.2×	1.1×	1.9×	0.5×	1.9×
	GO-75	1.0×	1.0×	1.1×	1.8×	0.5×	1.8×
	GO-50	0.8×	0.8×	1.0×	1.7×	0.4×	1.7×
	GO-25	0.7×	0.8×	0.9×	1.6×	0.4×	1.6×
24	GO-100	0.8×	1.3×	0.9×	1.50×	0.6×	1.8×
	GO-75	0.8×	1.23×	0.9×	1.4×	0.54×	1.8×
	GO-50	0.7×	1.21×	0.9×	1.3×	0.49×	1.7×
	GO-25	0.6×	1.1×	0.8×	1.03×	0.5×	1.6×

TABLE X: Scalability of GO Configurations: Execution Times in Seconds for PageRank and WCC on GridGraph.

k	Part. Algo.	PageRank			WCC		
		OK	WK	TW	OK	WK	TW
8	GO-100	6	19	128	3	7	48
	GO-75	7	22	134	4	8	48
	GO-50	7	21	128	4	7	49
	GO-25	8	20	126	5	7	44
16	GO-100	6	21	96	4	7	37
	GO-75	7	23	100	6	8	39
	GO-50	8	20	100	5	7	39
	GO-25	7	21	102	6	7	38
24	GO-100	8	16	100	5	6	33
	GO-75	7	19	110	6	7	34
	GO-50	8	20	99	6	8	37
	GO-25	7	18	109	7	7	40

C. Execution Times

Table VI (Right) shows the execution time in seconds for different GO configurations - GO-100, GO-75, GO-50, GO-25, and Mt-Metis. The execution times of GO-100 compare well with Mt-Metis and better for graphs with higher $\frac{|E|}{|V|}$ ratio (PK, OK WK) with speedups ranging from 1.3× to 1.7×. (see Table IX). For the two larger graphs the results are quite different – for UK2002 the execution times for Mt-Metis are roughly 2× faster than GO-100 while for WK, Mt-Metis runs 1.6× slower. Nevertheless, GO-100 produces partitioning with fewer edgecuts than Mt-Metis for both UK2002 and WK.

The runtimes of GO scale with graph size ranging from few seconds for the smallest graph to roughly 32 minutes for large graphs. Let us compare the execution times of GO-100 with GO-25. We observe that typically GO-25 exceeds the execution times of GO-100 by less than 50%. For the largest graph of UK2007 with over 3.3 billion edges, the execution time of GO-25 exceeds that of GO-100 by 8.5%, 13.4%, and 6.5% for 8-way, 16-way, and 24-way partitionings. Thus, we see that the runtimes of GO scale well with graph size, number of partitions, and amount of memory available to run.

Finally, we observed that the I/O times of GO-100 differ from the I/O times of GO-25 only by a small amount because the I/O performed by the out-of-core feature is highly efficient. Table VIII presents the I/O times of writing/reading of initial partitioning to/from Infinimem is small compared to rest of the I/O time (numbers in parenthesis in the table) for reading the initial graph and writing out the final partitioning.

D. GridGraph Performance vs. GO Partitioning

Next we study the impact of partitions produced on runtimes of graph algorithm on a state-of-the-art out-of-core system. We executed two graph algorithms, **PageRank** and **WCC** (*Weakly Connected Components*), on the GridGraph [25] out-of-core system using the partitions produced by GO-100, GO-75, GO-50 and GO-25. The execution times of GridGraph are given in Table X. As we can seen from this table, the execution times of all the GO configurations are comparable. In fact, the runtimes are same for most of the GO configurations. In other words, reducing memory to 25% does not result in any slowdowns on GridGraph as the quality of partitions produced does not change significantly from GO-100 to GO-25. Therefore we can conclude that the quality of partitions produced by GO is fairly insensitive to the aount of memory provided to GO. The above results are to be expected because, as we reduce the memory available to GO, the number of edgecuts does not change significantly.

In order to compare the effectiveness of GO's partitioning with Mt-Metis partitioning and other simple partitioning schemes, we ran the workloads of *PageRank* and *WCC* on GridGraph. We compare the running times of GridGraph

TABLE XI: Execution Times in Seconds for GO-100, Mt-Metis, Cyclic, and Block-Cyclic Partitionings on Medium Sized Graphs OK, WK and Large Graph TW for PageRank and Weakly Connected Components (WCC) on GridGraph.

k	Part. Algo.	PageRank			WCC		
		OK	WK	TW	OK	WK	TW
8	**Mt-Metis**	7	21	✗	4	8	✗
	GO-100	**6**	**19**	**128**	**3**	8	**48**
	Cyclic	8	**19**	133	5	8	50
	Block-Cyclic	8	24	130	5	**7**	49
16	**Mt-Metis**	9	22	✗	5	14	✗
	GO-100	**8**	**21**	**96**	**4**	**12**	**37**
	Cyclic	9	23	109	5	15	39
	Block-Cyclic	**8**	29	99	**4**	17	**37**
24	**Mt-Metis**	7	18	✗	**6**	9	✗
	GO-100	**6**	**16**	**100**	**6**	**7**	**33**
	Cyclic	7	20	101	8	9	44
	Block-Cyclic	7	21	105	9	11	35

on OK, WK and TW input graphs for multiple partitioning schemes. In addition to Mt-Metis and GO, we also collected the running times of GridGraph when using simple partitioning strategies, cyclic and block-cyclic, as these strategies have been used by existing out-of-core systems for evaluating graph queries to circumvent the memory intensive nature of graph partitioning. The results shown in Table XI demonstrate that running times for GO-100 partitioning are same or less than that for Mt-Metis, Cyclic and Block-Cyclic partitionings with the exception of *WCC* for k=8. Note that in Table XI the minimum execution times are shown in bold. This is to be expected because GO-100 always produces partitioning that is superior to the partitioning generated by other methods.

V. Conclusions

In this paper we presented the GO out-of-core graph partitioner that can function within the memory constraints imposed by the machine and successfully partition graphs that far exceed the size of a graph that can be held in memory. The execution time and memory requirements scale well with the graph size. For graphs that can be successfully partitioned using the in-memory Mt-Metis graph partitioner, GO produces high quality partitioning in terms of edgecuts and balance. In contrast to multilevel partitioning scheme used by Mt-Metis, GO is single level and it partitions the graph in two highly memory efficient parallel passes. For graphs that can be successfully partitioned by Mt-Metis on a single machine, GO produces balanced 8-way partitions with 11.8× to 76.2× fewer edgecuts using 1.9× to 8.3× lesser memory in comparable runtime.

Acknowledgements

This work is supported in part by National Science Foundation grants CCF-1813173, CCF-2002554, and CCF-2028714 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] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Technical Report SAND93-1301*, Sandia National Labs, 1993.

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

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

[8] 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.

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

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

[23] 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.

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

[25] 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.

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

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