Journal of Parallel and Distributed Computing 147 (2021) 152–165

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

CIC-PIM: Trading spare computing power for memory space in graph processing

Yongxuan Zhang [a,b,c], Hong Jiang [d], Fang Wang [a,b,*], Yu Hua [a,b], Dan Feng [a,b], Yongli Cheng [a,b,e], Yuchong Hu [a,b], Renzhi Xiao [a,b]

[a] Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Engineering Research Center of data storage systems and Technology, Ministry of Education of China, China
[b] School of Computer Science and Technology, Huazhong University of Science and Technology, China
[c] School of Mathematics and Computer, Yuzhang Normal University, China
[d] University of Texas at Arlington, United States of America
[e] College of Mathematics and Computer Science, Fuzhou University, China

ARTICLE INFO

Article history:
Received 8 October 2018
Received in revised form 2 November 2019
Accepted 11 September 2020
Available online 22 September 2020

MSC:
00-01
99-00

Keywords:
Graph processing
Parallel processing
Index compression
Shared-memory

ABSTRACT

Shared-memory graph processing is usually more efficient than in a cluster in terms of cost effectiveness, ease of programming and runtime. However, the limited memory capacity of a single machine and the huge sizes of graphs restrains its applicability. Hence, it is imperative to reduce memory footprint. We observe that index compression holds promise and propose CIC-PIM, a lightweight encoding with chunked index compression, to reduce the memory footprint and the runtime of graph algorithms. CIC-PIM aims for significant space saving, real random-access support and high cache efficiency by exploiting the ubiquitous power-law and sparseness features of large scale graphs. The basic idea is to divide index structures into chunks of appropriate size and compress the chunks with our lightweight fixed-length byte-aligned encoding. After CIC-PIM compression, two-fold larger graphs are processed with all data fit in memory, resulting in speedups or fast in-memory processing unattainable previously.

1. Introduction

Graphs are important data structures to conveniently model a wide range of real-world scenarios such as social networks, the Web, road networks, etc. With the increasing scales of graphs, graph processing has attracted considerable research interest [8,20,37,44,46–48,52]. A triangle, i.e., a subgraph of three vertices pairwise connected, is an important concept in graph structure analysis. Triangle Counting (**TC**), which obtains the number of triangles in a graph, is a fundamental tool in graph processing to compute important graph metrics such as clustering coefficient and transitivity ratio [51], and has been widely used in real-world applications [4,6,20,36,45,50,55,58,63,64,70]. In **common graph algorithms** [18] such as BFS and PageRank [59,60], a vertex only needs to access its own neighbors. However, TC algorithms are unique because a vertex usually needs to access the neighbors of its own neighbors [18]. Thus, the graphs used by state of the art TC algorithms usually need different processing (Section 2.2).

Given the tight coupling of different parts in a graph, graph processing in a shared-memory machine with all data fit in memory, i.e., shared-memory graph processing, is usually more efficient than in a cluster from the perspectives of efficiency, ease of programming and speed [20,30,59,60]. Current commodity servers can be equipped with hundreds of GBs or even TBs memory and tens of cores, enabling them to efficiently process graphs with billions of edges in memory [20,48,52,59]. This allows companies such as Twitter to process graphs in a single server [30].

The combination of the huge sizes of graphs and the limited memory capacity of a single server restrains the applicability of shared-memory graph processing [12]. The rapid growth of graph size and the relatively slow growth of memory capacity exacerbate the problem. On the other hand, the long-standing and ever-widening gap between the computing power of CPU and the bandwidth of memory, i.e., the memory wall problem, grows with more cores integrated in a single CPU. Therefore, trading the spare computing power of CPU for memory space, e.g., reducing data sizes by compression, is a reasonable solution

* Corresponding author at: Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Engineering Research Center of data storage systems and Technology, Ministry of Education of China, China.
E-mail addresses: zyx126com@126.com (Y. Zhang), hong.jiang@uta.edu (H. Jiang), wangfang@hust.edu.cn (F. Wang), csyhua@hust.edu.cn (Y. Hua), dfeng@hust.edu.cn (D. Feng), chengyongli@fzu.edu.cn (Y. Cheng), yuchonghu@hust.edu.cn (Y. Hu), xrz@hust.edu.cn (R. Xiao).

https://doi.org/10.1016/j.jpdc.2020.09.008
0743-7315/

to the aforementioned problem and can be increasingly reasonable with the gap becoming wider. To the best of our knowledge, no prior work tries to address the index compression problem of the popular graph representation CSR (Compressed Sparse Row, detailed in Section 2.1) in the large scale graph processing scenario, potentially missing an important opportunity to improve shared-memory graph processing.

Existing graph compression schemes only consider the sequential settings and are not suitable for share-memory processing [10–13,42,67], try to compress specific graph representations such as trees [21,28] and can only support limited graph algorithms, aim at accelerators such as FPGAs but CPUs [37,46] or can only gain reasonable results on graphs of specific scale range [8]. Ligra+ is a shared-memory graph processing system working on compressed CSR-represented graphs of any scale and efficiently supports a wide range of graph algorithms [60]. Nevertheless, Ligra+ does not include any TC algorithms and only compresses the adjacency list (**adjlist**), i.e., leaves index data uncompressed (detailed in Section 2.1). Our evaluations show that index data also need to be compressed due to its up to around 80% percentage, as detailed in Section 2.4.

To mainly compress the index structures of CSR-represented graphs, we propose a chunk-based index compression scheme, called *Chunked Index Compression for Parallel In-Memory graph processing* (**CIC-PIM**, detailed in Section 3.1). The key idea of CIC-PIM is to exploit the ubiquitous power-law and sparseness features (Section 2.4) of large scale graphs, and divide index structures into chunks of appropriate size, then compress the chunks with our novel lightweight fixed-length byte-aligned encoding.

Evaluations show that, CIC-PIM achieves space savings of more than 60% on index data and more than 50% on entire graphs while still improving speed compared with state of the art approaches. After CIC-PIM compression, graphs of up to 2-fold (the size ratio of uncompressed graphs to CIC-PIM-compressed graphs) larger than those uncompressed can be processed with all data fit in memory, resulting in speedups or fast in-memory processing unattainable previously. The CIC-PIM advantages stem from its unique compression-friendliness after chunking, real random-access-supported and cache efficient design of the compressed index structures, and lightweight decoding routine.

The contributions of the paper include:

- Mainly to compress the index structures of graphs, we design a scheme CIC-PIM, which achieves significant space savings while still improving speed.
- We conduct in-depth analysis to demonstrate the efficacy of the techniques in CIC-PIM.
- We perform extensive evaluations driven by nine real-world graphs to evaluate the efficacy of CIC-PIM. Results indicate notable improvements over state of the art approaches.

The rest of the paper is organized as follows. Section 2 presents the background information which motivates our studies. The CIC-PIM scheme and the related issues are described in Section 3. Section 4 shows and analyzes experimental results. In Section 5, existing solutions and related works are reviewed. Finally, Section 6 concludes the paper with remarks on future work.

2. Background and motivation

In this section, we first briefly introduce the widely used CSR, i.e., the representation of large scale graphs and sparse matrices, the efficient heuristics widely used by TC algorithms, and the vByte encoding to lay a foundation for our work, then present observations that motivate our work. For ease of reference, Table 1 lists frequently used notations and concepts.

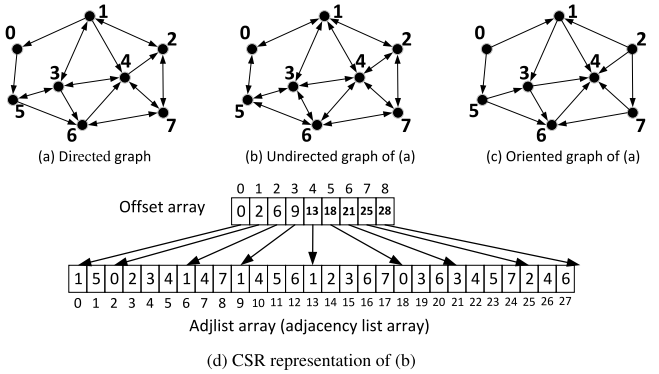


Fig. 1. Directed, undirected, oriented graphs, and CSR representation. In practice, each edge in undirected graphs is represented with two reciprocal directed edges. To utilize the symmetric feature of undirected graphs, only out-edges of each vertex are stored. As shown in the CSR of (b), adjlists of all vertices are merged into a single **adjlist array**, where each adjlist is sorted in the ascending order. The beginning offset (into adjlist array) of the adjlist of each vertex is stored in the **offset array**, where the additional, i.e., the last slot contains the number of edges (28) for convenience of programming. For a directed graph, because of its asymmetric feature, the in-edges of each vertex also need to be stored in another CSR structure. Note that, the oriented graph has significantly less directed edges than unoriented graphs (the directed and undirected graphs).

2.1. CSR representation and compressed graphs

Compressed Sparse Row (CSR [25,68]) is one of the most widely used representations of large scale graphs and sparse matrices due to its low memory requirements, simplicity, and high efficiency [1,5,43,52,59–61,70]. A toy graph and its CSR are shown in Fig. 1. Note that, in CSR degrees need not be stored because $degree_i = offset_{i+1} - offset_i$.

For a compressed graph, in addition to the offset array and the compressed adjlist array, a **degree array** is introduced to store degrees (shown in Fig. 4), because degrees cannot be obtained from the offset array due to the compression of the adjlist array with variable length encoding (detailed in Section 2.3). More specifically, because of the uncertainty of the codeword length of variable length encoding, $degree_i$ can no longer be calculated from $offset_{i+1}$ and $offset_i$. The offset array and degree array are collectively called **index data**, which is left uncompressed in Ligra+. A graph with a compressed adjlist array but uncompressed index data is defined to be **partially-compressed**, i.e., Ligra+-compressed and a graph with the adjlist array and index data both compressed is defined to be **fully-compressed**, i.e., CIC-PIM-compressed.

2.2. State of the art TC algorithms and orientation heuristics

Almost all the state of the art TC algorithms (e.g., [1,16,20,26,34,38,53,61,70]) leverage an efficient degree-based orientation heuristics, which is based on a total order $\prec$ of vertices, which is defined as follows [1]:

$u \prec v \Longleftrightarrow d_u < d_v$ *or* $(d_u = d_v$ *and* $u < v)$

Where d_u is the degree of the vertex with ID u. As shown in Fig. 1(c), every pair of reciprocal directed edges in the undirected graph is replaced with a single directed edge, whose direction is from the lower degree vertex to the other. This process results in the so-called **oriented graph** [16], which has only half of the directed edges of the undirected graph and only need to store out-edges in CSR. Accordingly, directed and undirected graphs are collectively called **unoriented graph**s, which are used by Ligra+ because only common graph algorithms included there.

As shown in Fig. 1, the number of directed edges hence the size of adjlist array in an oriented graph is usually significantly

Table 1
Notation and Concept.

Notation/Concept	Description
TC	Triangle Counting or Triangle Counting algorithm(s)
common graph algorithms	Algorithms in which a vertex only needs to access its own neighbors, in contrast to TC (Section 1 Par. 1)
adjlist	Adjacency list, i.e., the neighbor list of a vertex
partially-compressed graph	A graph with adjacency lists compressed but index data uncompressed, i.e., Ligra+-compressed graph (Section 2.1 Par. 2)
fully-compressed graph	A graph with adjacency lists and index data both compressed, i.e., CIC-PIM-compressed graph (Section 2.1 Par. 2)
adjlist array	The array containing the adjlists of all vertices (Section 2.1 Par. 2)
offset array	The array containing the beginning offsets (into the adjlist array) of adjlists of all vertices (Section 2.1)
degree array	The array containing the degrees of all vertices (Section 2.1)
index data/structures	The degree array and the offset array (Section 2.1 Par. 2)
orientation	The efficient heuristics used in almost all state of the art TC algorithms (Section 2.2)
oriented graph	The TC-algorithm-used graphs on which orientation has been executed. (Section 2.2)
unoriented graph	In contrast to oriented graphs, including directed and undirected graphs used by common graph algorithms (Section 2.2)
MTC	The fastest shared-memory TC algorithm working on uncompressed graphs (Section 2.2 Par. 4)
CMTC	The proposed MTC-based algorithm working on Compressed graphs (Section 3.3 and Fig. 8)

Algorithm: MTC

```
Input: an uncompressed graph G
Output: triangle count
1 t ← 0;
2 foreach u ∈ V in parallel do          // V - vertex set of G
3     foreach v ∈ A_u do                // A_u - adjlist of vertex u
4         t ← t + |Intersect(A_u, A_v)|;
5 return t;
```

Fig. 2. MTC – the simplified version of the fastest known shared-memory TC algorithm [20,61]. For each edge (u, v) (line 2 to 3), the intersection of the adjlists, i.e., neighbor lists of u and v is computed and the cardinality of the intersection is added to the counter (line 4).

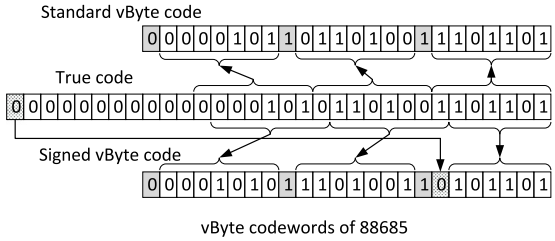


Fig. 3. How "88685" is encoded with vByte. vByte tries to remove the leading zeros in the true code to save space. In standard vByte, which is used to encode non-negative integers, the 7-bit chunks containing at least one 1 in the true code are encoded in the corresponding bytes of the codeword with the most significant bit (shaded) of each byte is set to 1 or 0 to indicate whether the following byte is also a part of the codeword or not. In signed vByte, the second most significant bit (dotted) of the first byte in the codeword is used to indicate the sign (0 for positive), and thus the first byte of the codeword only contains a 6-bit chunk of the true code.

smaller than that in the unoriented graph (directed and undirected graphs). On the other hand, the size of index data stays unchanged because it is determined by the number of vertices. Therefore, an oriented graph usually contains a higher proportion of index data than the unoriented graph.

TC algorithms working on oriented graphs are much faster than those working on unoriented graphs [1,16,20,26,53,61,70]. The speedup mainly stems from the reduced time complexity [1]. The fastest known shared-memory TC algorithm, which also leverages the orientation heuristics, comes from [20,61] and is called **MTC** (Multicore Triangle Counting) in this paper. For understandability, a simplified MTC is shown in Fig. 2. In our proposed work, elaborated in Section 3.3, MTC is modified to work on compressed graphs.

2.3. Variable-length byte-aligned encoding

The scheme used to compress the adjlist array in CIC-PIM is a variant of the standard variable-length byte-aligned encoding (**standard vByte**, shown in Fig. 3) [57]. To compress non-negative integers, the standard vByte tries to remove leading zeros (if any) to save space. A codeword of standard vByte consists of a variable number of bytes aligned to the physical byte boundary for fast access. In graph compression, the integers to be compressed can be negative. Standard vByte is modified to encode the possible negative integers. The second most significant bit of the lowest byte in the codeword is used to encode the sign, and the modified scheme is called **signed vByte**, as shown in Fig. 3. Ligra+ shows that vByte achieves a better compromise of runtime and space saving than other schemes.

In CIC-PIM, vByte is borrowed to compress the adjlist array. First, the deltas (i.e., differences) between consecutive elements in an adjlist are calculated. For the first element in the adjlist, the delta between the element and the vertex itself is calculated and thus maybe negative. Then the first delta is encoded with signed vByte and other deltas with standard vByte. Due to the byte-aligned feature, both the encoding and decoding routines of vByte are fast [57]. The decoding routine is called on the fly in graph algorithms and incurs limited runtime overhead.

2.4. Observations and motivations

Ligra+ [60], a shared-memory graph processing system working on compressed graphs of any scale, uses the popular CSR representation and supports many common graph algorithms efficiently. However, Ligra+ does not include any TC algorithm and only compresses the adjlist array. Furthermore, for the (partially) compressed graphs used by Ligra+, the percentage of index data is up to 43% with an average of 22%. *It is a considerable proportion, and we should compress the index data too.*

As stated in Section 2.2, oriented graphs used by TC algorithms contains larger portions of index data than the unoriented graphs used by Ligra+. Our evaluations confirm this prediction and show

that the percentages of index data in partially-compressed oriented graphs can be up to around 80% for sparse graphs with an average of 36% (raw data are omitted). *This more clearly suggests that we must compress the index data too.*

Large scale graphs are usually sparse, i.e., have low average degrees [45]. To find out to what extent the low average degrees are, we collect statistics of all the 85 graphs from SNAP[1], one of the most popular large scale graph repositories. Results show that the average degree of all graphs is 14.8 with two graphs having average degrees of more than 100 (127 and 337). Moreover, most large scale graphs are **power-law** graphs [27], i.e., they have highly skewed degree distributions. Specifically, power-law graphs usually have a small number of huge degree vertices and a vast majority of low degree vertices, i.e., vertices with degrees of no more than hundreds.

Given the sparseness and power-law features, if the index structures, i.e., offset array and degree array, are divided into equal-sized chunks containing hundreds of consecutive vertices each, the maximum degrees will be no larger than hundreds in most chunks that only contain low degree vertices. For each chunk, we choose the first offset as the reference and calculate the deltas between other offsets and the reference, the maximum offset-deltas are no larger than tens of thousands in most chunks. Therefore, the degrees and offset-deltas in most chunks are potentially small enough to be encoded with fixed-length byte-aligned codewords of no more than two bytes each (detailed in Section 3.1.3). For example, assuming each chunk contains 256 vertices and the maximum degrees in most chunk are less than 200, then in most chunks, degrees can be stored with one byte each ($200 < 2^8$) and offset-deltas can be stored with two bytes each ($255 \times 200 < 2^{16}$).

A small fraction of real-world graphs are not power-law graphs. However, given the ubiquitous sparseness feature of large scale graphs [45], the aforementioned arguments on the smallness of degrees and offset-deltas in most chunks of power-law graphs are also tenable for non-power-law graphs.

The considerable proportion of index data motivates us to design an index encoding scheme suitable for shared-memory graph processing. When dividing the index structures into chunks, the smallness of the maximum degrees and offset-deltas in most chunks inspires us to propose a scheme called CIC-PIM (Section 3.1).

3. Design and implementation

3.1. CIC-PIM

When compressing a graph, CIC-PIM first compresses the adjlist array (using vByte), then compresses the index structures and obtains the fully-compressed graph. During the processing of fully-compressed graphs, all decoding work is done on the fly. Whenever the original offset and degree of a vertex are needed, the decoding routine of CIC-PIM is called. After the original offset and degree are decoded, they are used by the decoding routine of vByte to obtain the original adjlist of the vertex.

3.1.1. Design

Because we need to compress the index structures of time-critical shared-memory graph processing, the design of CIC-PIM must achieve the following three goals simultaneously: reasonable space saving, real random-access-supported and cache-efficient compressed structures, and lightweight decoding routine. Real random-access support is necessary because most accesses to index are random. To achieve the goals, we adopt three techniques:

- **Considerable space saving via chunking of index structures**: As discussed in Section 2.4, after dividing the index structures into equal-sized chunks containing hundreds of vertices each, the degrees and offset-deltas in most chunks are no larger than tens of thousands, and can mostly be encoded with one or two bytes each. This ensures considerable space saving.
- **Real random-access support via fixed-length byte-aligned encoding**: All degrees and offset-deltas of the same chunk are encoded with byte-aligned codewords of the same length, i.e., fixed-length byte-aligned encoding. The fixed-length feature makes real random access possible, as detailed in Section 3.1.4. Fixed-length encoding is rarely adopted because of its usually low space saving [57]. However, due to the compression amenability after chunking, CIC-PIM achieves reasonable space saving (Section 4.5). Furthermore, the byte-aligned feature also contributes to the speed of the decoding routine (Section 3.1.4).
- **High cache-efficiency via Array-Of-Structures memory layout (AOS)**: The AOS memory layout [62] detailed in Section 3.2.1 is used for high cache-efficiency: The codewords of degrees and offset-deltas are interlaced in the compressed index structure, which ensures the codewords of the degree and offset-delta of the same vertex are placed in consecutive memory cells and reside in the same cache line with high probability (Section 3.1.2). The degree and offset of a vertex are usually obtained successively, and thus the AOS layout helps to effectively improve cache-efficiency. The reasons behind the improvement are detailed in Section 3.2.1.

Due to the careful design of CIC-PIM, we can implement a lightweight decoding routine without branch mispredictions to obtain original degrees and offsets, as detailed in Section 3.1.4. Therefore, CIC-PIM achieves its design goals by the combined use of the three techniques.

3.1.2. Implementation

A key part of the implementation of CIC-PIM is shown in Fig. 4. Note that, we only show the compression of index data, i.e., we assume that the adjlist array has been compressed (with vByte) in advance. For ease of depiction and description, we assume that original degrees and offsets are separately stored in two integer arrays. However, actually the original degrees and offsets are also interlaced in our implementation, i.e., they are also stored in AOS layout for high cache efficiency. Offset-deltas of a chunk are calculated before the compression of the chunk and stored in an integer array. The compressed index structures and adjlist array are separately stored in two byte arrays (the *compressed index data* array and *compressed adjlist* array).

The metadata of chunks are stored in a high level index structure (the *chunk index* array), which is implemented as an array of structures with four fields, i.e., AOS layout, instead of four different arrays for high cache efficiency, because the fields are usually accessed successively. The fields are used to store the following metadata of a chunk: the **reference offset**, i.e., the first offset; the start index; the length of one offset-delta codeword; the length of one degree codeword. Note that, reference offsets are subscripts in the *compressed adjlist* array and start indexes are subscripts in the *compressed index data* array. We use different terms to avoid having them mixed up. The metadata are left uncompressed for fast access. The size of metadata is small relative to the size of the index data itself, because each chunk contains hundreds of vertices.

[1] http://snap.stanford.edu/

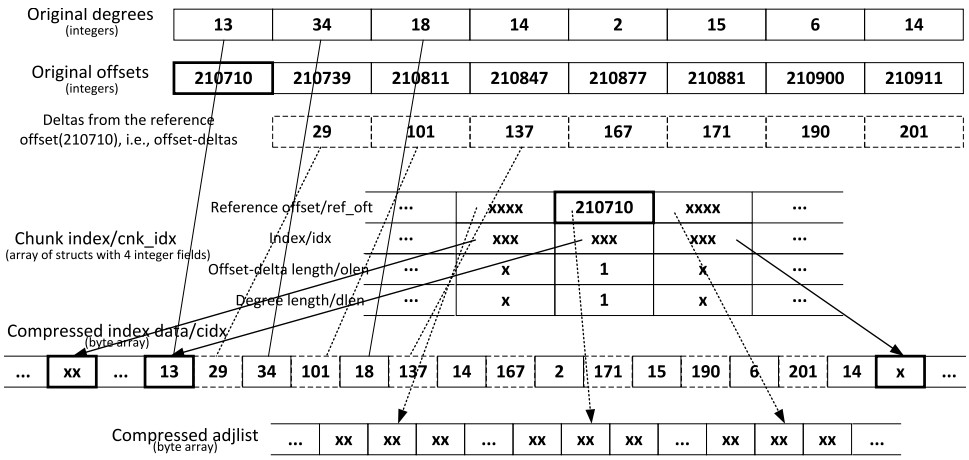


Fig. 4. CIC-PIM when encoding a chunk of index data. Note that, we only depict the compression of index data (original degrees and offsets), i.e., the adjlist array has been previously compressed and stored in the *compressed adjlist* array. Original offsets are the subscripts in the *compressed adjlist* array. For ease of depiction and description, we assume that original degrees and offsets are stored in two integer arrays. However, actually the original degrees and offsets are also interlaced as their codewords in our implementation, i.e., they are also stored in AOS layout for high cache efficiency. Original degrees and offset-deltas are compressed, interlaced and stored in the *compressed index data* array. The metadata of chunks are left uncompressed for fast access and stored in an array of structures (AOS) with four fields, i.e., the *chunk index* array.

3.1.3. Workflow

We take a chunk for example to show the workflow of index compression, as detailed in Fig. 4. For ease of depiction, the chunk size is assumed to be eight. Because the start vertex ID of a chunk can be calculated from the chunk ID and the chunk size, a dedicated chunking step is unnecessary. To compress the current chunk of index data, the encoding routine of CIC-PIM operates in the following order:

a. First the current index of the *cidx* array is put into the *idx* field of the current element in the *cnk_idx* array. Then the *original degrees* are scanned to obtain the maxima, which is used to determine the codeword length of every degree. In Fig. 4, the maxima is 34 and can be stored in one byte. Hence, every degree in the chunk is encoded with one byte.

b. The length of degree codeword (1) is put into the *dlen* field of the current element in the *cnk_idx* array. Next, the codeword of the first degree (13) is put into the *cidx* array and takes up one byte. The first degree needs special process because the corresponding offset, i.e., the *reference offset* are left uncompressed and stored in the *cnk_idx* array as stated in the next step.

c. The first *original offset*, i.e., the *reference offset* (210710) is put in the *ref_oft* field of the current element in the *cnk_idx* array. Then other *original offset*s are scanned and their deltas (i.e., differences) from the *reference offset* are calculated. The last delta, i.e., the maxima, determines the codeword length of every delta. In Fig. 4, the maxima is 201, and can be encoded with one byte.

d. Then the length of offset-delta codeword (1) is put into the *olen* field of the current element in the *cnk_idx* array. Next, all codewords of offset-deltas and the remaining codewords of degrees are interlaced and put into the *cidx* array sequentially, i.e., are stored in AOS layout.

3.1.4. Decoding routine

DEOD, the DEcoding routine of CIC-PIM to obtain the original Offsets and Degrees, is shown in Fig. 5. It is lightweight for three reasons: First, most of its work is done with bitwise operations on byte-aligned data; Second, no extra data need to be decoded due to the real random-access-supported design, i.e., DEOD need not decode any preceding data (Many existing schemes cannot support real random access and need to decode some preceding data [10–12]); Third, the if-else and other (not shown) branch statements in DEOD are predictable because the branch condition

Algorithm: DEOD

Input: a **fully-compressed** graph G and a vertex v
Output: original offset and degree of v

```
1  cnk_id ← floor(v/CHK_SZ); // CHK_SZ - #vertices in a chunk
2  in_chk_no ← v mod CHK_SZ;
3  ref_oft ← G.cnk_idx[cnk_id].ref_oft;
4  idx ← G.cnk_idx[cnk_id].idx;
5  olen ← G.cnk_idx[cnk_id].olen;
6  dlen ← G.cnk_idx[cnk_id].dlen;
7  if in_cnk_no = 0 then                    // the 1st vertex in a chunk
8      offset ← ref_oft;
9      degree ← the integer decoded from the consecutive dlen
           bytes starting from idx;
10 else
11     idx ← idx + dlen;
12     --in_cnk_no;                         // skip the special 1st vertex
13     idx ← idx + (olen + dlen) * in_cnk_no;
14     delta ← the integer decoded from the consecutive olen
           bytes starting from idx;
15     offset ← ref_oft + delta;
16     idx ← idx + olen;
17     degree ← the integer decoded from the consecutive dlen
           bytes starting from idx;
18 return offset and degree;
```

Fig. 5. DEOD – DEcoding routine of CIC-PIM to obtain the original Offset and Degree of a vertex. First, the ID of the chunk in which v lies and the in-chunk number of v are calculated based on the chunk size (line 1 to 2); The metadata of the chunk are read from the high level index structure, i.e., the *cnk_idx* array in Fig. 4 (line 3 to 6); If v is the first vertex in the chunk (line 7), v's offset, i.e., the *reference offset*, which was stored in the *cnk_idx* array without compression, is read directly (line 8). v's degree, which was compressed and stored in the *cidx* array, needs to be decoded (line 9). If v is not the first vertex in the chunk (line 10), because there is no offset-delta of the first vertex, variables *idx* and *in_cnk_no* are updated to skip the first vertex (line 11 to 12). Then, the starting address of the index data of v is calculated based on the codeword lengths of offset-delta and degree, i.e., *olen* and *dlen* (line 13). Finally, the offset-delta and degree are decoded successively (line 14 to 17).

statements can be evaluated in advance, i.e., they do not result in branch mispredictions [35]. In the graph algorithms working on fully-compressed graphs, whenever the adjlist of a vertex needs to be decoded, DEOD is called in advance to decode the offset and degree, then the offset and degree are used by the decoding routine of vByte to decode the adjlist.

3.2. Analysis

3.2.1. Effectiveness of AOS

The main conclusion of this section is as follows: Compared with the contrasting layout SOA (Structure Of Arrays [62]), the AOS layout helps to decrease the cache miss rates of index data

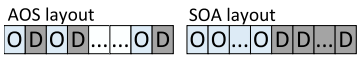


Fig. 6. The AOS and SOA memory layouts when used to stored the index data, i.e., the offsets and degrees in CIC-PIM where the AOS layout is adopted in several data structures for high cache efficiency.

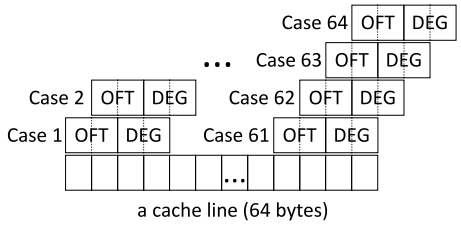


Fig. 7. When the AOS layout used, how the probability that the offset and degree of a vertex reside in the same cache line is calculated. A frame represents a byte or a memory cell. Assume that the offset and degree of a vertex take up 4 bytes totally and 2 byte each. We regard the offset in a certain cache line if the first of its 2 bytes resides in the cache line. Among the possible 64 cases, in the cases 1 to 61 the degree also completely reside in the same cache line, and thus the probability that the offset and degree reside in the same cache line is $61/64 = 0.95$.

access from nearly 100% to around 50% and gains a speedup of 12%.

When used to store the index data in CIC-PIM, the memory layouts are shown in Fig. 6. Existing works show that SOA is usually more efficient than AOS for parallel processing [32,56]. However, we will show than AOS is more efficient for graph processing. To simplify discussion, first, we assume that there is only one level of cache in the memory system (corresponds to L1 cache and the size is usually in dozens of KBs), and the other levels of caches (including caches from L2 cache to the last level cache) and the memory itself are uniformly viewed as RAM; Second, we further omit many details of hardware in discussion.

In the memory system of modern computer, when the data in a memory cell (can house a byte) is needed, CPU first search the copy of the data in the cache. If found, the need is met; If not found, CPU reads the data from memory cell, and simultaneously all the data in the chunk of consecutive memory cells containing the accessed cell are loaded into cache for near future use. The chunk is called a cache line and usually contains 64 or 128 bytes [22].

The index data of each vertex (offset and degree) is usually accessed randomly while the offset and degree are usually accessed successively. In AOS layout, the offset and degree of a vertex reside in the same cache line with high probability. Assume that both the offset and degree of a vertex take up 2 bytes each (after compression) and a cache line contains 64 bytes, the probability is $61/64 = 0.95$, as detailed in Fig. 7. After the offset of a vertex is accessed, the cache line containing the offset is loaded into cache simultaneously. The degree can now be found in cache with probability of 0.95 and the slow RAM access is avoided with the same probability. In SOA layout, the offset and degree of a vertex are so distant that they cannot reside in the same cache line, and thus cannot takes advantage of the chunked access feature of memory system.

Because the index data sizes of large scale graphs usually are much larger than the size of cache (usually MBs vs dozens of KBs), even after data warm-up there is only a small fraction of index data reside in cache. Given that the accesses to index data usually are random and the needed data cannot be correctly prefetched [22], the Cache Miss Rate (**CMR**) of the first access to a cache line should be nearly 100%. For SOA layout, the offset and degree of a vertex reside in two distant cache lines and thus the CMR when accessing them is nearly 100% ((≈100% + ≈100%) / 2 ≈ 100%). For AOS layout, the offset and degree of a vertex reside in the same cache line with a probability of 0.95, and hence the CMR is slightly more than 50% (the CMR of the access to the offset is nearly 100% and to the degree is 5%, and thus the average is 53%). The AOS layout of index data achieves an average 12% speedup than that of SOA (raw data are omitted). Because the accesses to index data only consist of a relatively small portion of data accesses, the speedup is relatively low.

3.2.2. Worst case index data space saving

The main conclusion of this section is that, the worst case space saving of CIC-PIM on index data is more than 53%. To simplify analysis, we assume (1) each chunk contains the degrees and offsets of 256 vertices as in our implementation; (2) vertex IDs, original offsets and degrees are represented with a 4-byte integer each; (3) the smallest degree is 1, i.e., there are no isolated vertices in graphs (Actually CIC-PIM can process graphs with isolated vertices). Because most large scale graphs are power-law graphs, we only analyze the index data space saving of power-law graphs here. The discussion on non-power-law graphs will be presented in Section 4.5.2.

Under the power-law degree distribution, a vertex has degree x with a probability of $P(x) \propto x^{-\alpha}$, where α is called *skewness exponent*, and the typical value of $\alpha = 2.2$ [24]. In other words, the probability density function of degree x is $f(x) = Cx^{-2.2}$, where C is a constant. Because $\int_1^{+\infty} f(x)dx = 1$, $C = 1.2$, and the exact probability density function is $f(x) = 1.2 * x^{-2.2}$. The probability that a degree lies in $[256, +\infty)$ is $P_1 = \int_{256}^{+\infty} f(x)dx = 0.00129$. The probability that a chunk contains at least one degree ≥ 256 is $P_2 = P_1 * 256 = 0.330$. The probability that a degree lies in $[2^{16}, +\infty)$ is 0.00000166. The probability that a chunk contains at least one degree $\geq 2^{16}$ is $256 * 0.00000166 = 0.000425$, which is negligible and need not be considered. Let $\bar{L_d}$ be the expected size (#bytes) of the compressed degree of a vertex, and $\bar{L_d} = 2 * P_2 + 1 * (1 - P_2) = 1.33$, where $2 * P_2$ corresponds to chunks containing at least one degree ≥ 256 and $1 * (1 - P_2)$ corresponds to chunks with every degree <256.

In aforementioned analysis, we have assumed that if a chunk contains at least one degree ≥ 256, codewords of 3 bytes each will be used to encode all the offset-deltas. This assumption is overstrict because 3-bytes codeword is needed only when the average degree in a chunk $\bar{d} \geq 256$, i.e., when $\bar{d} \geq 256$, $\bar{d} * 256 \geq 2^{16}$ and offset-deltas need 3 bytes each to be encoded (Recall that we do not consider vertices with degree $\geq 2^{16}$ due to the negligible probability). Let $\bar{L_o}$ be the expected size (#bytes) of the compressed offset-delta of a vertex, and $\bar{L_o} < 3 * P_2 + 2 * (1 - P_2) = 2.33$, where $3 * P_2$ corresponds to chunks containing at least one degree ≥ 256 and $2 * (1 - P_2)$ corresponds to chunks with every degree <256. Note that, because degrees are at least 1, the maximum offset-delta in each chunk ≥ 255 (the probability that equal to 255 is 2.9×10^{-64}) and thus at least 2 bytes needed to encode each of them. Taken together, the expected size of the compressed index data of a vertex $\bar{L} = \bar{L_d} + \bar{L_o} < 1.33 + 2.33 = 3.66$, and the space saving $> (8 - \bar{L})/8 = 54\%$. Due to the introduction of the metadata of chunks, an around 1% loss is caused and hence the space saving >53%.

It is hard to craft a model to predict the space saving of CIC-PIM on entire graph given that it is related to four variables: the average degree, the number of vertices, the skewness exponent and the chunk size. However, we can qualitatively analyze the space saving of CIC-PIM as follows. For graphs with relatively high average degrees, vByte will achieve reasonable space savings [60]. For graphs with low average degrees, the space savings of vByte are low due to the considerable proportion of index data. Nonetheless, given the reasonable space saving of CIC-PIM on index data, the space saving of CIC-PIM on entire graphs should also be reasonable.

Algorithm: CMTC

```
   Input: a compressed graph G
   Output: triangle count
1  T ← 0;
2  foreach u ∈ V in parallel do          // V - vertex set of G
3      Allocate an array A for each thread;
4      Call DEOD to decode the offset and degree of u when
       G is fully compressed or read the offset and degree
       when G is partially compressed;
       // deFirst - decoding routine of signed vByte
5      A[0] ← deFirst(CA_u); // CA_u - compressed adjlist of u
6      for j ← 0 to d_u − 1 do                // d_u - degree of u
7          v ← A[j];
8          Call DEOD to decode the offset and degree of v
           when G is fully compressed or read the offset and
           degree when G is partially compressed;
9          T ← T + |CIntersect(CA_u, CA_v, A, d_u, d_v)|;
10 return T

11 Procedure CIntersect
   Input: CA_u, CA_v, A, d_u, d_v
   Output: triangle count
12 u' ← A[0], v' ← deFirst(CA_v), t ← 0, i_1 ← 0, i_2 ← 0;
13 if v is the 1st neighbor of u then
14     while i_1 < d_u and i_2 < d_v do
15         if u' = v' then
               // deNext - decoding routine of standard vByte
16             u' ← deNext(CA_u), put u' in A;
17             v' ← deNext(CA_v), ++t, ++i_1, ++i_2;
18         if u' < v' then
19             u' ← deNext(CA_u), put u' in A, ++i_1;
20         else
21             v' ← deNext(CA_v), ++i_2;
22 else
23     while i_1 < d_u and i_2 < d_v do
24         if u' = v' then
25             ++t, ++i_1, ++i_2;
26             u' ← A[i_1], v' ← deNext(CA_v);
27         if u' < v' then
28             ++i_1, u' ← A[i_1];
29         else
30             v' ← deNext(CA_v), ++i_2;
31 return t
```

Fig. 8. CMTC – modified MTC algorithm working on partially- and fully-Compressed graphs. When working on fully-compressed graphs, **DEOD** is called in advance to decode the offset and degree whenever the adjlist of a vertex is to be decoded. During an intersection is computed, if needed, the adjlists of the two vertices u and v are decoded on the fly by calling **deFirst** and **deNext**, which are the decoding routines of signed vByte and standard vByte respectively. An optimization is introduced: An array A is created to avoid the repetitive decoding of the adjlist of u (line 3). Specifically, when the adjlist of u is intersected with the adjlist of u's first neighbor (line 13), the adjlist of u is decoded actually and stored in A (line 16 and 19). When the adjlist of u is intersected with the adjlists of u's remaining neighbors, the adjlist of u is directly obtained from A (line 26 and 28).

3.3. Implementation of TC algorithms

The fastest shared-memory TC algorithm MTC [20,61] mentioned in Section 2.2 is modified to work on partially- and fully-compressed graphs, and the new algorithm is called CMTC (Fig. 8). Orientation can be executed in preprocessing [1,16,38,53] or on the fly [26,34,61]. In [61], MTC is implemented with orientation executed on the fly because [61] needs to support edge sampling and approximate triangle counting. However, we focus on exact triangle counting, and given that orientation can be performed once and the resulting oriented graphs can be used many times, orientation is done in preprocessing in our implementation. Ligra+ does not take TC algorithms into consideration, and cannot utilize the orientation heuristics. Hence, we do not implement our work in Ligra+. Hereafter, we use the term *TC* or *TC algorithms* in place of MTC and CMTC for brevity.

3.4. Implementation of common graph algorithms

Although inspired by TC related issues, CIC-PIM is also similarly applicable to common graph algorithms. We take the state of the art BFS, PageRank, sparse matrix vector multiplication (SpMV), radii estimation (Radii) and connected components (CC) as examples to demonstrate the capability [43,59]. We first implement these algorithms working on uncompressed graphs, then naively modify them to obtain their implementations working on partially- and fully-compressed graphs. Hereafter, we use *common graph algorithms* in place of these algorithms and their implementations working on compressed graphs for brevity.

4. Evaluation and discussion

4.1. Evaluation settings

The server used is equipped with four Xeon E7-4820 v3 CPUs and 256 GB of memory. The OS is Ubuntu 14.04 with swap areas disabled to obtain real in-memory results. All programs, coded from scratch in C++ by the same person, are parallelized with OpenMP and compiled using G++ 4.9.2 with -O3 option. Programs are run with the default number of threads in OpenMP (96), i.e., twice the number of physical cores (48), except in the scalability evaluation (Section 4.7). Each time presented is the average time of three runs. Though the processing times of some relatively small graphs are short and hence notably vary from run to run, the average speedup percentages presented in Section 4.4 remain constant when we conduct the evaluations twice.

4.2. Datasets and preprocessing

The graphs used in evaluations are shown in Table 2. *clj*,[2] *slj*[2] and *twr*[2] represent social networks. *c09*,[3] *gsh*[4] and *c12*[3] are web graphs. These graphs are power-law graphs. To increase the diversity of datasets, three non-power-law graphs with different degree distributions are included: *osm*[5] is the street map of Europe; *nlp* is related to the optimization problem of linearized Karush–Kuhn–Tucker systems[6]; *3dg* is a 3D grid and generated with a tool from Ligra.[7]

To run TC algorithms, raw graphs are first processed into (uncompressed) oriented graphs. To run common graph algorithms, raw graphs are first processed into (uncompressed) unoriented graphs (directed or undirected). To obtain compressed graphs, uncompressed graphs are partially- and fully-compressed. The parallelized compression is fast due to the lightweight design of CIC-PIM. The compression times, obtained in the same server and shown in Table 3, are not included in runtimes because they can be amortized across many runs.

For common graph algorithms, whether raw graphs should be processed into directed or undirected graphs is determined by specific application scenarios. In our evaluations, the largest four graphs (from *twr* to *c12*) are processed into directed and the rest are processed into undirected, similar to what Ligra+ does.

4.3. Metrics of interest

Four metrics are used to evaluate the efficacy of our work: speedup compared with the state of the art approaches, the space saving of index data, the space saving of entire graph, and the multithreaded scalability.

2 http://snap.stanford.edu/
3 http://lemurproject.org/
4 http://law.di.unimi.it/
5 http://www.dis.uniroma1.it/challenge9/
6 https://sparse.tamu.edu/
7 https://github.com/jshun/ligra/

Table 2
Dataset Information.

Dataset	#Vertices	#Edges[a]	#Triangles	Storage Size (B)[b]
clj	3,997,962	34,681,189	177,820,130	148M
slj	4,847,571	42,851,237	285,730,264	182M
osm	173,789,186	182,620,014	102,623	1.33G
nlp	27,993,600	373,239,376	0	1.49G
3dg	299,418,309	898,254,927	0	4.46G
twr	41,652,230	1,202,513,046	34,824,916,864	4.63G
c09	4,780,950,910	7,811,398,073	31,013,037,486[c]	93.8G
gsh	988,490,691	25,690,705,118	910,140,734,636	103G
c12	6,257,706,595	66,539,548,496	3,058,034,046,618[c]	542G

[a]The number of oriented edges.
[b]Uncompressed storage size of the oriented graph.
[c]The results are different from those of [53]. We are confident in our results as we obtained the same results using another processing flow.

Table 3
Compression Time.

Dataset	Ligra+ (s)[a]	CIC-PIM (s)
clj	0.015	0.034
slj	0.017	0.040
osm	0.14	0.31
nlp	0.15	0.37
3dg	0.43	0.99
twr	0.46	1.07
c09	8.0	13.2
gsh	10.4	12.8
c12	5476	4963

For the huge *c12*, the compression must be done with the help of external storage (a SAS HDD) and is bottlenecked by I/O, which explains the long times. The fully-compressed *c12* is obtained in less time than the partially-, i.e., Ligra+-compressed graph because of its smaller size (Table 6) and hence less I/O. Other graphs are compressed with all data fit in memory and hence fast.
[a]Ligra+-, i.e., partially-compressed.

4.4. Speedup

4.4.1. Speedup for TC

As shown in Table 4, for *c12*, the huge size of uncompressed graph is more than 2-fold larger than the RAM capacity (542 GB vs 256 GB), resulting in out-of-memory error during the run of MTC. When CMTC processing the Ligra+-, i.e., partially-compressed graph (240 GB), because the graph size is very close to the RAM capacity and the swap area has been disabled, out-of-memory error also arises. After being fully compressed (165 GB), CMTC can process the graph with all data fit in memory, leading to fast in-memory processing (636 S).

For graphs other than *c12*, all data can be fit in memory even uncompressed. Compared with the speeds of MTC (on uncompressed graphs), the speedup percentages, defined as $(time_{old} - time_{new})/time_{old} \times 100\%$, of CMTC on CIC-PIM-, i.e., fully-compressed graphs range from −2% to 37% with an average of 11%. Also compared with the speeds of MTC, the speedup percentages of CMTC on Ligra+-, i.e., partially-compressed graphs range from −29% to 26% with an average of 3%. Thus, index compression with CIC-PIM not only leads to space saving but also achieves a speedup of 8%. The reasons for the moderate speedup are twofold. First, due to the careful design of CIC-PIM, the lightweight decoding routine incurs limited computing overhead, and the overhead is mostly offset by spare computing power. Second, the significantly reduced size of index data mitigates the memory wall problem and contributes to speedup. Therefore, generally speaking, speedup positively correlates with space saving. Moreover, the ratio of the compressed-graph size to the L3 cache capacity and the structure characteristics of graphs may also be important factors in some cases.

4.4.2. Speedup for common graph algorithms

For *c12*, the discussion is similar to that of TC, as stated in Section 4.4.1. For graphs except *c12*, all data can be fit in memory even uncompressed. Compared with the speeds of these algorithms on uncompressed graphs (in columns with the heading of Ligra), the average speedup percentages of BFS, PageRank, SpMV, Radii and CC on CIC-PIM-, i.e., fully-compressed graphs are 14%, 17%, 10%, 9% and 10% respectively. Also compared with the speeds of these algorithms on uncompressed graphs, the average speedup percentages on Ligra+-, i.e., partially-compressed graphs are correspondingly 6%, 2%, -1%, 5% and 2%. Because of the careful design of CIC-PIM and the alleviated memory wall problem due to index compression, speedups are achieved on all algorithms. Various graph algorithms have the same key operation in which the adjlist of a vertex is traversed with different computing, and thus there are no significant differences in speedups of different algorithms. More specifically, for Radii, the average speedups are relatively low (4%); for PageRank, the speedup is relatively remarkable (15%); for BFS, SpMV and CC, the speedups are moderate (8%, 11% and 8% respectively).

4.4.3. Runtime decomposition

To figure out the cost of decoding, we collect times spent on decoding and computing when processing CIC-PIM-compressed graphs and show the results in Table 5 (The time spent on other codes except decoding and computing are negligible). The time percentages of decoding range from 19% to 24% with an average of 22%. Specifically, for TC, BFS, PageRank, SpMV, Radii and CC, the percentages are 21%, 24%, 24%, 22%, 21% and 19% respectively. Though compared with PageRank and SpMV, in the other algorithms the computing needed after decoding the adjacency list of a vertex are relatively less, there are more branch mispredictions in the computing of the latter, and hence the percentage differences between algorithms are insignificant. Moreover, the percentage differences between graphs are also insignificant.

4.5. Index-data space saving

4.5.1. Index-data space saving for TC

For the oriented graphs used by TC, the index-data space savings are shown in the first *Index* column of Table 6. The space savings range from 59% to 82%, with an average of 69%. As discussed in Section 2.4, the noticeable space saving mainly comes from the smallness of the maximum degrees and offset-deltas in most chunks.

Higher index space savings are usually achieved on huge graphs including *c09*, *gsh* and *c12*, because in those graphs the offsets and/or degrees have to be represented with 8-byte integers, and leave larger portions of leading zeros to be removed. However, *nlp* is an exception. Its high index space saving comes

Table 4
Runtime.

Dataset	TC (s)			BFS (s)			PageRank (s)			SpMV (s)			Radii (s)			CC (s)		
	MTC[a]	Ligra+	CIC-PIM	Ligra	Ligra+	CIC-PIM	Ligra	Ligra+	CIC-PIM	Ligra	Ligra+	CIC-PIM	Ligra	Ligra+	CIC-PIM	Ligra	Ligra+	CIC-PIM
clj	.281	.273	**.265**	.047	.023	**.022**	.048	.043	**.031**	.145	.129	**.121**	1.42	1.06	**.99**	.190	.179	**.165**
slj	.413	.409	**.370**	.053	.027	**.025**	.052	.050	**.033**	.197	.221	**.164**	1.54	**1.18**	1.21	.255	.268	**.232**
osm	**.94**	1.21	.96	**.232**	.273	.247	.647	.773	**.571**	.810	.984	**.805**	84.1	91.3	**82.4**	**.170**	.188	.173
nlp	2.85	2.11	**1.80**	**1.34**	1.49	1.38	.332	.281	**.277**	1.67	1.44	**1.35**	36.5	33.8	**33.6**	.462	.417	**.411**
3dg	26.3	23.6	**19.4**	7.51	8.20	**7.45**	1.93	2.06	**1.73**	6.12	6.53	**5.76**	**324**	347	341	5.74	5.63	**5.09**
twr	47.2	**44.3**	45.9	**.793**	.889	.832	3.05	**2.61**	2.63	10.8	**10.1**	10.5	**11.0**	11.3	11.7	4.42	**4.26**	4.48
c09	75	80	**73**	9.13	9.97	**8.32**	21.4	24.3	**19.7**	37.1	41.2	**33.7**	153	167	**135**	29.5	32.1	**24.7**
gsh	183	**166**	168	11.27	**10.23**	10.36	38.1	**34.2**	35.7	84.1	**73.7**	75.4	189	**162**	168	33.4	27.5	**27.2**
c12	OOM	OOM	636	OOM	OOM	55	OOM	OOM	128	OOM	OOM	224	OOM	OOM	341	OOM	OOM	64

For each algorithm, the leftmost column (with the heading MTC or Ligra) contains the runtimes of the algorithm on uncompressed graphs; the middle column (with the heading Ligra+) contains the runtimes of the algorithm on partially-, i.e., Ligra+-compressed graphs; the rightmost column (with the heading CIC-PIM) contains the runtimes of the algorithm on fully-, i.e., CIC-PIM-compressed graphs. Generally speaking, CIC-PIM not only reduces data sizes but also helps to speed up processing due to its careful design, or results in fast in-memory processing unattainable previously.
[a]The heading is different from those of other algorithms because Ligra neither supports orientation nor includes any TC algorithms. OOM - Out of Memory.

Table 5
Decoding-Time Percentage.

Dataset	TC	BFS	PageRank	SpMV	Radii	CC
clj	23	25	23	25	20	18
slj	20	23	25	25	23	17
osm	21	26	23	19	22	17
nlp	18	21	27	19	24	20
3dg	18	24	25	25	22	16
twr	24	24	24	23	19	22
c09	21	21	23	20	21	21
gsh	22	23	22	21	18	23
c12	24	27	23	19	20	20

Table 6
Storage Space Saving.

Dataset	TC (%)			Common Graph Algorithms (%)		
	Ligra+	CIC-PIM	Index[a]	Ligra+	CIC-PIM	Index[a]
clj	26	39	61	40	47	60
slj	26	38	62	39	46	60
osm	−22	38	62	0	36	62
nlp	51	63	80	57	61	62
3dg	12	43	62	21	39	62
twr	44	48	59	45	51	57
c09	0	61	82	0	62	86
gsh	59	67	75	62	71	73
c12	56	69	81	57	70	82

All results are relative to the storage sizes of uncompressed graphs (Table 2). CIC-PIM averagely achieves space savings of more than 50% on entire graphs and more than 65% on index data. Note that, compared with the up to around 90% space saving of the techniques for sequential graph processing [12], the space saving of CIC-PIM is relatively low. It is because the application setting of CIC-PIM is the time-critical shared-memory graph processing, and it is hard to compress graphs aggressively without processing speed losses. The negative number (−22) means the increase of storage size.
[a]Space savings of CIC-PIM on index data.

from the fact that a large number of consecutive vertices become sink vertices (no out-edge) after orientation, and causes the compressed index sizes of many chunks that only contain sink vertices to be zero (Recall that only out-edges are stored in the CSR of oriented graphs, and thus all degrees and offset-deltas of the chunks that only contain sink vertices are zero).

4.5.2. Index-data space saving for common graph algorithms

For the unoriented graphs used by common graph algorithms, the index-data space savings range from 57% to 86% with an average of 67%, as shown in the last *Index* column of Table 6. The space savings result from the smallness of the maximum degrees and offset-deltas in most chunks, as discussed in Section 2.4. For TC, the average index-data space saving is 69%, as presented in Section 4.5.1. Compared with the 67% here, the slight 2% increase mainly stems from orientation, which firstly replaces every two reciprocal directed edges with one directed edge and thus reduces average degrees (i.e., increases sparseness), and secondly redirects each edge from the lower degree vertex to the other vertex and thus alleviates the skewness of degree distribution. The two operations of orientation both contribute to the smallness of the maximum degrees and offset-deltas in chunks and hence index-data space saving.

Higher index-data space savings are achieved on huge graphs due to the representation of their offsets and/or degrees with 8-byte integers, as discussed in Section 4.5.1. *nlp* is no longer an exception, because common graph algorithms work on unoriented graphs.

Next, we explain why the index data space savings on relatively small graphs where offsets and degrees are represented with 4-byte integers (from *clj* to *twr*) are all around 60%, especially why there are so many 62%s. We take a non-power-law graph *osm* as an example. *osm* is a road map and the degree of any vertex in the graph is between one and dozens. Thus, the degrees in every chunk can be encoded with one byte each, and the offset-deltas in every chunk can be encode with two bytes each. Hence, the space saving is $(8-1-2)/8 = 63\%$. Due to the introduction of the metadata of chunks, an around 1% loss of space saving is caused and hence the 62%. For another non-power-law graph *3dg*, each vertex has a degree of three and thus the discussions are clearly also tenable. For the relatively small power-law graphs (*clj*, *slj* and *twr*), because the degrees of the vast majority of vertices ≤ 255 ($\int_1^{255} 1.2 * x^{-2.2} dx = 0.999$), the space saving should be near to 62%. However, the existence of a tiny fraction of vertices with high degrees causes a space saving loss of no more than several percent (The analysis to determine the worst-case space saving, i.e., 53%, in Section 3.2.2 is overstrict). Thus, index-data space savings for the relatively small graphs lie around 60%.

4.6. Entire-graph space saving

The space savings of entire-graph storage and peak memory usage due to CIC-PIM compression are similar, and only the results of the former are presented.

4.6.1. Entire-graph space saving for TC

The space savings of compressed oriented graphs used by TC are shown in the second and third columns of Table 6. The additional space savings due to index compression range from 4% to 61% with an average of 24% (the differences of the two columns). The marked space savings come from the reasonable compression efficacy of CIC-PIM and the considerable proportion of index data. Thus, for graphs with high proportions of index

data, i.e., graphs with low average degrees such as *osm* (60%), *3dg* (31%) and *c09* (61%), high additional space savings are achieved.

Compared with uncompressed graphs, the space savings of CIC-PIM-, i.e., fully-compressed graphs range from 38% to 69% with an average of 52%, as shown in the third column of Table 6. The significant space savings are achieved from the compression efficacy of CIC-PIM on index data and the adjlist array. Higher space savings are usually achieved on graphs with huge number of vertices and dense graphs, i.e., graphs with high average degrees. The former include *c09* and *c12* and the latter include *twr* and *gsh*. For graphs with huge number of vertices, vertex IDs, offsets and degrees are represented with 8-byte integers and leave larger portions of leading zeros to be removed as stated in Section 4.5.1. For dense graphs, the adjlist of each vertex tends to contain more neighbors, i.e., there are averagely more integers (i.e., neighbors) scattered between 0 and n-1 (n is the number of vertices). Thus, the deltas between consecutive integers in an adjlist tend to be smaller and less bytes needed to encode these deltas.

For *nlp*, high space saving is gained because many vertex IDs in the adjlist of each vertex are sequential integers, i.e., the deltas between these consecutive integers are one, and only one byte needed to encode each of them.

The −22% space saving on Ligra+-, i.e., partially-compressed *osm* stems from its low average degree (1.1), which causes the space to increase because the introduction of the degree array more than offset the space saving gained from the compression of the adjlist array.

4.6.2. Entire-graph space saving for common graph algorithms

The space savings of the compressed unoriented graphs used by common graph algorithms are shown in the fifth and sixth columns of Table 6. The additional space savings due to index compression range from 4% to 62% with an average of 18% (the differences of the two columns). The marked space savings stem from the reasonable compression efficacy of CIC-PIM and the considerable proportion of index data. The reason for the lower percentage (18%) than that of TC (24%) is because the proportions of index data in unoriented graphs are lower than those in the oriented graphs used by TC, as stated in Section 2.2.

Compared with uncompressed graphs, the space savings of CIC-PIM-, i.e., fully-compressed graphs range from 36% to 71% with an average of 53%, as shown in the sixth column of Table 6. Because of the same reasons for TC (Section 4.6.1), higher space savings are usually achieved on graphs with huge number of vertices and dense graphs while *nlp* is an exception.

4.7. Scalability

To evaluate the multithreaded scalability of CIC-PIM, we show the speedups of CMTC (on fully-compressed graphs) and MTC (which works on uncompressed graphs). As the number of threads varies from 1 to 48 in multiples of 6, CMTC shows better scalability than MTC on all graphs and the scalability improvement is more significant on low average degree graphs (*3dg* and *osm*), as shown in Fig. 9. However, *nlp* is an exception because its average degree is high while its scalability improvement is most significant. Generally speaking, the better scalability of CIC-PIM mainly comes from the alleviated memory wall problem due to reduced data sizes.

When traversing the adjlist of a vertex, the access to the first neighbor is usually random, and the subsequent accesses to the rest neighbors are sequential. When processing graphs with short adjlists, i.e., graphs with low average degrees, there are higher portions of random access than those of graphs with high average degrees, and hence lower effective memory bandwidth. This means that when processing graphs with low average degrees, the memory wall problem is severer. Thus, the processing of graphs with low average degrees (*osm* and *3dg*) is more sensitive to data sizes and gains more significant scalability improvements due to CIC-PIM compression.

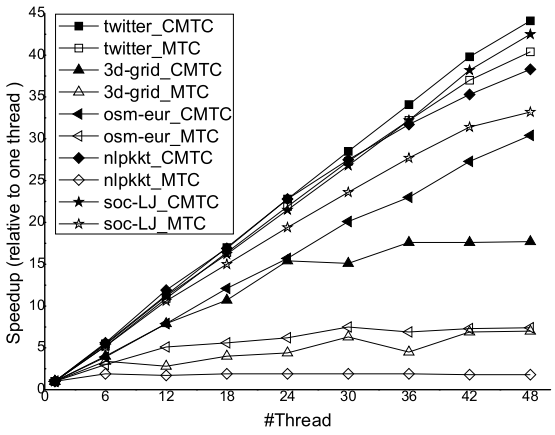


Fig. 9. Multithreaded scalability of CMTC on CIC-PIM-, i.e., fully-compressed graphs vs. that of MTC [20,61], which works on uncompressed graphs. Speedups are normalized to that of one thread. CMTC shows better scalability than MTC on all graphs and the scalability improvement is more significant on low average degree graphs (*3dg* and *osm*) while *nlp* is an exception because its average degree is high but the scalability improvement is significant.

For *nlp*, the exceptional result comes from its special structure feature. The matrix representation of *nlp* is a nearly diagonal matrix, and [31] shows that processing of this kind of graph scales well only with less than four threads. After the number of threads increasing beyond four, the memory bandwidth is quickly saturated and the processing times do not decrease any more, which explains the bad scalability of MTC on *nlp*. *nlp* contains a large number of consecutive vertices with degree of zero and the rest vertices with relatively high degree. Hence, during its processing the portion of random memory access is low, i.e., the effective memory bandwidth is high. When processing the graph (whether compressed or not) with one thread, the limited computing power of one core and the higher effective memory bandwidth make the CPU become bottleneck. When MTC processing the uncompressed graph with one thread, the higher effective memory bandwidth causes the processing is fast (5.2 S); When CMTC processing the fully-compressed graphs with one thread, the computation of decoding causes the runtime to increase significantly due to the limited computing power of one core (68 S). In contrast, when running with 48 threads, the memory is the bottleneck for both MTC and CMTC, and the processing times are comparable (2.85 vs 1.80). Thus, when the number of threads increases from 1 to 48, the speedup of CMTC is much higher than that of MTC (68/1.80 vs. 5.2/2.85).

We omit the results of the remaining four graphs for clearer visualization because they are similar to the results of some graphs shown in Fig. 9. Specifically, the result of *c09* is similar to that of *osm*; the results of *clj*, *gsh*, and *c12* are similar to that of *slj* or *twr*. Because various graph algorithms have the same key operation, there are no significant differences in speedups of different algorithms as stated in Section 4.4.2. Thus, the scalability of CIC-PIM on common graph algorithms is similar to that of TC and is omitted.

4.8. Impact of chunk size

The key design parameter in CIC-PIM is the chunk size, which should be an integer power of two. We evaluate the integer

powers from 64 to 4,096, and observe that space saving decreases slightly while speed increases slightly. With the increase of chunk size, the size of chunk metadata is reduced significantly and the cache hit rate of index data access increases, which explains the slight increase of speed. However, the growth of chunk size increases the possibility that each chunk contains high degree vertices and results in the increase of the compressed index data, which more than offsets the reduction of chunk metadata and explains the slight decrease of space saving. To achieve a good compromise, a chunk size of 256 is chosen.

5. Related work

5.1. Graph processing and compression

There are a large body of work on shared-memory graph processing [40,48,52,54,59] report solutions that can efficiently process large scale graphs. Nonetheless, these solutions do not use compression to reduce memory usage. In contrast, CIC-PIM aims to reduce memory usage while still improving speed.

Studies such as [10–13,42,67] focus on compressing graphs/matrices in sequential settings and achieve space savings of up to around 90%. Nevertheless, they are not suitable for shared-memory processing due to two reasons. First, they aim to specific applications such as neighborhood query in sequential settings and do not take multithreading into consideration. Second, they support random access by decoding extra data and compress data with variable length bit-aligned schemes, both of which are too heavy in time overhead for shared-memory graph processing. Though several more recent works aim at shared-memory settings [7,8,21,28,37,46]. They try to compress specific graph representations such as trees and hence can only support limited graph algorithms [21,28], aim at accelerators such as FPGAs but CPUs [37,46] or can only gain reasonable space saving on graphs of specific scale range [8]. CIC-PIM aims to compress the popular CSR representation of graphs of any scale in shared-memory settings and can support a wide range of graph and sparse matrix algorithms.

Index compression has been studied in the literature [9,14,57]. Nevertheless, these techniques are mainly used to compress inverted index or the index of database systems, and their decoding costs are too high to be practical for shared-memory processing. Compression techniques for the index data of graphs are proposed in [7,8,10–12,21] and usually achieve good results in some metrics. However, they are designed for applications in sequential settings, only support graphs of specific representations or only suitable for graphs of specific scale range.

5.2. Triangle counting

Triangle counting (TC) are generally classified into two categories: exact TC and approximate TC. Exact TC obtains the exact number of triangles and is further classified into shared-memory TC, single node out of core TC, distributed-memory TC and distributed out of core TC. As stated before, share-memory TC, i.e., TC on a single multicore node with all data fit in memory, is generally more efficient than on a cluster regarding cost effectiveness, ease of implementation and speed. However, the applicability of shared-memory TC is limited by the RAM capacity of a single node [19,20,29,33,61,69]. To effectively reduce the working set of shared-memory TC and fit larger graphs into the RAM of a single node, CIC-PIM try to compress the index structures of graphs.

Single node out of core TC leverages the large capacity of external storage of a single node to process larger graphs and shows high cost performance [15,17,44]. However, due to the low I/O bandwidth of a single node, these algorithms suffer from low speed and are much slower than shared-memory TC. Distributed-memory TC may process larger graphs than shared-memory TC [1,2,70]. However, due to the much longer network latency than local RAM and the communication intensive nature of TC, distributed-memory TC is usually significantly slower than shared-memory TC. Distributed out of core TC mostly refers to MapReduce algorithms which can usually process larger graphs [39,41,53,71]. However, due to the long network latency and low I/O bandwidths, works of this type are usually slower than shared-memory TC.

Approximate TC estimates the number of triangles mainly by sampling and usually works on stream graphs [23,36,58,65,66], although it can also work on static graphs [1,64]. Though most approximate TC algorithms aim at single node settings, some of them can also run on distributed systems [1,64]. By sampling, approximate TC works can usually reduce the working set by orders of magnitude times and improve processing speed dramatically. However, the drawback is that they cannot obtain exact results. There are some works leverage matrix computation to get the estimated number of triangles [3,49,50,63]. Though these works can fully use the massive legacy resources of matrix computation to simplify TC, they usually are compute-intensive and hence slow.

6. Conclusion

In this paper, we conduct research on graph compression related issues inspired by TC-related issues in shared-memory settings. Mainly to compress the considerable proportion of index data in graphs represented with the popular CSR format, we propose a scheme CIC-PIM, which achieves significant space savings while still improving speed on TC and a wide range of common graph algorithms. In further work, CIC-PIM can be extended to distributed settings.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by National Key R&D Program of China NO. 2018YFB1003305, NSFC No. 61772216, No. 61832020, No. 61821003, National Science and Technology Major Project No. 2017ZX01032-101, Fundamental Research Funds for the Central Universities. This work is also supported by NSFC 61502190 and CERNET Innovation Project NGII20170120.

References

[1] S. Arifuzzaman, M. Khan, M. Marathe, PATRIC: A parallel algorithm for counting triangles in massive networks, in: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management, ICKM'13, ACM, 2013, pp. 529–538.

[2] S. Arifuzzaman, M. Khan, M. Marathe, A space-efficient parallel algorithm for counting exact triangles in massive networks, in: Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications, HPCC'15, IEEE, 2015, pp. 527–534.

[3] A. Azad, A. Buluç, J. Gilbert, Parallel triangle counting and enumeration using matrix algebra, in: Proceedings of 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPSW'15, IEEE, 2015, pp. 804–811.

[4] Z. Bar-Yossef, R. Kumar, D. Sivakumar, Reductions in streaming algorithms, with an application to counting triangles in graphs, in: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'02, Society for Industrial and Applied Mathematics, 2002, pp. 623–632.

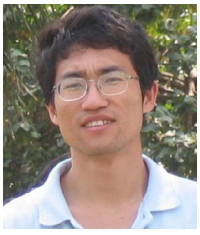
[5] S. Beamer, K. Asanović, D. Patterson, Direction-optimizing breadth-first search, Sci. Program. 21 (3–4) (2013) 137–148.
[6] L. Becchetti, P. Boldi, C. Castillo, A. Gionis, Efficient semi-streaming algorithms for local triangle counting in massive graphs, in: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, SIGKDD'08, ACM, 2008, pp. 16–24.
[7] M. Besta, T. Hoefler, Survey and taxonomy of lossless graph compression and space-efficient graph representations, arXiv preprint arXiv:1806.01799.
[8] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, T. Hoefler, Log(Graph): a near-optimal high-performance graph representation, in: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT'18, 2018, pp. 7:1–7:13.
[9] B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, R. Sherkat, Efficient index compression in db2 luw, Proc. VLDB Endow. 2 (2) (2009) 1462–1473.
[10] D.K. Blandford, G.E. Blelloch, I.A. Kash, Compact representations of separable graphs, in: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'03, Society for Industrial and Applied Mathematics, 2003, pp. 679–688.
[11] D.K. Blandford, G.E. Blelloch, I.A. Kash, An experimental analysis of a compact graph representation, in: Proceedings of the Workshop on Analytic Algorithmics and Combinatorics, ANALCO'04, 2004, pp. 49–61.
[12] P. Boldi, S. Vigna, The webgraph framework i: Compression techniques, in: Proceedings of the 13th International Conference on World Wide Web, WWW'04, ACM, 2004, pp. 595–602.
[13] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, P. Raghavan, On compressing social networks, in: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Ciscovery and Data Mining, SIGKDD'09, ACM, 2009, pp. 219–228.
[14] F. Claude, A. Fariña, M.A. Martínez-Prieto, G. Navarro, Indexes for highly repetitive document collections, in: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, ICKM'11, ACM, 2011, pp. 463–468.
[15] Y. Cui, D. Xiao, D.B. Cline, D. Loguinov, Improving i/o complexity of triangle enumeration, in: Proceedings of the 2017 IEEE International Conference on Data Mining, ICDM'17, IEEE, 2017, pp. 61–70.
[16] Y. Cui, D. Xiao, D. Loguinov, On efficient external-memory triangle listing, in: Proceedings of the 16th IEEE International Conference on Data Mining, ICDM'16, IEEE, 2016, pp. 101–110.
[17] Y. Cui, D. Xiao, D. Loguinov, On efficient external-memory triangle listing, IEEE Trans. Knowl. Data Eng. (TKDE) 31 (2019) 1555–1568.
[18] D.M. Da Zheng, R. Burns, J. Vogelstein, C.E. Priebe, A.S. Szalay, FlashGraph: Processing billion-node graphs on an array of commodity ssds, in: Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST'15, 2015, pp. 45–58.
[19] T.A. Davis, Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss, in: Proceedings of the 2018 IEEE High Performance Extreme Computing Conference, HPEC'18, IEEE, 2018, pp. 1–6.
[20] L. Dhulipala, G.E. Blelloch, J. Shun, Theoretically efficient parallel graph algorithms can be fast and scalable, in: Proceedings of the 2018 ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'18, 2018.
[21] L. Dhulipala, G.E. Blelloch, J. Shun, Low-latency graph streaming using compressed purely-functional trees, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'19, ACM, 2019, pp. 918–934.
[22] U. Drepper, What every programmer should know about memory, Red Hat, Inc.
[23] D. Ediger, K. Jiang, J. Riedy, D.A. Bader, Massive streaming data analytics: A case study with clustering coefficients, in: Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, IPDPSW'10, 2010, pp. 1–8.
[24] M. Faloutsos, P. Faloutsos, C. Faloutsos, On power-law relationships of the internet topology, in: ACM SIGCOMM Computer Communication Review, Vol. 29, ACM, 1999, pp. 251–262.
[25] T. Gao, Y. Lu, B. Zhang, G. Suo, Using the intel many integrated core to accelerate graph traversal, Int. J. High Perform. Comput. Appl. 28 (2014) 255–266.
[26] I. Giechaskiel, G. Panagopoulos, E. Yoneki, PDTL: Parallel and distributed triangle listing for massive graphs, in: Proceedings of the 44th IEEE International Conference on Parallel Processing, ICPP'15, IEEE, 2015, pp. 370–379.
[27] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: Distributed graph-parallel computation on natural graphs, in: Proceedings of 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI'12, 2012, pp. 17–30.
[28] T. Granskog, A. Strigér, A comparison of search times on compressed and uncompressed graphs, 2015.
[29] O. Green, L.-M. Munguía, D.A. Bader, Load balanced clustering coefficients, in: Proceedings of the 1st Workshop on Parallel Programming for Analytics Applications, PPAA'14, ACM, 2014, pp. 3–10.
[30] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, R. Zadeh, WTF: The who to follow service at twitter, in: Proceedings of the 22nd International Conference on World Wide Web, WWW'13, ACM, 2013, pp. 505–514.
[31] V. Gupta, H. Kim, K. Schwan, Evaluating Scalability of Multi-Threaded Applications on a Many-Core Platform, Tech. Rep., Georgia Institute of Technology, 2012.
[32] H. Homann, F. Laenen, SoAx: A generic c++ structure of arrays for handling particles in hpc codes, Comput. Phys. Comm. 224 (2018) 325–332.
[33] Y. Hu, H. Liu, H.H. Huang, Tricore: Parallel triangle counting on gpus, in: Proceedings of the 20th IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'18, IEEE, 2018, pp. 171–182.
[34] X. Hu, Y. Tao, C.-W. Chung, Massive.graph. triangulation, Massive graph triangulation, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD'13, ACM, 2013, pp. 325–336.
[35] H. Inoue, M. Ohara, K. Taura, Faster set intersection with simd instructions by reducing branch mispredictions, Proc. VLDB Endow. 8 (2014) 293–304.
[36] M. Jha, C. Seshadhri, A. Pinar, A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox, ACM Trans. Knowl. Discov. Data (TKDD) 9 (3) (2015) 15.
[37] S. Khoram, J. Zhang, M. Strange, J. Li, Accelerating graph analytics by co-optimizing storage and access on an fpga-hmc platform, in: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA'18, ACM, 2018, pp. 239–248.
[38] J. Kim, W.-S. Han, S. Lee, K. Park, H. Yu, Opt: a new framework for overlapped and parallel triangulation in large-scale graphs, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD'14, ACM, 2014, pp. 637–648.
[39] H. Kim, S. Kim, J.-K. Min, An efficient triangle enumeration on parallel and distributed frameworks, in: Proceedings of the 2018 IEEE International Conference on Big Data and Smart Computing, BigComp'18, IEEE, 2018, pp. 545–548.
[40] S. Ko, W.-S. Han, Turbograph++: A scalable and fast graph analytics system, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD'18, ACM, 2018, pp. 395–410.
[41] T.G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, C. Task, Counting triangles in massive graphs with mapreduce, SIAM J. Sci. Comput. 36 (5) (2014) S48–S77.
[42] K. Kourtis, G.I. Goumas, N. Koziris, Optimizing sparse matrix–vector multiplication using index and value compression, in: Proceedings of the 2008 Conference on Computing Frontiers, CF'08, ACM, 2008.
[43] K. Kourtis, G. Goumas, N. Koziris, Exploiting compression opportunities to improve spmxv performance on shared memory systems, ACM Trans. Archit. Code Optim. (TACO) 7 (3) (2010) 16.
[44] A. Kyrola, G. Blelloch, C. Guestrin, GraphChi: Large-scale graph computation on just a pc, in: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI'12, 2012, pp. 31–46.
[45] M. Latapy, Main-memory triangle computations for very large (sparse (power-law)) graphs, Theoret. Comput. Sci. 407 (2008) 458–473.
[46] J. Lee, H. Kim, S. Yoo, K. Choi, H.P. Hofstee, G.-J. Nam, M.R. Nutter, D. Jamsek, EXtrav: Boosting graph processing near storage with a coherent accelerator, Proc. VLDB Endow. 10 (2017) 1706–1717.
[47] H. Liu, H.H. Huang, Enterprise: breadth-first graph traversal on gpus, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'15, 2015, pp. 68:1–68:12.
[48] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, GraphLab: A new framework for parallel machine learning, in: Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence, UAI'10, 2010, pp. 340–349.
[49] T.M. Low, V.N. Rao, M. Lee, D. Popovici, F. Franchetti, S. McMillan, First look: Linear algebra-based triangle counting without matrix multiplication, in: Proceedings of the 2017 IEEE High Performance Extreme Computing Conference, HPEC'17, IEEE, 2017, pp. 1–6.
[50] J. Moody, Matrix methods for calculating the triad census, Social Networks 20 (4) (1998) 291–299.
[51] M.E. Newman, The structure and function of complex networks, SIAM Rev. 45 (2) (2003) 167–256.
[52] D. Nguyen, A. Lenharth, K. Pingali, A lightweight infrastructure for graph analytics, in: Proceedings of 24th ACM SIGOPS Symposium on Operating Systems Principles, SOSP'13, 2013, pp. 456–471.

[53] H.-M. Park, S.-H. Myaeng, U. Kang, PTE: Enumerating trillion triangles on distributed systems, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, SIGKDD'16, ACM, 2016, pp. 1115–1124.
[54] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, J. Leskovec, Ringo: Interactive graph analytics on gig-memory machines, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD'15, ACM, 2015, pp. 1105–1110.
[55] J. Riedy, H. Meyerhenke, D.A. Bader, D. Ediger, T.G. Mattson, Analysis of streaming social networks and graphs on multicore architectures, in: Proceedings of the 2012 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'12, IEEE, 2012, pp. 5337–5340.
[56] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, P. Dubey, Can traditional programming bridge the ninja performance gap for parallel computing applications?, in: Proceedings of the 39th IEEE Annual International Symposium on Computer Architecture, ISCA'12, IEEE, 2012, pp. 440–451.
[57] F. Scholer, H.E. Williams, J. Yiannis, J. Zobel, Compression of inverted indexes for fast query evaluation, in: Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'02, ACM, 2002, pp. 222–229.
[58] K. Shin, E. Lee, J. Oh, M. Hammoud, C. Faloutsos, Dislr: Distributed sampling with limited redundancy for triangle counting in graph streams, arXiv preprint arXiv:1802.04249.
[59] J. Shun, G.E. Blelloch, Ligra: A lightweight graph processing framework for shared memory, in: Proceedings of 18th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP'13, 2013, pp. 135–146.
[60] J. Shun, L. Dhulipala, G.E. Blelloch, Smaller and faster: Parallel processing of compressed graphs with ligra+, in: 2015 Data Compression Conference, DCC'15, 2015, pp. 403–412.
[61] J. Shun, K. Tangwongsan, Multicore triangle computations without tuning, in: Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE'15, IEEE, 2015, pp. 149–160.
[62] I.-J. Sung, G.D. Liu, W.-M.W. Hwu, DL: A data layout transformation system for heterogeneous computing, in: Innovative Parallel Computing, InPar'12, IEEE, 2012, pp. 1–11.
[63] C.E. Tsourakakis, Counting triangles in real-world networks using projections, Knowl. Inf. Syst. 26 (3) (2011) 501–520.
[64] C.E. Tsourakakis, U. Kang, G.L. Miller, C. Faloutsos, DOULION: Counting triangles in massive graphs with a coin, in: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, SIGKDD'09, ACM, 2009, pp. 837–846.
[65] A. Turk, D. Türkoglu, Revisiting wedge sampling for triangle counting, in: Proceedings of the 2019 The World Wide Web Conference, WWW'19, 2019, pp. 1875–1885.
[66] P. Wang, P. Jia, Y. Qi, Y. Sun, J. Tao, X. Guan, REPT: A streaming algorithm of approximating global and local triangle counts in parallel, in: Proceedings of the 35th IEEE International Conference on Data Engineering, ICDE'19, IEEE, 2019, pp. 758–769.
[67] J. Willcock, A. Lumsdaine, Accelerating sparse matrix computations via data compression, in: Proceedings of the 20th Annual International Conference on Supercomputing, ISC'06, ACM, 2006, pp. 307–316.
[68] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, Scientific computing kernels on the cell processor, Int. J. Parallel Program. (2007) 263–298.
[69] A. Yaşar, S. Rajamanickam, M. Wolf, J. Berry, Ü.V. Çatalyürek, Fast triangle counting using cilk, in: Proceedings of 2018 IEEE High Performance Extreme Computing Conference, HPEC'18, IEEE, 2018, pp. 1–7.
[70] Y. Zhang, H. Jiang, F. Wang, Y. Hua, D. Feng, X. Xu, LiteTE: Lightweight communication-efficient distributed-memory triangle enumerating, IEEE Access 7 (2019) 26294–26306.
[71] Y. Zhu, H. Zhang, L. Qin, H. Cheng, Efficient mapreduce algorithms for triangle listing in billion-scale graphs, Distrib. Parallel Databases 35 (2) (2017) 149–176.

Yongxuan Zhang received the B.E. degree in Computer Science and Technology from Nanchang Hangkong University, Nanchang, China, in 2005, and the Ph.D. degree in Computer Science and Technology from Huazhong University of Science and Technology, Wuhan, China, in 2020. He is currently an instructor of Mathematics and Computer School at Yuzhang Normal University. His present research interests include graph processing, high-performance computing, big data computing and machine learning.

Hong Jiang received the B.Sc. degree in Computer Engineering from Huazhong University of Science and Technology, Wuhan, China; the M.A.Sc. degree in Computer Engineering from the University of Toronto, Toronto, Canada; and the Ph.D. degree in Computer Science from the Texas A&M University, College Station, Texas, USA. He is currently Chair and Wendell H. Nedderman Endowed Professor of Computer Science and Engineering Department at the University of Texas at Arlington. His present research interests include computer architecture, computer storage systems and parallel I/O, highperformance computing, big data computing, cloud computing, performance evaluation. He is a Topic and Associate Editor of the IEEE Transactions on Computers and recently served as an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems. He has over 300 publications in major journals and international Conferences in these areas, including IEEE-TPDS, IEEE-TC, Proceedings of IEEE, ACM-TACO, ACM-ToS, USENIX ATC, FAST, EUROSYS, ISCA, MICRO, SOCC, LISA, SIGMETRICS, ICDE, DAC, DATE, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, INFOCOM, ICPP, etc. Dr. Jiang is a Fellow of IEEE, and Member of ACM.

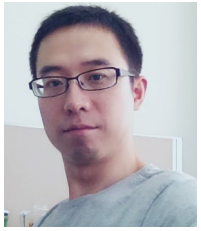
Fang Wang received the BE and master's degrees in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994 and 1997, respectively, and the Ph.D. degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2001, where she is currently a professor of computer science and engineering. Her research interests include distribute file systems, parallel I/O storage systems, and graph processing systems. She has more than 80 publications in major journals and conferences, including IEEE TC/TPDS/TNSM,ACM TACO, SC,MSST,DATE,HiPC, ICDCS, HPDC, ICCD, ICDE and ICPP.



Yu Hua received the B.E. and Ph.D. degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He is currently a professor at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing and network storage. He has more than 80 papers to his credit in major journals and international conferences including IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS), USENIX ATC, USENIX FAST, INFOCOM, SC, ICDCS, ICPP and MASCOTS. He has been on the organizing and program committees of multiple international conferences, including INFOCOM, ICDCS, ICPP, RTSS and IWQoS. He is a senior member of the IEEE and CCF, a member of ACM, and USENIX.

Dan Feng received the B.E., M.E., and Ph.D. degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEE-TC, IEEE- TPDS, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She serves on the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012. She is a member of IEEE and a member of ACM.

Yongli Cheng received the BE degree from the Chang'an University, Xi'an, China, in 1998, the MS degree from the FuZhou University, FuZhou, China, in 2010, and the Ph.D. degree from the Huazhong University of Science and Technology, Wuhan, China, 2017. He is currently a teacher of the College of Mathematics and Computer Science, FuZhou University currently. His current research interests include computer architecture and graph computing. He has several publications in major international conferences and journals, including HPDC, IWQoS, INFOCOM, ICPP, the Future Generation Computing Systems, IEEE/ACM Transactions on Networking, and Frontiers of Computer Science.



Yuchong Hu received the B.Eng. degree in Computer Science and Technology from Special Class for the Gifted Young (SCGY), the University of Science and Technology of China (USTC) in 2005, and the Ph.D. degree in Computer Software and Theory from the University of Science and Technology of China (USTC) in 2010. His research mainly focuses on designing and implementing intelligent reliability mechanisms, based on fault-tolerance (such as Network Coding or Erasure Coding), to improve reliability, performance and security for storage systems, which include cloud storage, big-data storage, deduplicated backup, heterogeneous/hierarchical storage, in-memory NoSQL database etc. He published 10 papers as the first/corresponding author in conferences FAST, INFOCOM, SoCC, and journals JSAC, TOS, TIT, TIFS, TPDS. He also published more than 40 papers in major journals and conferences, including TC, ATC, DSN, MSST, IWQoS, SRDS, ICPP, ICPADS, ISPA, ISIT, ICC, etc.

Renzhi Xiao received the B.E. degree in software engineering from Jiangxi University of Science and Technology, Nanchang, China, in 2013. He is currently working toward the Ph.D. degree majoring in computer architecture at Huazhong University of Science and Technology(HUST), Wuhan, China. His research interests include computer architecture, in-memory key–value store, non-volatile memory, and NVM-based data structures.