Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+

Julian Shun Laxman Dhulipala Guy E. Blelloch
Carnegie Mellon University
jshun@cs.cmu.edu, ldhulipa@andrew.cmu.edu, guyb@cs.cmu.edu

We study compression techniques for parallel in-memory graph algorithms, and show that we can achieve reduced space usage while obtaining competitive or improved performance compared to running the algorithms on uncompressed graphs. We integrate the compression techniques into Ligra, a recent shared-memory graph processing system. This system, which we call Ligra+, is able to represent graphs using about half of the space for the uncompressed graphs on average. Furthermore, Ligra+ is slightly faster than Ligra on average on a 40-core machine with hyper-threading. Our experimental study shows that Ligra+ is able to process graphs using less memory, while performing as well as or faster than Ligra.

1 Introduction

Graphs algorithms have many applications, such as in analyzing social networks, biological networks and unstructured meshes in scientific simulations. Due to the recent growth in data sizes, improving the running time and space usage of graph algorithms has become very important. Recent work has focused on speeding up graph algorithms via parallelization, with a number of programming frameworks appearing in recent years (see, e.g., [11, 14, 21, 22, 24] among many others). Processing graphs in shared memory can be significantly faster than doing so in distributed memory, as recent work has shown [24]. However, this requires the graphs to fit in the memory of a single node. While the largest publicly available real-world graphs today can fit in the main memory of a single multicore machine, which can be configured with over a terabyte of memory, the cost of machines increases with memory size. Similarly, the cost of renting machines in the cloud increases with their memory sizes. Therefore, reducing the sizes of graphs is crucial in reducing the cost of large-scale graph analytics. Furthermore, it allows even larger real-world graphs that will become available in the future to fit on a single node. As such, we are motivated to study compression in large-scale graph algorithms as a means of reducing space. Additionally, we are interested in knowing if using compression can speed up parallel graph algorithms.

As for previous work on using compression to reduce memory usage in graph algorithms, Blandford et al. [2, 3] experiment with a variety of graph compression techniques, and study the performance of graph algorithms on compressed graphs, which requires on-the-fly decoding. Their techniques can reduce space usage by up to a factor of 3–6 compared to normal adjacency arrays. However, they only study the techniques in a sequential setting, and for only three specific algorithms—depth-first search, PageRank and bipartite matching. They show that the algorithms on compressed graphs are about 25% slower. In this work, we parallelize their compression techniques, and study the performance of a broad class of parallel graph algorithms on much larger graphs than used in [2, 3]. We show that sequentially, the algorithms on compressed graphs are indeed often slower, however in parallel they become competitive with or faster than the algorithms on uncompressed graphs.

1068-0314/15 $31.00 © 2015 IEEE
DOI 10.1109/DCC.2015.8

IEEE computer society

This is because graph algorithms are memory-bound, and memory is a larger bottleneck in parallel due to multiple cores competing for bandwidth—therefore reducing the memory footprint is more important, while at the same time decoding becomes less of an overhead as it has better parallel scalability relative to the rest of the computation.

To experiment with a wide variety of graph applications, we parallelize and integrate the compression and decoding techniques described by Blandford et al. [2, 3], as well as a compression idea from the sparse matrix-vector multiplication literature [18], into the Ligra shared-memory graph processing framework [24]. Our extended framework, which we call Ligra+, uses less space than Ligra, while providing comparable or improved performance.

Ligra+ is able to represent a variety of synthetic and real-world graphs using 49–56% of its original size on average, depending on the compression scheme. The performance of the graph algorithms in Ligra+ ranges from 2.2x faster to 1.1x slower than the original Ligra system. In many cases, Ligra+ outperforms Ligra due to its smaller memory footprint, and is about 14% faster on average when using the fastest compression scheme. Using compression, Ligra+ is able to process graphs using less memory, and fit larger graphs in memory, while performing just as well as or better than Ligra. As the compression techniques are part of a graph processing framework, users can easily work with compressed graphs without worrying about the implementation details. Applications written in Ligra can also be used in Ligra+, as the interfaces are the same.

Other Related Work. There has been a large amount of work on compressing graphs, especially planar graphs and graphs with constant genus (see, e.g., [3] and the references within). Recent work [13, 20] has used compression to reduce graph sizes in the MapReduce setting. Their focus is on reducing the storage size on disk because a large portion of the running time of MapReduce is from disk I/O's, and they show performance improvements for MapReduce graph algorithms. However, the techniques are not used to reduce the in-memory space usage. In contrast, our work focuses on reducing the in-memory space usage while maintaining or improving performance, so it becomes necessary to efficiently decode on-the-fly.

Other work has focused mainly on compressing Web and social network graphs (see e.g. [1, 5, 10]). Most of these works have not been used to improve the performance of general graph algorithms. The techniques that have been applied to graph algorithms are particular to the algorithm and compression scheme [7, 8, 12, 15, 23], and not used in a general framework. The algorithms are also studied in the sequential setting.

Running algorithms on compressed inputs has been previously explored in the setting of sparse matrix-vector (spMV) multiplication [4, 9, 17, 18, 25]. Like graph algorithms, spMV is also a memory-bound computation, and so better improvements are observed in parallel. These papers show promising results, but only study the specific spMV computation. In our work, we study the impact of compression in a broad class of parallel graph algorithms.

2 Preliminaries

We denote a directed unweighted graph by $G(V, E)$ where V is the set of vertices and E is the set of (directed) edges in the graph. A weighted graph is denoted by $G = (V, E, w)$, where w is a function which maps an edge to a real value (its weight). The vertices are assumed to be indexed from 0 to $|V| - 1$. $N^+(v)$ denotes the out-neighbors of vertex v in G and $deg^+(v)$ denotes the out-degree of v in G. Similarly, $N^-(v)$ and $deg^-(v)$ denote the in-neighbors and

in-degree of v in G. We assume that there are no self-edges or duplicate edges in the graph.

The *difference encoding* scheme takes a vertex v's adjacency list, $\{v_0, v_1, \ldots, v_{deg(v)-1}\}$, given in increasing order and encodes the differences, $\{v_0 - v, v_1 - v_0, \ldots, v_{deg(v)-1} - v_{deg(v)-2}\}$. Blandford et. al [2] describe a class of variable-length codes, known as k-bit codes, which encode (compress) an integer x as a series of k-bit blocks. Each block uses one bit as a *continue bit*, which indicates if the following block is also a part of x's encoded representation. To encode x, we first check if $x < 2^{k-1}$. If this is the case, we simply write the binary representation of x into a single block, and set the continue bit to 0. Otherwise, we write the binary code for $x \mod 2^{k-1}$ in the block, set the continue bit to 1, and then encode $\lfloor x/2^{k-1} \rfloor$ in the subsequent blocks. Decoding works by examining blocks until a block with a continue bit of 0 is found. The decoded value in the i^{th} examined block is multiplied by $2^{i(k-1)}$, and added to the result. For values that can be negative (e.g. the first edge of a vertex), an extra bit in the first block is used as the sign bit, and decoding is modified accordingly.

2.1 Ligra Framework

In this section, we review the Ligra framework for shared-memory graph processing [24]. Ligra supplies a ***vertexSubset*** data structure used for representing a subset of the vertices. Ligra provides two simple functions, one for mapping over vertices and one for mapping over edges. **vertexMap** takes as input a vertexSubset U and a boolean function F, applies F to all vertices in U, and returns a vertexSubset containing vertices $U' \subseteq U$ such that for all $u \in U'$, $F(u) = true$. Note that F can side-effect data structures associated with the vertices, and does so in applications in Ligra. **edgeMap** takes as input a graph $G(V, E)$, vertexSubset U, boolean update function F and boolean conditional function C; it applies F to all edges $(u, v) \in E$ such that $u \in U$ and $C(v) = true$ (call this set of edges E_a), and returns a vertex subset U' containing vertices v such that $(u, v) \in E_a$ and $F(u, v) = true$. Again, F can side-effect data structures associated with the vertices. The programmer must ensure the parallel correctness of the functions passed to vertexMap and edgeMap.

vertexSubset Implementation. A vertexSubset $U \subseteq V$ in Ligra can be represented either sparsely or densely. If U is represented sparsely, then Ligra stores it as an array of size $|U|$ containing the vertex IDs (in any order). If U is represented densely, then Ligra uses a boolean array of size $|V|$, where index i stores *true* if and only if $i \in U$.

vertexMap Implementation. In Ligra, vertexMap is implemented as a simple parallel for-loop, applying the input function F over each element in the input vertexSubset, and then a parallel filter is used to keep only the vertices such that applying F returned *true*.

edgeMap Implementation. While the Ligra programmer needs to call only edgeMap, two versions of edgeMap are used under the hood—edgeMapSparse and edgeMapDense. For performance reasons, Ligra switches between the two versions based on the size of the input vertexSubset and the sum of the out-degrees of its vertices.

edgeMapSparse (pseudo-code shown in Figure 1) works as follows: for each vertex v in the input vertexSubset U (in parallel), it loops through the out-neighbors ngh $\in N^+(v)$ (in parallel) and if C(ngh) returns *true* it applies $F(v,$ ngh); if $F(v,$ ngh) returns *true*, then ngh is added to the output vertexSubset. edgeMapSparse operates on the sparse representation of a vertexSubset. As a vertex may be added to the output vertexSubset multiple times, duplicates must be removed before returning the vertexSubset (however, for efficiency this can be turned off by the programmer if the semantics of the F and C functions do not allow

```
1: procedure edgeMapSparse(G, U, F, C)
2:     Out = {}
3:     parfor v ∈ U do
4:         parfor ngh ∈ N+(v) do
5:             if (C(ngh) == 1 and F(v, ngh) == 1) then
6:                 Add ngh to Out
7:     Remove duplicates from Out
8:     return Out
```

Figure 1: edgeMapSparse implementation

```
1: procedure edgeMapDense(G, U, F, C)
2:     Out = {}
3:     parfor v ∈ {0, ..., |V| − 1} do
4:         if (C(v) == 1) then
5:             for ngh ∈ N−(v) do
6:                 if (ngh ∈ U and F(ngh, v) == 1) then
7:                     Add v to Out
8:                 if (C(v) == 0) then break
9:     return Out
```

Figure 2: edgeMapDense implementation

for duplicates).

edgeMapDense (pseudo-code shown in Figure 2) works as follows: for each vertex $v \in V$, it checks if $C(v)$ returns *true*; if so, it loops through the in-neighbors $N^-(v)$, checks if ngh $\in U$ and if so, applies $F(\text{ngh}, v)$. The vertices v such that $F(\text{ngh}, v)$ returned *true* are added to the resulting vertexSubset. edgeMapDense works on the dense representation of a vertexSubset. It uses an optimization where each vertex stops looping over its in-neighbors once $C(v)$ becomes $false$ (Line 8 in the pseudo-code). Note that the outer loop is parallel, but the inner loop is sequential to enable the optimization which breaks out of the loop early. However for vertices with high in-degree, Ligra loops over the in-neighbors in parallel, and does not check $C(v)$ for early termination.

The edgeMap function provided to the programmer calls edgeMapDense or edgeMapSparse based on the number of vertices and outgoing edges in the vertexSubset. When the number is greater than some threshold, then edgeMapDense is used, and otherwise edgeMapSparse is used. By default, the threshold is set to $|E|/20$, but if desired, the programmer can provide another threshold value. The input vertexSubset is automatically converted to sparse or dense representation based on whether edgeMapSparse or edgeMapDense is used.

Graph Representation. Ligra represents graphs using the adjacency array format. Two edge arrays, one for in-edges and one for out-edges, are kept for the entire graph (for symmetric graphs, only one edge array is kept). Each vertex stores a pointer to the start of its edges in each array, as well as its in-degree and out-degree.

3 Ligra+ Implementation

We now describe how we develop Ligra+ to support processing of compressed graphs.

Encoding. We use two types of k-bit codes in Ligra+—byte codes and nibble codes, which correspond to 8-bit and 4-bit codes, respectively. Byte codes are fast to decode, as compressed blocks lie on byte-aligned boundaries. Nibble codes lie on 4-bit boundaries, and are slower to decode due to the extra bit arithmetic required. Blandford et al. [2] show that 2-bit codes and gamma codes (effectively 1-bit codes) do not provide much additional space savings compared to nibble codes, while being more expensive to decode, so we opted not to use them.

For byte codes, we also use an idea from the spMV compression literature [18] which reduces the decoding time. Instead of storing a variable-length code for each value, we find consecutive groups of elements that require the same number of bytes (1 to 4 bytes) to store. For each group we store an 8-bit header indicating the number of bytes each element requires (2 bits of the header) and the size of the group (6 bits of the header, which allows for groups of up to size 64). This technique slightly increases the space usage, but decreases the decoding time as there is no longer a continue bit which needs to be checked to figure out when to stop decoding. This allows us to unroll the decoding loop for an element as we know how many bytes it requires, and therefore reduces branch mispredictions. We refer to

this scheme as ***run-length encoded byte codes***. Note that with this scheme, each byte can use all 8 bits for data, as it no longer needs to store a continue bit.

We implement an encoder program that generates a binary file representing a compressed graph using one of the coding schemes. The encoding is parallelized over the vertices. For each vertex in the graph, we first sort the edges in ascending order, and then compress the edge set of each vertex by encoding the differences between consecutive edges. For the first edge of each vertex, we encode the difference between the source and the target vertex (which can be negative), with an additional sign bit in the first block. The run-length encoded byte codes do this as well for the first edge of each vertex (the discussion in the previous paragraph is only applied to the remaining edges). We maintain a single array of compressed edge values, and store the vertex offsets into the array. To process the vertices in parallel in the applications, vertex degrees must be known before decoding so that appropriate offsets into shared arrays can be computed. In Ligra+, the vertex degrees are not implicit from the offsets, while in Ligra they are. Hence we store the vertex degrees as well. We do not compress the vertex offsets and degrees, since for many real-world graphs the number of edges is much larger than the number of vertices, so the space savings are low. For asymmetric input graphs, the in-edges for each vertex are also generated and encoded.

Decoding. The vertexSubset and VERTEXMAP implementations in Ligra+ are the same as in Ligra, since vertices are not compressed. We modify the two implementations of EDGEMAP, so that the neighbors of a vertex are decoded using a special function. In particular, Lines 4–6 of the EDGEMAPSPARSE pseudo-code in Figure 1 and Lines 5–8 of the EDGEMAPDENSE pseudo-code in Figure 2 are replaced by a call to DECODESPARSE and DECODEDENSE, respectively.

We first describe our sequential implementations of DECODESPARSE and DECODEDENSE for the variable-length codes, and then discuss how to parallelize them in the next sub-section. The implementations use two decoding functions ***FirstEdge*** and ***NextEdge***. FirstEdge takes as input a pointer into the compressed edge array, and decodes one value representing the difference between the edge and source vertex (which can be negative). It then modifies the pointer to point to the start of the next value in the compressed edge array. NextEdge takes as input a pointer into the compressed edge array, decodes one value representing the difference between consecutive edges (which can only be positive), and modifies the pointer to point to the start of the next value in the compressed edge array. The decoding functions decode byte codes or nibble codes following the procedure described in Section 2.

The pseudo-code for DECODESPARSE is shown in Figure 3. It takes as input the source vertex v, its degree (deg), a pointer to the start of its out-neighbors in the compressed array of out-edges (outEdges), the functions F and C, and a pointer to the output vertexSubset of EDGEMAPSPARSE (Out). It decodes the first neighbor by calling the function FirstEdge, which returns the difference between the source and target vertex, and then adds the value of the source vertex v to the result to obtain the value of the neighbor (Lines 4–5). The result is assigned to the variable prevEdge to allow for decoding of subsequent edges (Line 8). In later iterations, the difference between the previous edge and current edge is obtained by calling the function NextEdge; the edge value is obtained by adding the difference to the value of prevEdge (Lines 6–7), and then subsequently assigned to prevEdge (Lines 8). As in EDGEMAPSPARSE, we apply the function C to ngh, and if it returns *true*, we apply F to (v, ngh); if F returns true then we add the neighbor to the output vertexSubset (Lines 9–10).

```
1: procedure decodeSparse(v, deg, outEdges, F, C, Out)
2:     prevEdge = −1
3:     for j = 0 to deg−1 do                ▷ Loop over out-neighbors
4:         if j == 0 then
5:             ngh = FirstEdge(outEdges) + v
6:         else
7:             ngh = NextEdge(outEdges) + prevEdge
8:         prevEdge = ngh
9:         if (C(ngh) == 1 and F(v, ngh) == 1) then
10:            Add ngh to Out
```

Figure 3: decodeSparse implementation

```
1: procedure decodeDense(v, deg, inEdges, F, C, Out, U)
2:     prevEdge = −1
3:     for j = 0 to deg−1 do                ▷ Loop over in-neighbors
4:         if j == 0 then
5:             ngh = FirstEdge(inEdges) + v
6:         else
7:             ngh = NextEdge(inEdges) + prevEdge
8:         prevEdge = ngh
9:         if (ngh ∈ U and F(ngh, v) == 1) then
10:            Add v to Out
11:        if (C(v) == 0) then break
```

Figure 4: decodeDense implementation

The pseudo-code for decodeDense is shown in Figure 4. It takes the same arguments as decodeSparse, except that the compressed edge array is for the in-edges (inEdges) instead of the out-edges, and it also takes the input vertexSubset U to edgeMapDense. Decoding the edges is done in the same way as in decodeSparse. When decodeDense is called with vertex v, it is assumed that $C(v)$ is *true*. As in the original edgeMapDense, it checks if an in-neighbor ngh is in the input vertexSubset U, and if so applies F to (v, ngh); if F returns *true* then v is added to the resulting vertexSubset (Lines 9–10). The optimization of breaking early is done on Line 11.

For run-length encoded byte codes, we modify the decoding procedures to process groups of edges after reading the header. Lines 9–10 of decodeSparse and Lines 9–11 of decodeDense are applied immediately after each neighbor ID is decoded.

Parallel Decoding. Although for most graphs, decoding the edges sequentially for each vertex gives performance competitive with Ligra, we found that for some applications on certain graphs, it was up to 2 times slower. In these cases, parallelizing over the vertices was not sufficient due to the highly skewed distributions of degrees. Therefore we designed a parallel decoding scheme, where vertices with degree greater than some threshold T split their edges into chunks each containing T edges (except for possibly the last chunk), and the first edge of each chunk is difference encoded with respect to the source vertex. For each vertex, offsets into its chunks of edges are stored, except for the first chunk. Thus, for vertices with only one chunk (degree at most T), no extra storage is required. For each vertex, the different chunks of the edge array are decoded in parallel, as we have the offset to the start of each chunk. For decodeDense, the optimization of breaking early is applied inside each chunk. The threshold T represents a trade-off between parallelism and space overhead. In our experiments we set the threshold T to be 1,000, which we found to work best overall, although we found the performance to be similar across a wide range of T (from 100 to 10,000). We found that the storage required for the additional offsets is minimal for this range of T, as there are at most $|E|/T - |V|$ offsets needed for the graph. We note that the papers describing compression in parallel spMV do not perform parallel decoding within rows of the matrix (analogously, the edges of a vertex).

Graph Storage. Two arrays for edges are used—one for the compressed in-edges and one for the compressed out-edges. Vertex offsets into the edge arrays and their degrees are stored in a separate array, uncompressed. For symmetric graphs, only one edge array is required, and for asymmetric graphs, both the in-edges and out-edges are required.

Weighted Graphs. For weighted graphs, we encode the edge weights using difference encoding with respect to the value 0, and use a bit in the first block as the sign bit. Decoding is done in the same manner as decoding the first edge of a vertex, but relative to the value 0. The edge targets and weights are interleaved to improve cache locality. The FirstEdge and

Input Graph	Num. Vertices	Num. Directed Edges	Ligra	Ligra+ (byte)	Ligra+ (byte-RLE)	Ligra+ (nibble)
randLocal	10,000,000	98,201,048	433 MB	228 MB	246 MB	221 MB
3D-grid	9,938,375	59,630,250	278 MB	219 MB	209 MB	209 MB
soc-LJ	4,847,571	85,702,474	362 MB	188 MB	204 MB	178 MB
cit-Patents	6,009,555	33,037,894	156 MB	107 MB	117 MB	105 MB
com-LJ	4,036,538	69,362,378	294 MB	152 MB	166 MB	143 MB
com-Orkut	3,072,627	234,370,166	950 MB	440 MB	466 MB	421 MB
nlpkkt240	27,993,601	746,478,752	3.1 GB	1.06 GB	1.16 GB	815 MB
Twitter	41,652,231	1,468,365,182	12.08 GB	6.17 GB	6.46 GB	5.95 GB
uk-union	133,633,041	5,507,679,822	45.9 GB	15.5 GB	16.2 GB	10.9 GB
Yahoo	1,413,511,391	12,869,122,070	62.8 GB	37.9 GB	39.3 GB	34.4 GB

Table 1: Graph input sizes and storage sizes, including both vertices and edges.

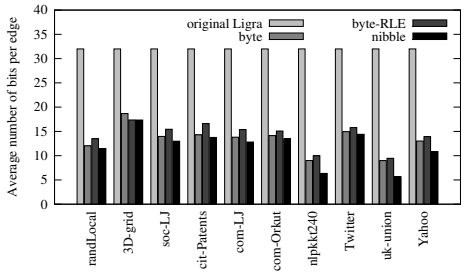


Figure 5: Average number of bits per edge required for the different coding schemes.

NextEdge functions are modified to decode the target of an edge along with its weight.

For run-length encoded byte codes, we find groups of edges that require at most x bytes for the difference with the previous edge and y bytes for the weight, where $x \in \{1,2,3,4\}$ and $y \in \{1,4\}$. The header byte uses 3 bits to store the (x,y) combination and 5 bits for the size of the group (allowing for groups of up to size 32). The decoding functions are modified accordingly to decode groups of edge targets/weights after reading the header.

Comparison to Ligra. The user interface to Ligra+ is the same as in Ligra, so applications developed using Ligra are compatible with Ligra+. Only the graph representation and the implementations of EDGEMAP have changed, and the user is not exposed to this.

4 Experiments

In this section, we analyze the effect of compression on the space usage and running time of a variety of graph applications using a collection of large-scale graphs. We experiment with six graph applications developed in the original Ligra framework (see [24] for details on the implementations): breadth-first search (***BFS***), betweenness centrality computation from a source vertex (***BC***), graph radii estimation (***Radii***), connected components (***Components***), ***PageRank*** (one iteration) and ***Bellman-Ford*** shortest-paths. The first five applications run on unweighted graphs, while Bellman-Ford runs on weighted graphs. In this paper, we only compare Ligra+ with Ligra as the goal of our experimental study is to observe the impact of graph compression on running time and space usage, while keeping other factors the same. We note that Ligra has been compared with several other graph processing systems in [24], and shown to often outperform them on a per-core basis.

Experimental Setup. We run our experiments on a 40-core Intel machine (with two-way hyper-threading) with 4×2.4GHz Intel 10-core E7-8870 Xeon processors (with a 1066MHz bus and 30MB L3 cache) and 256GB of main memory. The programs use Cilk Plus to express parallelism and are compiled with the `icpc` compiler (version 12.1.0) with the `-O3` flag. The reported times are based on a median of three trials.

Input Graphs. We use a set of synthetic and real-world graphs, whose sizes are shown in Table 1. ***randLocal*** is a random graph where every vertex has five edges to neighbors chosen with probability proportional to the difference in the neighbor's ID value from the vertex's ID. ***3D-grid*** is a grid graph in 3-dimensional space where every vertex has six edges, each connecting it to its 2 neighbors in each dimension. For the real-world graphs, we obtained the ***soc-LJ***, ***cit-Patents***, ***com-LJ*** and ***com-Orkut*** graphs from `http://snap.stanford.edu/`, and we symmetrized the graphs. ***nlpkkt240*** is a graph obtained from `http://www.cise.ufl.edu/research/sparse/matrices/`. ***Twitter*** is a graph of the Twitter network [19]. ***uk-union*** is a graph generated from snapshots of a subset of the UK web network [6]. ***Yahoo*** is a web

Input Graph	BFS								BC								Radii							
	orig.		byte		byte-RLE		nibble		orig.		byte		byte-RLE		nibble		orig.		byte		byte-RLE		nibble	
	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})
randLocal	1.46	0.055	1.93	0.056	1.8	**0.054**	3.23	0.08	4.82	**0.152**	6.36	0.167	5.33	0.159	7.95	0.228	9.22	0.286	10.7	0.295	9.82	**0.284**	17.1	0.433
3D-grid	1.47	0.214	1.3	0.216	1.26	**0.214**	1.66	0.233	4.75	0.559	4.64	0.572	5.36	**0.558**	5.78	0.588	135	**5.23**	173	5.57	139	**5.25**	244	7.73
soc-LJ	0.634	0.028	0.677	**0.027**	0.676	0.026	0.902	0.031	2.6	**0.093**	3.37	0.108	3.11	0.1	5.08	0.139	8.06	0.22	10.6	0.233	10.6	**0.218**	15	0.363
cit-Patents	0.639	**0.029**	0.758	0.03	0.752	0.03	1.08	0.037	2.1	**0.086**	2.59	0.091	2.49	0.091	3.62	0.115	4.75	**0.149**	6.91	0.16	5.81	0.157	8.61	0.224
com-LJ	0.523	**0.023**	0.539	**0.023**	0.54	0.023	0.708	0.026	2.12	**0.082**	2.89	0.091	2.57	0.087	4.14	0.121	7.75	**0.212**	8.73	0.222	8.23	0.213	14	0.343
com-Orkut	0.663	**0.029**	0.899	0.031	0.789	**0.029**	1.65	0.049	4.38	**0.14**	5.98	0.163	4.94	0.142	9.59	0.268	13.2	0.355	14.4	0.367	12.3	**0.323**	26.1	0.645
nlpkkt240	10.3	0.489	9.41	0.463	8.74	**0.466**	14.2	0.517	33.3	1.34	33.3	1.28	28.1	**1.23**	35.7	1.39	897	22.6	1120	24.4	906	**21.1**	1820	39.1
Twitter	6.91	**0.27**	8.79	0.274	8.33	0.268	13	0.347	40.1	4.62	47.4	**3.16**	44.9	3.53	75.2	3.78	172	7.46	193	7.26	172	**7.13**	392	10.2
uk-union	48.5	2.29	45.9	1.48	37.6	**1.34**	60.4	1.99	128	5.4	131	4.05	101	**3.46**	177	5.47	664	32	462	16.7	383	**14.5**	718	25.6
Yahoo	124	4.68	113	3.98	128	**3.8**	161	4.81	458	13.8	510	13.6	438	**12.4**	767	19	1390	36.4	1440	35.2	1250	**32.7**	2280	53.7

	Components								PageRank								Bellman-Ford							
	orig.		byte		byte-RLE		nibble		orig.		byte		byte-RLE		nibble		orig.		byte		byte-RLE		nibble	
	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})	(T_1)	(T_{40})
randLocal	2.03	**0.074**	2.87	0.08	2.51	0.074	4.94	0.116	1.74	0.062	1.87	0.062	1.79	**0.062**	3	0.082	9.64	0.329	10.1	0.325	10.1	**0.326**	15.6	0.432
3D-grid	1.02	**0.635**	1.36	0.772	1.06	0.68	2	1.21	0.871	0.04	0.799	0.039	0.823	**0.036**	1.2	0.048	24.7	1.36	23.4	1.14	23.2	**1.16**	31.5	1.31
soc-LJ	2.37	**0.074**	3.32	0.083	2.84	0.075	5.5	0.132	1.85	0.059	1.88	0.061	1.79	**0.055**	3.11	0.088	3.63	0.139	4.87	0.141	4.38	**0.138**	6.99	0.203
cit-Patents	1.16	**0.044**	1.66	0.05	1.52	0.046	2.61	0.069	0.884	**0.032**	0.857	0.034	0.849	**0.032**	1.33	0.042	3.28	**0.14**	3.97	0.145	4.05	0.145	6.02	0.179
com-LJ	1.87	**0.061**	2.63	0.067	2.26	0.062	4.47	0.108	1.51	0.049	1.5	0.045	1.42	**0.041**	2.46	0.068	4.02	0.124	3.59	0.127	3.62	**0.128**	6.65	0.181
com-Orkut	3.7	0.108	4.31	0.119	3.74	**0.094**	8.52	0.223	4.53	0.158	4.16	0.146	3.98	**0.144**	7.44	0.236	5.55	0.241	6.26	0.246	5.78	**0.228**	13.2	0.427
nlpkkt240	10.6	0.547	14	0.596	10.5	**0.49**	23	0.927	7.36	0.24	6.89	0.224	8.09	**0.226**	9.2	0.269	142	4.88	144	4.46	141	**4.43**	265	6.91
Twitter	76.4	3.35	82.2	2.42	72.3	**2.27**	147	3.83	48.6	2.68	73.5	2.74	70.5	**2.66**	95.2	3.15	41.3	1.14	50.2	1.11	34.9	**1.06**	65.7	1.68
uk-union	71.1	5.57	53.2	2.73	45.7	**2.61**	76.1	3.9	74.4	4.89	56.9	**2.26**	52.3	2.24	64.8	2.65	42.9	2.9	45.1	1.74	42.8	**1.53**	63.8	2.34
Yahoo	307	12.1	309	10.7	271	**9.84**	500	15.8	263	8.2	258	**7.73**	238	7.39	347	9.79	176	**6.28**	225	6.54	210	6.11	331	8.92

Table 2: Running times on 40 cores with hyper-threading on different applications for the original Ligra (orig.), Ligra+ using byte coding (byte), byte coding with run-length encoding (byte-RLE) and nibble coding (nibble).

graph obtained from `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g` and we symmetrized it to get a larger graph. Twitter and uk-union are asymmetric, and the rest of the graphs are symmetric. We remove all self and duplicate edges from the graphs.

Compression Quality. We experiment with several graph reordering schemes (e.g. [3, 16]) to improve the locality (i.e. renumber vertices such that the IDs of vertices and their neighbors are close), and hence compression of the graphs. While for most graphs, applying the best reordering algorithm improves compression, the locality of our real-world graphs is already quite good without reordering. In our experiments, we use the best ordering for each graph, but confirmed that reordering is not always necessary to obtain good compression.

Figure 5 compares the average bits per edge required for byte coding (***byte***), run-length encoded byte coding (***byte-RLE***), and nibble coding (***nibble***) using the best reordering algorithm for each graph. For reference, we also show that the uncompressed graph in Ligra requires 32 bits per edge. For the input graphs, all three coding schemes use many fewer bits per edge than in Ligra (at most 19 bits per edge). Among the three coding schemes, nibble codes require the least space, followed by byte codes, and finally byte-RLE codes.

In Table 1, we list the size required to store each graph in Ligra, Ligra+ with byte coding, run-length encoded byte coding, and nibble coding. This includes the edges, vertex offsets, and vertex degrees (for Ligra+). For graphs that have a high vertex-to-edge ratio (i.e. 3D-grid) the space savings of Ligra+ compared to Ligra are smaller, since Ligra+ does not compress vertices. However, for graphs with good compression and/or low vertex-to-edge ratio, such as nlpkkt240 and uk-union, the space savings are up to 3x for byte and byte-RLE coding and 4x for nibble coding. *On average, byte codes, byte-RLE codes and nibble codes reduce the space to about 53%, 56% and 49% of the uncompressed size, respectively.*

Running Time. In Table 2, we report the times using a single-thread (T_1) and times using 40 cores with hyper-threading (T_{40}) for each application on each input graph. The time for encoding graphs is not included in the running times, as this process only needs to be done once per graph and is hence the cost is amortized across all subsequent computations on the graph. However, we note that the encoding step is quite efficient as it essentially amounts to a

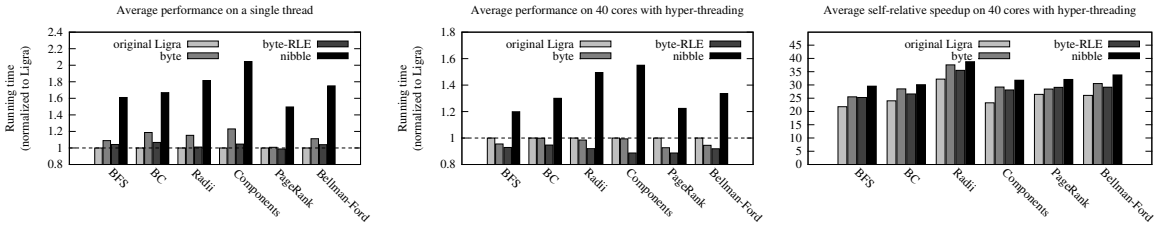


Figure 6: Average performance of Ligra+ relative to Ligra for each application on a single-thread (**left**) and on 40 cores with hyper-threading (**center**). Average self-relative speedup over all inputs for each application on 40 cores with hyper-threading (**right**).

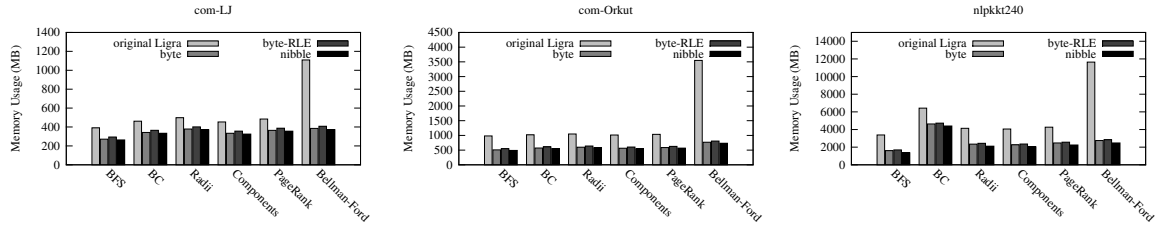


Figure 7: Peak memory usage of graph algorithms on com-LJ (**left**), com-Orkut (**center**) and nlpkkt240 (**right**).

scan over each vertex's edges, and is done in parallel. We also plot the average performance per application of Ligra+ with each encoding scheme relative to Ligra in Figure 6 (left and center). We see that sequentially, Ligra+ is slower on average than Ligra for all of the applications except PageRank, but in parallel, Ligra+ with byte-RLE or byte codes is faster on all applications. In parallel, Ligra+ using nibble codes is still generally slower than Ligra due to the high overhead of decoding, but not by as much as on a single thread (see Figure 6). Decoding nibble codes is slower than decoding byte and byte-RLE codes because the operations are not on byte-aligned memory addresses. Ligra+ with byte-RLE codes is generally faster than with byte codes because there is a lower decoding overhead.

Graph algorithms are memory-bound, and the reason for the improvement in the parallel setting is because memory is more of a bottleneck in parallel than in the sequential case, and so the reduced memory footprint of Ligra+ is important in reducing the effect of the memory bottleneck. In addition, the decoding overhead is lower in parallel than sequentially because it gets better parallel speedup relative to the rest of the computation.

Overall, Ligra+ is at most 1.1x slower and up to 2.2x faster than Ligra on 40 cores with hyper-threading. *On average, over all applications and inputs, Ligra+ using byte-RLE codes is about 14% faster than Ligra in parallel and about 8% faster using byte codes*. In parallel, Ligra+ using nibble codes is about 35% slower than Ligra on average. The graphs with better compression (e.g. nlpkkt240 and uk-union) tend to have better performance in Ligra+. For the larger graphs, Ligra+ outperforms Ligra in most cases because vertices tend to have higher degrees and neighbors no longer fit on a cache line, making the reduced memory footprint a more significant benefit. Sequentially, Ligra+ is slower than Ligra by about 3%, 13% and 73% on average when using byte-RLE, byte, and nibble codes, respectively.

We plot the average parallel self-relative speedups (T_1/T_{40}) over all inputs of each of the coding schemes per application in Figure 6 (right). Both Ligra and Ligra+ achieve good speedups on the applications—at least a factor of 20 for Ligra and 25 for Ligra+. The three compression schemes all achieve better speedup than Ligra. Again, this is because compression alleviates the memory bottleneck which is a bigger issue in parallel, and the overhead of decoding is lower because it has better parallel scalability relative to the rest of the computation.

Memory Usage. In Figure 7, we plot the peak memory usage of the applications using Ligra

and Ligra+ for several graphs. For all graphs, Ligra+ has a lower peak memory usage than Ligra. Since the applications use auxiliary data structures of size proportional to the number of vertices, for graphs with a low vertex-to-edge ratio (i.e. com-Orkut and nlpkkt240), we see a significant saving in memory usage with Ligra+ compared to Ligra, and for other graphs the saving is lower.

5 Conclusion

In this work, we develop Ligra+ by integrating compression techniques into Ligra, a large-scale shared-memory graph processing framework that supports a broad set of graph algorithms. Using Ligra+, we are able to process graphs in parallel faster and using less memory than Ligra. This makes Ligra+ attractive for processing large graphs in shared memory. Ideas for future work include integrating other compression schemes into Ligra+ and exploring the performance of compression methods in other graph processing systems.

Acknowledgments. This work is supported by the National Science Foundation under grant number CCF-1314590, the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program, and a Facebook Graduate Fellowship.

References

[1] M. Adler and M. Mitzenmacher. "Towards Compressing Web Graphs". In *DCC*. 2001.
[2] D. K. Blandford et al. "An Experimental Analysis of a Compact Graph Representation". In *ALENEX*. 2004.
[3] D. K. Blandford et al. "Compact Representations of Separable Graphs". In *SODA*. 2003.
[4] G. E. Blelloch et al. "Hierarchical Diagonal Blocking and Precision Reduction Applied to Combinatorial Multigrid". In *SC*. 2010.
[5] P. Boldi and S. Vigna. "The webgraph framework I: compression techniques". In *WWW*. 2004.
[6] P. Boldi et al. "A Large Time-Aware Graph". In *SIGIR*. 2008.
[7] N. Brunelle et al. "Algorithms for Compressed Inputs". In *DCC*. 2013.
[8] G. Buehrer and K. Chellapilla. "A scalable pattern mining approach to web graph compression with communities". In *WSDM*. 2008.
[9] A. Buluç et al. "Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication". In *IPDPS*. 2011.
[10] F. Chierichetti et al. "On Compressing Social Networks". In *KDD*. 2009.
[11] J. Gonzalez et al. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In *OSDI*. 2012.
[12] D. Hannah et al. "Analysis of Link Graph Compression Techniques". In *European Conference on Advances in Information Retrieval*. 2008.
[13] U. Kang et al. "GBASE: an efficient analysis platform for large graphs". *VLDB J.* (2012).
[14] U. Kang et al. "PEGASUS: mining peta-scale graphs". *Knowl. Inf. Syst.* (2011).
[15] C. Karande et al. "Speeding Up Algorithms on Compressed Web Graphs". In *WSDM*. 2009.
[16] G. Karypis and V. Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". *SIAM J. Sci. Comput.* (1998).
[17] K. Kourtis et al. "CSX: an extended compression format for spmv on shared memory systems". In *PPOPP*. 2011.
[18] K. Kourtis et al. "Exploiting compression opportunities to improve SpMxV performance on shared memory systems". *TACO* (2010).
[19] H. Kwak et al. "What is Twitter, a social network or a news media?" In *WWW*. 2010.
[20] Y. Lim et al. "SlashBurn: Graph Compression and Mining beyond Caveman Communities". *TKDE* (2014).
[21] A. Lugowski et al. "A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis". In *SDM*. 2012.
[22] G. Malewicz et al. "Pregel: a system for large-scale graph processing". In *SIGMOD*. 2010.
[23] K. H. Randall et al. "The Link Database: Fast Access to Graphs of the Web". In *DCC*. 2002.
[24] J. Shun and G. E. Blelloch. "Ligra: A Lightweight Graph Processing Framework for Shared Memory". In *PPoPP*. 2013.
[25] J. Willcock and A. Lumsdaine. "Accelerating sparse matrix computations via data compression". In *ICS*. 2006.