GraphMP

论文 system Paper

current out-of-core systems still have much lower performance (5-100M edges/s) than in-memory approaches (1-2B edges/s)

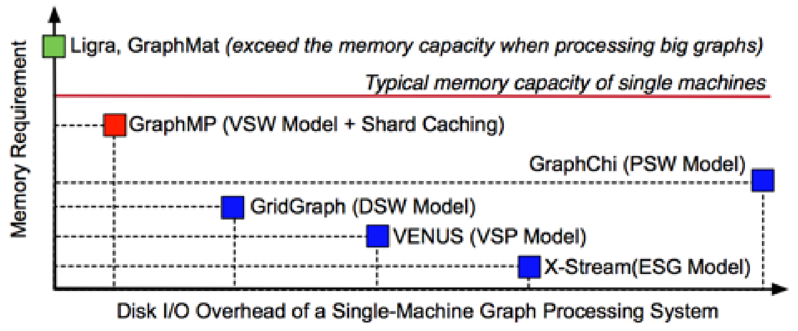


Fig. 1: GraphMP is a single-machine semi-external-memory graph processing system. Compared to in-memory systems like Ligra and GraphMat, GraphMP can handle big graphs on a single machine, since it does not store all graph data in memory. Compared to out-of-core systems (e.g., GraphChi, X-Stream, VENUS and GridGraph), GraphMP could fully utilize available memory of a typical server to reduce disk I/O overhead.

GraphMP

- does not need to store all edges in memory (compared to in-memory)
- requires more memory to store all vertices (compared to out-of-core)
- is designed to run on a commodity muti-core server with HDDs.

GraphMP Highlights:

- **vertex-centric sliding window (VSW)**
- **selective scheduling**, so that inactive shards can be skipped to avoid unnecessary disk accesses and processing
- **compressed shard cache mechanism** to fully utilize available memory to cache a partition of shards in memory

System Design

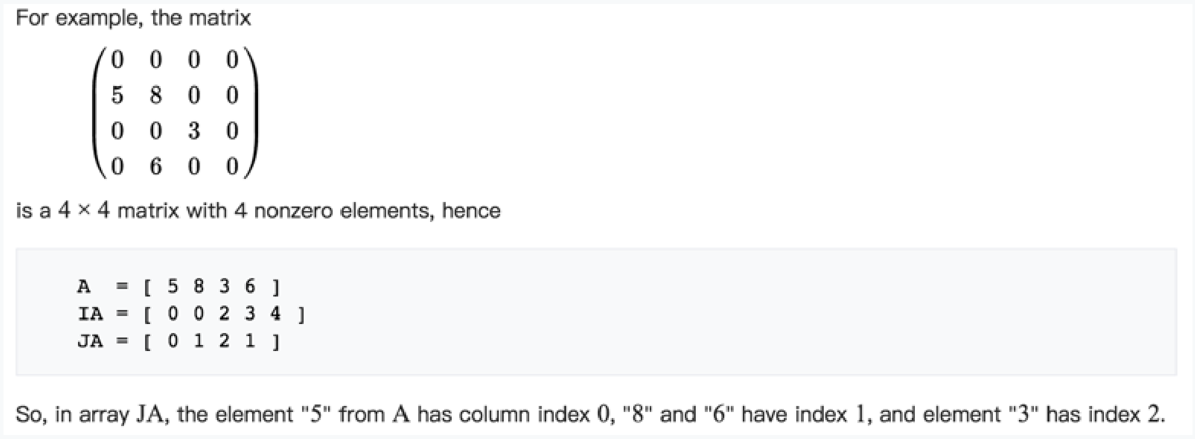
Graph Sharding and Data Storage

Compared to GraphChi, GraphMp groups edges in a shard by their destination, and stores them in key-values pair $(id(v), \Gamma_{in}(v))$.
The number of shards and vertex intervals are chosen with:

1. any shard can be completely loaded into the main memory
2. the number of edges in each shard is balanced

> In this work, each shard approximately contains 18-22M edges, so that a single shard roughly needs 80MB memory.

CSR
`col` array stores all edgens' column indices in row-major order
`row` array records each vertex's adjacency list distribution

> **Compressed sparse row (CSR, CRS or Yale format)**
>
> - The array A is of length NNZ and holds all the nonzero entries of $\mathbf{M}$ in left-to-right top-to-bottom ("row-major") order.
> - The array IA is of length $m + 1$. It is defined by this recursive definition:
>   - $IA[0] = 0$
>   - $IA[i] = IA[i-1]$ + (number of nonzero elements on the $(i-1)$-th row in the original matrix)
> - The third array, JA, contains the column index in $\mathbf{M}$ of each element of A and hence is of length NNZ as well.
>
> 
>
> For example, the matrix
>
> $$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$
>
> is a 4 × 4 matrix with 4 nonzero elements, hence
>
> ```
>    A  = [ 5 8 3 6 ]
>    IA = [ 0 0 2 3 4 ]
>    JA = [ 0 1 2 1 ]
> ```
>
> So, in array JA, the element "5" from A has column index 0, "8" and "6" have index 1, and element "3" has index 2.

preprocess an input graph, and generates all edge shards and metadata files

1. Scan the whole graph to get its basic information, and record the in-degree and out-degree of each vertex.
2. Compute vertex intervals to guarantee that, (1) each shard is samll enough to be loaded into memory, (2) the number of edges in each shard is balanced.
3. Read the graph data sequentially, and append each edge to a shard file based on its destination and vertex intervals.
4. Transform all shard files to the CSR format, and persist the metadata files on disk.

The Vertex-Centric Sliding Window Computation Model

Selective Scheduling

Given a shard, if **all source vertices of its associated edges are inactive**, it is an inactive shard. An inactive shard would not generate any updates in the following iteration. Therefore, it is unnecessary to load and process these inactive shards.

Bloom filters
For each shard, GraphMP manages a Bloom filter to record the source vertices of its edges.
GraphMP only enables selective scheduling when the ratio of active vertices is lower than a threshold.

Compressed Edge Caching

Motivation:

> For example, given a server with 24 CPU cores and 128GB memory, when running PageRank on a graph with 1.1 billion vertices, GraphMP uses 21GB memory to store all data, including SrcVertexArray, DstVertexArray, the out- degree array, Bloom filters, and the shards under processing.

build an in-application cache system.

Compress shard and store in cache system.
Compressors: **snappy** and **zlib**

- model-1: uncompressed shards
- model-2: snappy compressed shards
- model-3: zlib-1 compressed shards
- model-4: zlib-3 compressed shards

model 1->2->3->4 higher compression ratio but longer decompressing time

Quantitative Comparion

Category	PSW (GraphChi)	ESG (X-Stream)	VSP (VENUS)	DSW (GridGraph)	VSW (GraphMP)
Data Read	$C\|V\| + 2(C+D)\|E\|$	$C\|V\| + (C+D)\|E\|$	$C(1+\delta)\|V\| + D\|E\|$	$C\sqrt{P}\|V\| + D\|E\|$	$\theta D\|E\|$
Data Write	$C\|V\| + 2(C+D)\|E\|$	$C\|V\| + C\|E\|$	$C\|V\|$	$C\sqrt{P}\|V\|$	0
Memory Usage	$(C\|V\| + 2(C+D)\|E\|)/P$	$C\|V\|/P$	$C(2+\delta)\|V\|/P$	$2C\|V\|/\sqrt{P}$	$2C\|V\| + ND\|E\|/P$

$\delta \approx (1 - e^{-d_{avg}/P})P, 0 \le \theta \le 1$

GraphChi

For each iteration, GraphChi uses three steps to processes one shard:

1. loading its associated vertices, in-edges and out-edges from disk into memory (Read $C|V| + 2(C+D)|E|$);
2. updating the vertex values (Memory Usage $(C|V| + 2(C+D)|E|)/P$);
3. writing the updated vertices (which are stored with edges) to disk (Write $C|V| + 2(C+D)|E|$).

X-Stream

1. loads its associated vertices and processes it out-edges, generating and propagating updates to corresponding values on disk (Read $C|V| + D|E|$; Write $C|E|$);
2. processes all updates and uses them to update vertex values on disk (Read $C|E|$; Write $C|V|$).

X-Stream only needs to keep the vertices of a partition in memory, so the memory usage is $C|V|/P$.

VENUS

1. loading a v-shard into the main memory (Read $D|E|$);
2. processing its corresponding g-shard in a streaming fashion (Memory Usage $C(2+\delta)|V|/P$);
3. writing updated vertices to disk (Write $C|V|$).

GridGraph

the $|V|$ vertices are divided into $\sqrt{P}$ equalized vertex chunks and $|E|$ edges are partitioned into $\sqrt{P} \times \sqrt{P}$ blocks according to the source and destination vertices.

GraphMP

GraphMP keeps all source and destination vertices in the main memory during the vertex-centric computation. There- fore, GraphMP would **not incur any disk write** for vertices in each iteration until the end of the program.
N CPU cores to process P edge shards in parallel (Read $D|E|$, due to the compressed edge cache mechanism, actual size is $\theta D|E|$).
Each shard needs incoming vertices and outgoing vertices, and each CPU core loads $|E|/P$ edges in memory, os the total memory usage is $2C|V| + ND|E|/P$.